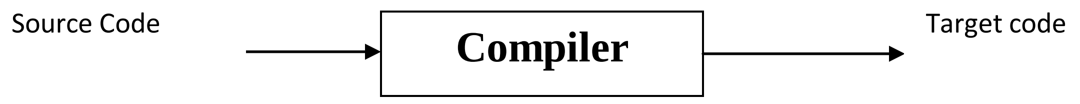
United College of Engineering and Research, Prayagraj
Computer Science and Engineering and Department
Compiler Design
Unit 1

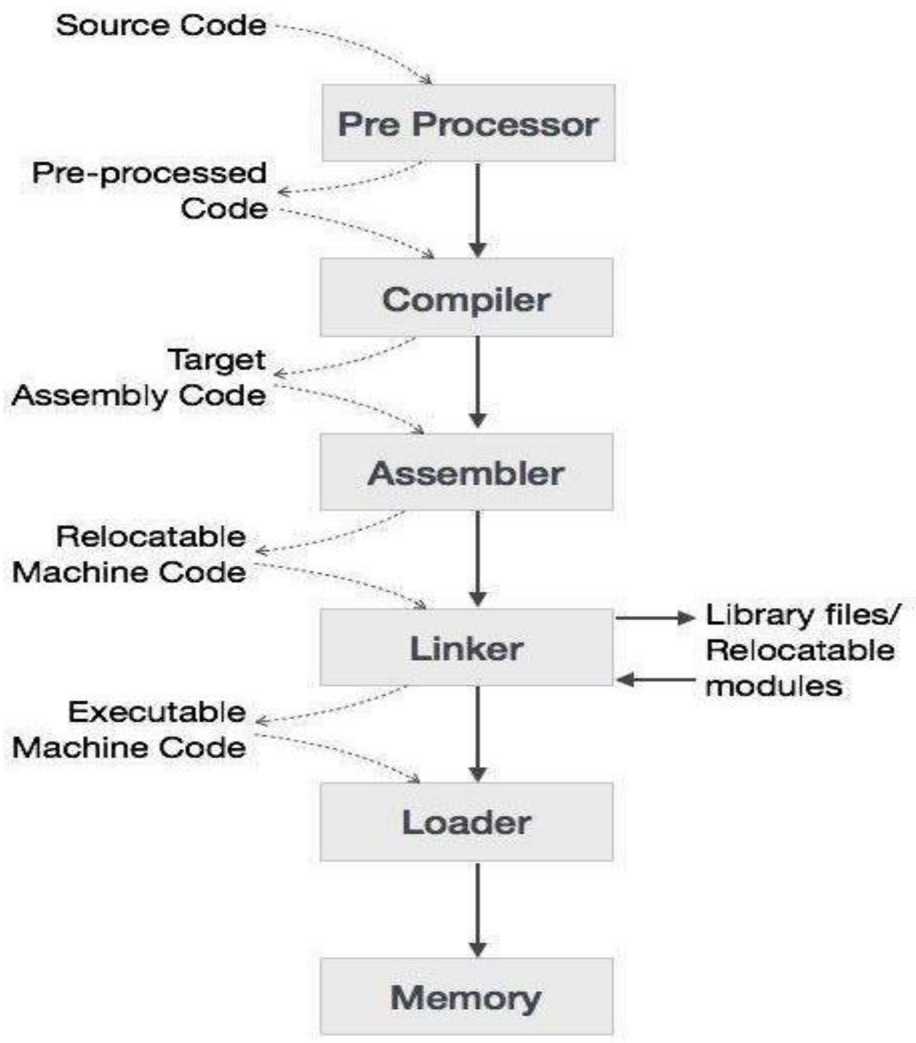
Semester: 5th (CSE) Prepared By: Prabhat Shukla

Compiler:-

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.



Language Processing System:-



Preprocessor:-

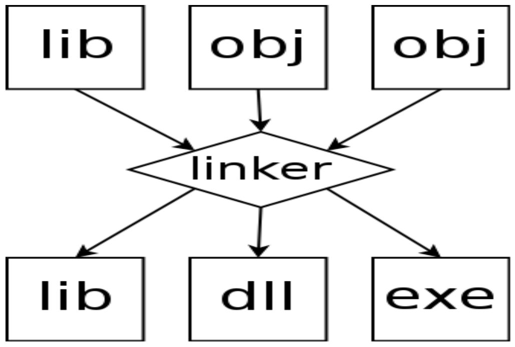
A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation; file inclusion, language extension, etc.

Assembler:-

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker:-

Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.



Types of linker:-

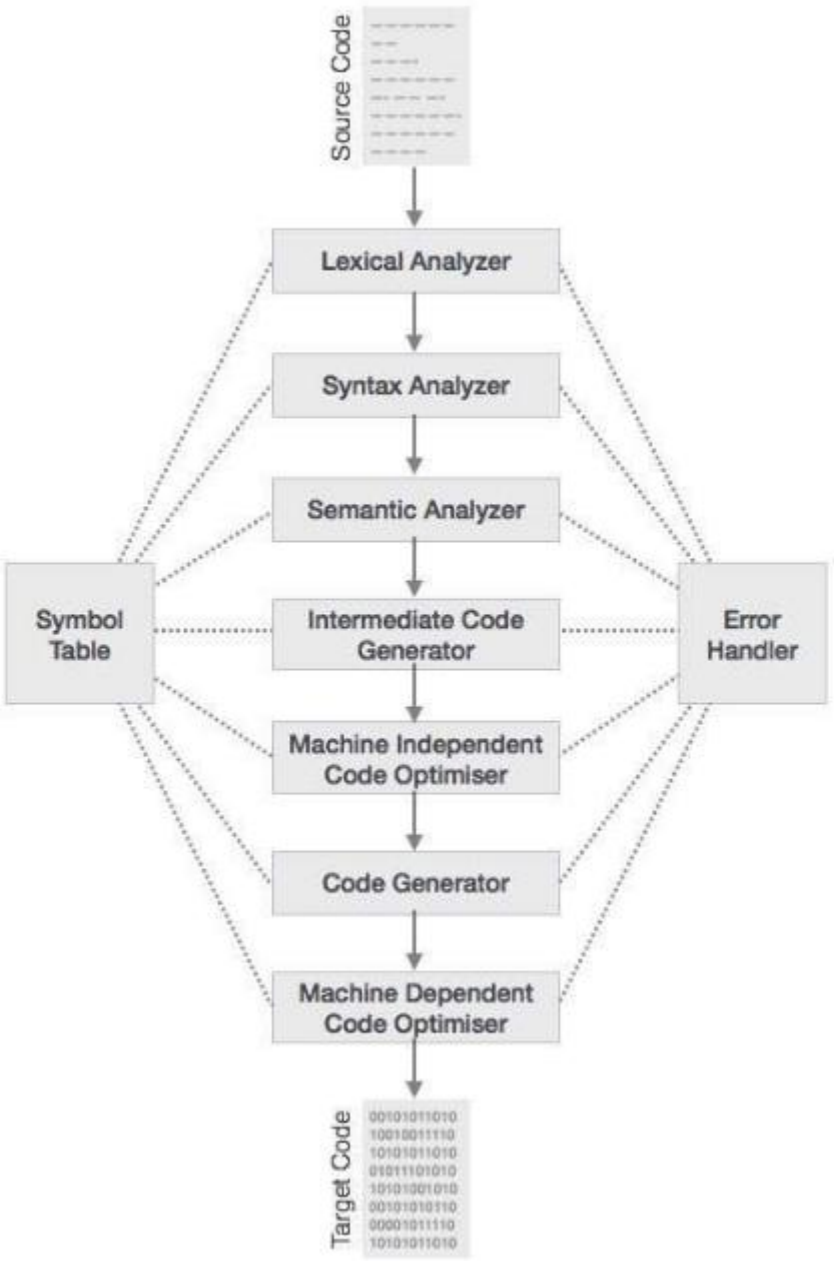
Dynamic Linker:- Many OS environment allow dynamic linking, that is postponing of the resolution of some undefined symbol or library function until a program is run.

Static Linker:- It is the result of linker copying all library routines used in the program into the executable image. This may require more disk-space and memory than dynamic linking.

Loader:-

Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

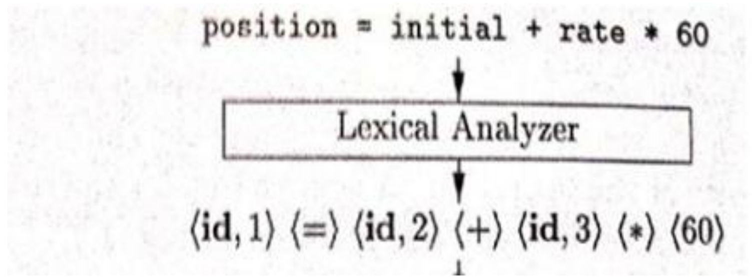
Phases of Compiler:-

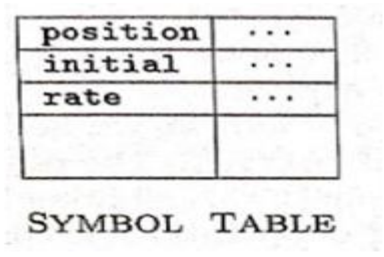


example. float position, initial, rate;
position=initial+rate*60

Lexical Analysis:-

This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens.


position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩

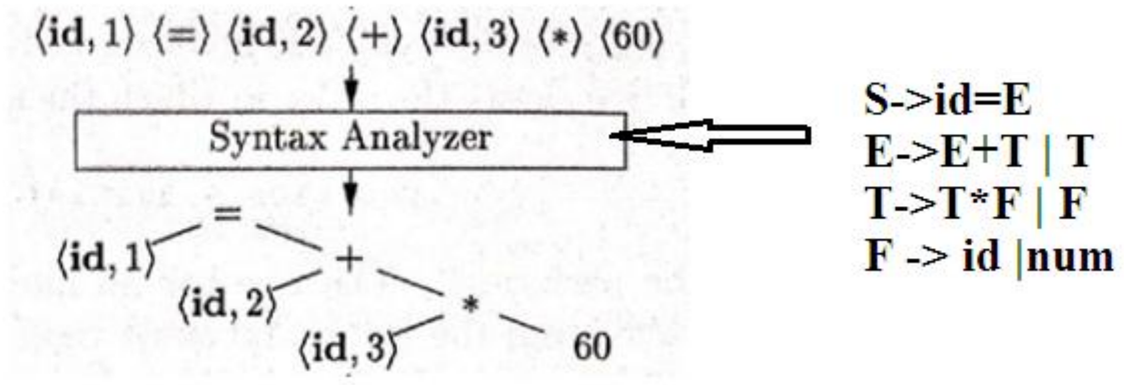


position	...
initial	...
rate	...

SYMBOL TABLE

Syntax Analysis:-

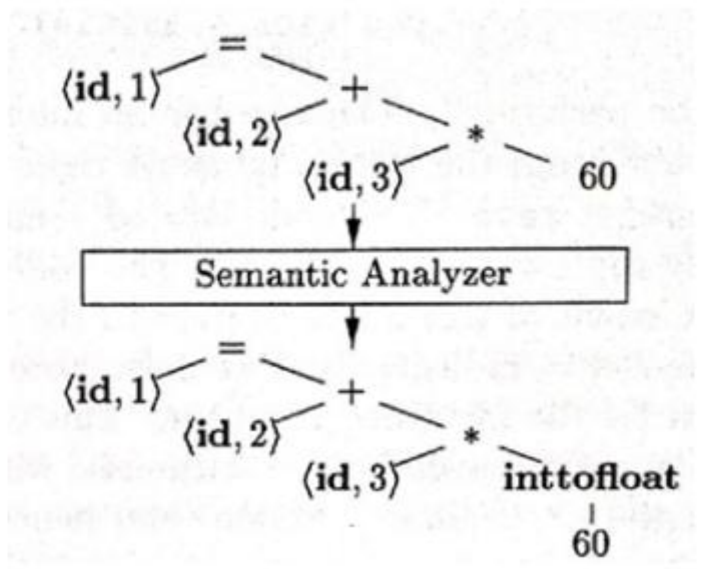
It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.



S->id=E
E->E+T | T
T->T*F | F
F -> id |num

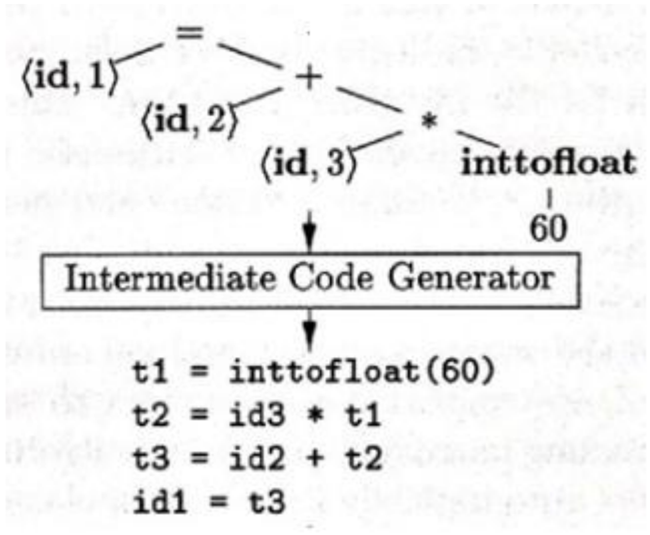
Semantic Analysis:-

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.



Intermediate Code Generation:-

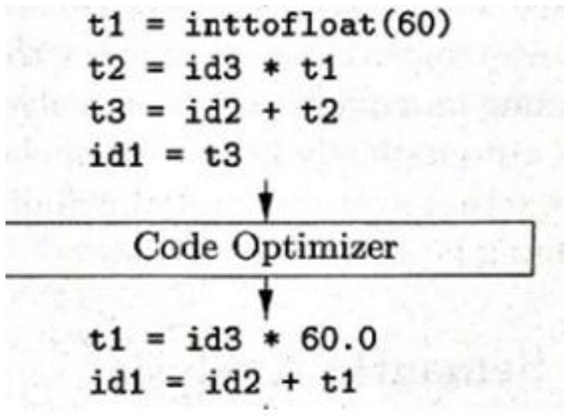
After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.



```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimization:-

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

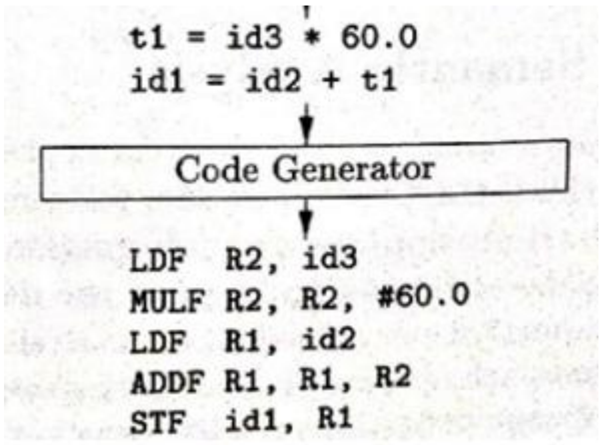


```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

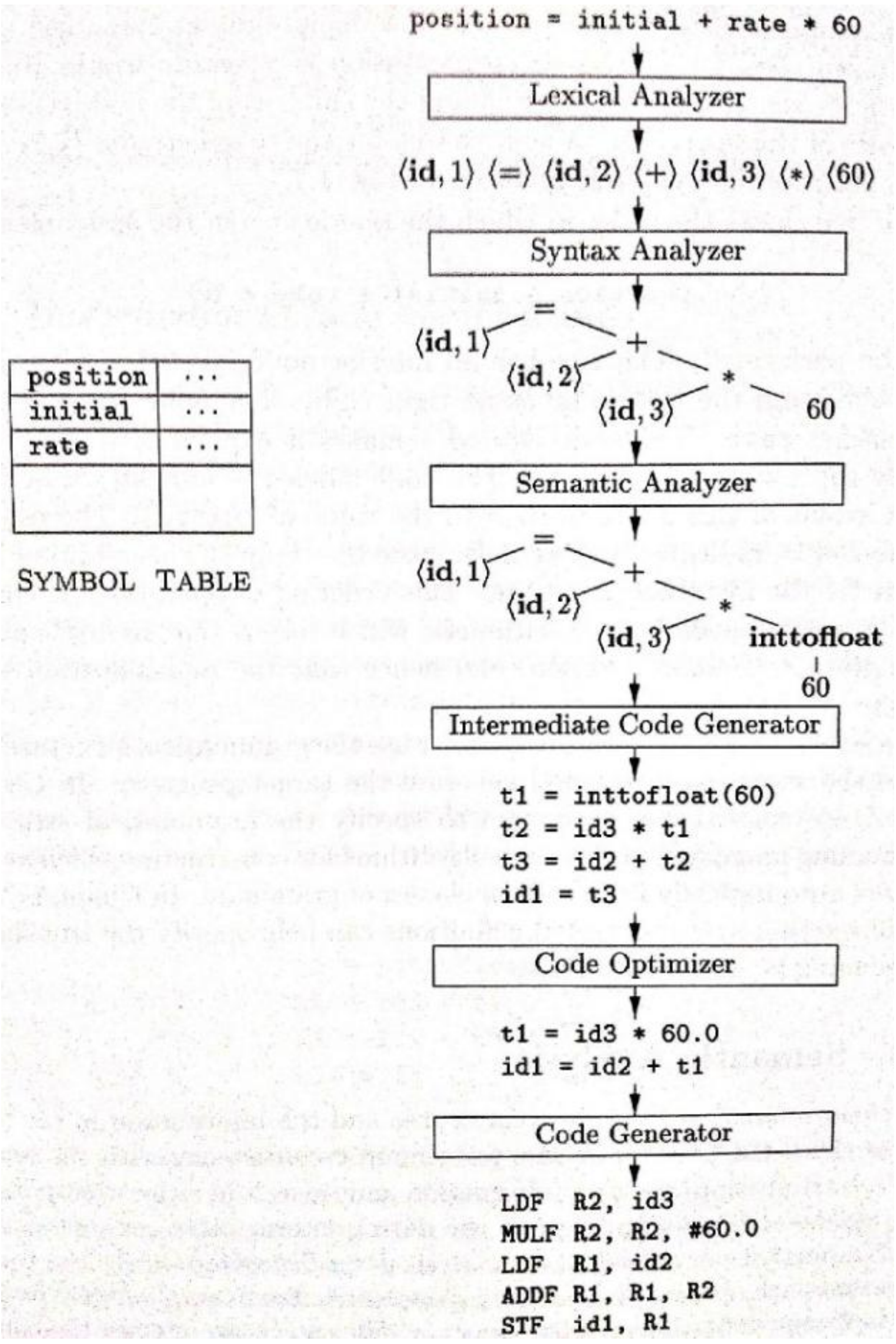
Code Generation:-

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.



Symbol Table:-

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

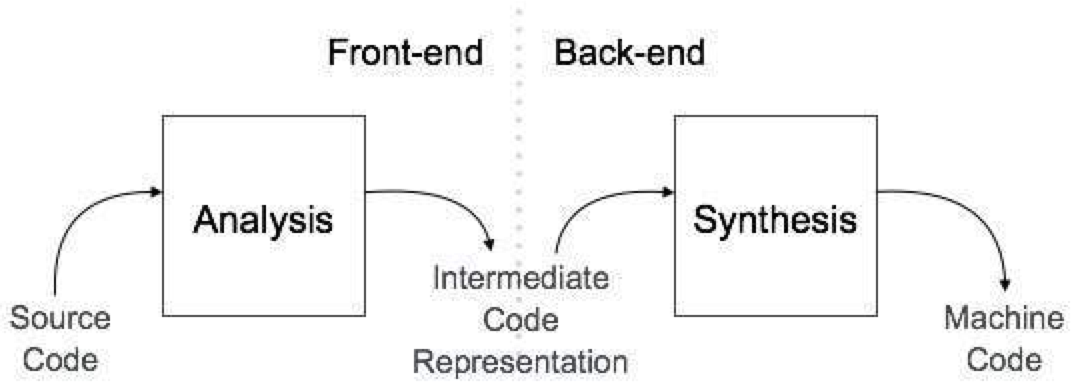


position	...
initial	...
rate	...

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```



Analysis Phase:-

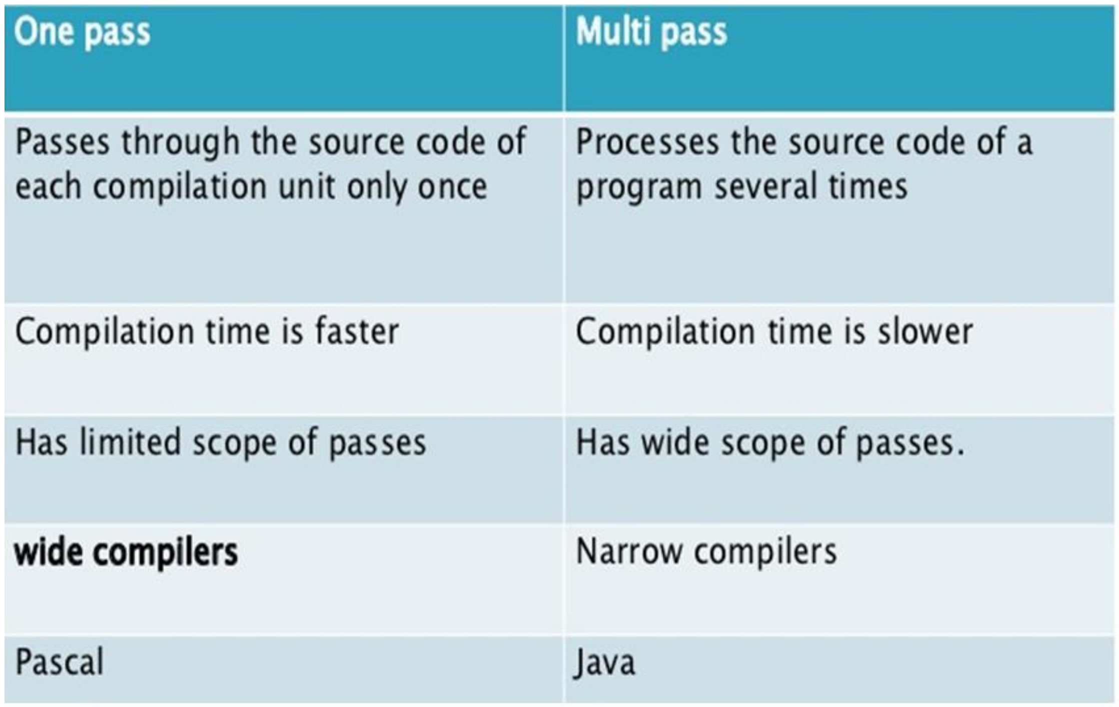
The **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate representation of the source program.

Synthesis Phase:-

The **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.

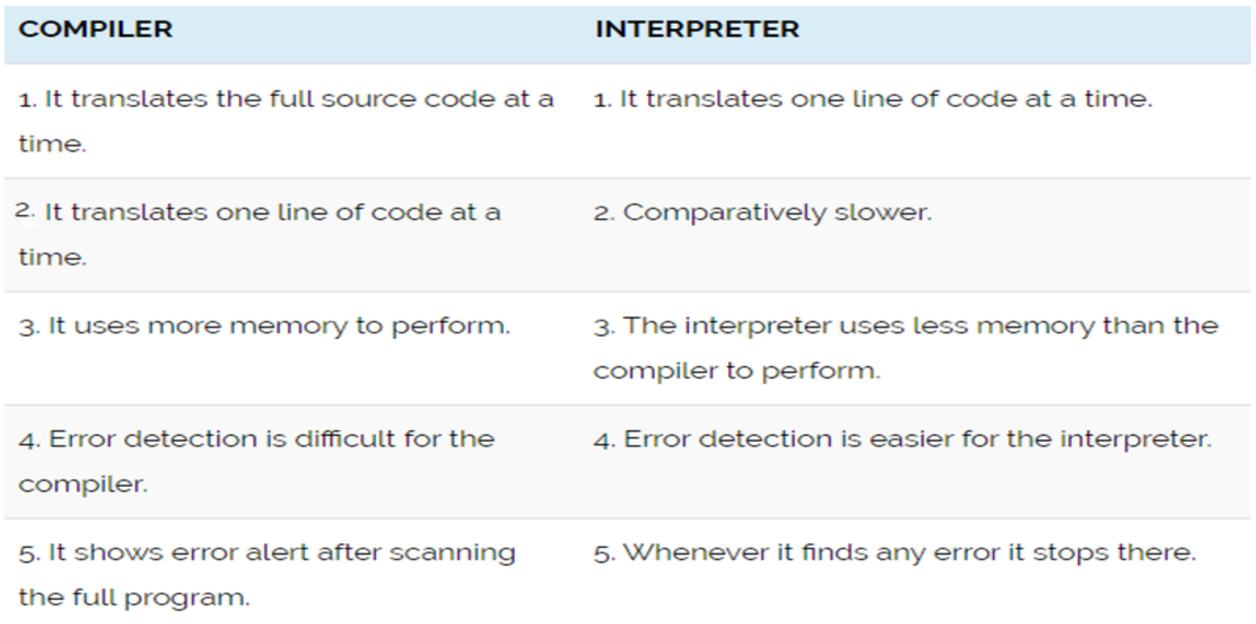
A compiler can have many phases and passes.

- **Pass** : A pass refers to the traversal of a compiler through the entire program.
- **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.



One pass	Multi pass
Passes through the source code of each compilation unit only once	Processes the source code of a program several times
Compilation time is faster	Compilation time is slower
Has limited scope of passes	Has wide scope of passes.
wide compilers	Narrow compilers
Pascal	Java

Compiler Vs Interpreter:-

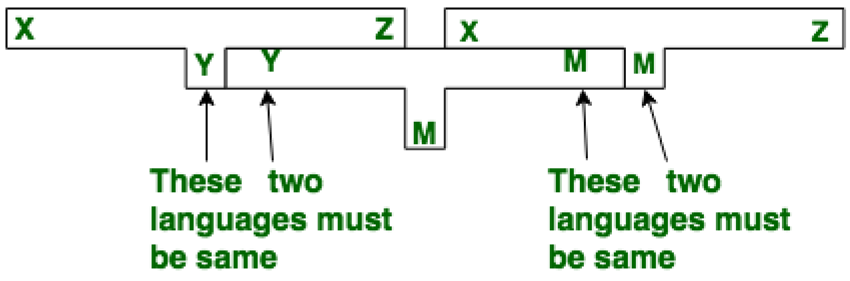


COMPILER	INTERPRETER
1. It translates the full source code at a time.	1. It translates one line of code at a time.
2. It translates one line of code at a time.	2. Comparatively slower.
3. It uses more memory to perform.	3. The interpreter uses less memory than the compiler to perform.
4. Error detection is difficult for the compiler.	4. Error detection is easier for the interpreter.
5. It shows error alert after scanning the full program.	5. Whenever it finds any error it stops there.

Bootstrapping and Cross Compiler:-

Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.

Suppose we want to write a cross compiler for new language X. The implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create XYZ. Now if existing compiler Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ. That means a compiler for source language X that generates a target code in language Z and which runs on machine M.


X
Z
Y
Y
M
X
M
M
Z
These two languages must be same
These two languages must be same

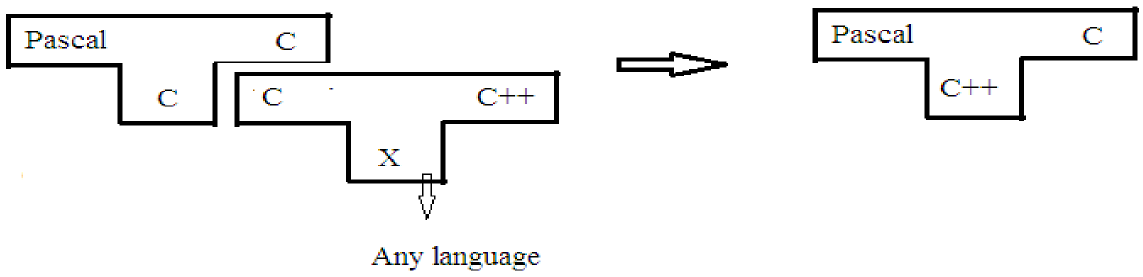
Cross Compiler:-

A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler.

Question:-

A pascal translator written in C language that takes pascal code and produces C as o/p. Create a pascal translator in C++ for the same

Sol:-



Finite Automata:-

Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata is a recognizer for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand.

The mathematical model of finite automata consists of:

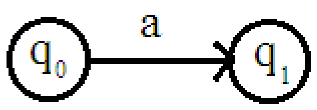
- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state (q0)
- Set of final states (qf)
- Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), $Q \times \Sigma \rightarrow Q$

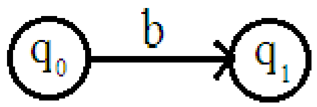
Thomson's Construction Rule:-

Construction of an NFA from a Regular Expression
INPUT: A regular expression r over alphabet C.
OUTPUT: An NFA N accepting L(r)
METHOD: Begin by parsing r into its constituent sub expressions. The rules for constructing an NFA consist of basis rules for handling sub expressions with no operators, and inductive rules for constructing larger NFA's from the NFA's for the immediate sub expressions of a given expression.
R.E.

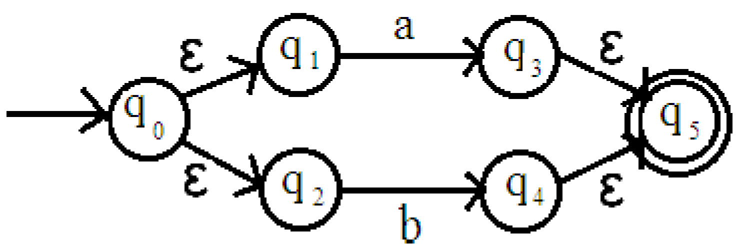
Case 1: **R=a**



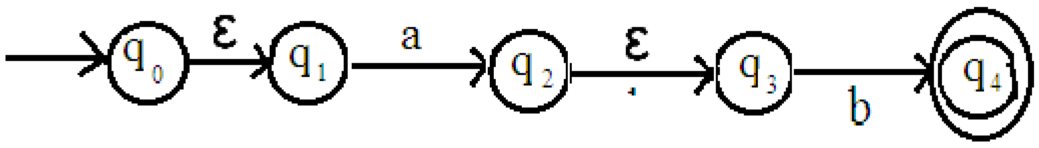
Case 2: **R=b**



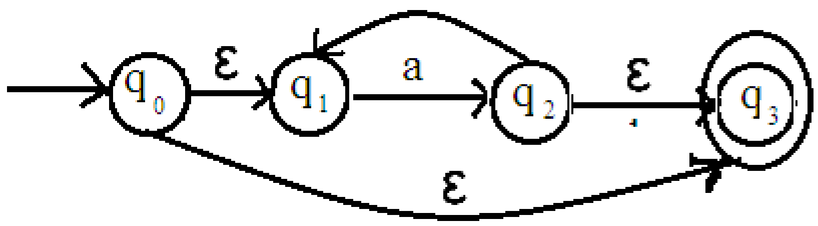
Case 3: **R=a+b**
R=a | b



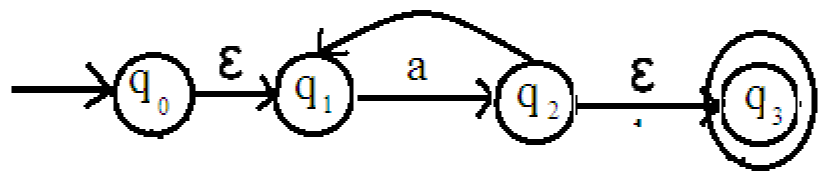
Case 4: **R=ab**



Case 5: **R=a***



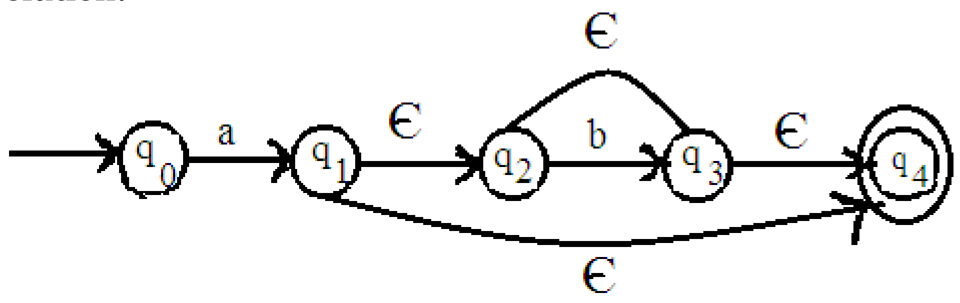
Case 6: **R=a+**

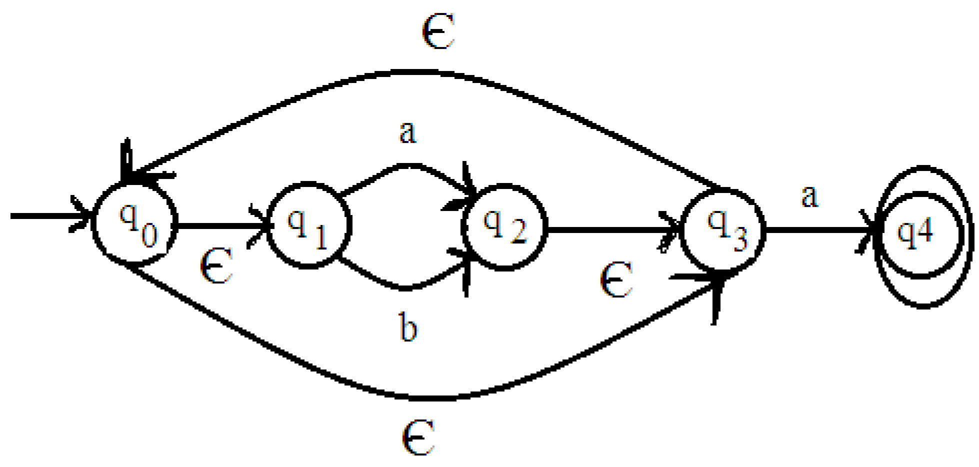


Ex. Draw the NFA for the regular expression

R.E. = a b*
R.E. = (a | b)* a
R.E. = (a | b)*
R.E. = (a b | b a)*

Solution: -





Similarly we will do remaining part.

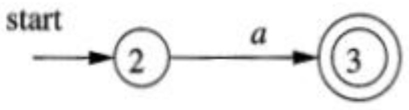
Subset Construction Algorithm:-

Step 1:- Put ϵ- closure (S_0) an unmarked state into the set of DFA.

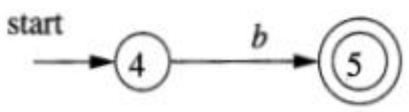
```
Step 2:-  while(there is one unmarked state s1 in DFA)
          begin
                mark s1
                for each input symbol a do
                begin
                      s2← ϵ-closure(move(s1,a))
                      if(s2 is not in DFA)
                      {
                            Then add s2 into DFA as an unmarked state
                      }
                      Dtrans[s1,a]←s2
                End
          End
```

Ex. Design a DFA for (a | b)* ab
Solution:-

NFA for a (i.e) subexpression $r_1 \rightarrow N$ (r_1)

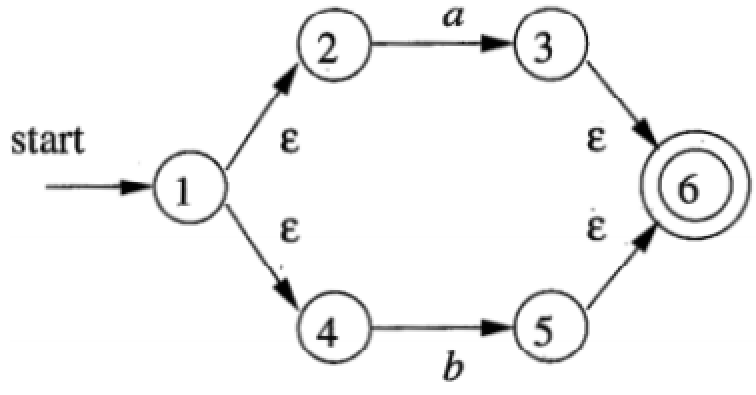


NFA for b, Subexpression $r_2 \rightarrow N$ (r_2)



NFA for a | b, (i.e) combine N (r_1) & N(r_2)

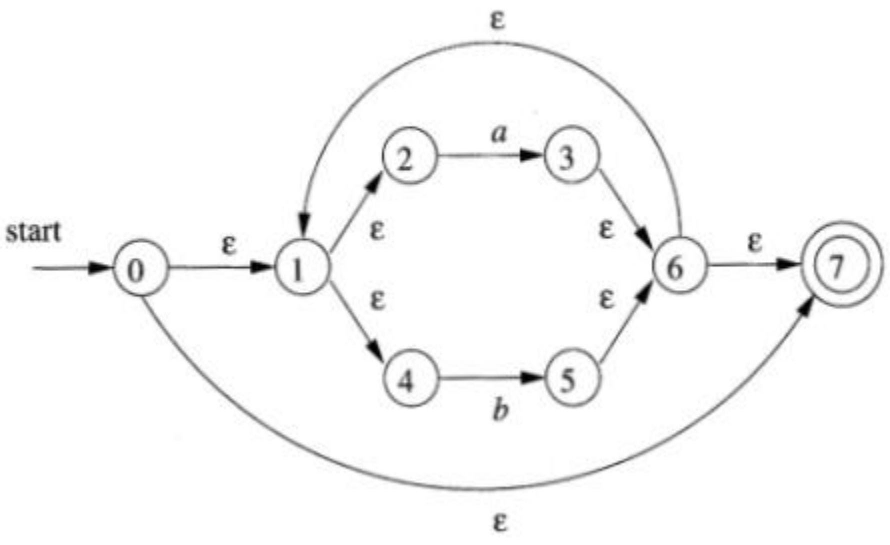
$r_3 = r_1 | r_2$



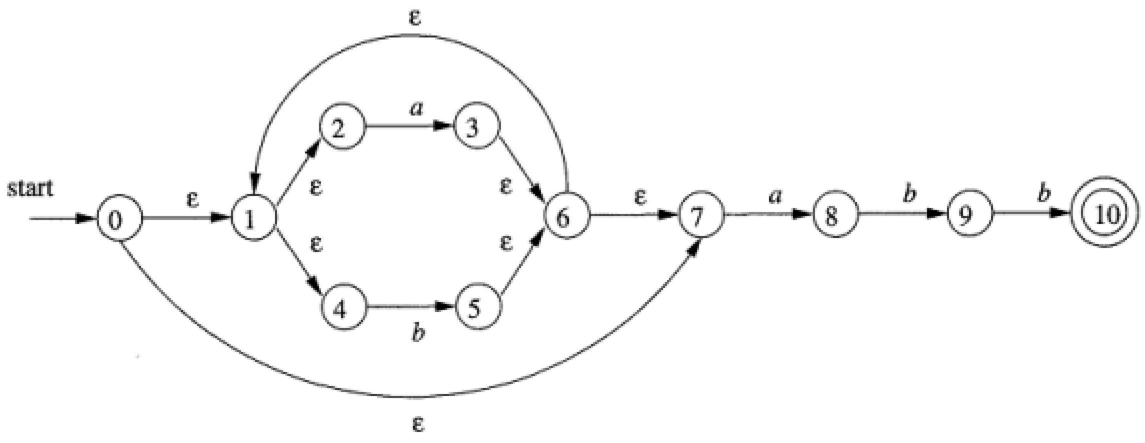
NFA for (a|b)*

i.e. for r4 = r3   r3=r1|r2

r5=(r3)*



NFA for (a|b)*abb:



ε – closure (0) = {0, 1, 2, 4, 7 } = A

Dtran[A, a] = ε – closure (move(A, a))
= ε – closure (3, 8)
= {1, 2, 3, 4, 6, 7, 8 }
= B

Dtran[A, b] = ε – closure (move(A, b))
= ε – closure (5)
= {1, 2, 4, 5, 6, 7 }
= C

Dtran[B, a] = ε – closure (move(B, a))
= ε – closure (3, 8)
= {1, 2, 3, 4, 6, 7, 8 }
= B

Dtran[B, b] = ε – closure (move(B, b))
= ε – closure (5, 9)
= {1, 2, 4, 5, 6, 7, 9 }
= D

Dtran[C, a] = ε – closure (move(C, a))
= ε – closure (3, 8)
= {1, 2, 3, 4, 6, 7, 8 }
= B

Dtran[C, b] = ε – closure (move(C, b))
= ε – closure (5)
= {1, 2, 4, 5, 6, 7 }
= C

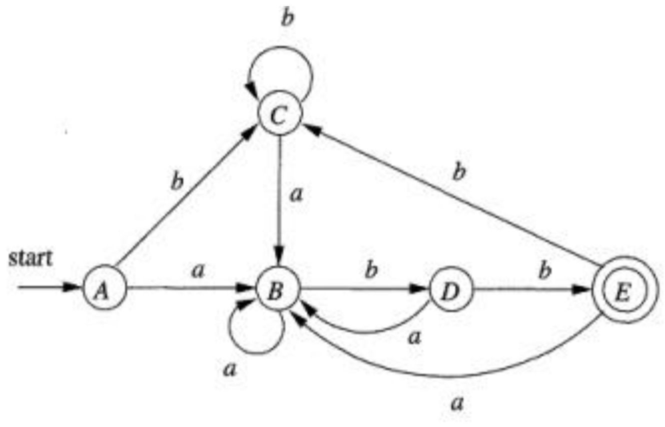
Dtran[D, a] = ε – closure (move(D, a))
= ε – closure (3, 8)
= {1, 2, 3, 4, 6, 7, 8 }
= B

Dtran[D, b] = ε – closure (move(D, b))
= ε – closure (5, 10)
= {1, 2, 4, 6, 7, 10 }
= E

Dtran[E, a] = ε – closure (move(E, a))
= ε – closure (3, 8)
= {1, 2, 3, 4, 6, 7, 8 }
= B

Dtran[E, b] = ε – closure (move(E, b))
= ε – closure (5)
= {1, 2, 4, 5, 6, 7 }
= C

NFA State	DFA State	a	b
{0, 1, 2, 4, 7 }	A	B	C
{1, 2, 3, 4, 6, 7, 8 }	B	B	D
{1, 2, 4, 5, 6, 7 }	C	B	C
{1, 2, 4, 5, 6, 7, 9 }	D	B	E
{1, 2, 4, 6, 7, 10 }	E	B	C

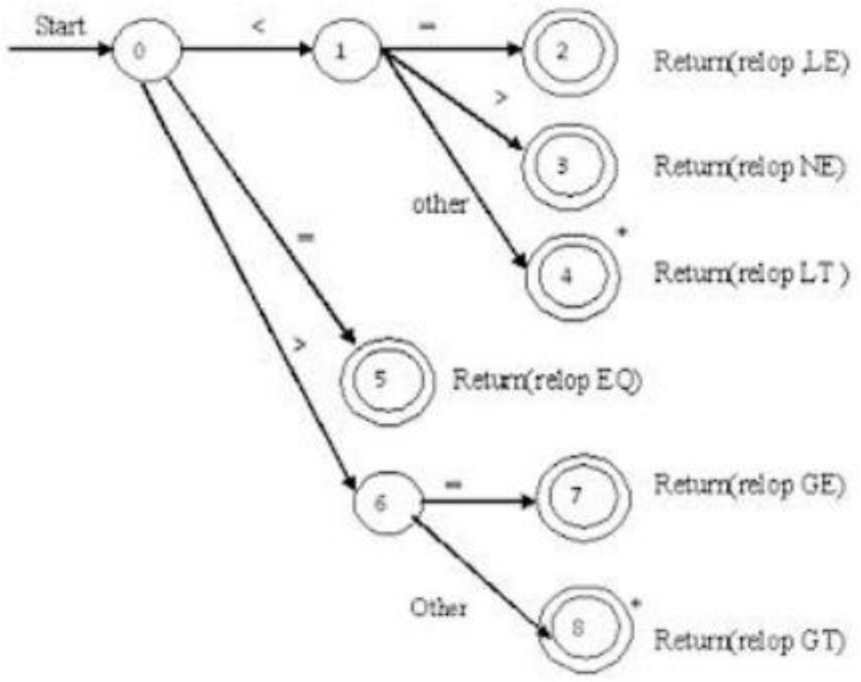


TRANSITION DIAGRAM:-

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. Edges are directed from one state of the transition diagram to another, each edge is labeled by a symbol or set of symbols. If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.
2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.
3. One state is designed the state ,or initial state ., it is indicated by an edge labeled “start” entering from nowhere .the transition diagram always begins in the state before any input symbols have been used. As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.



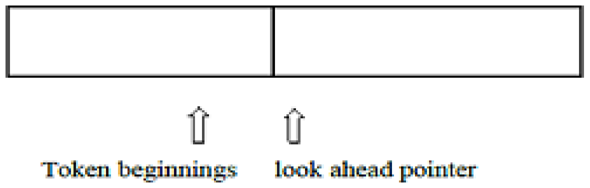
INPUT BUFFERING

The LA scans the characters of the source program one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Buffering techniques:

1. Buffer pairs
2. Sentinels

The lexical analyzer scans the characters of the source program one at a time to discover tokens. Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer. Figure shows a buffer divided into two half of, say 100 characters each. One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the token is discovered .we view the position of each pointer as being between the character last read and the character next to be read. In practice each buffering scheme adopts one convention either a pointer is at the symbol last read or the symbol it is ready to read.

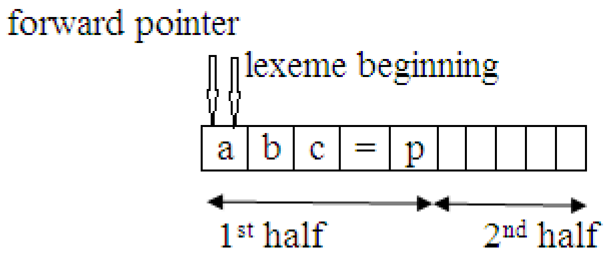


Token beginnings look ahead pointer The distance which the look ahead pointer may have to travel past the actual token may be large. For example, in a PL/I program we may see:

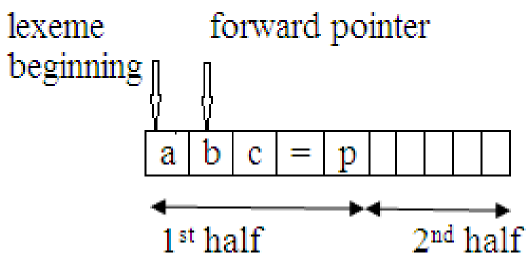
DECALRE (ARG1, ARG2… ARG *n*) without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file. Since the buffer shown in above figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered. In the above example, if the look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been grouped into tokens. While we can make the buffer larger if we chose or use another buffering scheme, we cannot ignore the fact that overhead is limited.

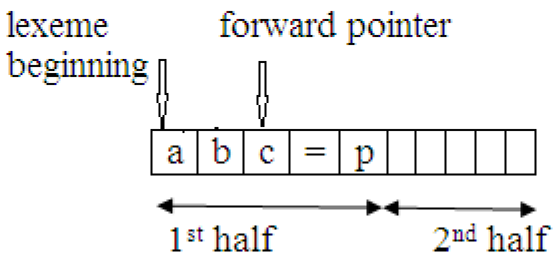
```
if forward at end of first half then begin
          reload second half;
          forward := forward +1
end
else if forward at end of second half then begin
          reload first half;
          move forward to beginning of first half
end
else forward := forward + 1;
```

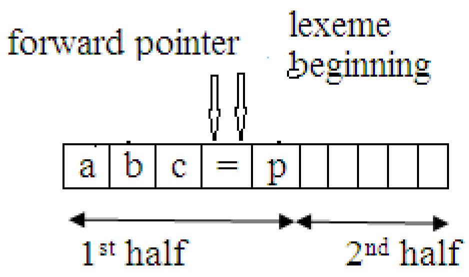
Ex:-
abc = pqr*xyz



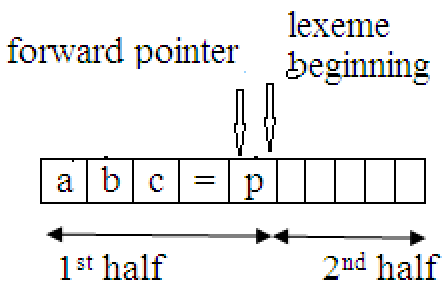
1. A buffer is divided into two half, each containing N-bit of info. One pointer mark the beginning of the token ,while another pointer look ahead, scan ahead of the beginning pointer until the token is discovered.



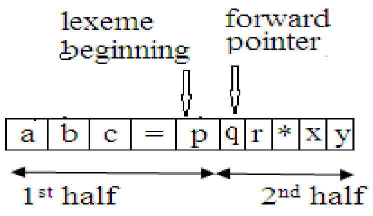


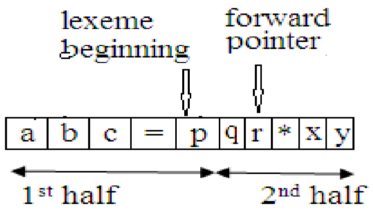


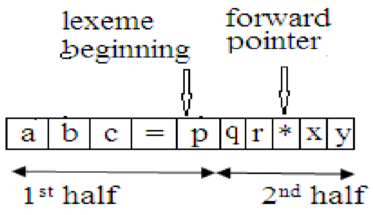
2. When forward pointer move to character = it identifies "**abc**" as an identifier. At the time lexeme beginning ptr directly jumps to character =



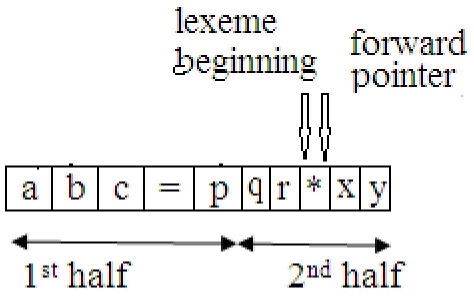
3. When forward pointer moves to index no=4 recognize = as an assignment operator and move both forward and beginning pointer to index no 4

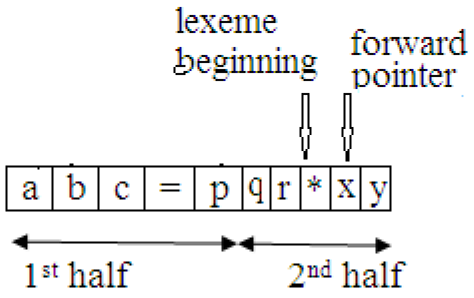




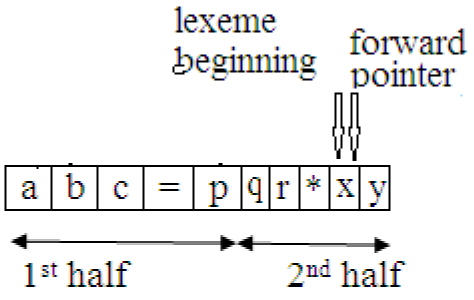


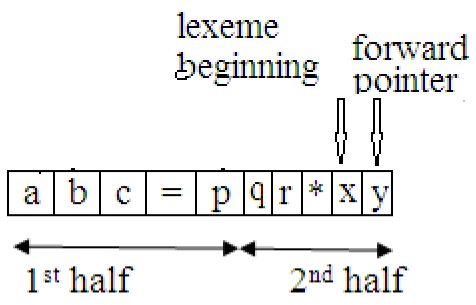
4. When forward pointer moves to index =7 recognize pqr as an identifier and moves both forward and beginning pointer at index =7

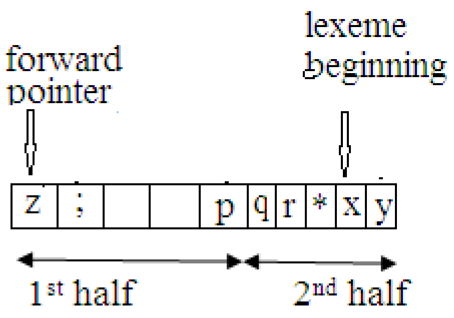


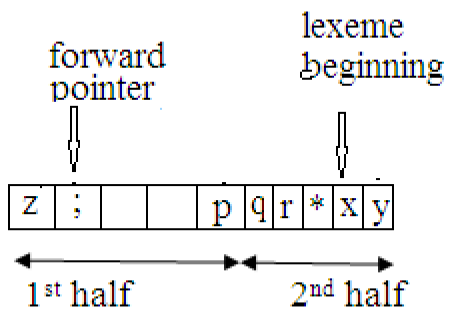


5. When forward pointer move to index 8 recognize * as an operator and set lexeme beginning pointer to index 8









6. When forward pointer move to index ; recognize xyz as an identifier and set lexeme beginning pointer to index ;

TOKEN, LEXEME, PATTERN:-

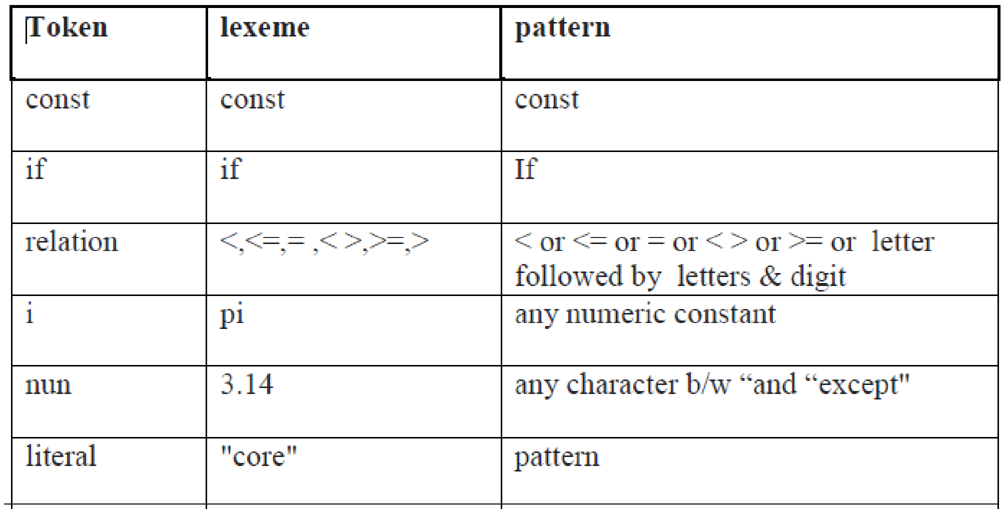
Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

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

Formal Grammar:

- Formal grammar is a set of rules. It is used to identify correct or incorrect strings of tokens in a language. The formal grammar is represented as G.
- Formal grammar is used to generate all possible strings over the alphabet that is syntactically correct in the language.
- Formal grammar is used mostly in the syntactic analysis phase (parsing) particularly during the compilation.

Formal grammar G is written as follows:

G = <V, N, P, S>

Where:

N describes a finite set of non-terminal symbols.
V describes a finite set of terminal symbols.
P describes a set of production rules
S is the start symbol.

Example:

$S \rightarrow a\ A\ |b$
$A \rightarrow a\ A\ |\ a$

Backus Normal Form

- Backus Normal Form is a notation techniques for the context free grammar.
- It is often used to describe the syntax of languages used in computing.
- Such as computer programming languages, document formats and so on. .
- They are applied wherever exact descriptions of languages are needed.

Example:-Write a BNF grammar for the language of University of Lucknow course codes.

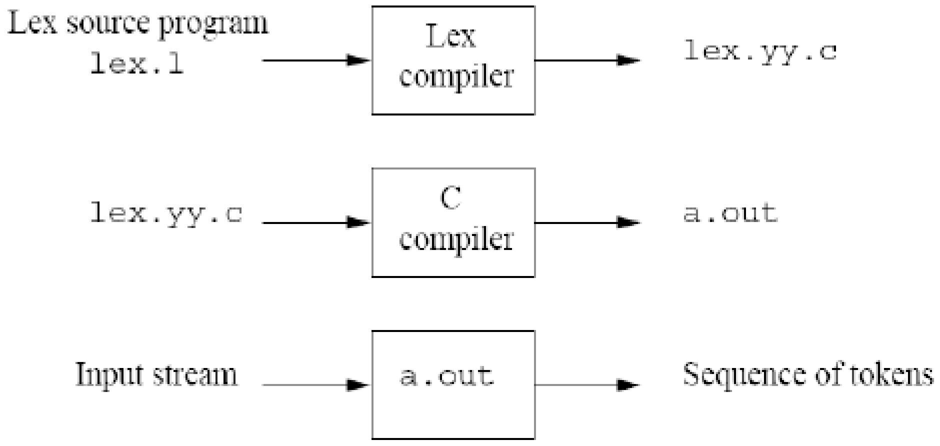
- **Example sentences:**
- **CSI3125**
- **MAT2743**
- **PHY1200**
- **CHE6581**
- **CSI9999**

Solution:

- <coursecode> ::= <acadunit> <coursenumber>
- <acadunit> ::= <letter> <letter> <letter>
- <coursenumber> ::= <year> <semesters> <digit> <digit>
- <year> ::= <ugrad> | <grad>
- <ugrad> ::= 0 | 1 | 2 | 3 | 4
- <grad> ::= 5 | 6 | 7 | 9
- <semesters> ::= <onesemester> | <twosemesters>
- <onesemester> ::= <frenchone> | <englishone> | <bilingual>
- <frenchone> ::= 5 | 7
- <englishone> ::= 1 | 3
- <bilingual> ::= 9
- <twosemesters> ::= <frenchtwo> | <englishtwo>
- <frenchtwo> ::= 6 | 8
- <englishtwo> ::= 2 | 4
- <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Lex:-

The Lex compiler is a tool that allows one to specify a lexical analyzer from regular expressions.



I

Structure of Lex Program:-

```
%{
                Declarations/Definition Section
%}

%%
Rule Section
%%
User Defined Section
```

Definition Section:-

The definition section creates an environment for the lexer, which is a C-code. This area of lex specification is separated by "%{" and "%}" it contains C statements such as global deceleration , commands, library files and other deceleration which will be copied to the lexical analyzer when it passed through the lex tool.

```
%{
        #include<stdio.h>
%}
```

Rule Section:-

It contains the pattern and actions that specify the lex specification. A pattern is in the form of a regular expression to match the string.

Once the pattern is matched the corresponding action part is invoked. The action part contains normal C language statements.

They are enclosed within braces **“{” and “}”**, if there is more than one statement then make these component statement into single block of statements.

```
%%
        [a]     {printf(“Alphabet a”);}
%%
```

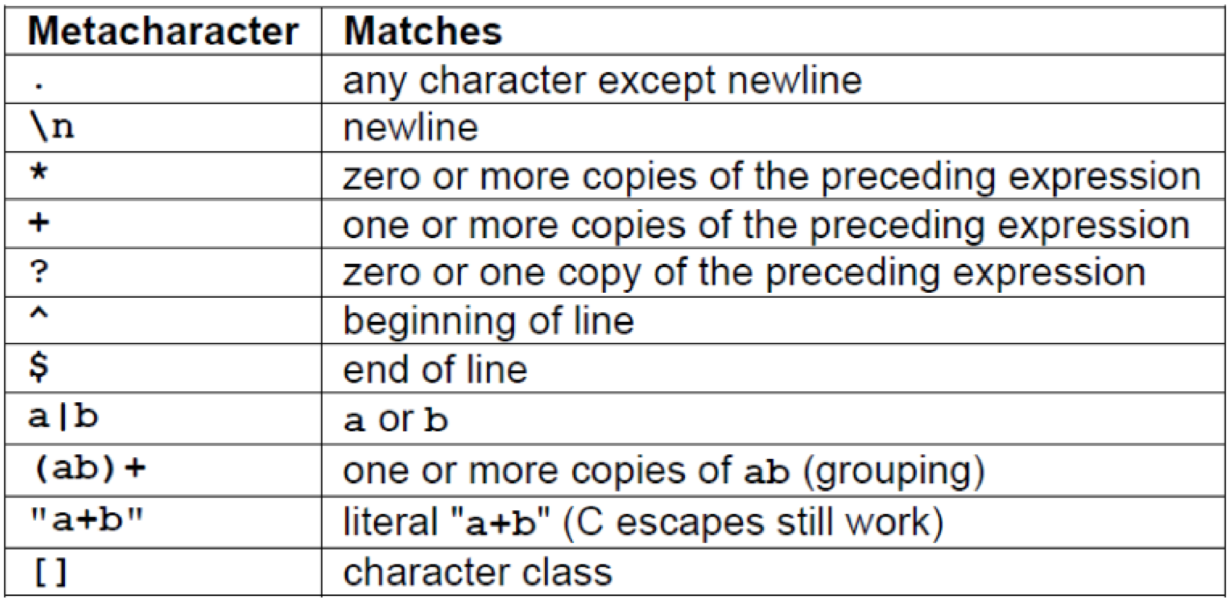
User Defined Section:-

This section contains any valid C-code. Lex copies the contents of this section into generated lexical analyzer.

Lex itself does not produce an executable program, instead , it translate the lex specification into a file containing a C subroutine called yylex.

All the rules in the rule section will automatically be converted into C statement by the lex tool and will be put under the function name yylex().

Whenever we call the function yylex() in main function even though we have not defined it anywhere in the program.



Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Ambiguity:-

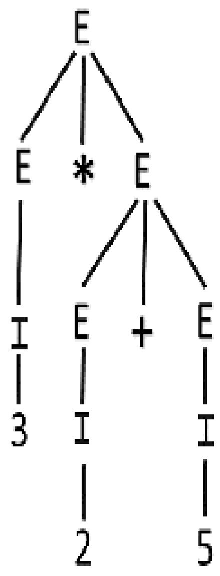
- A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be **ambiguous**.
- To show that a grammar is ambiguous all we need to do is find a terminal string that is the yield of more than one parse tree.
- Since a string with more than one parse tree usually has more than one meaning, we need to design unambiguous grammars for compiling applications, or to use ambiguous grammars with additional rules to resolve the ambiguities.

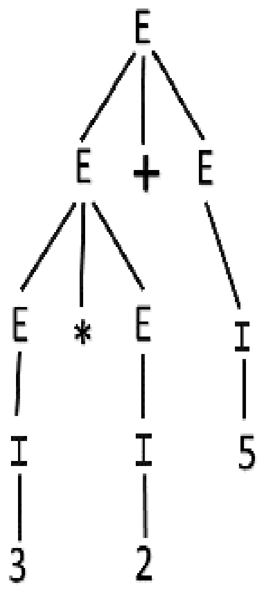
Example: Let us consider a grammar G with the production rule

$E \rightarrow I$
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$I \rightarrow \varepsilon \mid 0 \mid 1 \mid 2 \mid \ldots \mid 9$

Solution:

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:





Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is ambiguous.