

# Compiler Design

## UNIT - 1

### **Short Question:-**

1. Differentiate between dynamic loader and linker?
2. What is assembler?
3. Draw the transition diagram for an identifier?
4. Draw the transition diagram for relational operators?
5. Describe the language denoted by the following regular Expression  $(1+0)^*$ .
6. What is a cross-compiler?
7. Discuss the utility of macros?
8. What do you mean by a regular expression?
9. Differentiate between compiler and interpreter?
10. Discuss the merits and demerits of the single pass compiler and multipass compiler?
11. Describe various compiler writing tools?
12. Explain the term bootstrapping with example?
13. What is the role of a lexical analyzer? Enumerate the issues handled by lexical analyzer?
14. Differentiate between linker and loader

### **Medium Question:-**

1. Discuss input buffering and preliminary scanning in lexical analysis?
2. How is a Finite automaton useful for Lexical Analysis?
3. Why do we divide the compilation into phases?
4. Discuss the challenges in compiler design?
5. How bootstrapping is done in more than one machine?
6. Discuss the subset construction algorithms?
7. Explain the term token, lexeme, pattern?
8. Write the algorithm for moving forward pointer in "input buffering" Scheme ?
9. Construct a minimal DFA which accept set of all strings over  $\{a,b\}$  in which every „a“ should be followed by „bb“
10. How is finite Automation useful for Lexical Analysis?

### **Long Question:-**

1. Construct minimal DFA for the following regular expression  $(a|b)^*a(a|b)$
2. Construct a minimal DFA which accepts set of all strings over  $\{a,b\}$  in which no. of a "s are divisible by 3 and no. of b"s is divisible by 3
3. Show the construction of NFA for the following Regular Expression  $(a|b)^*a(a|b)(a|b)$
4. Construct NFA for the following RE using Thomson"s construction:  $(0|1)^*0(0|1)^*$
5. Explain the phases of the compiler in detail. Write down the output of each phase for the expression  $a=b+c*50$ .

=====

### Short Answers:-

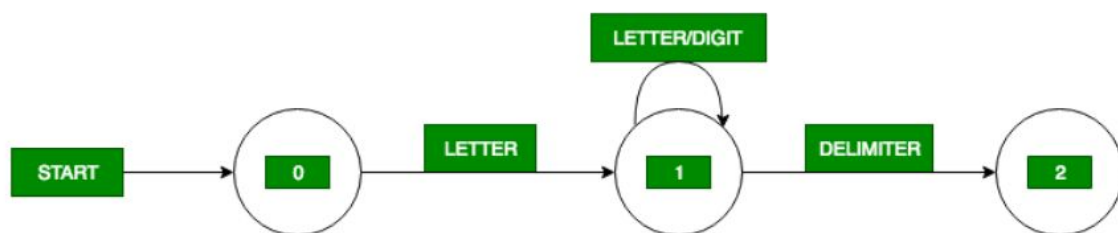
1. **Dynamic loading** is concerned with loading the subroutines of a program as and when required during run-time, instead of loading all the subroutines at once before the program execution starts. This is useful in efficient memory usage since many subroutines may not be called at all.

So basically what we do here is we load the main module first and then during execution we load some other module only when it's required.

Whereas, **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 modules/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

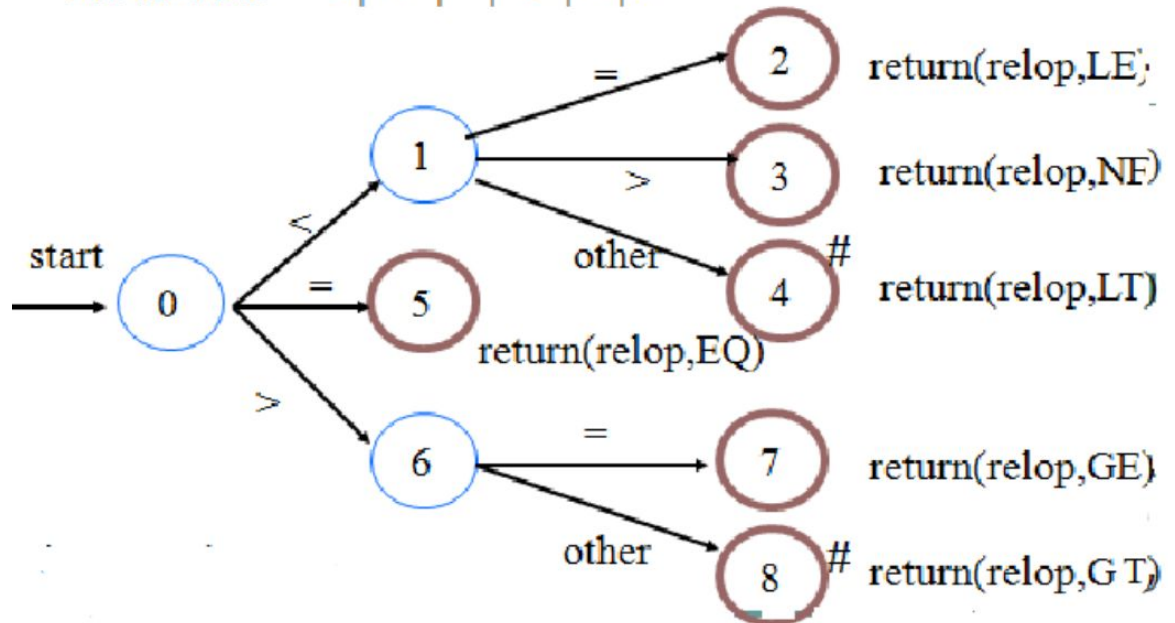
2. **Assembler:-** An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

3. Transition Diagram of identifier :-



4. Transition Diagram of relational operator :-

Ex :RELOP = < | <= | = | <> | > | >=



5. Regular expression  $(1+0)^*$  :-

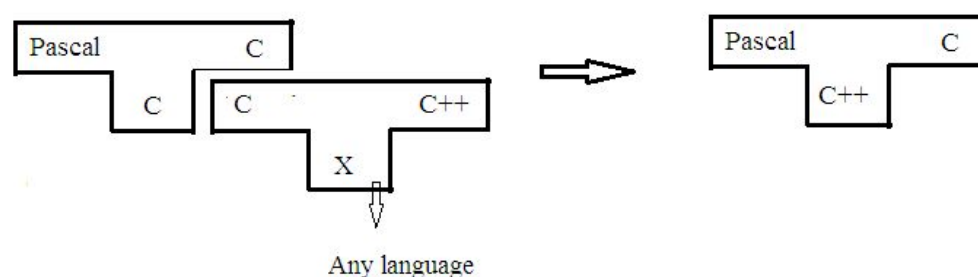
$(a+b)^*$

Set of strings of a's and b's of any length including the null string. So  $L = \{ \epsilon, a, b, aa, ab, bb, ba, aaa, \dots \}$

6. **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 an Android smartphone is a cross compiler.

**E.g.** 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



7. A **macro** (which stands for "**macroinstruction**") is a programmable pattern which translates a certain sequence of input into a preset sequence of output. Macros can make tasks less repetitive by representing a complicated sequence of keystrokes, mouse movements, commands, or other types of input.

In computer programming, macros are a tool that allows a developer to reuse code. For instance, in the [C programming language](#), this is an example of a simple macro definition which incorporates [arguments](#):

```
#define square(x) ((x) * (x))
```

After being defined like this, our macro is used in the code body to find the square of a number.

8. A **Regular Expression** can be defined as follows –

## REGULAR EXPRESSIONS

Definition :- The formal definition of regular expression over  $\Sigma$  is as follows:-

- i) Any terminal symbol (i.e. an element of  $\Sigma$ ),  $\wedge$  and  $\phi$  are regular expressions. when.
- ii) The union of two regular expressions  $R_1$  and  $R_2$ , written as  $R_1 + R_2$  is also a regular expression.
- iii) The concatenation of two regular expressions  $R_1$  and  $R_2$ , written as  $R_1 R_2$ , is also a regular expression.
- iv) The iteration (or closure) of a regular expression  $R$ , written as  $R^*$ , is also a regular expression.
- v) If  $R$  is a regular expression, the  $(R)$  is also a regular expression.
- \* This rule will influence the order of evaluation of a RE.
- \* In the absence of parentheses, we have hierarchy of operations as follows:- iteration (closure), concatenation, and union.
- vi) The RE over  $\Sigma$  are precisely those obtained recursively by the application of the rules 1-5 once or several times.

E.g.  $(1+0)^*$  etc.

9. Compiler v/s 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.

#### 10. Single Pass Compiler :-

- Merits :-
  - Single pass compiler is faster and smaller than the multi pass compiler.
  - Single pass compiler is one that processes the input *exactly once*, so going directly from lexical analysis to code generator, and then going back for the next read.
- Demerits :-
  - Single pass compiler is less efficient in comparison with a multipass compiler.
  - We can not optimize very well due to the limited context of expressions.
  - As we can't backup and process it again so grammar should be limited or simplified.

#### Multi Pass Compiler :-

- Merits :-
  - Consumes less memory than a single pass compiler.
  - Multipass compiler is more efficient in comparison with a single pass compiler.
- Demerits :-
  - The Multi pass compiler is slower than the Single pass compiler.
  - Takes more time as compared to single pass compiler

#### 11. Some commonly used compiler construction tools include:(Surya k notes vala)

- **Parser Generator –**  
It produces syntax analyzers (parsers) from the input that is based on a grammatical description of a programming language or on a context-free grammar. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.  
Example: PIC, EQM
- **Scanner Generator –**

It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language. It generates a finite automaton to recognize the regular expression.

Example: Lex

- **Syntax directed translation engines –**

It generates intermediate code with three address formats from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produce the intermediate code. In this, each node of the parse tree is associated with one or more translations.

- **Automatic code generators –**

It generates the machine language for a target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator. A template matching process is used. An intermediate language statement is replaced by its equivalent machine language statement using templates.

- **Data-flow analysis engines –**

It is used in code optimization. Data flow analysis is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another. Refer – [data flow analysis in Compiler](#)

- **Compiler construction toolkits –**

It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler

**Short form of above answer:-**

**Parser Generator –**

(It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar.)

**Scanner Generator –**

(It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language.)

**Syntax directed translation engines –**

(It generates intermediate code with three address formats from the input that consists of a parse tree.)

**Automatic code generators –**

(It generates the machine language for a target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator.)

**Data-flow analysis engines –**

(It is used in code optimization. Data flow analysis is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another. )

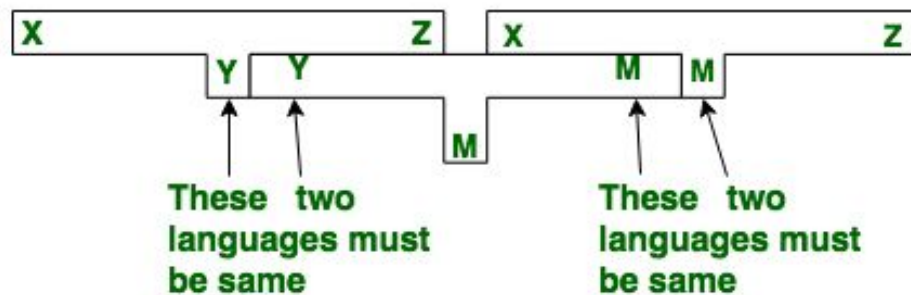
**Compiler construction toolkits –**

(It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.)

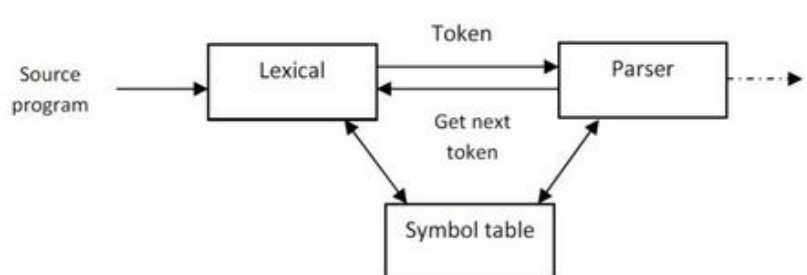


12. **Bootstrapping** is a process in which simple language is used to translate a more complicated program which in turn may handle a more complicated program. This complicated program can further handle even more complicated programs and so on.

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.



13. Role of lexical analyzer:- As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as **output a sequence of tokens** for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis.



The issues handled by lexical analyzer:-

- It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- **Stripping out comments and whitespace** (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- Another task is **correlating error messages** generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
- In some compilers, If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

#### 14. Linker vs Loader :-

BASIS FOR COMPARISON	LINKER	LOADER
Basic	It generates the executable module of a source program.	It loads the executable module to the main memory.
Input	It takes as input, the object code generated by an assembler.	It takes executable module generated by a linker.
Function	It combines all the object modules of a source code to generate an executable module.	It allocates the addresses to an executable module in main memory for execution.
Type/Approach	Linkage Editor, Dynamic linker.	Absolute loading, Relocatable loading and Dynamic Run-time loading.

=====

#### Medium Answers:-

1. The LA scans the characters of the source program one at a time to discover tokens. Because large amounts 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:

- Buffer pairs
- Sentinels

For preliminary scanning of lexical analyzer :- [see from slide 39](#)

(OR) 🙋

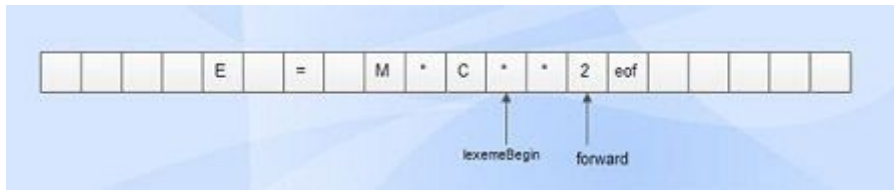
Preliminary scanning:- (jo bheje ha usi ko explain kar dena apni language me both buffer pair and sentinels)

#### Buffer Pairs

Because of the large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Fig shows the buffer pairs which are used to hold the input data.





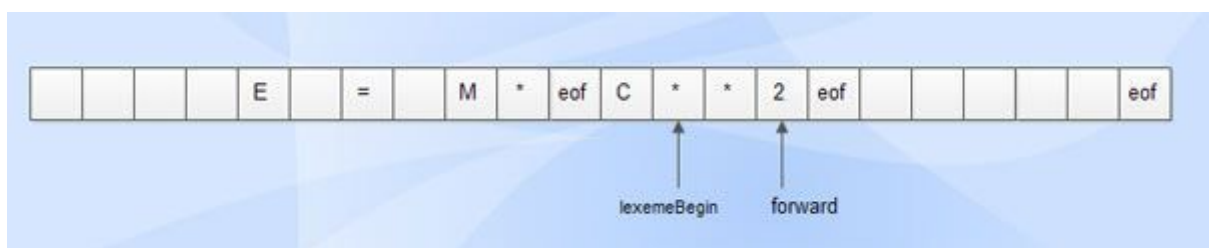
## Sentinels

- In the previous scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.

**Test 1:** For end of buffer.

**Test 2:** To determine what character is read.

- The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program.(eof character is used as sentinel).



## 2. Usefulness of Finite Automata for Compiler :-

- Lex translates regular expressions into transition diagrams and then transition diagrams into c code to recognize tokens in the input stream. Here, Finite Automata is used in this process as :-

RE -> NFA -> DFA -> C code

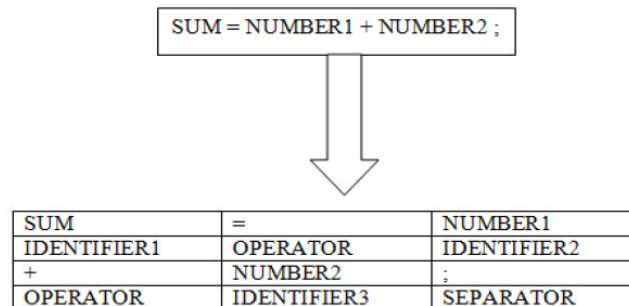
- A RECOGNIZE takes language L and string x as input, and responds YES if x is an element of L, or NO otherwise. Here, Finite Automata is used as RECOGNIZER.
- A FA is DETERMINISTIC if there is only one possible transition of some <state, input> pair.
- A FA is NON-DETERMINISTIC if there is more than one possible transition of some <state, input> pair.
- But both DFAs and NFAs recognize the same class of languages.

### Example :-

A lexeme is the unit derived from the source program with each group is related to anyone symbolic category.

The given source code "SUM=NUMBER1+NUMBER2; " is having six lexemes. The lexical analyzer reads the input character by character until finding the lexeme. First character 'S' is obtained. Then the character 'U' followed by 'M' and '='. The last character '=' omitted. The first three characters combined to form the lexeme "SUM".

Then the character '=' and 'N' are scanned by the lexical analyzer. The last character 'N' omitted to form the lexeme "=". Then the remaining lexemes "NUMBER1", "+", "NUMBER2", ";" identified by the lexical analyzer. The pattern matching technique is used to match the lexeme with the token type.



3. Compilers are large and complex programs so we divide compilers into different phases on the basis of their complexity. There are four compulsory phases of compiler; Lexical Analysis, Syntax Analysis, Semantic Analysis and Target Code Generation. In the first phase of the compiler we define lexical rules by regular expression. Lexical Analysis is also called Scanning. In this phase source program is checked to have valid characters and words. After detailed analysis, input streams of tokens are stored in a buffer. The second phase of the compiler is Syntax Analysis. In this phase we define syntax rules by Context Free Grammar. Syntax Analysis is also called parser. During parsing, syntax rules are checked. Input of this phase is tokens and output is parse tree or syntax tree. Third phase of the compiler is Semantic Analysis. Basically the word semantic means the "meaning".

In this phase we define semantic rules by attribute grammars. Semantic Analysis has two types; Declaration Checking and Type Checking. The input of this phase is syntax tree and output is attribute grammar. The last and final phase is Target Code Generation. After analysis of source code is completed, then the last step is converting it into target language. The other optional phases are Source Code Optimizer, Target Code Optimizer and Intermediate Code Generator.

#### 4. Following are the CHALLENGES while designing COMPILERS

*many variations:-*

- many programming languages (eg, FORTRAN, C++, Java)
- many programming paradigms (eg, object-oriented, functional, logic)
- many computer architectures (eg, MIPS, SPARC, Intel, alpha) many operating systems (eg, Linux, Solaris, Windows)

*Qualities of a compiler (in order of importance):-*

- the compiler itself must be bug-free it must generate correct machine code
- the generated machine code must run fast
- the compiler itself must run fast (compilation time must be proportional to program size) the compiler must be portable (ie, modular, supporting separate compilation) it must print good diagnostics and error messages

- the generated code must work well with existing debugger must have consistent and predictable optimization.

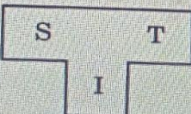
Building a compiler requires knowledge of:-

- programming languages (parameter passing, variable scoping, memory allocation, etc)
- theory (automata, context-free languages, etc)
- algorithms and data structures (hash tables, graph algorithms, dynamic programming, etc)
- computer architecture (assembly programming) software engineering.

5. Bootstrapping in more than one machine:-

**Bootstrapping :**

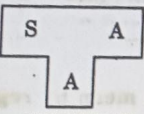
1. Bootstrapping is the process of writing a compiler (or assembler) in the source programming language that it intends to compile.
2. Bootstrapping leads to a self-hosting compiler.
3. An initial minimal core version of the compiler is generated in a different language.
4. A compiler is characterized by three languages :
  - a. Source language ( $S$ )
  - b. Target language ( $T$ )
  - c. Implementation language ( $I$ )
5.  ${}^S C_I^T$  represents a compiler for Source  $S$ , Target  $T$ , implemented in  $I$ . The  $T$ -diagram shown in Fig. 1.4.1 is also used to depict the same compiler :



The diagram is a T-shaped box. The top horizontal bar contains 'S' on the left and 'T' on the right. The vertical stem of the T contains 'I'.

**Fig. 1.4.1.**

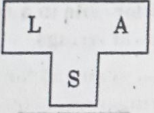
- a. Create  ${}^S C_A^A$  a compiler for a subset,  $S$ , of the desired language,  $L$ , using language  $A$ , which runs on machine  $A$ . (Language  $A$  may be assembly language.)



The diagram is a T-shaped box. The top horizontal bar contains 'S' on the left and 'A' on the right. The vertical stem of the T contains 'A'.

**Fig. 1.4.2.**

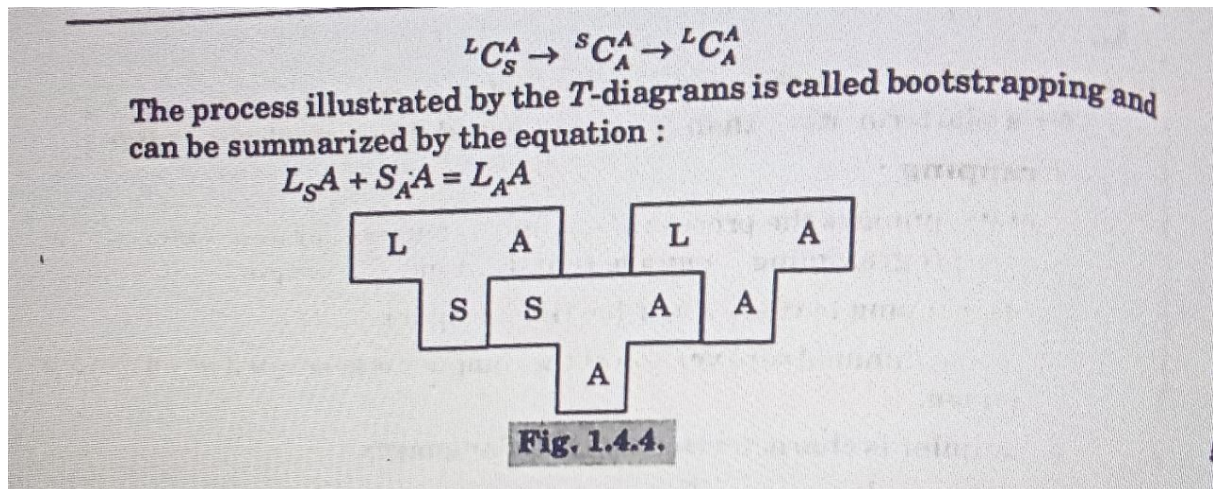
- b. Create  ${}^L C_S^A$ , a compiler for language  $L$  written in a subset of  $L$ .



The diagram is a T-shaped box. The top horizontal bar contains 'L' on the left and 'A' on the right. The vertical stem of the T contains 'S'.

**Fig. 1.4.3.**

- c. Compile  ${}^L C_S^A$  using  ${}^S C_A^A$  to obtain  ${}^L C_A^A$ , a compiler for language  $L$ , which runs on machine  $A$  and produces code for machine  $A$ .



## 6. Subset Construction Algorithm:-

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

Step 2:- while(there is one unmarked state  $s_1$  in DFA)

begin

mark  $s_1$

for each input symbol  $a$  do

begin

$s_2 \leftarrow \epsilon\text{-closure}(\text{move}(s_1, a))$

if( $s_2$  is not in DFA)

{

Then add  $s_2$  into DFA as an unmarked state

}

$D\text{trans}[s_1, a] \leftarrow s_2$

End

End

**Ex. Design a DFA for  $(a \mid b)^* abb$**

**Solution:-**



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

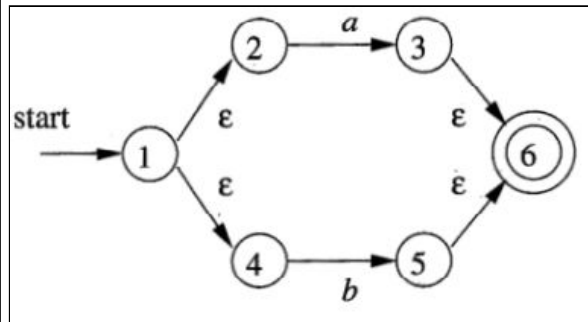


NFA for b, Subexpression  $r_2 \rightarrow N(r_2)$



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

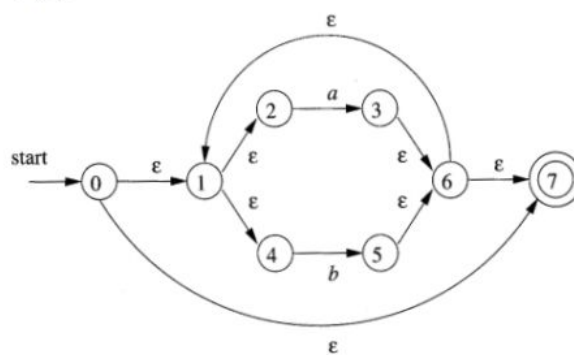
$$r_3 = r_1 | r_2$$



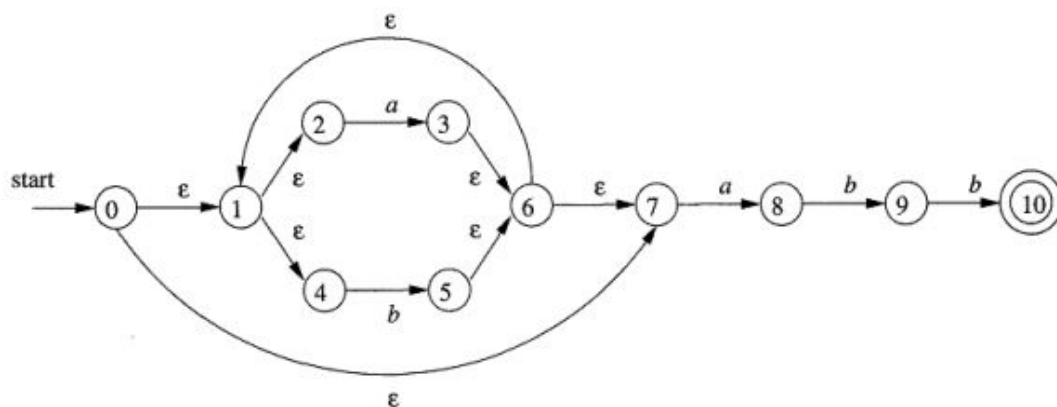
**NFA for  $(a|b)^*$**

i.e. for  $r_4 = r_3$        $r_3 = r_1 | r_2$

$$r_5 = (r_3)^*$$



NFA for  $(a|b)^*abb$ :

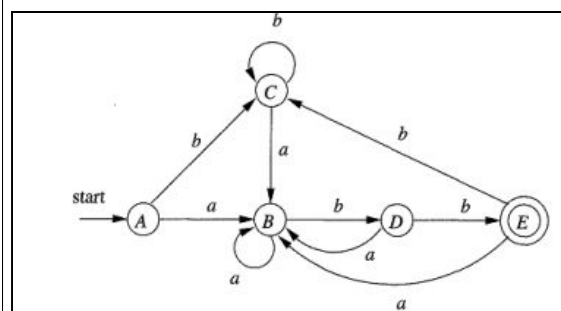


$\epsilon$  - closure (0) = {0, 1, 2, 4, 7} = A  
  
 $Dtran[A, a] = \epsilon$  - closure (move(A, a))  
 $= \epsilon$  - closure (3, 8)  
 $= \{1, 2, 3, 4, 6, 7, 8\}$   
 $= B$   
  
 $Dtran[A, b] = \epsilon$  - closure (move(A, b))  
 $= \epsilon$  - closure (5)  
 $= \{1, 2, 4, 5, 6, 7\}$   
 $= C$   
  
 $Dtran[B, a] = \epsilon$  - closure (move(B, a))  
 $= \epsilon$  - closure (3, 8)  
 $= \{1, 2, 3, 4, 6, 7, 8\}$   
 $= B$

$Dtran[B, b] = \epsilon$  - closure (move(B, b))  
 $= \epsilon$  - closure (5, 9)  
 $= \{1, 2, 4, 5, 6, 7, 9\}$   
 $= D$   
  
 $Dtran[C, a] = \epsilon$  - closure (move(C, a))  
 $= \epsilon$  - closure (3, 8)  
 $= \{1, 2, 3, 4, 6, 7, 8\}$   
 $= B$   
  
 $Dtran[C, b] = \epsilon$  - closure (move(C, b))  
 $= \epsilon$  - closure (5)  
 $= \{1, 2, 4, 5, 6, 7\}$   
 $= C$   
  
 $Dtran[D, a] = \epsilon$  - closure (move(D, a))  
 $= \epsilon$  - closure (3, 8)  
 $= \{1, 2, 3, 4, 6, 7, 8\}$   
 $= B$   
  
 $Dtran[D, b] = \epsilon$  - closure (move(D, b))  
 $= \epsilon$  - closure (5, 10)  
 $= \{1, 2, 4, 6, 7, 10\}$   
 $= E$   
  
 $Dtran[E, a] = \epsilon$  - closure (move(E, a))  
 $= \epsilon$  - closure (3, 8)  
 $= \{1, 2, 3, 4, 6, 7, 8\}$   
 $= B$

$Dtran[E, b] = \epsilon$  - closure (move(E, b))  
 $= \epsilon$  - 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



**7. 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:**



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

8. The algorithm for moving forward pointer in "input buffering" scheme :-

```

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;

```

E.g. [see from slide 39](#) ( vahi jo bheje ha buffer pair ka usi ko samjha do apni language me )

9. **Nahi aa raha** 😞

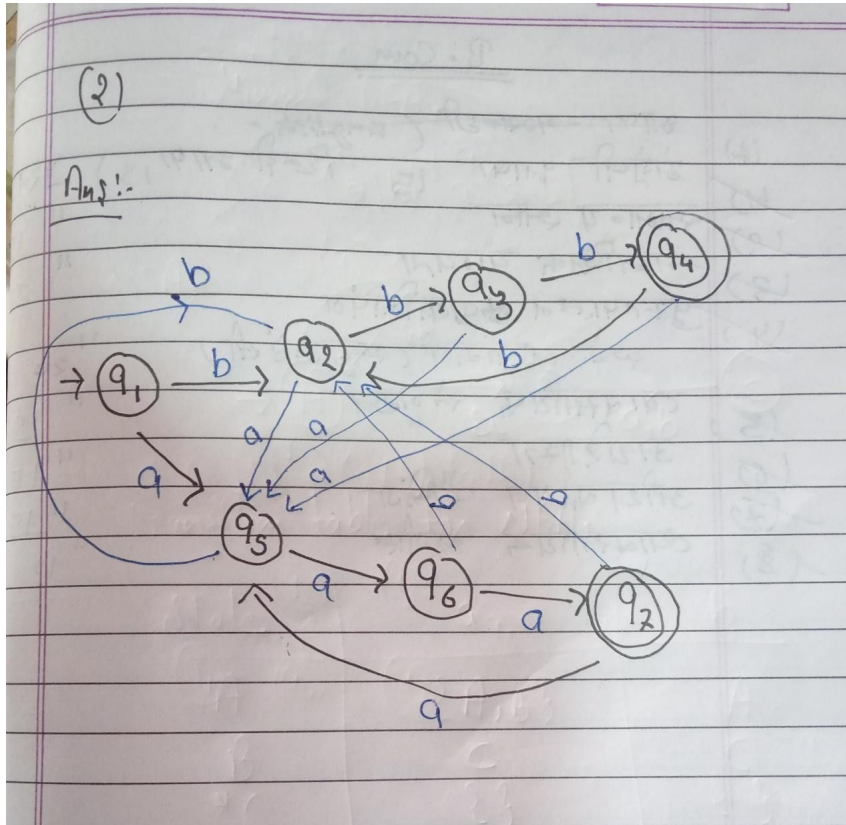
=====

### Long Answers:-

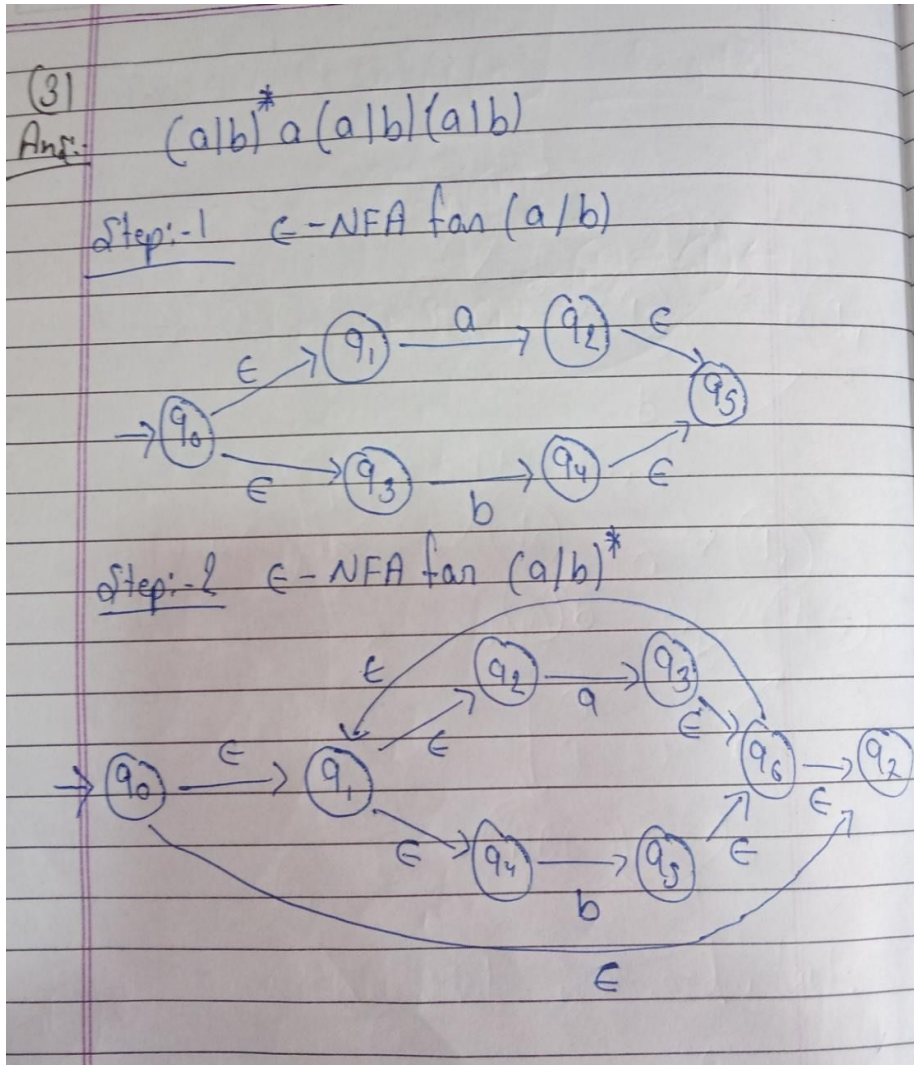
1. DFA for regular expression  $(a|b)^*a(a|b)$

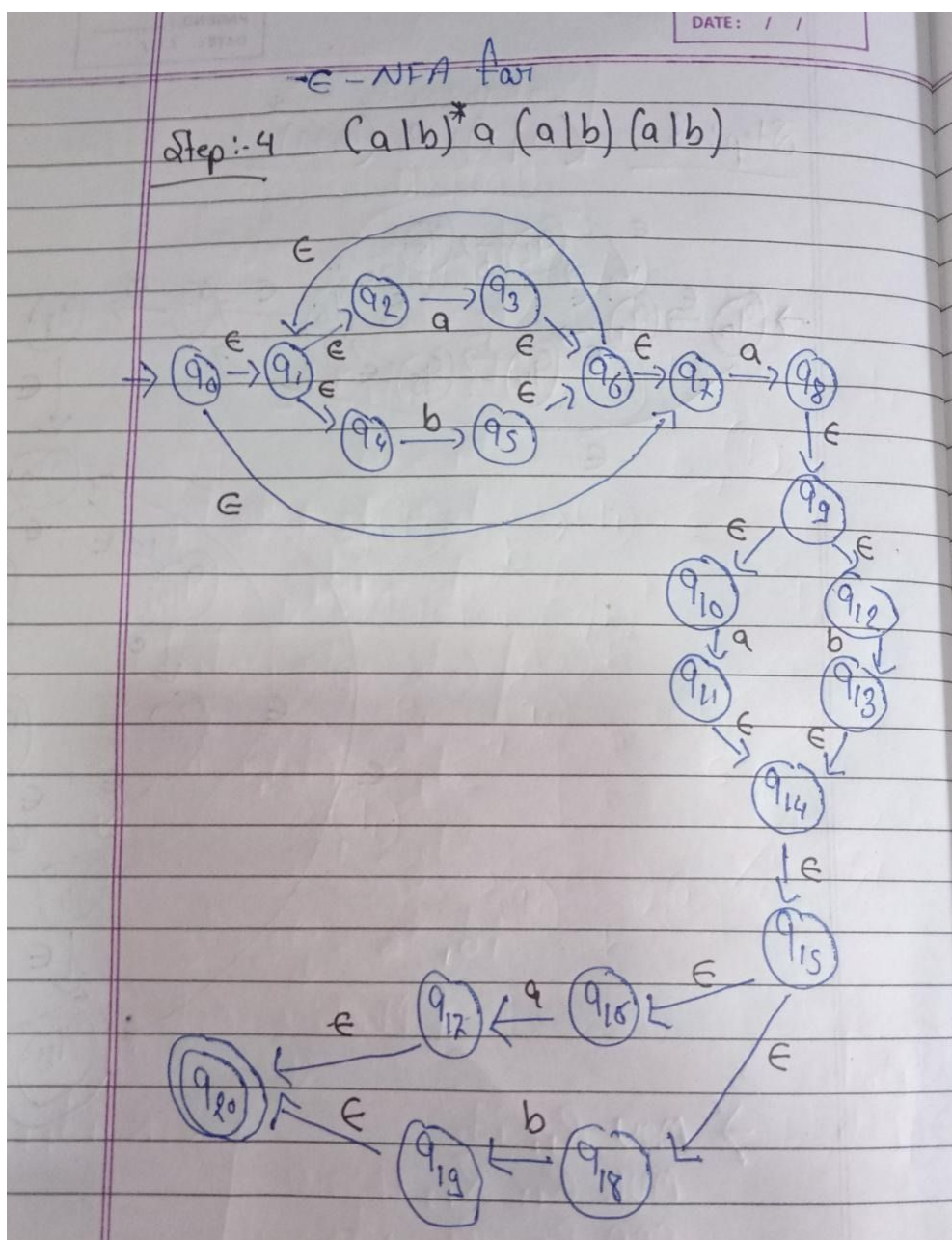
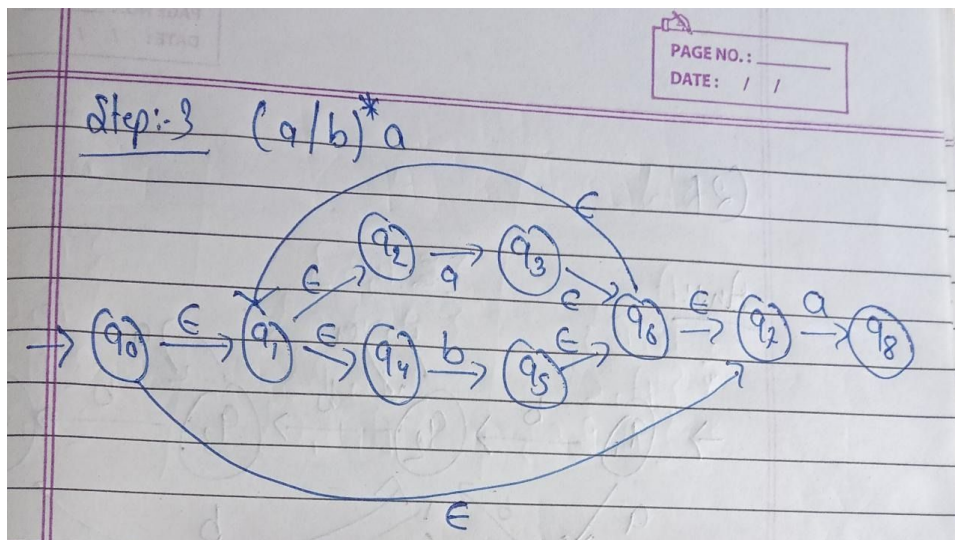
[See surya's solution](#)

2. Minimal DFA which accepts a set of all strings over  $\{a,b\}$  in which no. of a's are divisible by 3 and no. of b's is divisible by 3.



3. Show the construction of NFA for the following Regular Expression  $(a|b)^*a(a|b)(a|b)$

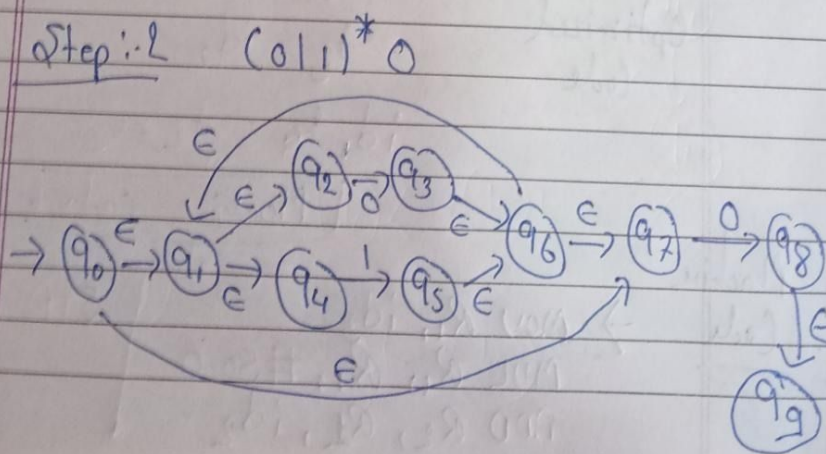
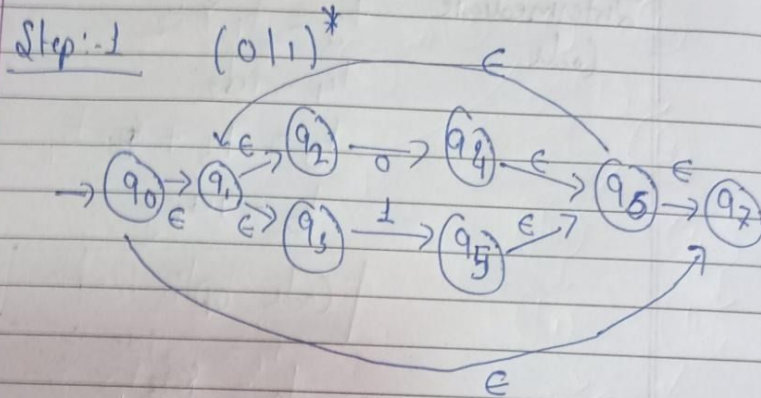


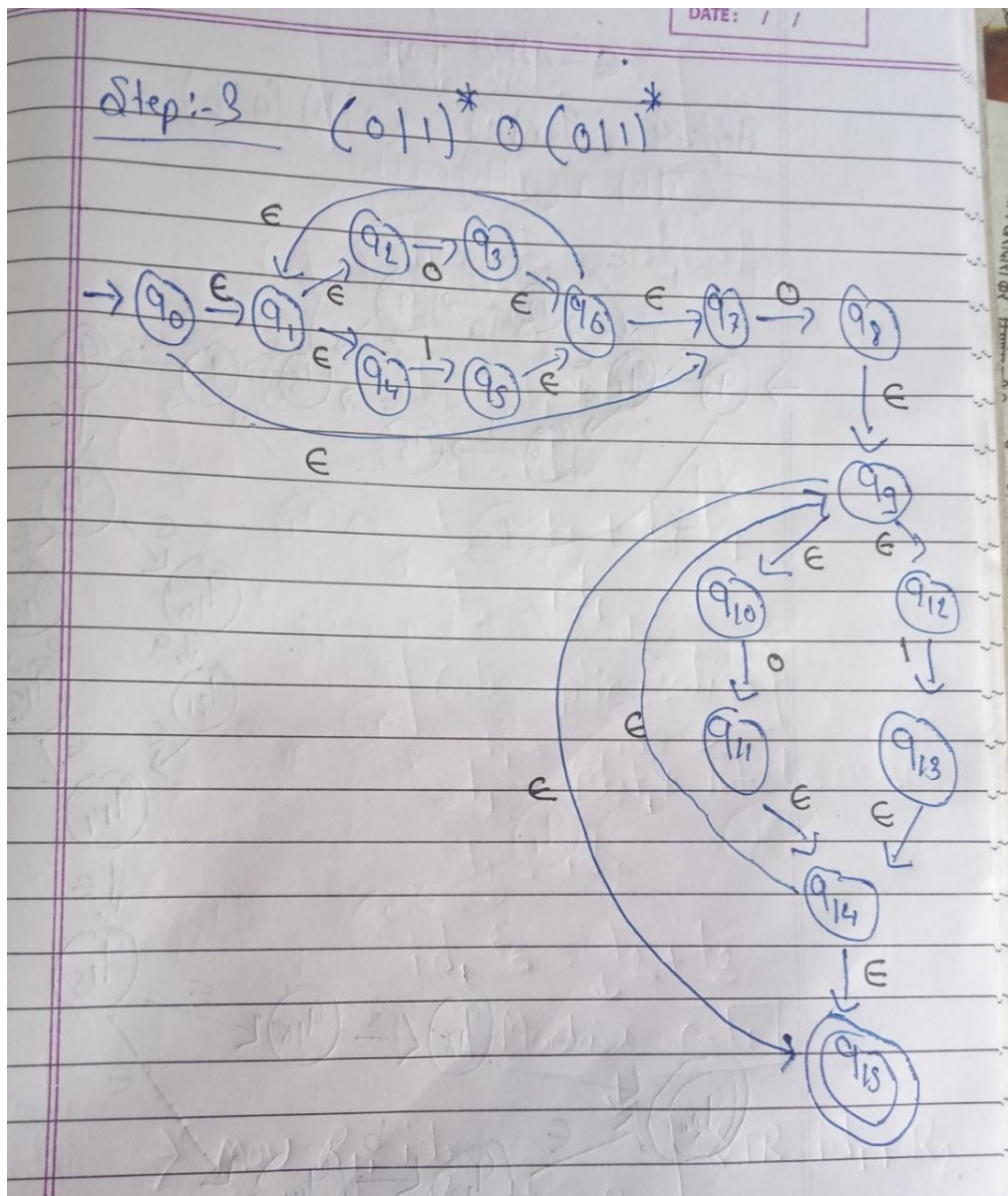




4. Construct NFA for the following RE using Thomson's construction:  $(0|1)^*0(0|1)^*$

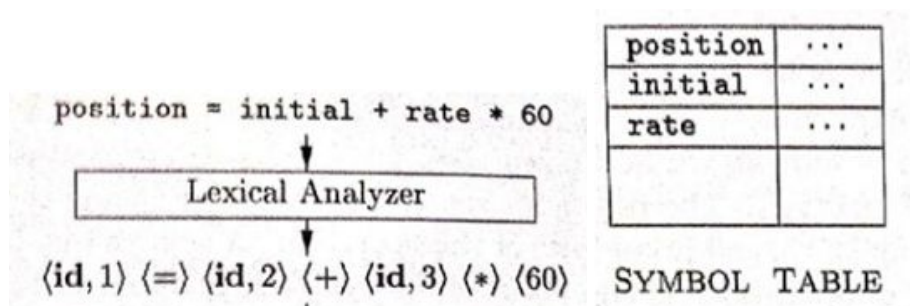
(4) Using Thomson's Construction,  
Construct NFA,  
 $(0|1)^*0(0|1)^*$





## 5. Lexical Analysis:-

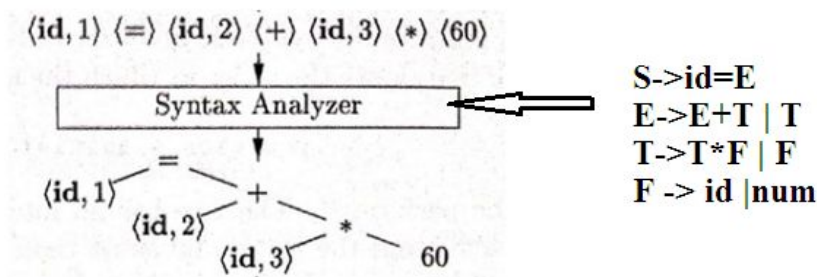
This phase scans the source code as a stream of characters and converts it into meaningful lexemes. A Lexical analyzer represents these lexemes in the form of tokens.



## Syntax Analysis:-

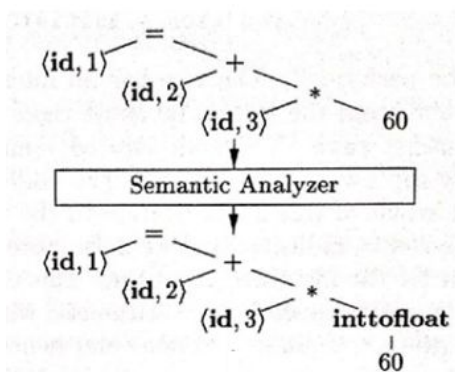
It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.





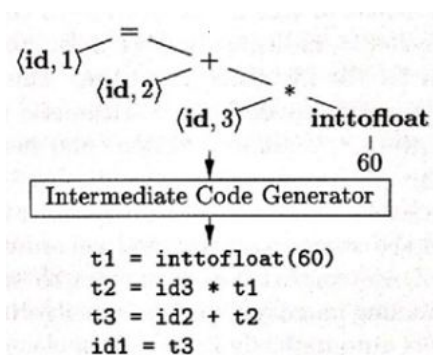
### Semantic Analysis:-

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:-

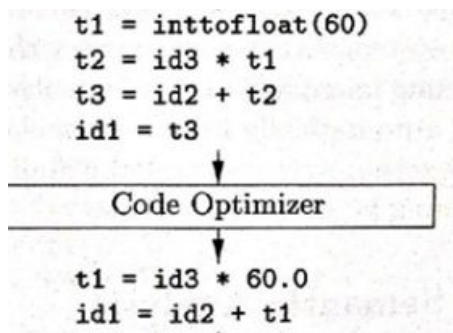
After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it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



### 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).



### **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) relocatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

