UNIT-I

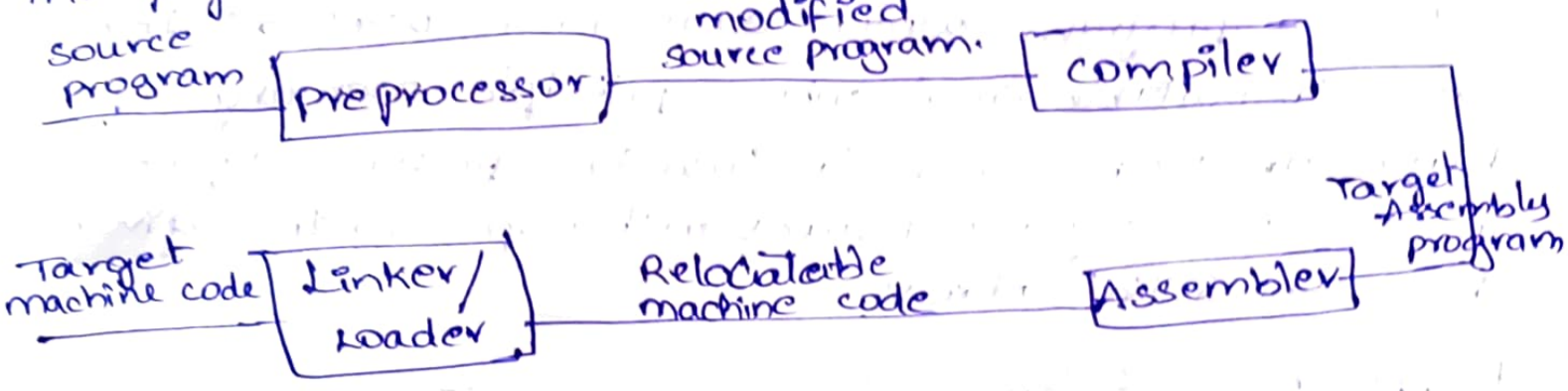
Language processing / Processors:-

Language processing system is a system that translates the source language which is taken as input into machine language.

→ This translation can be done by dividing the source file into modules. These modules are as follows

1. preprocessor
2. compiler
3. Assembler
4. Linker / Loader

The typical language processing system is shown in the figure below.



1. preprocessor:-

It is a special program processing, the code prior to the actual translation to perform necessary functions like deleting comments, adding necessary files and doing macro substitutions.

2. compiler:-

A compiler is a program that converts a sources program into a target program.

3. Assembler:

Assembler is a translator which translates assembly language program into object code. This program specifies symbolic form of the machine language of the computer.

4. Linker / Loader :-

Linker :-

It is a program that links the two objects files containing the compiled or assembled code to form a single file which can be directly executable. It is also responsible for performing the following functions.

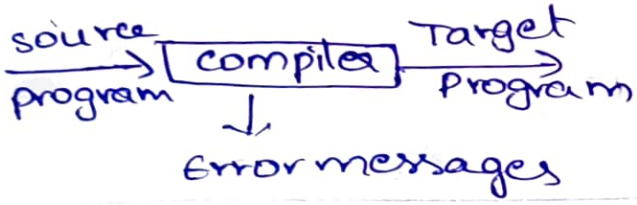
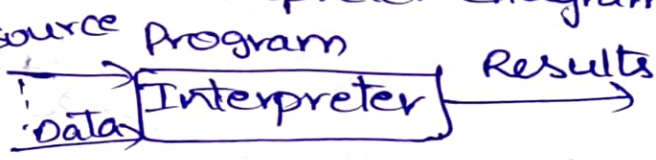
1. Linking the object program with that of the code for standard library functions and
2. Resources provided by the operating system like memory allocators and input and output devices.

Loaders :-

It is a program which loads or resolves all the code (relocatable code) whose principle memory references have undetermined initial locations present anywhere in the memory. It resolves with respect to the given base or initial address.

Differences between compiler and Interpreter

compiler	Interpreter
1. compiler is a program used to convert high level language to machine code and it executes whole program at a time.	1. It is same like compiler, but it executes only one statement at a time.
2. Some compilation languages are C, C++, FORTRAN, PASCAL, ADA. etc.	2. Some interpretation languages are LISP, ML, PROLOG, Small Talk. etc.

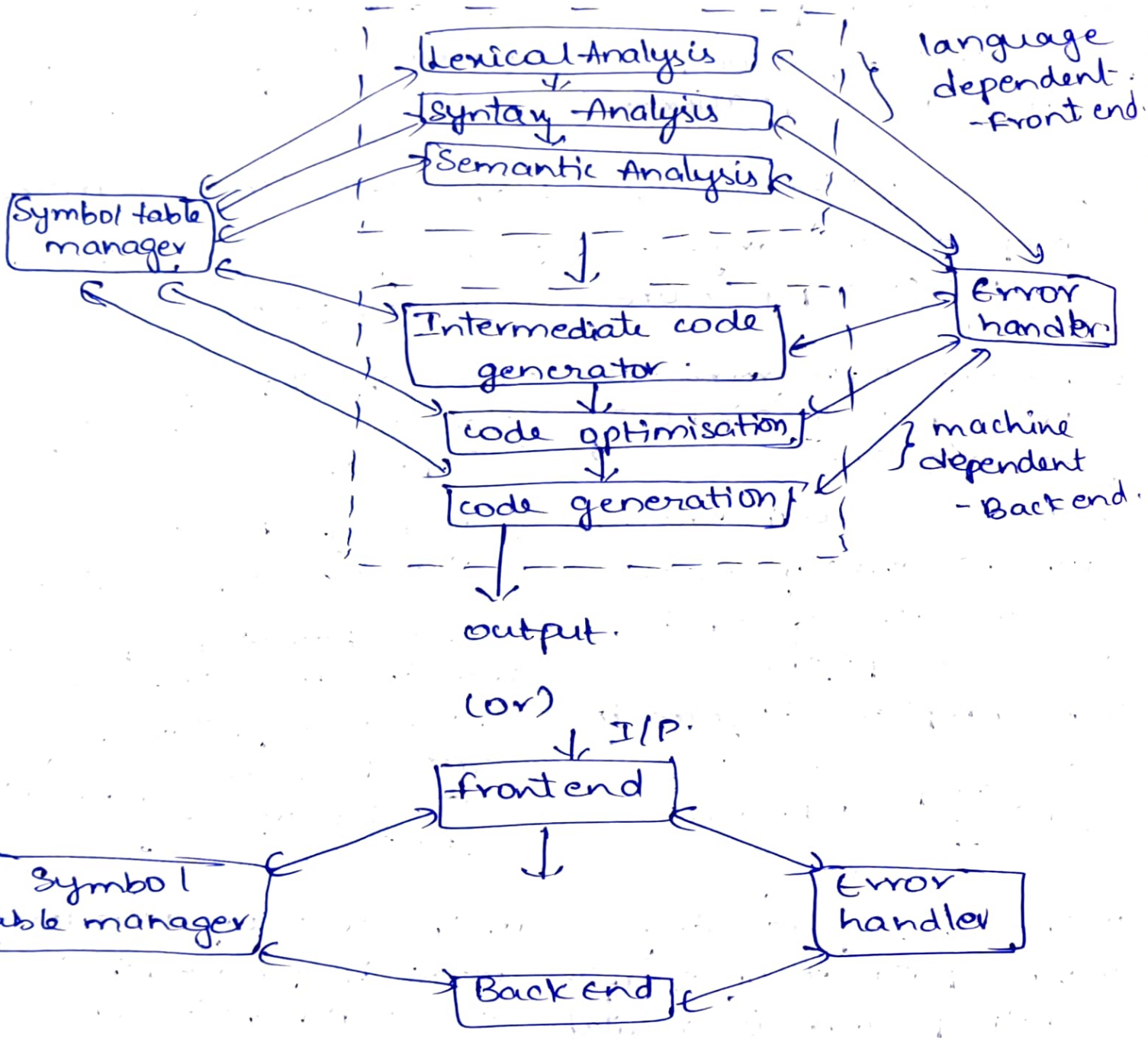
3. The translation process carried out by a compiler is termed as compilation.	3. The translation process carried out by the interpreter is termed as interpretation
4. processing time is less	4. processing time is more
5. compilers are large in size and occupy more memory	5. Interpreters are smaller than compilers
6. The object program execution is not carried out by the compiler.	6. The object program execution is carried out by the interpreter.
7. It process each program statement exactly once.	7. It might process some statements repeatedly.
8. The execution speed is more in compiler.	8. The execution speed is less in an interpreter.
9. The compiler diagram is 	9. The Interpreter diagram is 
10. The cost of decoding must be paid at once after the entire program is executed.	10. The cost of decoding must be paid each time the statement is to be executed
11. It is also called as software translation	11. It is also called as software simulation.

Structure of a compiler :-

→ A compiler is a program that reads a program return in one language and translates it into an equivalent program in another language. and the compiler reports to its users, the presence of error in the source program.

→ The program written is called source program

→ The program translated is called target program.



→ A compiler can be broadly divided into two parts, based on whether the phase is language dependent or machine dependent.

Phases of a compiler

A compiler is divided into no of parts, segments or sections session and each session is known as a phase.

→ A compiler is divided into two parts
- Analysis part (front end)
- Synthasis part (Back end)

→ The Analysis part is machine independent and language dependent.

→ The synthasis part is machine dependent and language independent.

→ The analysis part is divided into three phases
a) Lexical Analyzer (or) scanner
b) Syntax analyzer (or) parser
c) semantic Analyzer

→ The synthasis part is divided into three phases
d) Intermediate code generator.
e) code optimizer
f) code generator

→ Two additional activities of a compiler are
g) Symbol table manager h) error handler

a) Lexical Analyzer :- (scanner)
- The lexical analyzer does lexical analysis
- This phase reads the characters in the source program and groups them into a stream of tokens
- TOKens are the sequences of characters having a collective meaning

• The character sequence forming a token is called the lexeme.

Ex:- Consider an assignment statement.

position := initial + rate * 60

Sol

Lexeme	Token
position	id
:=	Assignment symbol
initial	id
+	plus sign (Addition)
rate	id
*	multiplication
60	numeric literal

The statement after the lexical analysis is given by $id_1 := id_2 + id_3 * 60$

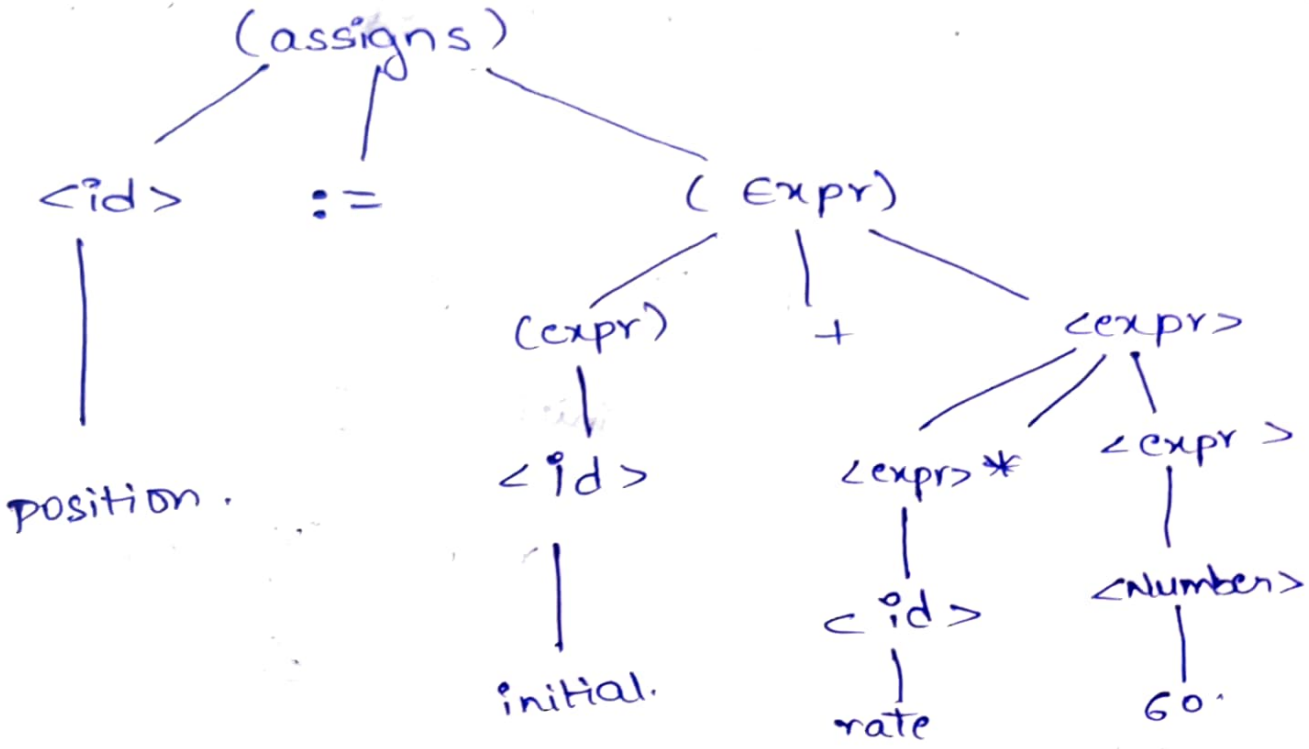
where id_1, id_2, id_3 are tokens for position, initial and rate respectively.

b) Syntax Analyzer (Parser):-

• The syntax Analyzer does Syntax Analysis.

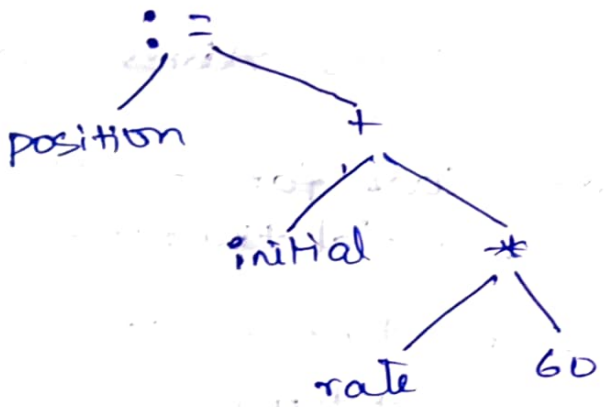
• It groups the tokens into grammatical parsers phrases represented by a hierarical structure called a parse tree.

• The parse tree for the input string $id_1 := id_2 + id_3 * 60$ is



It can be also represented by a syntax tree, which is compressed representation of the parse tree.

→ The syntax tree is,



c) Semantic Analyzer

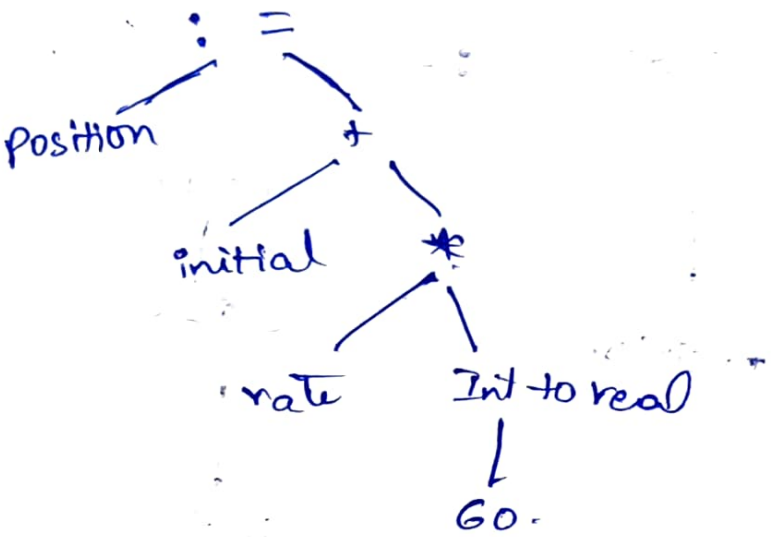
- Semantic Analyzer does the semantic Analysis.
- Semantic Analysis checks the source program for semantic errors. It performs type checking

Example:- for the input string, $id_1 := id_2 + id_3 * 60$.

Let all identifiers are real, then we have to convert the number 60 as int to real.

The abstract syntax tree after Semantic Analysis

C.



D) Intermediate Code Generator

→ After syntax and semantic Analysis, the compiler generator in intermediate representation of the source program.

→ one popular type is "Three Address code".

Example:-

→ The three Address code for the statement position := initial + rate * 60 is given as

temp1 := Int to Real (60)

temp2 := id3 * temp1.

temp3 := id2 + temp2.

id1 := temp3.

e) Code optimizer :-

→ The code optimization phase (includes) improves intermediate code.

→ The code $id_1 := id_2 + id_3 * 60$ can be optimized as;

temp1 := id_3 * 60.0

id_1 := id_2 + temp1

f) Code generator:-

→ The code generation phase converts the intermediate code into a targetcode

→ The target code consisting of sequenced machine code (or) assembly code that perform the same task.

example:- Assembly code for $id_1 := id_2 + id_3 * 60$ is,

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

→ The # signifies that 60.0 is treated as constant.

g) Symbol Table Manager:-

→ A symbol table management (or) Bookkeeping is a position of the compiler which keeps track of the names used by the program and records information such as datatype, precision, and so...on.

→ The data structure used to record this information is called symbol table.

→ This datatabStructure allows us to find the record for each identifier quickly and to store (or) retrieve data quickly.

h) Error Handler :-

→ Error Handler is invoked when a fault in the source program is detected.

→ The syntax and semantic Analysis phases usually handle a large number of errors.

Example:- Error detected in different phases.

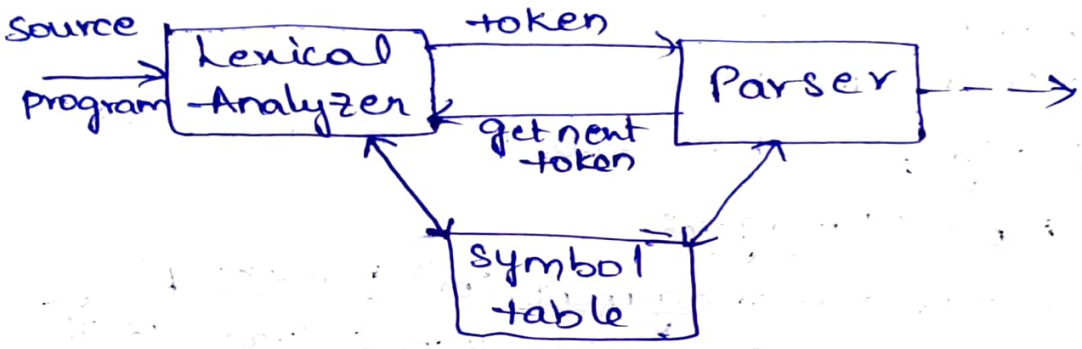
Lexical Analysis — Token Misspelled

Syntax Analysis — missing paranthesis (or) semicolon

Semantic Analysis — incompatable types
Intermediate CG — statements never reached
code generator — toolarge to fit in a word.

Lexical Analysis :- (The Role of Lexical Analysis)

→ The lexical analysis is the first phase of a compiler
→ Its task is to read the input characters and produce as sequence of tokens as output.
→ The interaction of lexical Analyser with parser is



→ upon recieving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token..

(*) Need for Lexical Analysis :-

→ The purpose of splitting the analysis of a sequence source program into three phases, i.e., lexical analysis, syntax analysis and semantic analysis is to simplify the overall design of the compiler.
→ In lexical analysis, it is easier to specify the structure of the source program.
→ Lexical Analyzer keeps track of line numbers, producing an output listing if necessary, stripping out white space such as redundant blanks and tabs and deleting comments.

(*) Tokens, patterns and lexemes:-

→ Token is a sequence of characters having a collective meaning.

→ pattern is a rule describing the set of strings in the input for which the same token is produced as output (or) It is a rule describing the set of lexemes that can represent a particular token in the source program.

→ Lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

→ In most programming languages keywords, operators, identifiers, constants, literal strings and punctuation symbols such as paranthesis, commas and semicolons are treated as tokens

Ex:- if(a<b)

Lexeme	Token	Pattern
if	identifier	if
(	left paranthesis	(or)
a	identifier	letter followed by letter (or) digit.
<	relational operator.	< or <= or > or >= or <> or ==
b	identifier	letter followed by letter (or) digit
)	Right paranthesis	(or)

(*) Attributes for Tokens:-

→ Lexical analyzer provides additional information to distinguish between the similar type of pattern that match a lexeme.

→ The lexical analyzer collects the attributes (properties or features) of tokens as the additional information.

→ A token has a single attribute, i.e., a pointer to the symbol table entry in which the information about the

token is kept

Ex :- E = M * C ** 2

lexeme	Tokens	Attribute values
E	id	Pointer to symbol-table entry for E
:=	assign-op	—
M	id	pointer to symbol-table entry for M
*	Multi-op	—
C	id	pointer to symbol-table entry for C
**	exp-op	—
2	number	2

Input Buffering

→ There are three general approaches to the implementation of a lexical analyzer.

(i) use a lexical analyzer generator such as lex compiler to produce the lexical analyzer from a regular-expression based specification.

In this case, the generator provides routines for reading and buffering the input.

(ii) write the lexical analyzer is a conventional system-programming language using the I/o facilities of that language to read the input

(iii) write the lexical analyzer is assembly language and explicitly manage the reading of input.

→ The lexical analyzer is the only phase of the compiler that reads the source of program character-by-character

Buffer-pairs :-

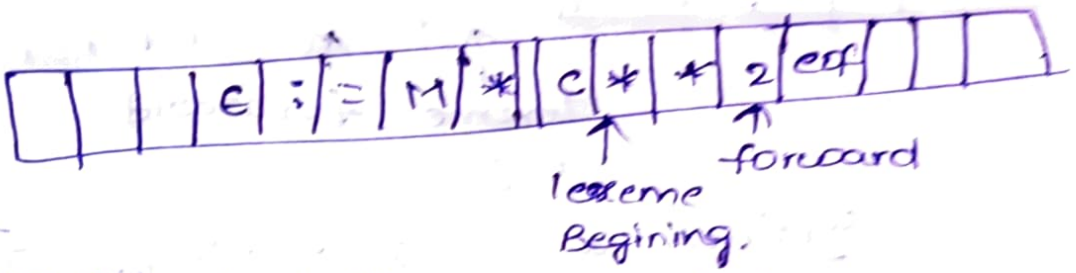
→ The lexical analyzer scans the characters of the source program one at a time to discover tokens.

→ There are many schemes that can be used to buffer input.

→ one class of schemes here,

- A Buffer divided into two N-character halves.
- 'N' is the number of characters on one disk block.

ex:- 1225.



∴ An input buffer in two halves

- We read 'N' input characters into each half of the buffer with one system read command, rather than invoking a read command for each input character.
- If fewer than 'N' characters (into each h) remain in the input, then a special character eof is read into the buffer after the input characters.
- eof marks the end of the sourcefile and is different from any input character.

→ Two pointers to the input Buffers are maintained.

→ The string of characters between the two pointers is the current lexeme

(i) Lexeme begining pointer

(ii) forward (or) lookhead pointer.

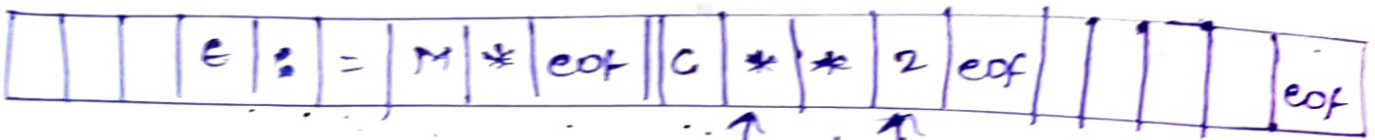
Sentinels:-

→ Except at the ends of the buffer halves, the code requires tests for each advance of the forward pointer.

→ we can reduce the two tests to the one if we extend each buffer half to hold a sentinel character at the end.

→ The sentinel is a special character [eof] that cannot be part of the source program.

→ The same buffer management with sentinels added is



lexeme Beginning

forward

Sentinels at end of each Buffer Half

⑨ Token Specifications :-

(i) Alphabets, Strings, languages and language operators.

(ii) Regular expressions

(iii) Regular Definations (or) Regular Grammary

(iv) Notational shorthands (or) Extensions of Regular expressions.

(ii) Regular Expression :-

→ A Regular expression is any well-formed formula constructed over union, concatenation and closure

→ Each regular expression (r) denotes a language L(r)

→ It is having a set of defining rules.

- 'ε' is a regular expression that denotes {ε}, i.e., the set containing the empty string
- If 'a' is a symbol in 'Σ', then 'a' is a regular expression that denotes {a}, i.e.,

the set containing string 'a'.

- Suppose 'r' and 's' are two regular expressions denoting the languages L(r) and L(s). Then
  - → r+s is a regular expression denoting L(r) ∪ L(s)
  - → r·s is a regular expression denoting L(r)·L(s)
  - → r* is a regular expression denoting L(r*)

A language denoted by a regular expression is said to be regular set

→ There are three operations on regular expressions.

a) Union, denoted by + (or) |

b) concatenation, denoted by dot(·) (or) no symbol.

c) closure, denoted by *.

Precedence of operations

→ Unary operator '*' has the highest precedence and is left associative.

→ concatenation operator has the second highest precedence and is left associative.

→ Union operator '+' has the lowest precedence and is left associative

ex:- ab^*+a has the steps

b^*

ab^*

ab^*+a

Examples of Regular expressions:-

→ Let $\Sigma = \{a, b\}$

* The regular expression a/b denotes the set {a,b}
* The regular expression (a+b)(a+b) denotes {aa,ab, ba, bb}, the set of all strings of a's and b's of length two
* The regular expression a^* denotes $\{\epsilon, a, aa, aaa, \ldots\}$ the set of all strings of zero (or) more a's.

* $(a+b)^*$ denotes the set of all strings containing zero (or) more instances of a's (or) b's.

"Algebraic Laws" for Regular Expressions:-

AXIOM.	Description
$r/s = s/r$	/ is commutative
$r/(s/t) = (r/s)/t$	/ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s/t) = rs/rt$ $(s/t)r = sr/tr$	concatenation distributes over /
$\epsilon r = r\epsilon = r$	ϵ is the identity element for concatenation
$r^* = (r/\epsilon)^*$	relation between * and ϵ
$r^{**} = r^*$	* is idempotent

Reserved words and Identifiers: (Recognition of Tok

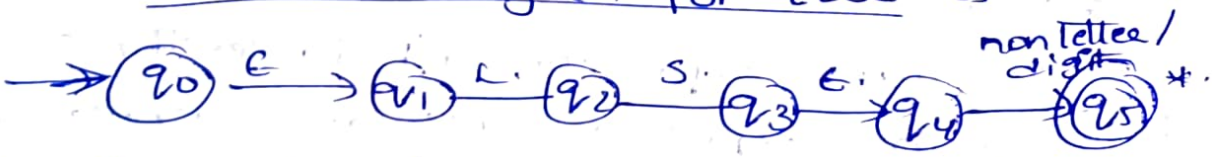
→ A Reserved word is also called as Keyword.

→ Keywords are the words whose meaning has been already explained in the compiler.

→ Example keywords are,

auto, break, case, char, const, continue, do, default, else, enum, if, for, while, - - - - -

→ The transition diagram for ELSE is



→ An identifier is any user-defined name

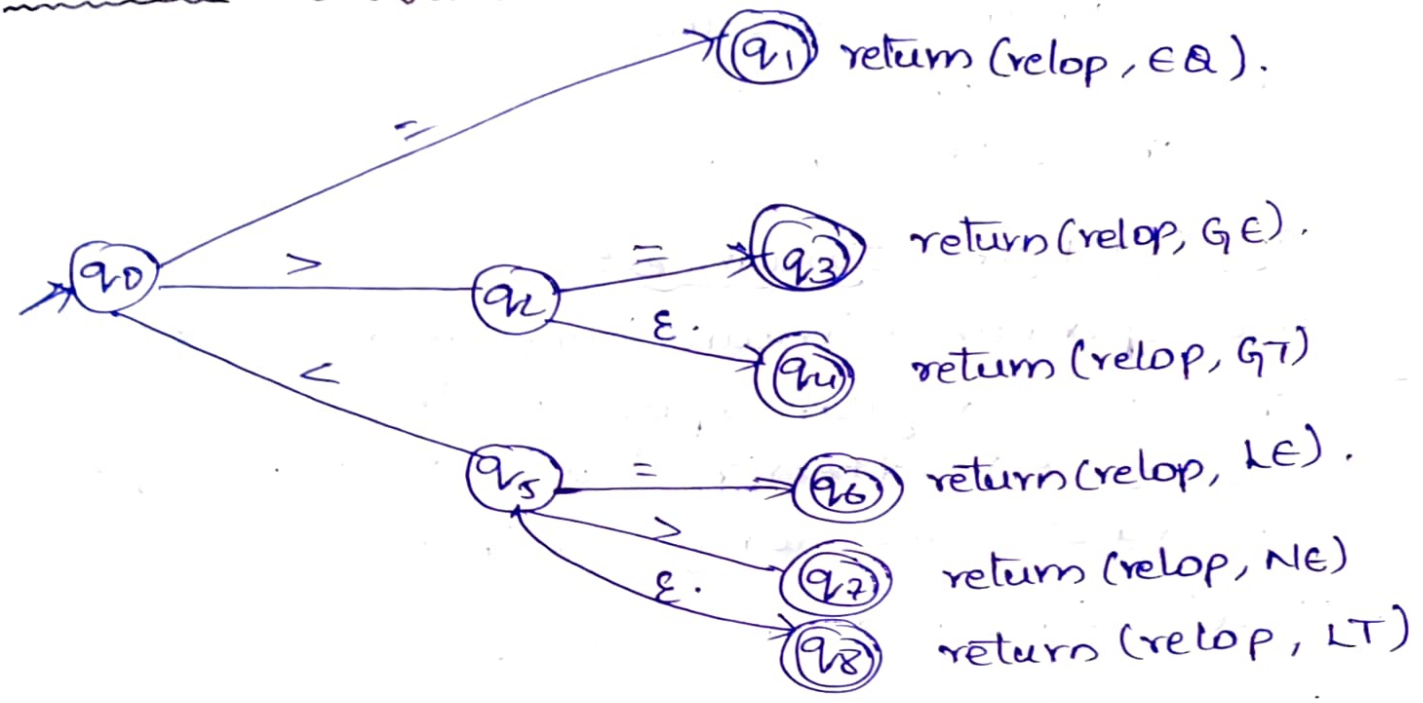
example:- a, b, c, sum, sum(), print(), display();

→ An identifier cannot start with a digit.

→ Keywords (or) Reserved words cannot be represented as Identifiers.

→ Transition diagram for identifier is

Transition Diagram for Relational operators :-



(iii) Regular Definations :-

→ If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form.

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\cdots$$
$$d_n \rightarrow r_n$$

where, each d_i — distinct name

r_i — Regular expressions over the symbols and previously defined values/names.

Ex:- Regular defination for identifier in pascal language.

letter → A|B|----|z|a|b|---|z.

digit → 0|1|----|9

id → letter(letter|digit)*.

(iv) Notational shorthands:-

a) one (or) more Instances

b) zero (or) one Instance

c) character classes.

a) one (or) More Instances:-

→ The operator '+' means "one (or) more Instances of"

→ If "r" is a regular expression that denotes the language L(r), then $(r)^+$ is a regular expression that denotes the language $(L(r))^+$

Ex:- $a^+ = \{a, aa, aaa, \ldots\}$.

b) Zero (or) one Instance:-

→ The operator '?' means "zero (or) one Instance of"

→ The notation r? is a shorthand for r/ε.

Ex:- a? = a/ε

c) character classes:-

→ It uses the Symbols [ ]

→ The notation [abc] where a,b,c are alphabet symbols denotes the regular expression a/b/c

→ An abbriveated character class such as [a-z] denotes the regular expression a/b/---/z

→ A pascal identifier is represented by a regular expression as

[A-z a-z] [A-z a-z 0-9]*

1) LEX Tools :- Lex Analyser Generator - LEX

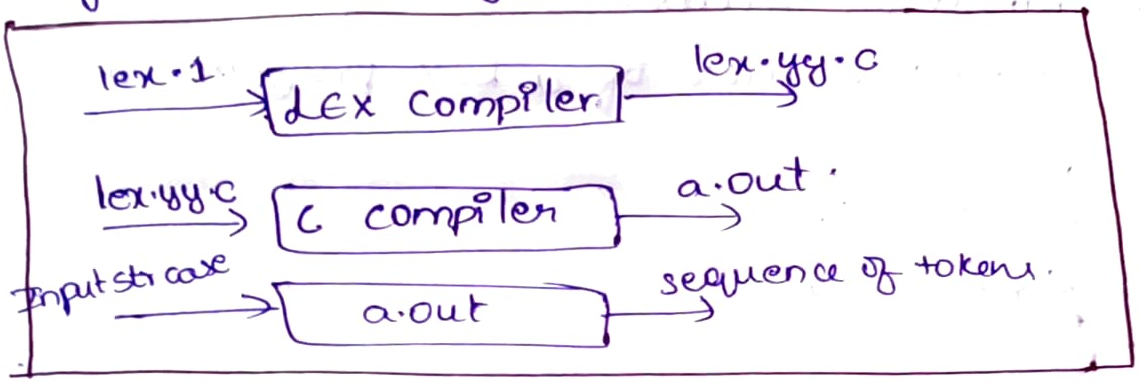
→ There is a wide range of tools for construction of lexical Analysis

→ The majority of these tools are based on regular expressions.

→ one of the traditional tools of regular expressions is LEX (or) lex.

→ we refer to the tool as the LEX compiler and its input specification as the lex language.

→ creating lexical Analyzer with lex is represented as,



→ A specification of the lexical analyser is prepared by creating a program lex.l is the lex language

→ The lex.l is run through the lex compiler to produce a 'c' program lex.yy.c.

→ The program lex.yy.c consists of tabular representation of a transition diagram constructed from the regular expression of lex.l, together with a standard routine that uses the table to recognize lexemes.

* LEX Specifications :-

→ A lex program consists of three parts.

declarations
%%
translation rules
%%
auxiliary procedures

→ The declaration section includes, the declaration of
- Variables
- manifest constants and
- regular definitions

→ A manifest constant is an identifier that is declared to represent a constant

→ The regular definitions are statements and are used as components of the regular expressions appearing in the translation rules.

→ The Translation rules of a LEX program are statements of the forms

P_1 {action 1}
P_2 {action 2}
— — — — —
— — — — —
Pn {action n}.

- where pi is a regular expression.

→ Auxilary procedures are needed by the actions and can be seperatedy compiled and loaded with the lexical Analyzer.

*) Example:-

i) write a lex program to identify comments in the program

```
%. { /* Lex program to identify comments in the program*/}
comments %%        {/* (letter/digit/whitespace)+*/}.
{comments}.        { /* No action is taken, No value is returned*/}
%%
install_id()       { /* Install lexemes in the symbol table */}
install_num()      { /* Install numbers as lexemes in the
                        Symbol table */}.
```

LEX PROGRAM to Recognize the decimal number:

→ Lex program is a tool that is used to generate lexical analyzer (or) Scanner.

→ The tool is often referred as lex compiler and its input specification as Lex language.

→ Lex program to recognize decimal numbers is,

```
%{
/* The program to recognize decimal number */
#include<stdio.h>
int q;
%}
%%[0-9]+[.]
{
for(i=0;i<yyleng;q++)
{
if (yytext[i]='.')
{
printf("%s is a decimal number",yytext);
}.
}.
printf("%s is not a decimal number", yytext);
}.
%%
main()
{
printf("\n enter any number \n");
yylex();
}
int yywrap()
{
return 1;
}.
```

output:-

```
[ ]# lex decimal.l
[ ]# cc lex.yy.c
[ ]# ./a.out
enter any number
0.25
0.25 is a decimal number.
```

Computation:-

The process of solving a problem to obtain a res is called computation.

- computation process can be represented by using mathematical models.

Types of mathematical models:-

The mathematical models are four types they are

- finite automata
- pushdown automata
- Linear bounded automata
- Turing machine.

Finate Automata

Finate Automata is understanding only one language, that language is called Regular Language (RL) followed by with the help of regular grammer (RG)

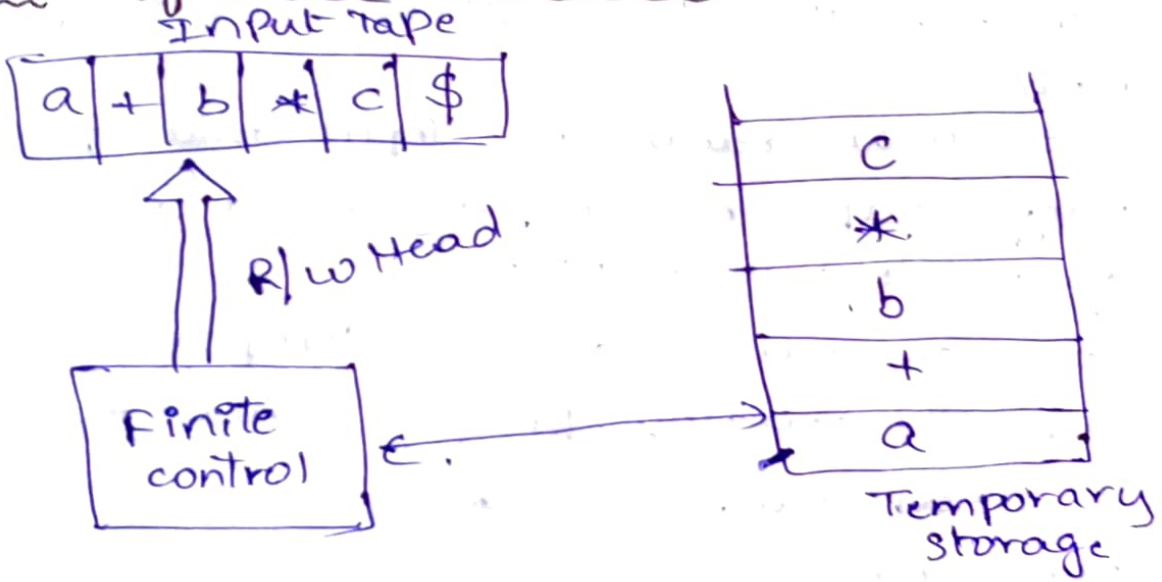
* Introduction
* components of Finate Automata
* Elements of Finate Automata.

* Representation of Finate Automata
* Examples.

Introduction:-

→ It is a self operating machine
→ It is a system which obtains, transforms, transmits and uses information to perform its functions without direct participation of humans.
→ Finite Automata is used in mathematical analysis
→ Application of finite automata is lexical analysis phases in compiler design.

Model of Finate Automata



• Components of Finate Automata

- Input Tape
- RBad /write Head.
- Temporary stage
- Finate control

- Input Tape:-
  - It is a memory storage used to store the input data, memory area is divided into number of cells.

• Each cell can hold only one input symbol at a time.
• The input string ends with end marker ($)

~Read/write Head:
• It is used by the finate control to read the input data from left side to right side in the input tape.
• Read write Head can move from and reads only one input symbol at a time

~ Finate control :-
• It controls the entire process of a system or machine
• Finate control reads the input data from the input buffer tape by moving read or write head from left to right.
• It stores the reading data in temporary storage.
• It stops the reading process when the read or write head reaches to end marker ~~to~~ in the input head tape.

~

• Elements of finate Automata

1. State 2. Transitions 3. Transition diagram/state diagram 4. Transition table.

1. state :- It is a behaviour that produce an action.
* It is a location in input buffer.
* The finite control reads the data from one state to another state in the buffer.
* states are classified into four types.

(i) initial state (or) starting state
(ii) Final state (or) Acceptance state
(iii) Intermediate state
(iv) Invalid state (or) Dead (or) Trap state

(i) Initial state (or) starting state :-

*The finite control can starts its reading process from a state is called initial state.

(ii) Finite state (or) Acceptance state :-

Finite control can stops its reading process after complete or end of given string then it reaches a state that state is called final state

(iii) Intermediate state :-

The states between starting state and final state are called Internal (or) Intermediate state.

(iv) Invalid state (or) Dead (or) Trap state :-

It is a invalid state when the finite control reads the input data from dead state then it always goes to dead state

2. Transitions :-

* It means moving one place to another place
* It is defined by Transition function.
* Transition function is denoted by δ.

3. Transition Diagram :-

* It is a graphical representation of finite automata.
* It is directed graph
* It is constructed by using the following symbols.

Symbol	Meaning
○	State
→ ○	initial state
◎	final state
◌	dead state
→	transition.

Ex:-

$\delta(q_0, a) = q_1$

$\delta(q_0, b) = q_0$

$\delta(q_1, b) = q_2$

$\delta(q_1, a) = q_1$

$\delta(q_2, a) = q_2$

$\delta(q_2, b) = q_2$.

4. Transition Tables:-

*It is a tabular representation of finite automata

*It is combination of rows and columns. Here rows represents states. Columns represents input symbols. The Entry in between row and column is always states.

eg:-

State	Input Symbols	
	a	b
→ q_0	q_1	q_0
q_1	q_1	q_2
◎ q_2	q_2	q_2

• Representation of Finite Automata

Mathematically a finite automata is a 5-tuple machine like $M = (Q, \Sigma, \delta, q_0, F)$ where

$Q \rightarrow$ set of states

Q is a finite and non-empty set of states

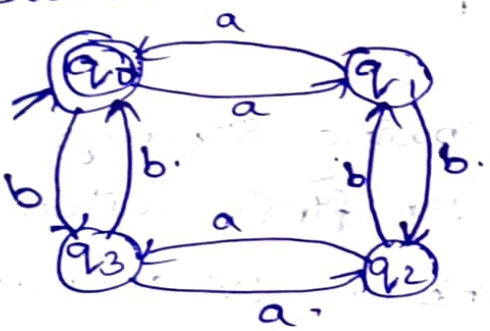
$\Sigma \rightarrow$ It is finite and non-empty set of input symbols

$\delta \rightarrow$ It is a transition function which is defined as $\delta: Q \times \Sigma \rightarrow Q$.

$q_0 \rightarrow$ It is a initial state or starting state and $q_0 \in Q$.

$F \rightarrow$ It is a finite set of final states and $F \subseteq Q$.

Note:- * Finite Automata contains one initial state
* Finite automata contains one or more final states



Mathematically it is represented as

$M = (Q, \Sigma, \delta, q_0, F)$ where

$Q = \{q_0, q_1, q_2, q_3\}$

$\Sigma = \{a, b\}$.

δ is a transition function which is defined as

$\delta: Q \times \Sigma = Q$.

Transition Table :-

δ: State	Input symbols a	b
→ (q0)	q1	q3
q1	q0	q2
q2	q3	q1
q3	q2	q0.

$q_0 = \{q_0\}$.

$F = \{q_0\}$.

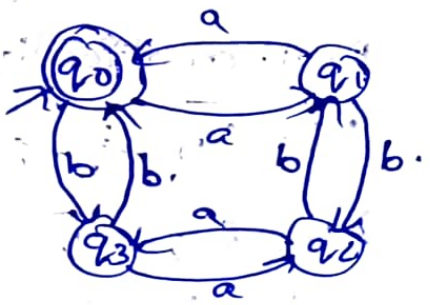
Acceptance of a string by FA

Let w be a given string and finite automata M contains Q states like $Q = \{q_0, q_1, q_2 \ldots q_f\}$ where q_0 is the initial state q_f is the final state. The string 'w' is accepted by machine M. If and only if $\delta(q_0, w) = q_f$.

Ex:- Check whether the following strings are accepted or not by the given finite automata.

(i) aabb (ii) ababa (iii) aabbaa (iv) abababb.

Sol The given finite automata.



$M = (Q, \Sigma, \delta, q_0, F)$ where

$Q = \{q_0, q_1, q_2, q_3\}$.

$\Sigma = \{a, b\}$.

δ: State	Input a	b
→ (q0)	q1	q3
q1	q0	q2
q2	q3	q1
q3	q2	q0

$q_0 = \{q_0\}$

$F = \{q_0\}$.

(i) $w = aabb$

$\delta(q_0, aabb) = \delta(q_1, abb)$

$= \delta(q_0, bb)$

$= \delta(q_3, b)$

$= q_0 \in F.$

∴ The given string $w = aabb$ is accepted by F.A M.

(ii) $w = ababa$

$\delta(q_0, ababa) = \delta(q_1, baba)$

$= \delta(q_2, aba)$

$= \delta(q_3, ba)$

$= \delta(q_0, a)$

$= q_1$

∴ The given string is $w = ababa$ is not accepted by F.A.

(iii) $w = aabbaa$

$\delta(q_0, aabbaa) = \delta(q_1, abbaa)$

$= \delta(q_0, bbaa)$

$= \delta(q_3, baa)$

$= \delta(q_0, aa)$

$= \delta(q_1, a)$

$= q_0 \in F.$

∴ The given string $w = aabbaa$ is accepted by F.A.M.

(iv) $w = ababbabb$

$\delta(q_0, ababbabb) = \delta(q_1, babbabb)$

$= \delta(q_2, abbabb)$

$= \delta(q_3, babb)$

$= \delta(q_0, abb)$

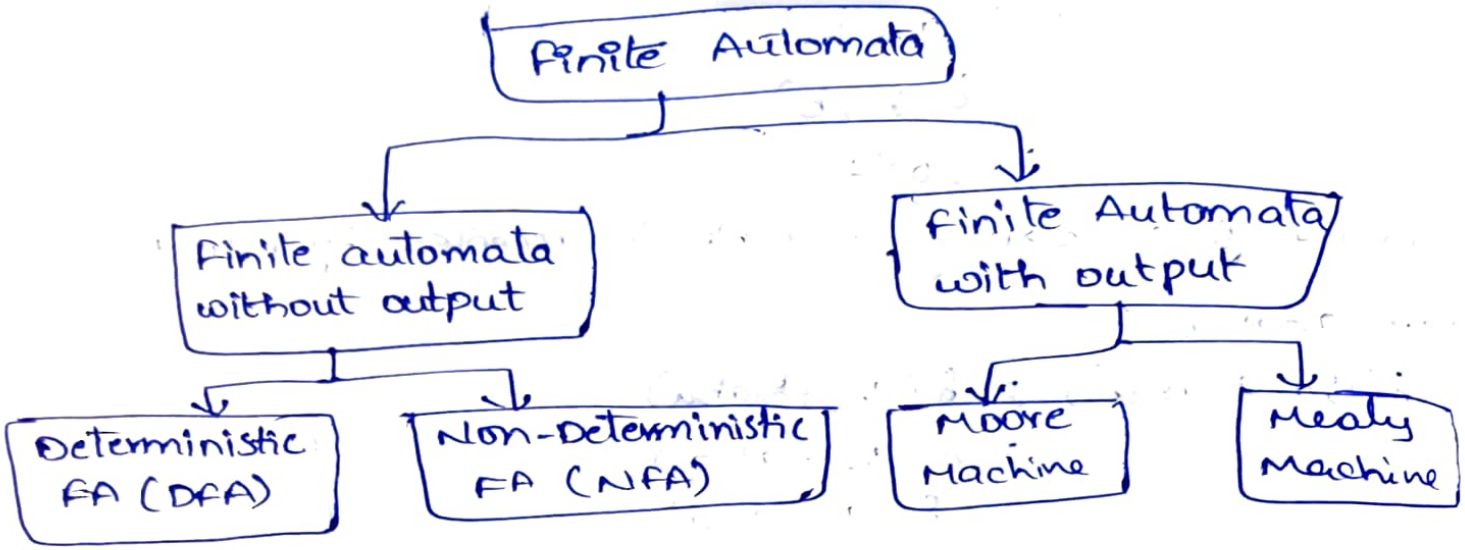
$= \delta(q_1, bb)$

$= \delta(q_2, b)$

$= q_1$

∴ The given string $w = ababbabb$ is not accepted by F.A.

Types Of FA :-



Bootstrapping :-

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..

writing a compiler for any high level language is a complicated process. It takes a lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages to clearly understand the Bootstrapping technique consider a following scenario.

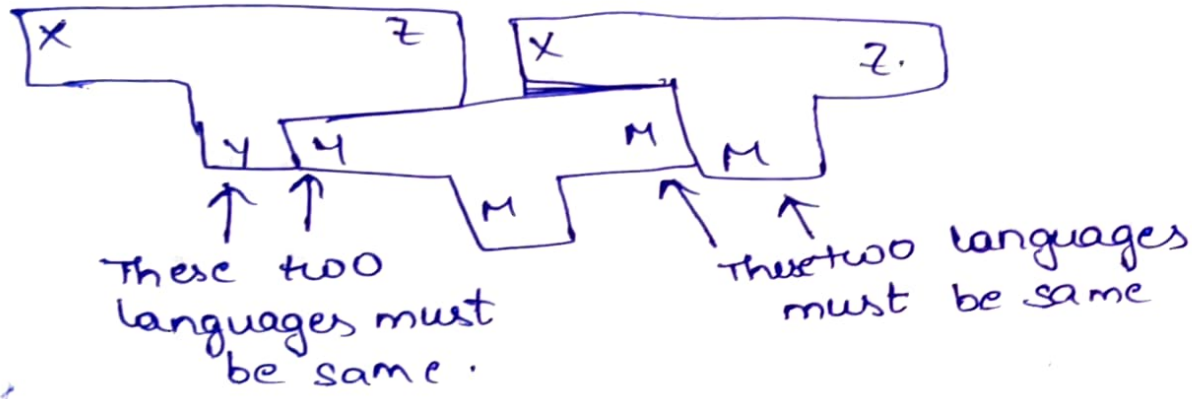
Suppose we want to write a cross compiler for new language X. the implementation language of this compiler is say Y and target code being generated is in language Z. That is, we create X-Y-Z. Now if existing compiler Y runs on machine

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.

Following diagram illustrates the above scenario.

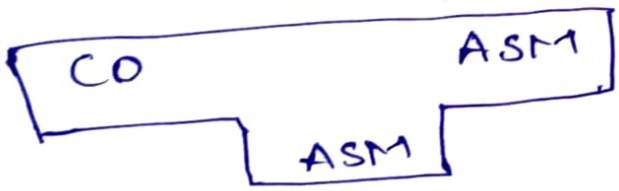
Example:

We can create compiler of many different forms. Now we will generate:



Compiler which takes C language and generates an assembly language as an output with the availability of a machine of assembly language.

- Step-1: First we write a compiler for a small of C in assembly language

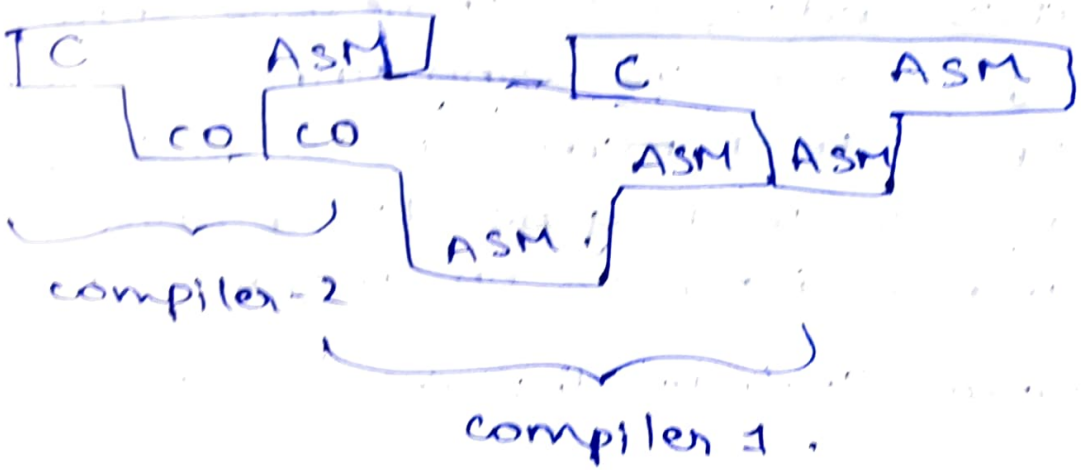


→ compiler 1

- step-2: Then using with small subset of C i.e., CO, for the source language c the compiler is written.

→ compiler-2

- step-3: Finally we complete the second compiler. using compiler 1 the compiler 2 is compiled.



- step 4:- Thus we get a compiler written in ASM which compiles C and generates code in ASM.