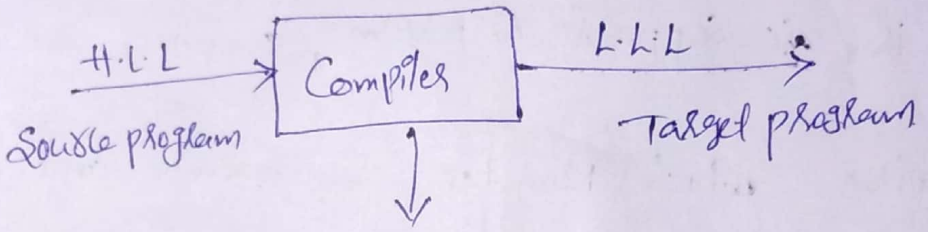
Introduction to Compiler Design:

Over view:-

Compiler: A compiler is a program that can Read a program in one language - the Source language - and Translate it into an equivalent program in another language - the target language.

* An important Role of the compiler is to Report any errors in the source program that detects during the Translation process.



Why should we study compiler Design?

Compilers are every where, Many applications of compiler Technology.

1, Machine code Generation for high level languages

2, Software Testing

3, program optimization

4. Malicious code detection

A compiler is a possibly the most complex system software.

System Software: Set of programs are to be used to perform a particular task.

Ex: operating system.

Application software: Set of applications which can be used to perform required task.

Ex: MS Excel, MS word.

List of compilers: • C compiler, • C++, • ALGOL, • BASIC, • COBOL, • Java.

Depending on how they have been constructed (or) on what function they are supposed to perform, compilers are classified as.

cross-compiler: which runs on one machine and generates a code for another machine.

Incremental-compiler: that allows a modified portion of a program to be recompiled.

Ideal compiler: * Take less time for compilation
* be smaller in size (it is not suitable for large code)
* be written in high level languages.

Programming languages:

* P.L; it is used to communication between the instructions.

P.L classified into 3 types.

1, low level (or) Machine level

2, high level

3, Assembly language (mnemonic code)

Ex: ADD, SUB, DIV, MUL.

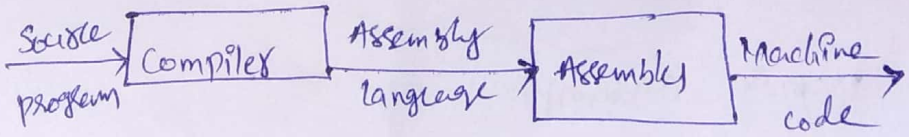
Note: Machine language and Assembly languages are example of low level language.

Translators: Translator is, convert one form of program into another form.

* Compilers
* Assembles
* Interpreter.

* Assembles converts, Assembly language program to machine level language.

Assembles:



* An Assembly code is a mnemonic version of machine code. The Typical Assembly instructions are as given below.

```
MOV  a, R,
MUL  #5, R,
ADD  R,, b.
```

The Assembler converts these instructions in the binary language which can be understood by the machine. Such a binary code is often called as machine code. This machine code is a Relocatable machine code that can be passed directly to the loader / linker for execution purpose.

The Assembler converts the assembly program to low level machine language using Two Passes.

** Note: A pass means one complete scan of the input program. The end of second pass is the Relocatable machine code.

Interpreter: An interpreter is a kind of translator. The source program gets interpreted every time it is to be executed, and every time the source program is analysed. Hence interpretation is less efficient than compiler.

* The interpreters do not produce object code.

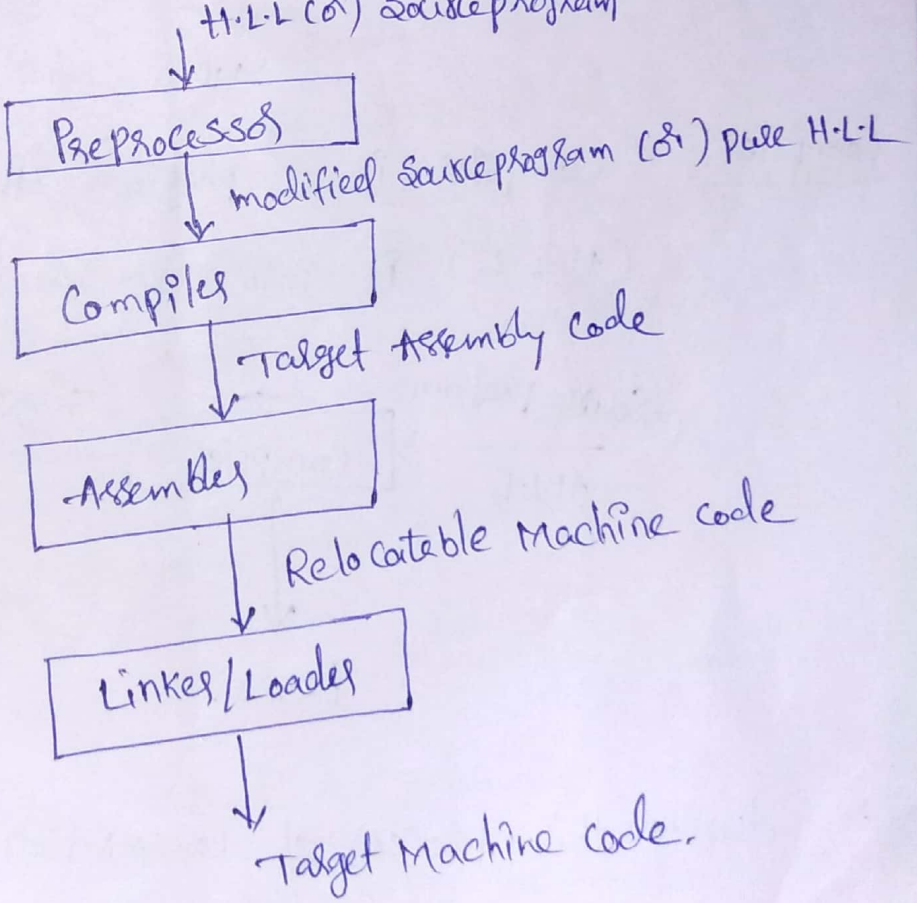
* Memory consumption is more.

Language Processing system:

→ We have learn that any computer system is made of Hardware and software. Hardware is just a piece of Mechanical devices and it's functions are being controlled by a compatible software.

→ Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming.

→ Hardware understands a language, which humans can not understand, so we write programs in high-level languages which is easier for us to understand and Remember.



Preprocessor: It is a part of a compiler, that produces input for a compiler.

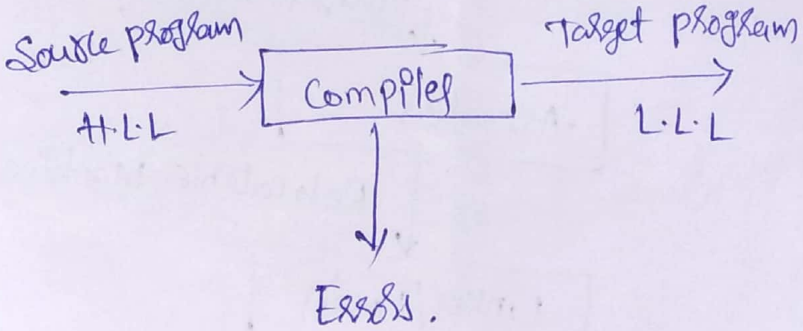
→ It deals with 2 Types of functions **File Inclusion** **Macro processing.**

File Inclusion: A preprocessor may include header files into the program text

Macro-processing: Macro is a small set of instructions, while we are Running a program, whenever macro name is identified then that name is Replaced by "macro definition"

Ex: #define PI 3.14 → Macro definition
↓
Macro.

Compiler: Compiler is a program, that Translate one language (H.L.L) it into Equivalent Target language (L.L.L)



* during this process of conversion, it return Errors.
—these are handled by the Error handler.

Assemblers: For every platform (Hardware + O.S) we are having an Assembler. These are not universal, since for each platform we have one.

→ It converts an Assembly language into machine code i.e, binary Representation.

→ the o/p of an Assembler is called an object file, which conteins a combination of machine instructions as well as the data Required to place these instructions in memory.

Loaders: Loader is a part of operating system and it is Responsible for loading executable files into memory and execute them. It calculates size of the program and creates memory space for it.

Linker: Linking, it is the proces of putting togather ollies programs files and functions that are Required by the program.

→ Linker Resolves External memory Addresses.

Ex: if the program is using exp() function, then the object code of this function should be braught from the "math library" of the system and linked to the main program.

Goals of Compiles:

→ It maintains the meaning of the program that is compiled

→ Compiles must improve the performance of i/p programs i.e, speed of the programs.

→ Reduce the time required for compilation so, as to support Rapid development and debugging cycles.

→ Gives Accurate Error message.

Differences between Compiles and Interpreter:

Compiles	Interpreter
→ Compiles checks (or) scans the entire h.l. program at once.	→ Interpreter Translates one statement at a time.
→ The Compilers produce object code	→ The Interpreters do not produce object code.
→ The Compiles is a complex program and it Requires large amount of memory.	→ Interpreters are simples and give us improved debugging environment.
→ Processing time is less	→ Processing time is more.
→ Some Compilation languages are C, C++, FORTRAN, PASCAL, Ada etc. (it is also called s/w Translation)	→ It is also called as software simulation. Some Interpretation languages are ML, LISP, PHython, Prolog etc.

Phases of compilation:-

There are Two parts of compilation

1, Analysis part

2, Synthesis part.

Analysis part consists of 3 phases.

1, Lexical Analysis phase

2, Syntax Analysis phase

3, Semantic Analysis phase.

The synthesis part consists of 3 phases.

4, Intermediate code Generation

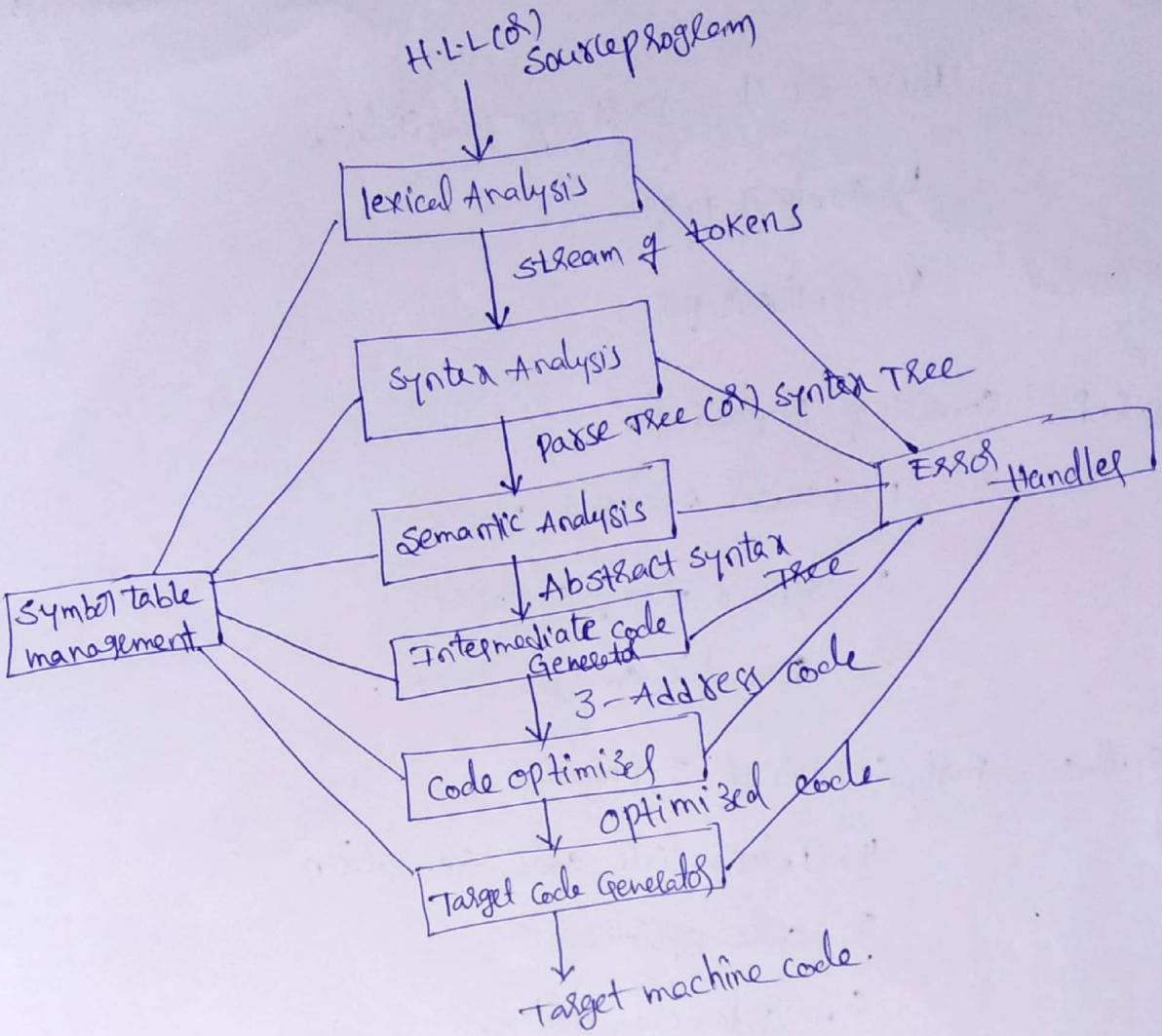
5, Code optimization

6, Code Generation.

* Analysis part, it is also called as Front end of a compiler.
* synthesis part, it is also called as Back end of a compiler.

→ Analysis phase breaks up the source program into constituted pieces and imposes a grammatical structure on them. It uses this structure to create an Intermediate Representation of the source program.

→ Synthesis phase constructs the desired target program from the Intermediate Representation and the Information in the Symbol Table.



Lexical Analysis Phase:

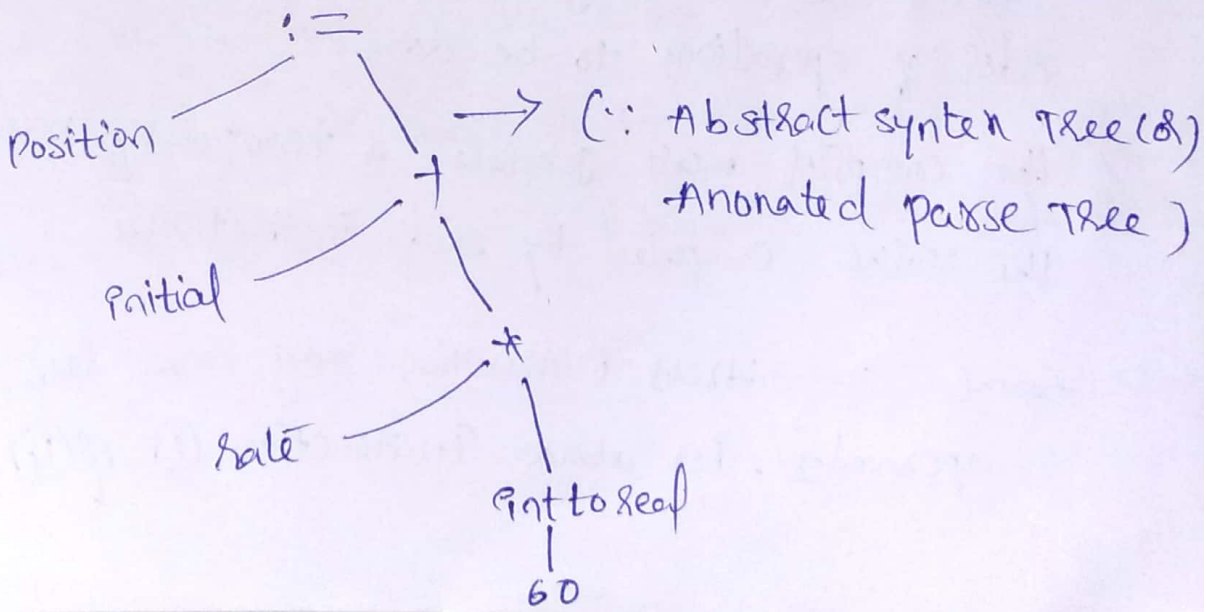
* Lexical Analysis phase is also called scanner.
* This phase Reads the characters in the source Program and groups them into a stream of tokens.
* token is a sequence of character (or) string having a collective meaning.
* The character sequence forming a token is called the lexeme for that token.

Semantic Analysis phase:

* Semantic Analysis checks the source program for semantic errors. (matching if-else stmt, missing parenthesis, Typechecking).
* It uses the hierarchical structure determined by the syntax analysis phase to identify the operators and operands of expressions and statements.
* Semantic analysis performs Type checking. i.e, it checks that each operator has operands that are permitted by the source language specifications.

Ex: if a real number is used to index an array i.e, A[1.5] then the compiler will report an error. This error is handled during Semantic Analysis.

Suppose it is assumed that all identifiers (id_1, id_2, id_3) have been declared to be real. But 60 assumes itself to be an integer. when * is applied to a real rate and integer 60, integer is converted into real during the type checking process. This is achieved from the operator inttoreal, which converts an integer into real.



Intermediate Code Generation:

* After syntax and semantic analysis compilers generate an intermediate representation of the source program. This representation should be easy to produce and easy to translate into the target program.

* Thus the intermediate code Generation phase transforms the parse tree into an intermediate-language representation of the source program.

One popular type of intermediate language is called as "Three-address" code which is like the assembly language. Three-address code for the statement

position := initial + rate * 60 is given as.

$temp_1 := intToReal(60)$

$temp_2 := id_3 * temp_1$

$temp_3 := id_2 + temp_2$

$id_1 := temp_3$

3-Address code has several properties. They are,

→ Each 3-Address instruction has atmost one operator in addition to the assignment. compiler has to decide the order of operations to be done.

→ The compiler must generate a temporary name to hold the value computed by each instruction.

→ Some 3-Address instruction may have less than ~~3 opera~~ 3-operands. In above instruction (1) & (4).

Consider An Assignment statement

Position := Initial + Rate * 60

- this would be grouped into the following tokens.

1, the identifier position
2, the Assignment symbol :=
3, the identifier initial
4, the plus sign
5, the identifier Rate
6, the multiplication sign
7, the number.

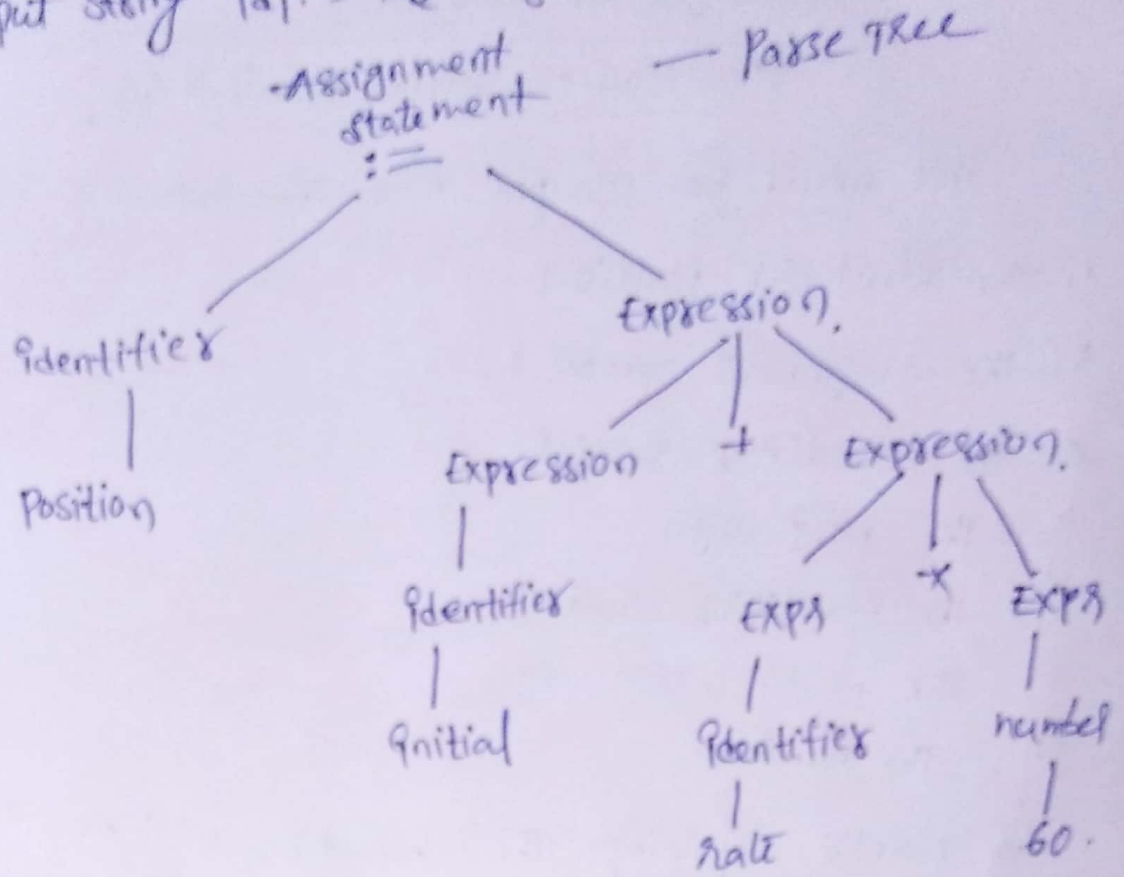
The blanks separating the characters will be eliminated during lexical analysis. when the lexical Analyser finds the identifier position, it generates a token, say id. the identifiers position, Rate and initial are the lexemes. the statement after lexical analysis is given by.

$id_1 := id_2 + id_3 * 60$ (i: id_1, id_2, id_3 are tokens for initial, position, Rate).

Syntax Analysis Phase:

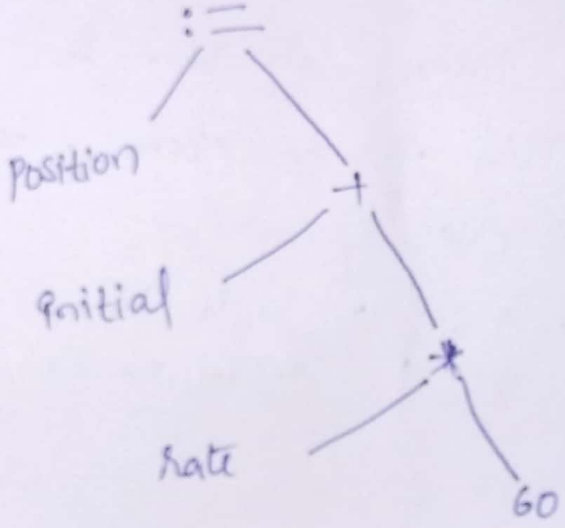
* the syntax Analysis phase is also called Parser.
* in this phase, the tokens generated by the lexical analyser are grouped together to form a hierarchical structure called 'Parse Tree'.

For the input string $id_1 := id_2 + id_3 * 60$



— Parse Tree

Syntax Tree:



* → A syntax tree is a compressed Representation of the parse tree. In which, operators appear as the interior nodes and operands of an operator are the children of node for that operator.

Code optimization: Code optimization phase improves the intermediate code i.e, it reduces the code by removing the repeated (or) unwanted instructions from the intermediate code. The above given intermediate code can be optimised to the following code.

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Target code Generation: Code generation phase converts the intermediate code into a target code consisting of sequenced machine code (or) Assembly code that performs the same task. The above optimized code can be written using Registers R1 and R2 which is as given below.

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

Here # signifies that 60.0 is treated as constant.

f is a floating point number.

Symbol Table management: It collects the information about "data objects" that appear in source program

Ex: whether the variable Represents int, float, etc

attributes may provide information about the storage allocated for a name, its type, its scope.

* The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

Error handler: In compilation, each phase detects errors, these errors are must be reported to error handlers to handle the errors so that the compilation can proceed.

LEX: LEX is a unix utility which generates the lexical Analyser. A LEX lexer is very much faster in finding the tokens as compared to the handwritten LEX program in C. The LEX program consists of 3 parts

1. Declaration section 2. Rule section 3. Procedure section.

%{
Declaration section → declaration of variable constants can be done.
%}
%%
Rule section → consists of Regular Expression with associated actions.
%%
Auxiliary procedure section. → in which all the required procedures are defined.

* Sometimes these procedures are required by the Actions in the Rule section.

Describe the output for the various phases of compiler with Respect to following statements. (7)

Position := Initial + Rate * 60

Sol: The o/p at each phase during compilation is as given below.

Position := Initial + Rate * 60

↓

Lexical Analysis

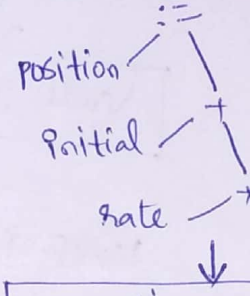
↓

$id_1 := id_2 + id_3 * 60$.

↓

Syntax Analysis

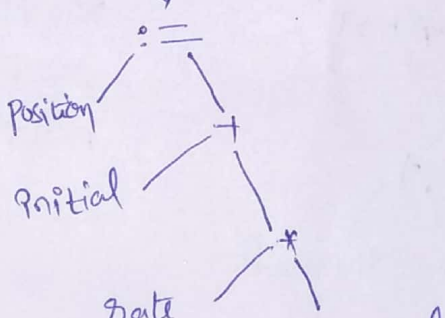
↓



↓

Semantic Analysis

↓



1	Position	....
2	Initial	....
3	Rate	....

symbol table

↓

Intermediate code Generator

↓

$temp_1 := Int\ to\ Real(60)$

$temp_2 := id_3 * temp_1$

$temp_3 := id_2 + t_2$

$id_1 := temp_3$

↓

Code optimizer

↓

$temp_1 := id_3 * 60.0$

$id_1 := id_2 + temp_1$

↓

Code Generator

↓

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

(Machine code)

2. write o/p at all phases of the compiler for the below statement.

$i = i * 70 + j + 2$.

↓

lexical Analysis

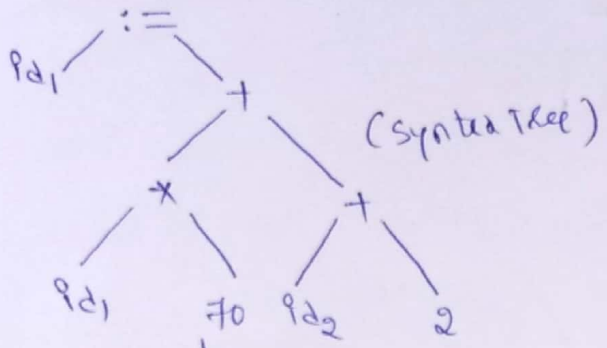
↓

$id_1 := id_1 * 70 + id_2 + 2$ (tokens)

↓

syntax analysis

↓

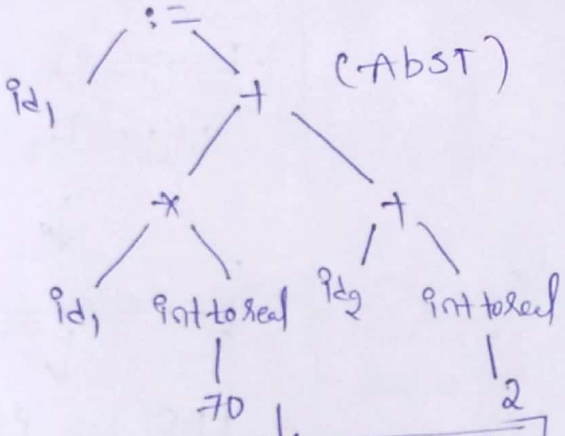


(Syntax Tree)

↓

Semantic Analysis

↓



(ABST)

↓

Intermediate Code Generation

↓

$temp_1 := int\ to\ real(70)$

$temp_2 := id_1 * temp_1$ (3-Address code)

$temp_3 := int\ to\ real(2)$

$temp_4 := id_2 + temp_3$

$temp_5 := temp_2 + temp_4$

$id_1 := temp_5$

↓

code optimisation

↓

$temp_1 := id_1 * 60.0$

$temp_2 := id_2 + 2.0$

$temp_3 := temp_1 + temp_2$

$id_1 := temp_3$ (optimized code).

↓

code generator

↓

```
MOVF id1, R1
MULF #60.0, R1
MOVF id2, R2
ADDF #2.0, R2
ADDF R2, R1
MOVF R1, id1.
```

(Machine code)

show the o/p generated by each phase for the following expression.

(19)

$x := (a+b) * (c+d)$.

↓

Lexical Analysis

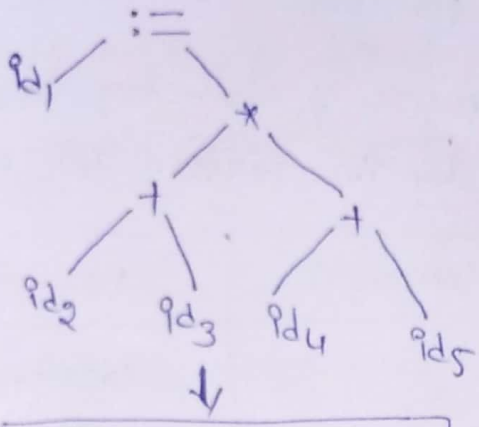
↓

$id_1 := (id_2 + id_3) * (id_4 + id_5)$

↓

Syntax Analysis

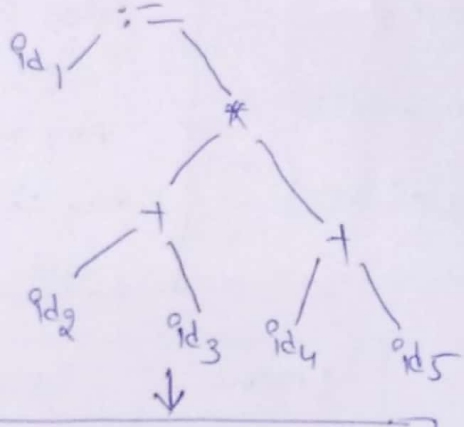
↓



↓

Semantic Analysis

↓



↓

Intermediate code Generator

↓

$temp_1 := id_2 + id_3$

$temp_2 := id_4 + id_5$

$temp_3 := temp_1 * temp_2$

$id_1 := temp_3$.

↓

Code Generation

↓

```
MOV  id2, R0
ADD  id3, R0
MOV  id4, R1
ADD  id5, R1
MUL  R1, R0
MOV  R0, id3.
```

(Machine code)

Token, pattern, Lexeme.

Token: it describes the stream of characters having a collective meaning. Ex: identifiers, keywords, constants, operators.

Pattern: Set of Rules that describe the token. 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: Lexeme is a sequence of characters in the sourceprogram that is matched by the pattern for a token.

Ex: 1

Token	Sample lexemes	Description of the pattern
const	const	constant
if	if	if
Relation	<, <=, =, <>, >, >=	< (or) <= or = or <> or > or >=
id	pi, count, D2	Letter followed by letters (or) digits.
num	3.1416, 0, 6.02E23	Any numeric constant
literal	"language processor"	Any characters enclosed b/w " and ".

Ex: 2
```
int MAX (int a, int b)
{
if (a>b)
return a;
else
return b;
}
```

(*: the blank and new line characters can be ignored).

Lexeme	token.
int	keyword
MAX	identifier
(	operator
int	keyword
a	identifier
,	operator
⋮	⋮

Role of lexical Analyser (or) Need for lexical analyser:

* lexical Analyser it is first phase of compiler, the lexical Analyser reads the characters from source program from left to right one character at a time and generates the sequence of tokens.

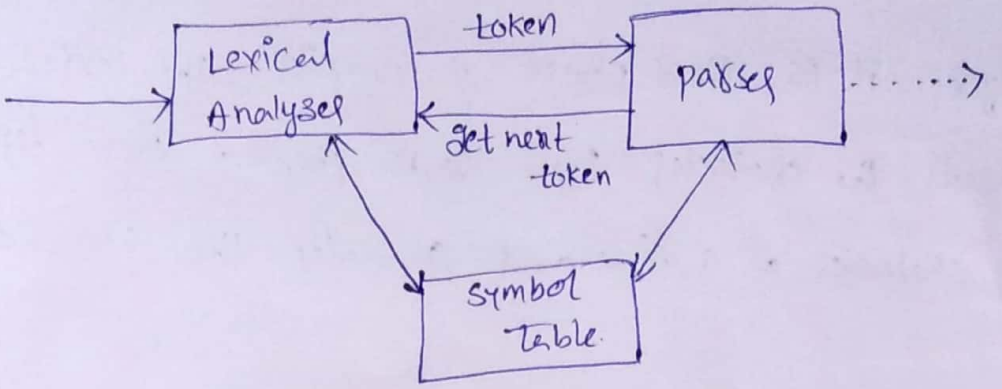
* Each token is a single logical cohesive unit such as identifier, keyword, operators and punctuation marks.

* and it also count, to tells the howmany Tokens are generated from Given source program.

Ex:
```
int max(x,y)
    int x,y;
      /* find max of x and y*/
    {
     return (x>y ? x:y);
    }
```
total 25 tokens

Ex: `printf("%d Hai", &x);` — 8 tokens

* LEX, which is an automated tool it is used to generate lexical Analyser.



* Lexical Analyser Reads the characters from source program, line by line.
* Lexical Analyser use the symbol table to produce tokens into parser.
* upon Receiving a "get next token" command from the parser, the lexical analyser Reads i/p characters until it can identify the next token.

Functions of lexical Analyser:

1, it produces stream of tokens
2, it eliminates comments, and white space which are in the form of blank, tab and new line characters from the source program.
3, it generates a symbol table which stores the information about identifiers, constants encountered in the i/p.
4, it keep track of line numbers.

5, It Reports the errors encountered while generating the tokens.

Lexical Errors: These type of errors can be detected during lexical analysis phase. Typical lexical phase errors are

1, Exceeding length of identifier (or) numeric constants

2, Appearance of illegal character (Ex: print("Helloworld");$)

3, unmatched string. (Ex: comment */ and of the comment is present, but beginning of comment is not)

* if the string ff is encountered in 'c' program i.e,

```
fi(a==f(n))...
```

Lexical Analyser can not tell whether fi is a misspelling of the keyword if (or) an undeclared function identifier.

because, None of the patterns for tokens matches a prefix of the Remaing to Input. So, This type of errors are solved by Error Recovery action (or) Error Recovery Solution.

Panic mode Recovery:

* Deleting successive character — printff
* Deleting missing character — Psinf → printf
* Replacing incorrect characters by correct characters → Prinff → Printf
* Transposing Two adjacent characters — rpintf ↔ Printf.

Attributes For Tokens:

The tokens and associated attribute values for the statement

F = M * C ** 2.

<id, pointer to symbol-table entry for F>

<assign_op>

<id, pointer to symbol-table entry for M>

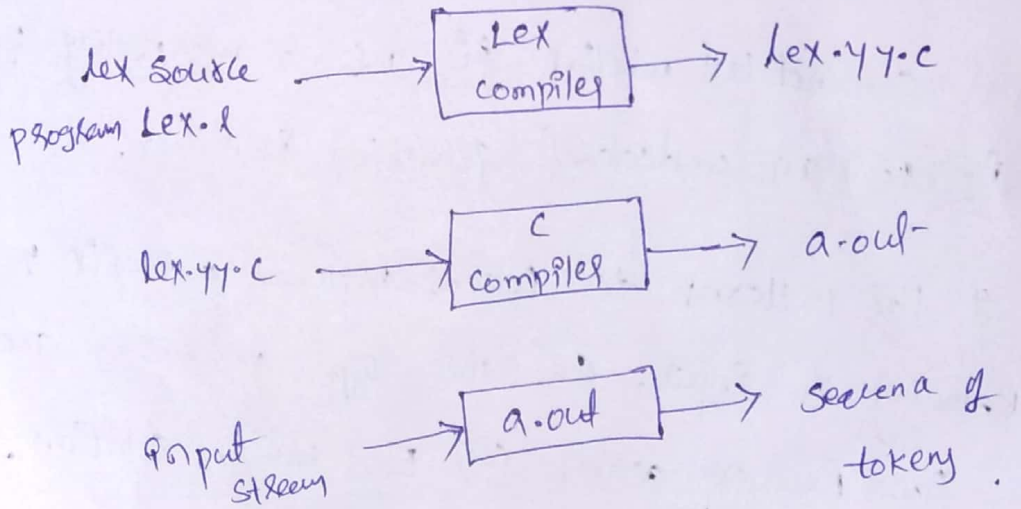
<mult_op>

<id, pointer to symbol-table entry for C>

<exp_op>

<num, integer value 2>.

Lexical Analyser Generator:

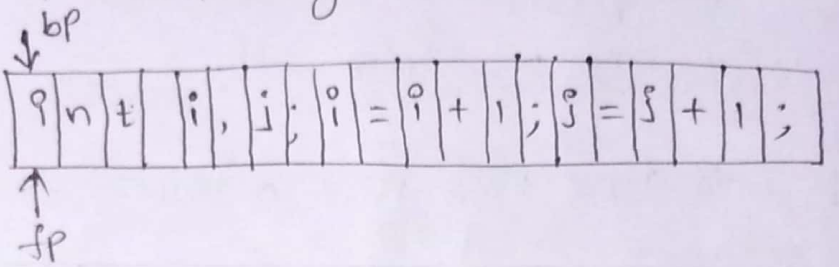


Input Buffering:

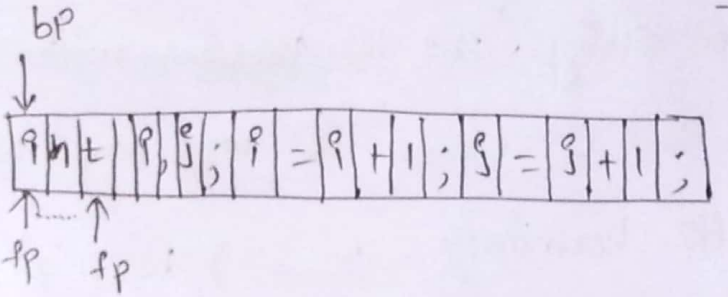
* The lexical analyser scans the i/p string from left to right one character at a time. it uses two pointers.

1. Lexeme begin_ptr
2. Forward_ptr.

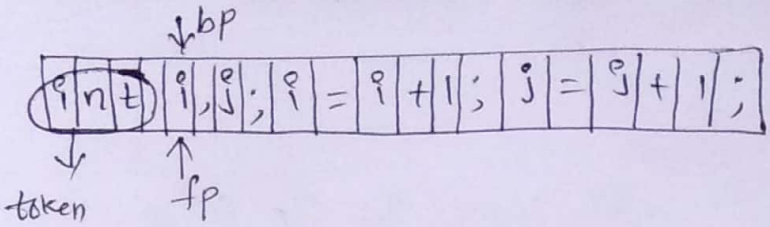
* These are the pointers to keep track of the position of the i/p scanned. Initially both the pointers point to the first character of the i/p string as shown below.



* The forward_ptr moves a head to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. in above as soon as forward_ptr (fp) encounters a blank space the lexeme "int" is identified.



* the fp will be moved a head at white space. when fp encounters white space, it ignore and moves a head. then both the begin_ptr (bp) and forward_ptr (fp) are set next token i.



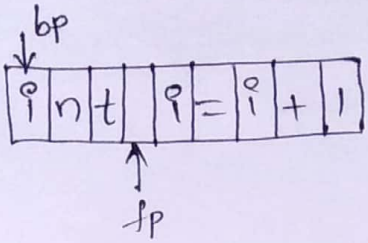
* The i/p character is then read from secondary storage. but Reading in this way from secondary storage is costly. (System calls). i.e, if the program size is larger one it needs more system calls. to read the tokens from memory we can overcome this problem with the help of buffering technique

buffering: A block of characters to be read into the buffer using only one system call.

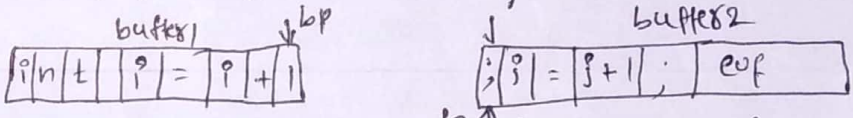
→ A block of Data First Read Into a Buffer, and then scanned by lexical analyser. There are Two methods used in this context

1, one buffer scheme
2, Two buffer scheme.

one buffer scheme: In this, only one buffer is used to store the i/p string. But the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan Rest of the lexeme the buffer has to be refilled, that makes overwriting the First part of lexeme.



Two Buffer scheme: To overcome the problem of one buffer scheme, in this method two buffers are used to store the i/p string. the first buffer and second buffer are scanned alternately. when end of current buffer is Reached the other buffer is filled. to identify the boundary of first buffer end of file (eof) characters should be placed at the end of first buffer. Similarly in second buffer also.



* This eof characters introduced at the end is called sentinel which is used to identify the end of buffer.

Code:

```
if(fp==eof(buff1))  /* encounters end of first buffer */
{
    /* Refill buffer2 */
fp++
}
else if (fp==eof(buff2) /* encounters end of second buffer */
{
    /* Refill buffer1 */
fp++
}
else if(fp==eof(input))
return; /* terminate scanning */
```

```
else
fp++;
/* Still Remaining i/p has
   to scanned */
```

(8)

```
switch (*forward++) {
    case eof:
        if (forward is at end of first buffer)
        {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer)
        {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    cases for the other characters
}
```

Specification of Tokens: (Regular Grammar and Regular Expression)

(strings, languages and operations on languages)

String:- (Sequence of characters):

* A string over an alphabet set Σ is a finite sequence of symbols from Σ, denoted by w.

Examples: 1, $\Sigma = \{a, b, c\}$ then 2, $\Sigma = \{0, 1\}$ then

a, b, c, abc, bca, cab, etc are strings. 1, 0, 11, 10, 01, ...

operations on strings:

|w| or |w|

The length of string s, usually written |s|, is the number of occurences of symbols in s.

Ex: Rama krishna

$|w| = 10$.

The empty string can be denoted as 'ε' & 'λ'. ({ε} A string that consists of zero symbols).

Prefix: Prefix of string 's' is any string obtained by Removing zero (or) more symbols from the end of 's'. ie, Removing leading symbols of that string.

Ex: banana → ban and ε are prefixes of banana

suffix: Suffix of a string 's' is any string obtained by Removing zero (or) more symbols from begining of 's'. ie, Removing tailing symbols of that string.

Ex: banana, → nana, ε are suffixes of banana

Substring: A substring of 's' obtained by deleting any prefix and any suffix from s. Removing prefix and suffix.

Ex: banana → nan

Subsequence: A subsequence of s is any string formed by deleting zero (or) more not necessarily consecutive positions of 's'

Ex:- Banana → bean is a subsequence of banana.

Concatination: If x and y are strings, then concatination of x and y, denoted by xy.

Ex: x=010, y=1

z=0101

Identity Element: the empty string is the identity under concatination. that is, for any string s, $\epsilon s = s\epsilon = s$.

Language: A language is a set of strings of symbols from some alphabet languages like ϕ, a empty set ϵ are also examples

Ex: $\Sigma = \{a\}$

$\Sigma^* = \{\epsilon, a, aa, aaa, \ldots\}$

↳ language.

Ex: $\{0,1\} = \Sigma$

$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, \ldots\}$

operations on language:-

L_1, L_2 are 2 languages the union denoted $L_1 \cup L_2$ is a language containing all string from both languages

$L_1 \cup L_2 = \{w / w$ in L_1 or w is in $L_2\}$

Example:

Ex-1 Let $L_1 = \{0, 01, 10, 11\}$

$L_2 = \{0, 01, 001\}$

$\boxed{L_1 \cup L_2 = \{0, 01, 10, 11, 001\}}$

Ex-2 $L_1 = \{0, 00, 000 \ldots\}$ and $L_2 = \{0, 00, 000 \ldots\}$

$\boxed{L_1 \cup L_2 = \{0\}^*}$

Concatenation: $L_1 L_2$ denoted by $L_1 L_2$ is a language containing the strings from L_1 followed by strings from L_2.

i.e. $L_1 L_2 = \{w_1 w_2 / w_1$ is in L_1 and w_2 is in $L_2\}$

Ex: $L_1 = \{a, ab, bb, b\}$

$L_2 = \{bb\}$

$L_1 L_2 = \{abb, abbb, babb, bbb\}$

Kleene closure: Language L is denoted as L^*, is a language containing all the strings obtained by concatinating zero(0) or more strings from L.

$$\boxed{L^* = \bigcup_{i=0}^{\infty} L^i}$$

Positive closure (L^+): it is denoted as L^+, is the same as the kleene closure, but without the term L^0. that is 'ϵ' will not be in L^+ unless it is in L itself.

L^4 : it is the set of all 4-letter strings.

Operation	Definition:
union	$L_1 \cup L_2 = \{s \mid s \in L_1 \text{ or } s \in L_2\}$
Concatenation	$L_1 L_2 = \{sp \mid s \in L_1 \text{ and } s \in L_2\}$
Kleene closure	$L^* = \bigcup_{i=0}^{\infty} L^i$
positive closure	$L^+ = \bigcup_{i=\phi}^{\infty} L^i$.

Regular Expression:- the language accepted by ~~FA~~ are easily describe by simple ~~Expressions~~ called Regular Expressions. R.E useful for Representing certain sets of strings in an algebraic cost. ie, we are able to describe identities by giving names to sets of letters and digits and using the language operators union, Concatenation, and closure.

Let Σ be an alphabet the Regular Expression over Σ and the sets they denote are defined as follows.

- ϕ is a Regular expression and denotes empty set
- ε is a regular Expression and denotes the set $\{\varepsilon\}$, ie, the set containing empty string.

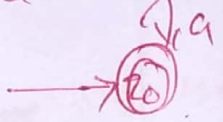
* the union of TWO R.E R_1 and R_2 Return as $R_1 + R_2$ is also in R.E
* the concatination of TWO R.E R_1 and R_2 Return as $R_1 \cdot R_2$ is also in R.E

* If R is an R.E then R^* is also R.E.

Regular Set:- Any set Represented by R.E is called as "Regular Set". It is a set of strings from which ∃ some F.A which accepts the set it is Represented by '{}'.

Ex: Regular Set = $\{\epsilon, a, aa, aaa, \dots\}$

R.E = a^*

F.A = 

R.E	Regular Set
a	{a}
a+b	{a,b}
ab	{a,b}
a*	{ε,a,aa,aaa,...}
(a+b)*	{a,b}*

Regular Set	R.E
{101}	101
{abba}	abba
{01,10}	01+10
{ε,ab}	ε+ab
{abb,a,b,bba}	abb+a+b+bba
{ε,0,00,000...}	0*
{1,11,111...}	1+
{ε,a,aa,ab,ba,bb...}	(a+b)*

Problems on R.E:

* all the strings of 0's and 1's $= \{\epsilon, 0, 1, 10, 11, 01, 00 \ldots\} = (0+1)^*$
* ending with 00 $= (0+1)^*00$
* begins with 0 ending with 1 $= 0(0+1)^*1$
* Set of all strings having even no of 1's $= \{\epsilon, 11, 1111, \ldots\} = (11)^*$
* odd no of 1's $= 1(11)^*$
* strings of 0's and 1's with at least 2 consecutive 0's

$$= (1+0)^*00(1+0)^*$$

* Set of all strings with 1100 as substring

$$(0+1)^*1100(0+1)^*$$

* Set of all strings with one (or) more 0's followed by 1

$$0^+1.$$

Algebraic laws for R.E:

These laws are used to identify the base on which the expression can be modify we apply the following Associative Rels

1. union operation on R.E are compitative

   i.e, $R+S = S+R$

2. union is Associative

$$(R+S)+t = R+(S+t)$$

3. Concatination operations are Associative

$$(R.S).t = R.(S.t)$$

4) Concatination is Right distributive over union

$(R+S)t = Rt+St$

5) Concatination is left distributive over union

$t(R+S) = tR+tS.$

6) $\phi^* = \epsilon.$

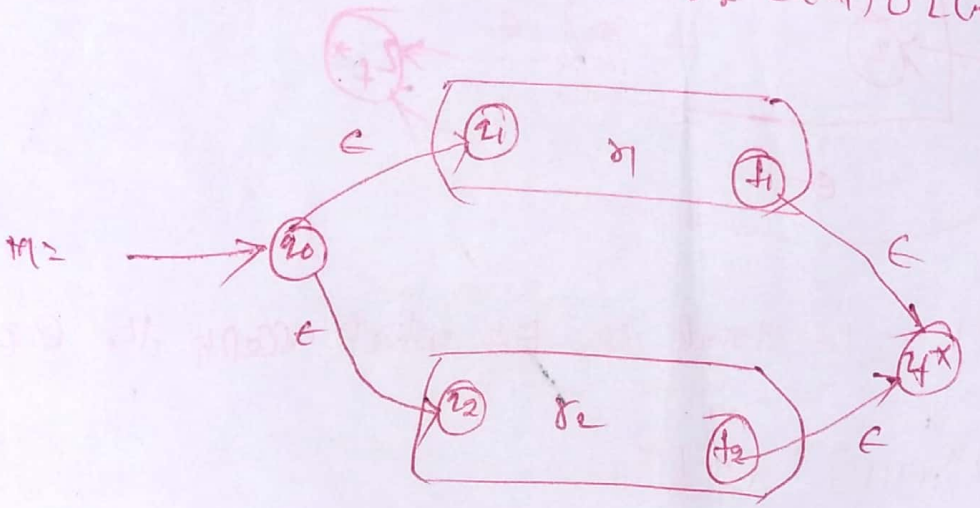
Operation on R.E: (union)

Let $\gamma_1 \& \gamma_2$ be Two R.E's with less than i operation that of FA's $m_1 \& m_2$ given by

$m_1 = \{Q_1, \Sigma_1, \delta_1, z_1, \{f_1\}\}$ and

$m_2 = \{Q_2, \Sigma_2, \delta_2, z_2, \{f_2\}\}$

To show that R formed using union operation of $\gamma = \gamma_1 + \gamma_2$ of automate on M. which accepts the langu $L(m_1) \cup L(m_2)$

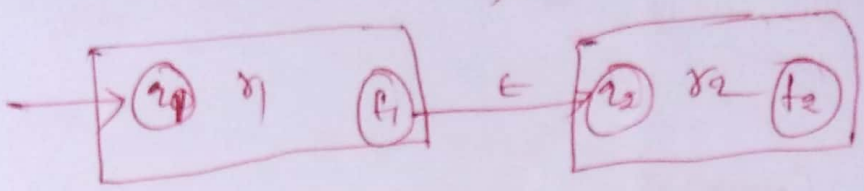


Concatination: Let γ_1, γ_2 be Two R.E's with less than i operation that of F.A's $m_1 \& m_2$ given by

$m_1 = \{Q_1, \Sigma_1, \delta_1, z_1, \{f_1\}\}$

$m_2 = \{Q_2, \Sigma_2, \delta_2, z_2, \{f_2\}\}$

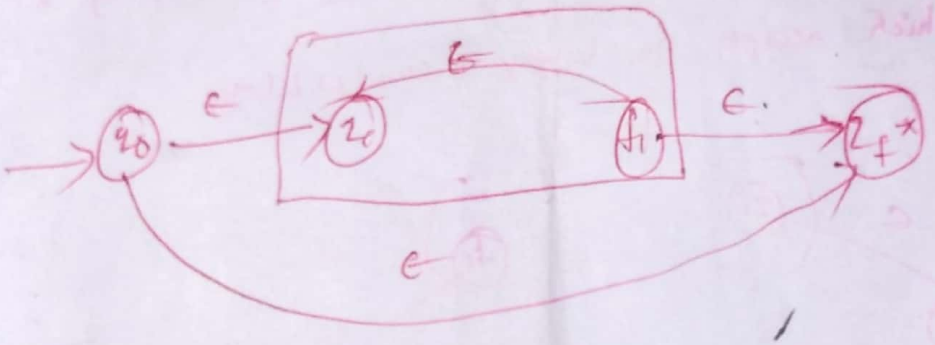
To show that γ found using concatenation operation as $\gamma = \gamma_1 \cdot \gamma_2$ as automata on M. which accepts the language $L(M_1) \cup L(M_2)$



Closure operation: let γ_1 be the R.E but less than γ_1 operation that of the finite automata m_1 given by

$m_1 = \{Q, \Sigma, \delta, q_1, f\}$

to show that γ found using closure operation as $\gamma = \gamma_1^*$ as automata m which accepts the langue $L(m_1^*)$



Ex: 1, $10 + (0+11)0^*1$ — Construct the FA which accepts the R.E

2, $\gamma = 01[[(10)^* + 111]^* + 0]^*1$

3, $(a+b)^* abb$

* Construct NFA with ϵ to DFA.

Components of Regular Expressions:

x → the character

. → any character usually except a new line.

R^* → 0 (or) more occurrences

R^+ → one (or) more occurrences

R_1R_2 → R_1 followed by R_2

R_2R_1

R4 → the set of + occurrences of strings.

Regular definition: If notation would continue, we may wish to give names to certain Regular Expression and use those names in subsequent Expressions, as the names were symbols may.

If Σ is an alphabet of basic symbol then a regular definition is a sequence of the form.

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\vdots$$
$$d_n \rightarrow r_n.$$

where

Each d_i is a new symbol, not in Σ and not the same as any other of the d_i's

Each r_i is a R.E over the alphabet $\Sigma \cup \{d_1, d_2, \ldots d_{i-1}\}$

Ex: 'C' identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of 'C' identifiers.

We shall conventionally use Italics for the symbols defined in R.D

letter → A|B|....|Z|a|b|...|z|

digit → 0|1|...|9

id → letter (letter|digit)*

Ex: Unsigned numbers (integer & floating point) are strings such as 5280, 0.01234, 6.336E4 (or) 1.89E-4

digit → 0|1|...|9

digits → digit digit*

optionalFraction → .digits | ε — empty string

optionalExponent → (E(+|-|ε) digits) | ε — empty string

number → digits optionalFraction optionalExponent.

Using shorthand, we can Rewrite the Regular definitions

letter → [A-Za-z]

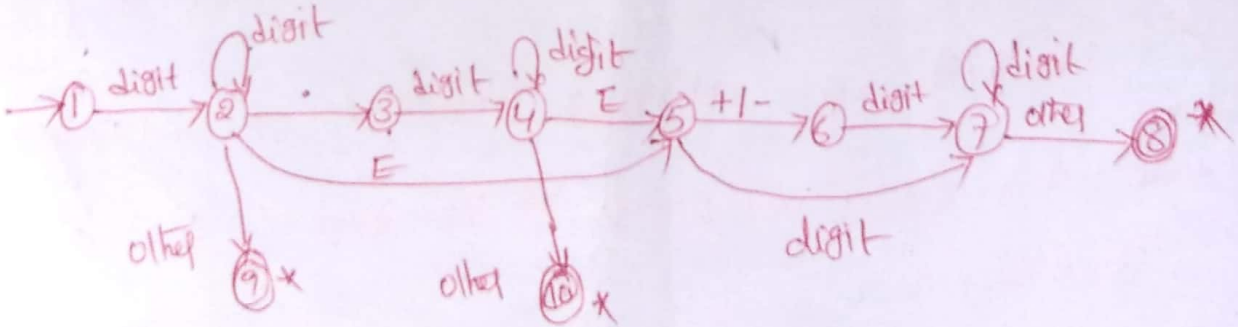
digit → [0-9]

id → letter (letter|digit)*

digits → digit+

number → digits (.digits)? (E[+-]? digits)?

Transition Diagram:



Recognition of Tokens:

* For a programming language there are various types of tokens such as Identifiers, keywords, constants and operators and so on. The token is usually Represented by a token type and token value.

Token Type	Token Value.

* The token type tells us the category of token and token value gives us the information Regarding token. The token value is also called token attribute. During lexical analysis process the symbol table is maintained. The token value can be a pointer to symbol table in case of identifier and constants. The lexical analyser reads the input program and generates a symbol table for tokens.

Ex: We will consider some encoding of tokens as follows.

Token	code	value
if	1	-
else	2	-
while	3	-
for	4	-
identifier	5	Ptr to symbol Table
constant	6	Ptr to symbol table
<	7	1
<=	7	2
>	7	3
>=	7	3
<>	7	5

(	8	1
)	8	2
+	9	1
-	9	2
=	10	-

our lexical Analyser will generate Following token stream.

```
if (a<10)
    i = i+2;
else
    i = i-2;
```

1, (8,1), (5,100), (7,1), (6,105), (8,2), (5,107), 10, (5,107), (9,1), (6,110), 2, (5,107), (5,107), (9,2), (6,110).

The corresponding symbol table for identifiers and constants will be,

Location counter	Type	Value
100	identifier	a
105	constant	10
107	identifier	i
110	constant	2

Transition Diagram for Recognition of token:

Lexical Analyser to scan the i/p string, and search for where the lexeme end.

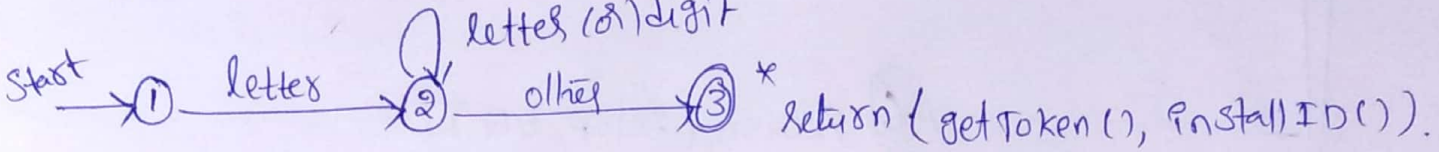
* Recognition of identifiers
* Recognition of delimiter
* " " keyword
* " " operators
* " " Number.

Recognition of Identifiers:

letter → [A-Z a-z]

digit → [0-9]

digits → digit+



Recognition of Delimiters:

WS → (blank | tab | newline)+

WS → Delimiter+

Recognition of Keywords:

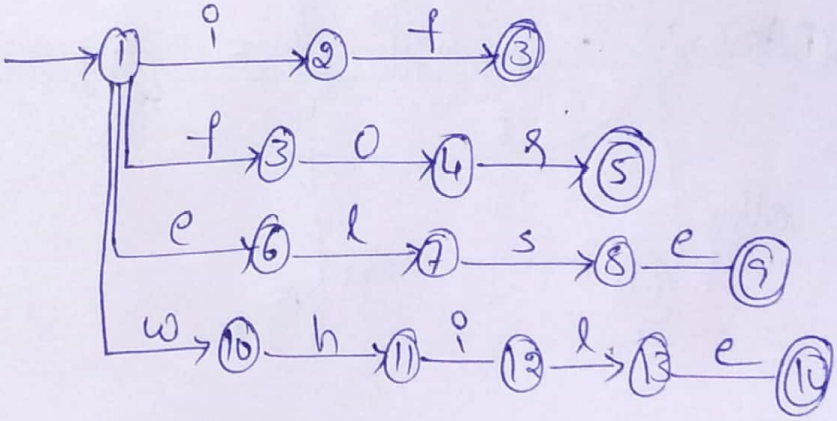
Keywords are Reserved words, these values can not change during compilation.

if → if

then → then

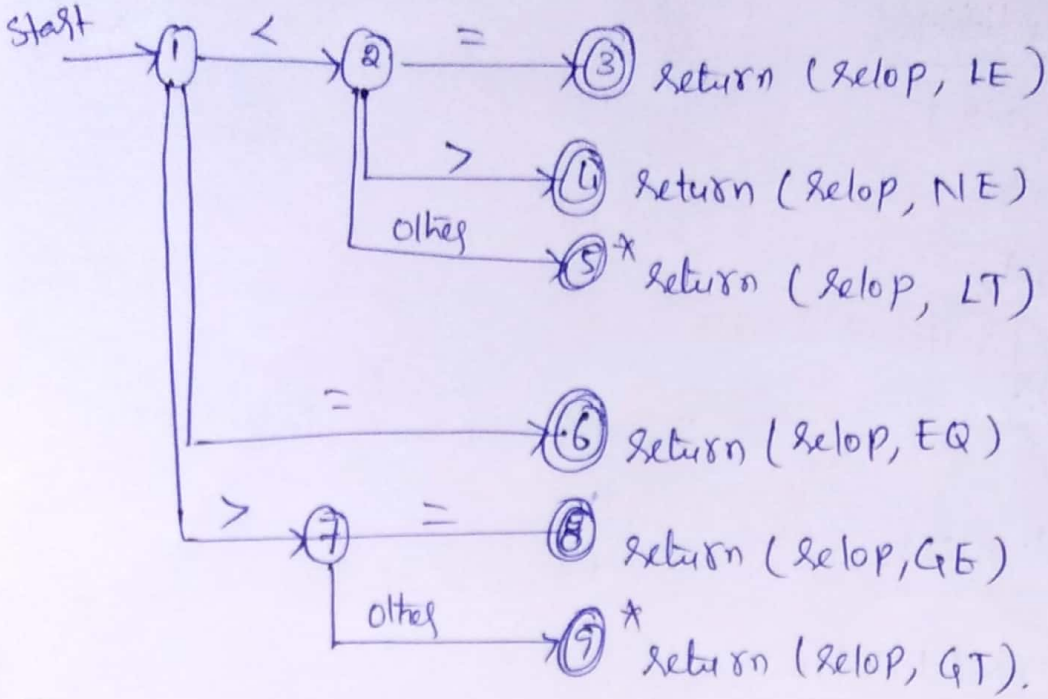
else → else

while → while.



Recognition of operators:

Relop → > | >= | < | <= | = | <>

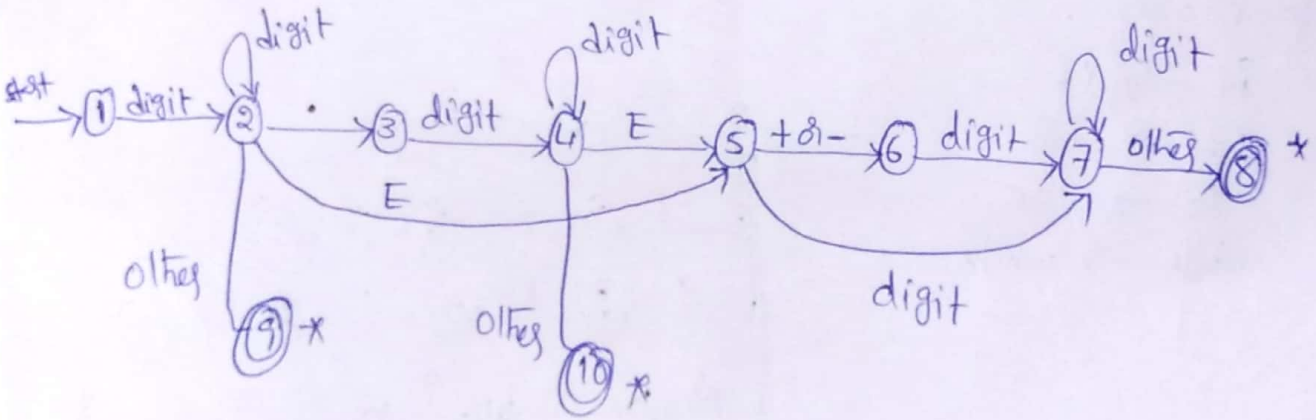


Recognition of number:

digit → [0-9]

digits → digit+

Number → digits (.digits)? (E [+-]? digits)?



Transition Diagram:-

An Automation is self-operating machine, a system which obtains transforms, transmits and uses information to perform the functions without direct human participation.

* An Automation is a model of computation consisting of set of states and discrete i/p's and o/p's.

* A start state an i/p alphabet and transition function that maps i/p symbol and current state to the next state.

* computation begins in the start state with an i/p string. it changes to new state depending on the transition functions

FA can be Represented a 5-tuple

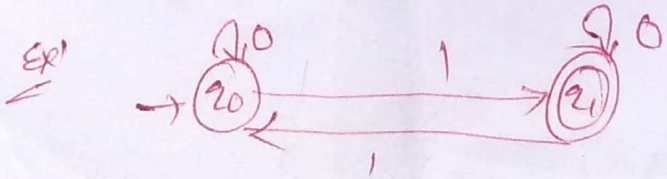
$M = (Q, \Sigma, \delta, q_0, F)$

Representation of FA: It can be done in 3 ways

1, transition diagram
2, transition table
3, transition function

T.D: FA is Represented in the form of a finite directed labelled graph called as transition graph. in which each vertex Represents as state and the directed edges indicate the transition of state.

Ex:



T.T It is a conventional tabular Representation of T.D.

It accepts 2 arguments and returns a value.

Rows - States

column - i/p symbol.

	0	1
q_0	q_0	q_1
q_1	q_1	q_0

Transition function: this is a function which describes the change of states during the transition. it maps a transition from one state to another state on some i/p symbol.

$$\delta = Q \times \Sigma \to Q$$

Types of F.A:

1. DFA
2. NFA.

DFA: have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state

* it doesn't accept 'ε' as i/p
* Almost one transition

Ex:

it is in D.F.A.

T.T

mathematical function of DFA

$$M_D = (Q, \Sigma, \delta, q_0, F)$$

$$\delta = Q \times \Sigma \to Q$$

N.F.A: have no Restriction on the labels of their edges.

→ A symbol can label several edges out of the same state, and ε, the empty string is possible label.

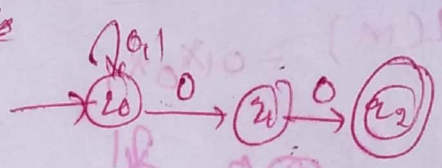
* at least one transition
* it accepts ε' as i/p symbol

Mathematical Def of NFA

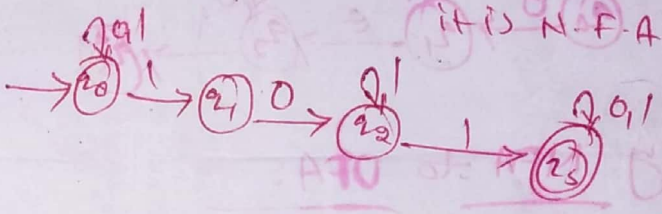
$M_N = \{Q, \Sigma, \delta, q_0, F\}$

$\delta = Q \times \Sigma \to 2^Q$ states.

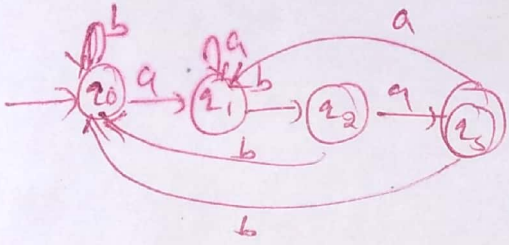
Ex



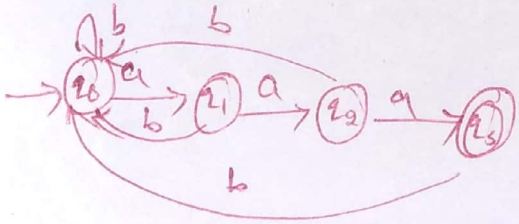
it is N.F.A



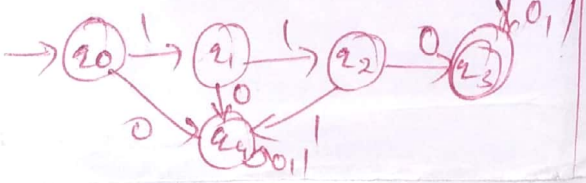
Construction DFA accepting all string over {a,b} ending in ab



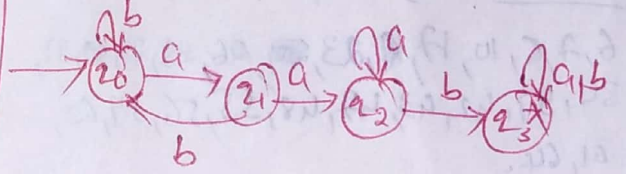
Ending with aaa



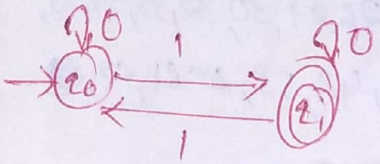
L = {w/w starts with 110}



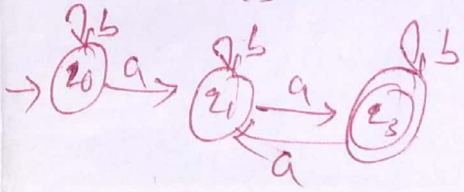
substring



odd no of 1's



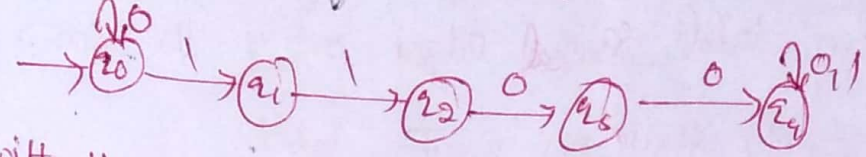
even no a's



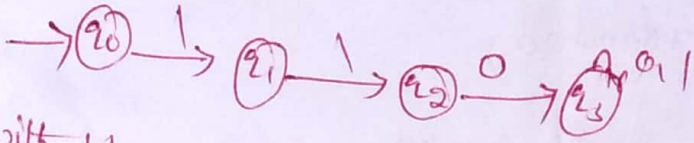
Example aabaa

NFA:

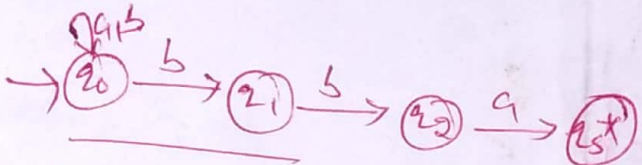
Contain 1100 as substring



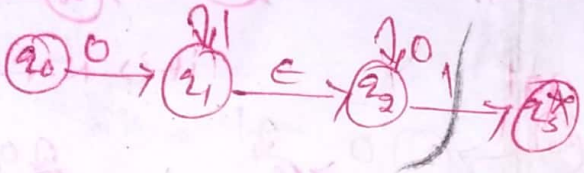
Starts with 110



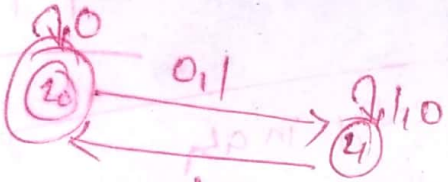
ending with bba



L(M) = 01*0*1



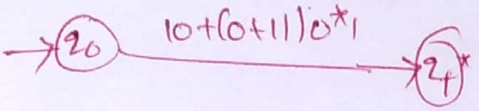
Converting NFA to DFA:

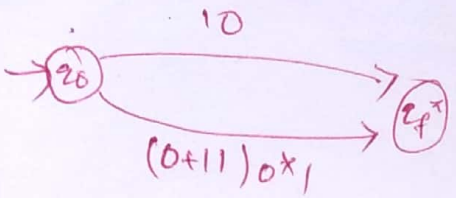


Construct the FA which accepts the R.E

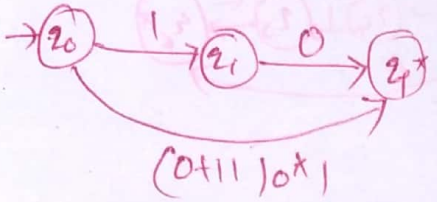
$10+(0+11)0^*1$

$\delta = 10+(0+11)0^*1$

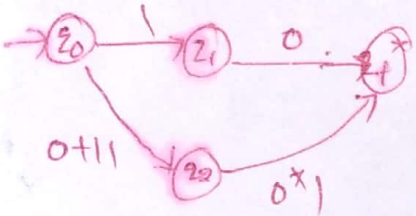




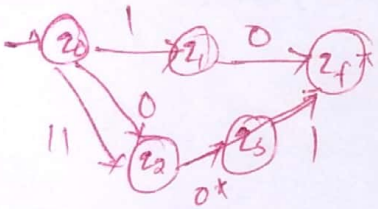
on eliminating union

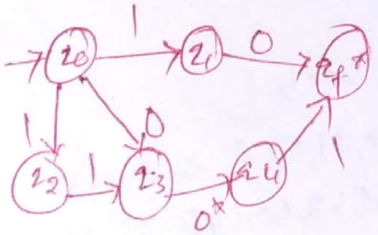


On eliminating concatination

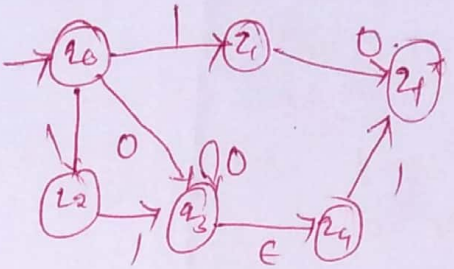


on eliminating concatination

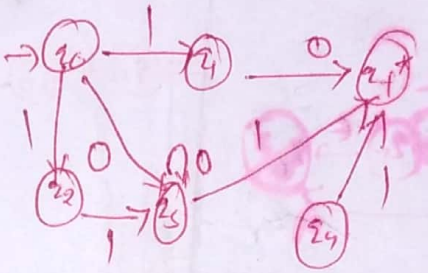




Elimination



Elimination ε



Construction of DFA

	0	1
q0	q5	q2 q1
q1	q4	ϕ
q2	ϕ	qf
q3	q3	qf
q4	ϕ	q1
qf	ϕ	ϕ

Q = {[q0], [q5] [q1,q2], [qf]}

	0	1
[q0]	q5	[q1 q2]
[q3]	[q3]	[qf]
[q1 q2]	[qf]	[q3]
[qf]	ϕ	ϕ

Construct NFA with ∈-moves for the following RE

$01^* + 10^*$

$r = 01^* + 10^*$

$r_1 \; r_2$

$r_1 = 01^*$

$r_3 \; r_4$

$r_3 = 0$

$r_4 = 1^*$

$r_2 = 10^*$

$r_5 \; r_6$

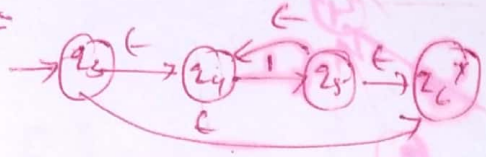
$r_5 = 1$

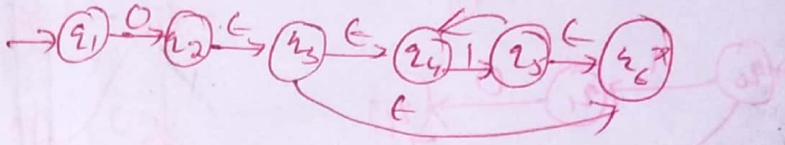
$r_6 = 0^*$

Construction of r_1:-

$r_1 = r_3 \cdot r_4$

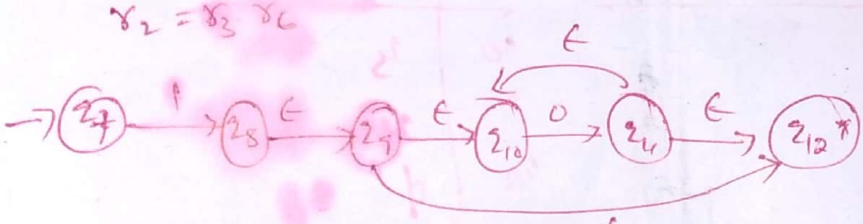
$r_3 =$ 

$r_4 =$ 

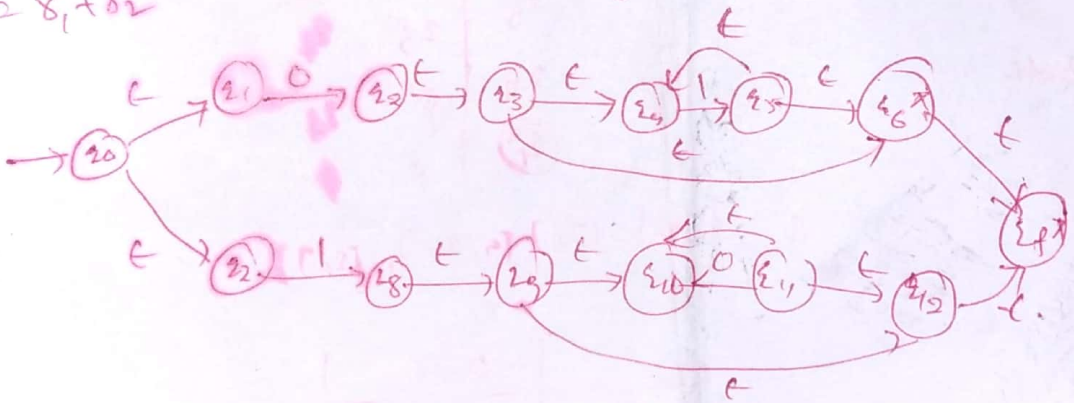


Construction of r_2:-

$r_2 = r_5 \; r_6$

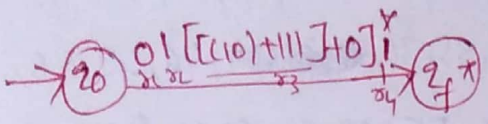


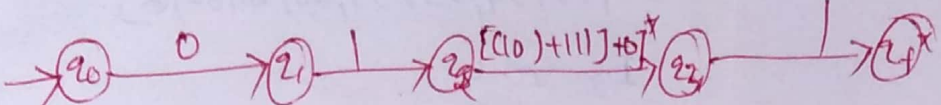
$= r_1 + r_2$

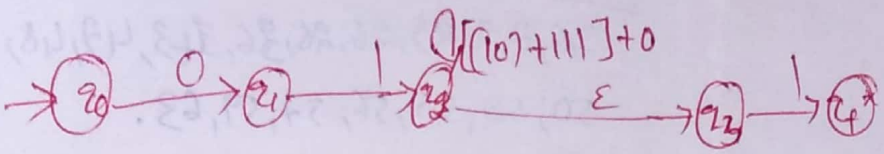


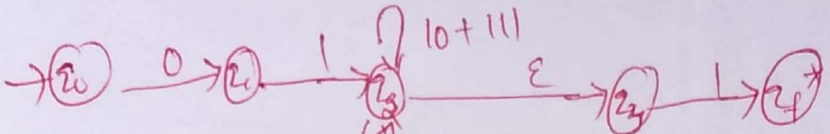
Construct FA equivalence to the following R.E:

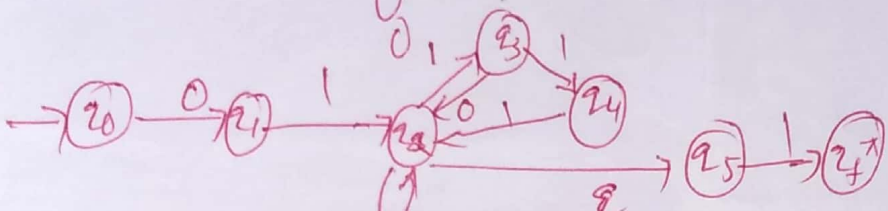
$$R = 0 | [[(10)+111]+0]^*$$

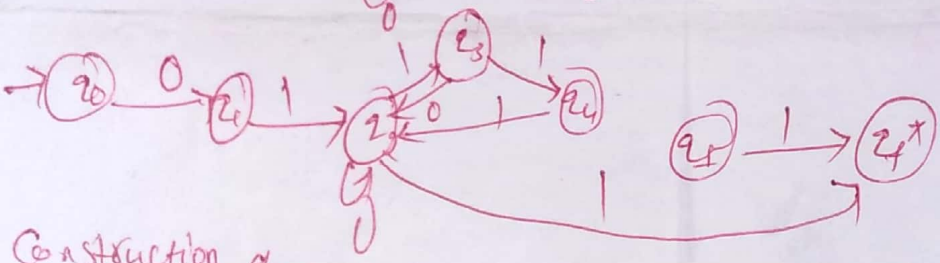










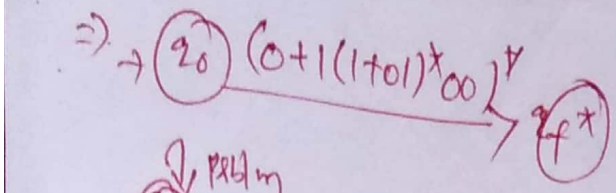


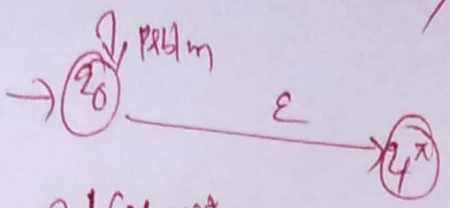
Construction of DFA:

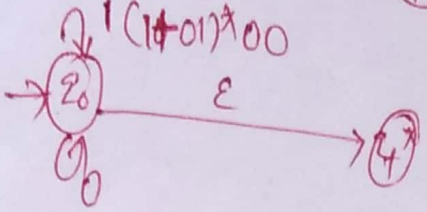
	0	1
q0	q1	ϕ
q1	ϕ	q2
q2	q2	{q3, q4}
q3	q2	q4
q4	ϕ	q2
q5	ϕ	q4*
q4*	ϕ	ϕ

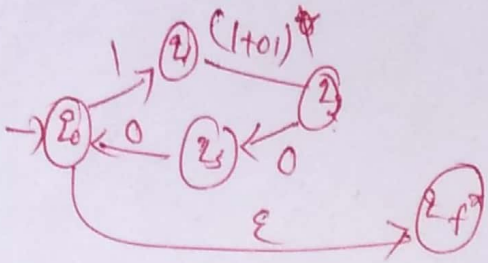
	0	1
q0	{q1}	ϕ
q1	ϕ	{q2}
q2	{q2}	{q3, q4}
{q3, q4*}	{q2}	{q4}
{q4}	ϕ	{q2}

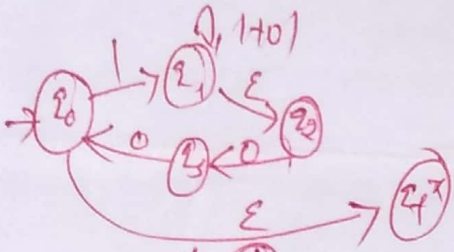
$(0+1(1+01)^*00)^*$

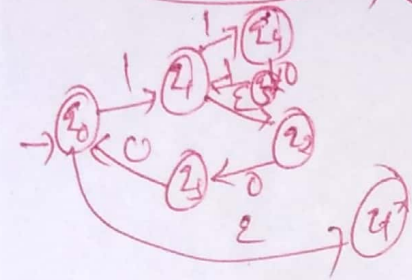

$(0+1(1+01)^*00)^*$

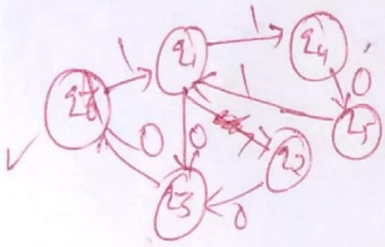

Problem
ε


$1(1+01)^*00$
ε
0


$(1+01)^*$
1
0
0
ε


$1+01$
1
ε
0
0
ε


1
1
0
0
ε


1
1
0
0
ε

1
1
0
0
0