Compiler Design :- Unit-1 :-

1. Translator
2. Phases.

Compiler :-

Compiler can ~~convert~~ translate a source program to target program.

Source prog. → Compiler → Target prog.

C, C++, etc.. (Compiler)

assembly language

Interpretor :-

Translates the code line by line.

Language processor :-

Pre processor | feeds source prg to compiler.

↓

Compiler | source prg to target prog.

↓

Assembler | assembly language to machine level language

↓

Loader / Linker → Links all the files.

(Single point of execution)

Loader: loads the machine level language to destination

Compiler.

Analysis	Synthesis
Front end	Back end.

Analyze Source prog

→ Analyze :-

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer

① Lexical Analyser :-

(Eg:) $c = a + b * 10;$

$\boxed{id_1 = id_2 + id_3 * 10;}$ (tokens)

②. Syntax Analyzer :-

```
        =
      /   \
   id1     +
         /   \
      id2     *
            /   \
         id3     10;
```

(parse tree)

③ Semantic

=

id_1

id_2

15m

→ Phases

1. Lexical
2. Syntax
3. Semanti
4. Interme
5. Code
6. Code

④ Inte

conv

code

t_1

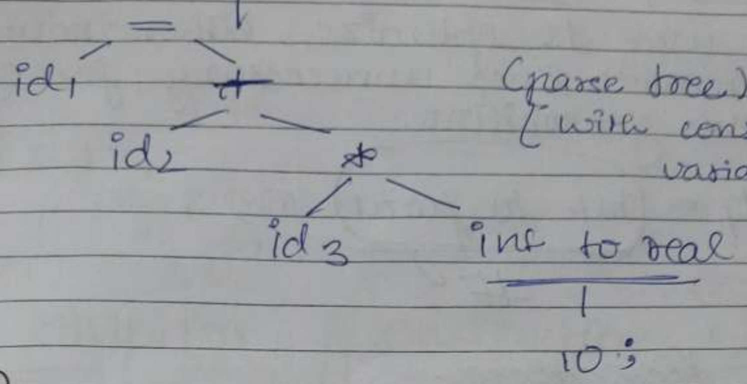
t_2

t_3

t

id

③ Semantic Analyzer :-



(parse tree)
[with constant variation]

16m

→ Phases of compiler :-

① Lexical analyzer
② Syntax analyzer
③ Semantic analyzer
④ Intermediate code generator
⑤ Code optimizer
⑥ Code generator.

} Front End. (①–④)

} Back End. (⑤–⑥)

④ Intermediate code Generator :-

converts semantic analyze to 3 address code.

t_1 = int to float ~~??~~ 10;
t_2 = id_3 * t_1;
t_3 = id_2 + t_2;
t_4 = t_3;
id_1 = t_4;

(5) Code optimizer :-

It is used to optimize either redundancy or remove lines unnecessary for effective output generation.

$t_1 = \boxed{\text{int to float}}\ 10;$

#

$t_1 = id_3 * \#10;$

$t_2 = id_2 + t_1;$

$id_1 = t_2;$

Optimized Code.

(6) Code Generator :-

It is used to produce the target program.
Assembly language is produced without any redundancy.

```
mov  #10 , R1
mov  id3 , R2
mul  R2 , R1
mov  id2 , R3
add  R3 , R1
mov  R1 , id1
```

Q What is
One cor
2m
Pass :-
(5m)
→ Com
(1) Parser Genera
(2). Scan Gen
(3) Syn Tran
Seman analys

Q What is a pass?

One complete scan of a program is called as pass.

2m Pass:
- 4m 1 Pass → 6 phases of compiler.
- 4m 2 Pass → (differentiates between front end & back end.)

(5m)

→ **Compiler Construction Tools :-**

① **Parser Generator** →

Tokens
↓
[Syntax Analyser]
↓
Parse Tree.

② **Scanner Generator** →

Source Program
↓
[Lexical Analyser]
↓
Token.

③ **Syntax Directed Translation Engine** →

Parse tree
↓
[Syntax Directed translation engine]
↓
3 address code

||

Symantic analyzer + ICG

④ Automatic Code Generator :-

||

Optimiser + Code Generator

3 address code
↓
[ACG]
↓
Assembly language

⑤ Compiler Construction :-

tool.exe (converts to executable file).

Activity - 1 :-

position = initial + rate * 100;

↓

Lexical Analyser

↓

$id_1 = id_2 + id_3 * 100;$

↓

Syntax Analyser

↓

```
 id1 =
      +
  id2    *
     id3   100;
```

↓

Symantic Analyser

↓

```
 id1 =
      +
  id2    *
     id3   int to real
                |
               100;
```

↓

Intermediate Code Generator

↓

$t_1 = \#100;$

$t_2 = id_3 * t_1;$

$t_3 = id_2 + t_2;$

$t_4 = t_3;$

$id_1 = t_4;$

↓

Code Optimizer

↓

$t_1 = id_3 * \#100;$

$t_2 = id_2 + t_1;$

$id_1 = t_2;$

↓

Code Generator

mov #100, R_1
mov id_3, R_2
mul R_2, R_1
mov id_3, R_3
add R_3, R_1
mov R_1, id_1

Parsv

→ Input Buffering :-

1. Begin Pointer - ~~Exists~~ Reads input from blank space.
2. Forward Pointer - Reads the buffer and calls the BP when a blank space is found.

bp ↓ (i), bp ↓ (m)

i	n	t		m	a	i	n	(	(		)	

fp ↑ (i) - - - >

Activity - I

Q. Find the number of tokens which is given in the following input.

```
int main()
{
printf("SRM University");
return 0;
}
```

bp ↓ (i), bp ↓ (m)

i	n	t		m	a	i	n	(	)	

fp ↑ (i), fp ↑ (t), fp ↑ (m), fp ↑ ())

'int', 'main', '(', ')', '{', 'printf', '(', '"SRM University"', ')', ';', 'return', '0', ';', '}'

Total = 14 tokens.

→ **Specification of token :-**

1. **Token :-** Token validates an identifier, operator or a keyword.

2. **Lexeme :-** It is predefined rules

   Eg :- int main ( )
   keyw keyw op op } → defined by lexeme.

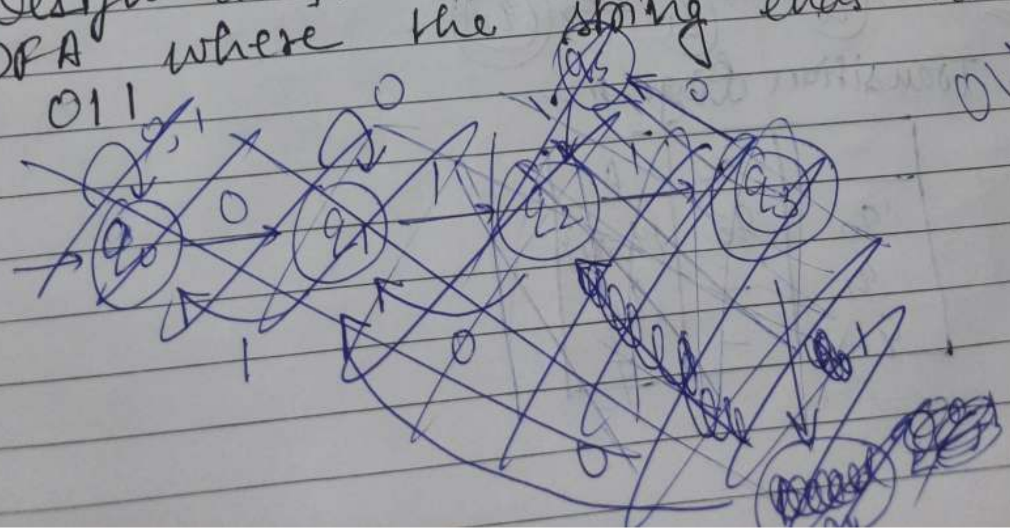
3. **Pattern :-** It is a set of rules that define a token.

→ **Finite Automata :-**

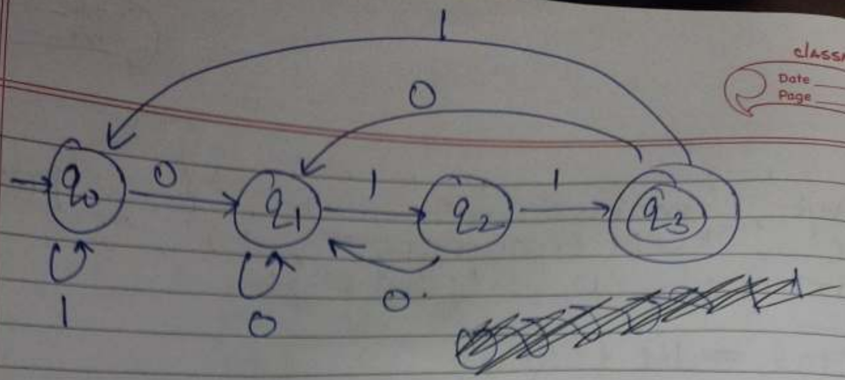
Activity - 2.

Q. Design a state transition diagram for DFA where the string ends with "011"

011

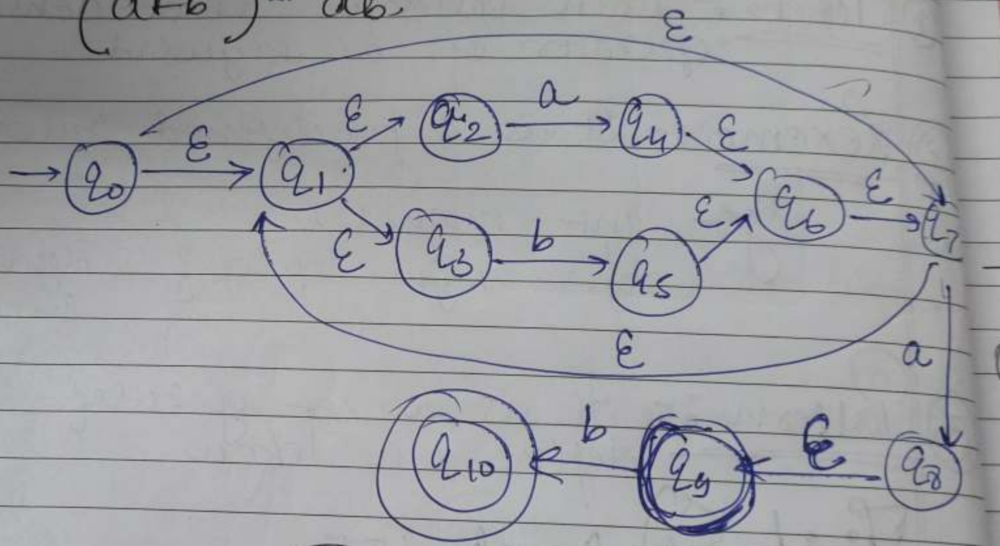


011



Q Draw an ε-NFA for given exp.

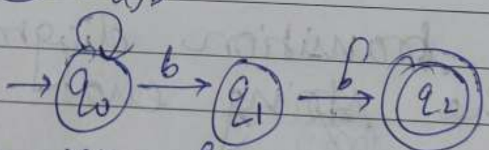
$(a+b)^* ab$



→ * NFA to DFA (8m)

Step-1 ① q.

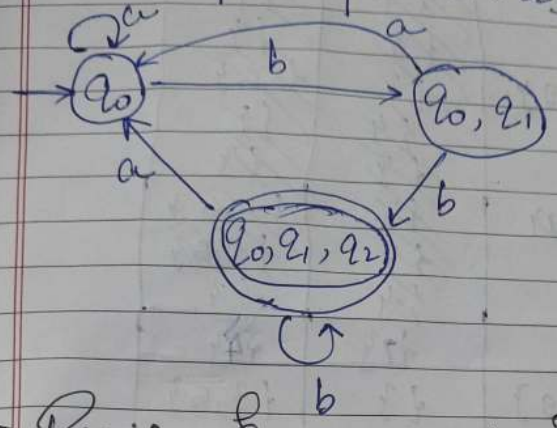
② $T_{a,b}$

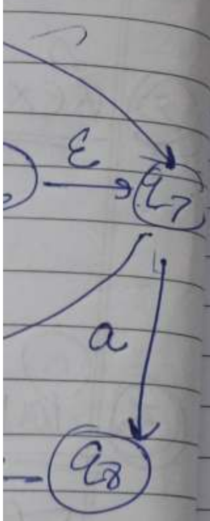
Eg:- 

① Transition Diagram.

	a	b
q_0	q_0	$\{q_0, q_1\}$
q_1	–	q_2
q_2	–	–

	a	b
q_0	q_0	$\{q_0, q_1\}$
$\{q_0, q_1\}$	q_0	$\{q_0, q_1, q_2\}$
* $\{q_0, q_1, q_2\}$	q_0	$\{q_0, q_1, q_2\}$

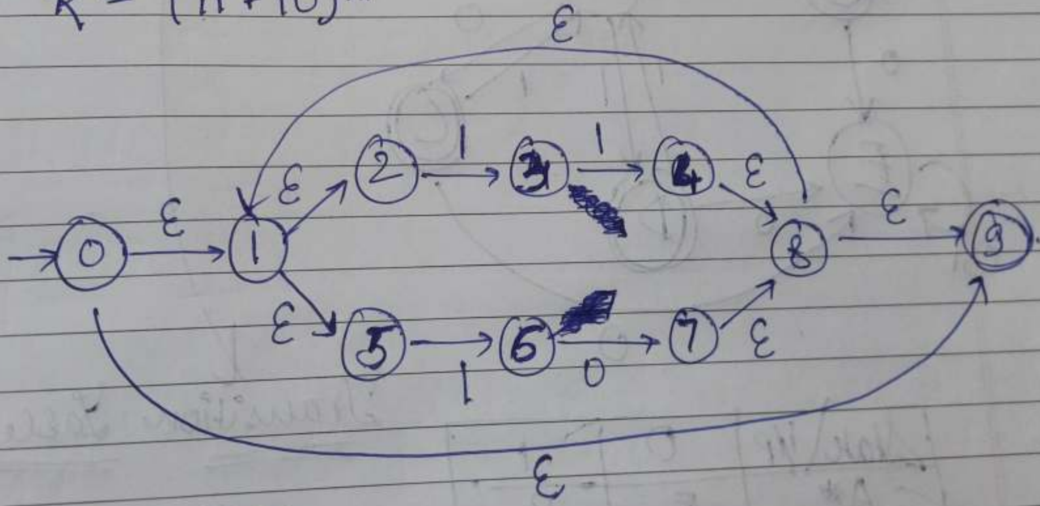




→ Regular Expression to DFA :- 10 m

① $RL \rightarrow RE \rightarrow \varepsilon NFA \rightarrow DFA \rightarrow \text{min } DFA$

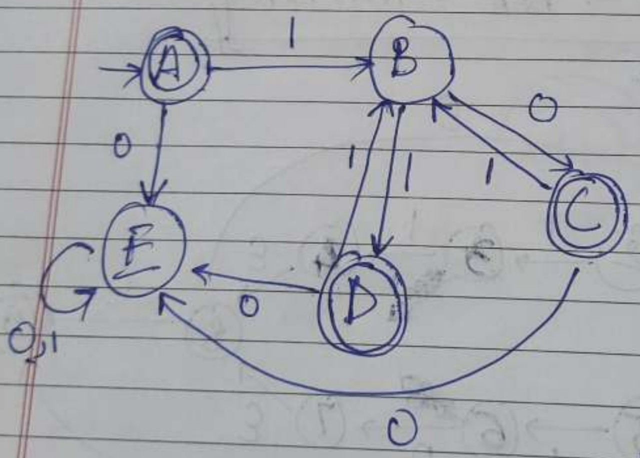
$R = (11 + 10)^*$



Transition Table

ENFA to DFA :-

X	Y = ε closed(X)	δ(Y,0)	δ(Y,1)
→ A {0}*	{0,1,2,5,9}	{ }	{3,6}
1	{2,5}	{ }	{3,6}
2	{ }	{ }	{ }
3	{3}	{ }	{ }
4	{8,1,2,5,9}	{ }	{3,6}
5	{5}	{ }	{ }
6	{6}	{ }	{ }
7	{8,1,2,5,9}	{ }	{3,6}
8	{1,2,5,9}	{ }	{3,6}
9	{ }	{ }	{ }
B {3,6}	{3,6}	{7}	{4}
C {7}*	{7,8,1,2,5,9}	{ }	{3,6}
D {4}*	{4,8,1,2,5,9}	{ }	{3,6}
E { }	{ }	{ }	{ }

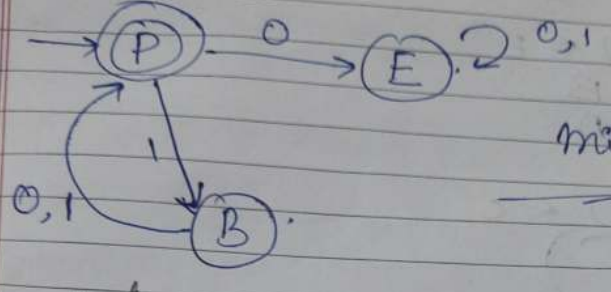


Transition Table :-

State \ i/p	0	1
A*	E	B
B	C	D
C*	E	B
D*	E	B
E	E	E

P

State\ip	0	1
* P	E	B
B	P	P
E	E	E

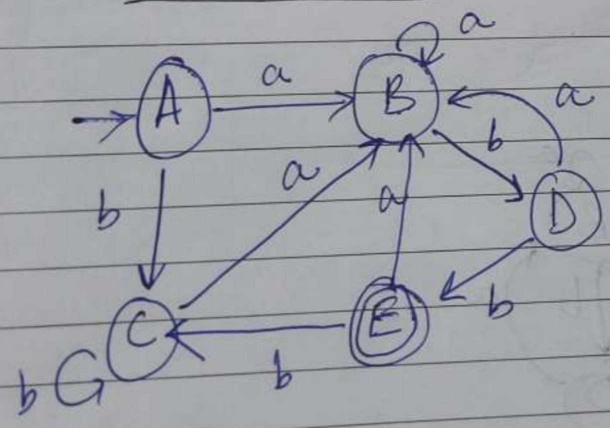


min DFA

→ Minimization of DFA.

Eg:- Construct min-DFA Automata.

State\I/P	a	b
A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C



(i) $\pi = \{A, B, C, D, E\}$

$\pi' = \{A, B, C, D\} \quad \{E\}.$

(2) $\delta(A,a)\ (A,b) \leftarrow A$

$\delta(A,a) = B$
$\delta(A,b) = C$

$\delta(B,a)\ (B,b) \leftarrow B$
$\delta(B,a) = B$
$\delta(B,b) = D$

$\delta(C,a)\ (C,b) \leftarrow C$
$\delta(C,a) = B$
$\delta(C,b) = C$

$\delta(D,a)\ (D,b) \leftarrow D$
$\delta(D,a) = B$
$\delta(D,b) = E$

$\delta(E,a)\ (E,b) \leftarrow E$
$\delta(E,a) = B$
$\delta(E,b) = C$

(3)

	a	b
A	B	A
B	B	D
D	B	E
*E	B	A