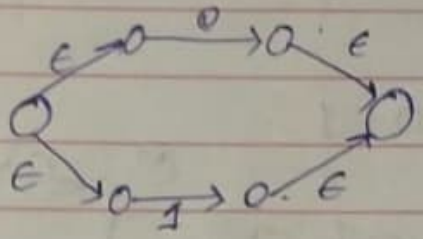
I - ASSIGNMENT

(Start Writing From Here)

Q) Convert the given Regular expression to Finite Automata.
$(0+1)^* 1 (0+1)$

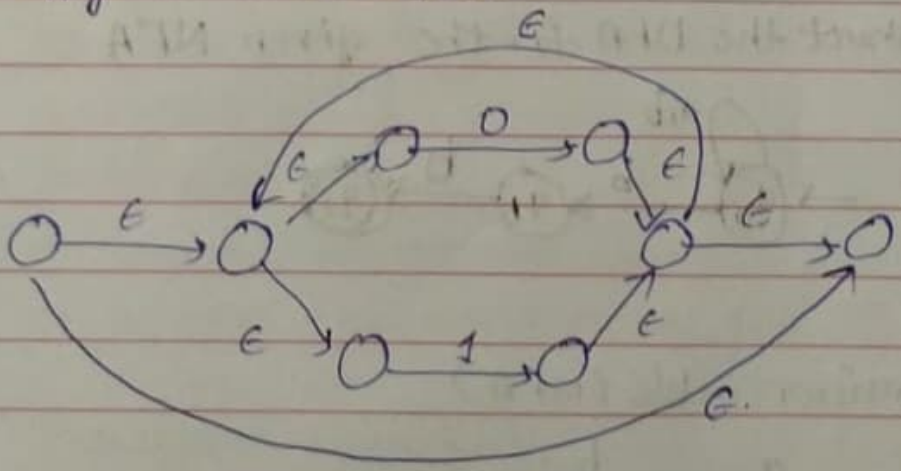
A: Given Regular expression $(0+1)^*$.

For representing the regular expression consider 2 state 0 and 1.

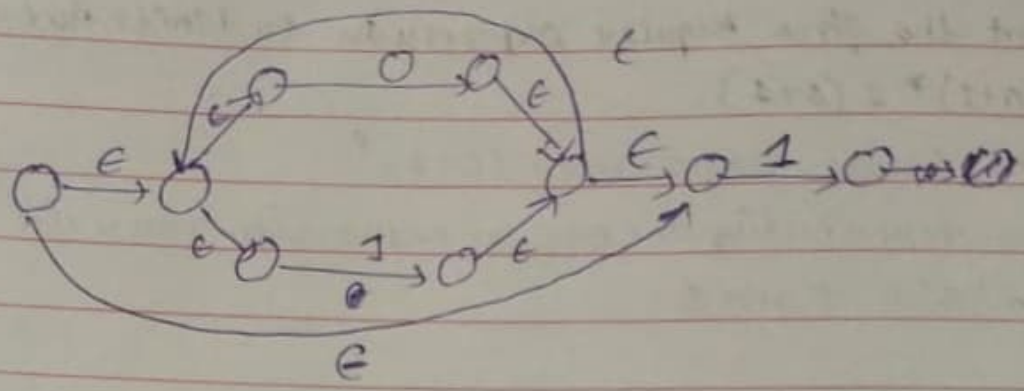


→ For union (or) + operation take 2 more states and apply union symbol.

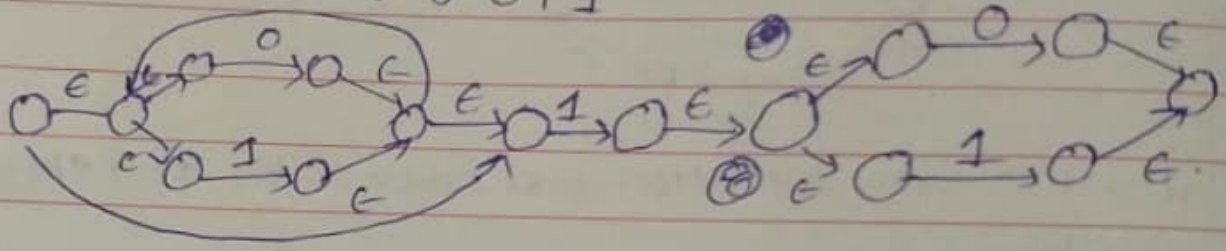
→ To apply $(0+1)^*$ take 2 more states



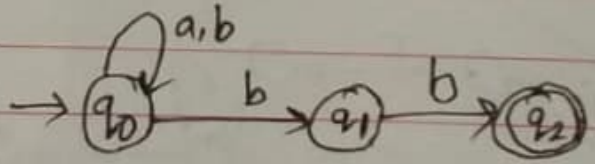
→ Another 2 states 1 and 0+1, to move to one consider another state.



→ Another state is 0+1



2(a) Construct the DFA for the given NFA



Sol): Transition Table (NFA)

	a	b
→ q_0	q_0	$\{q_0, q_1\}$
q_1	ϕ	q_2
(q_2)	ϕ	ϕ

DFA Transition table.

	a	b.	
$\rightarrow q_0$	q_0	$\{q_0, q_1\}$	$\{\{q_0, q_1\} \cup q_3\}$
$\{q_0, q_1\}$	q_0	$\{q_0, q_1, q_2\}$	$\{q_0 \cup q_3\} \cup \{q_0 \cup q_3\}$
$\{q_0, q_1, q_2\}$	q_0	$\{q_0, q_1, q_2\}$.	$q_0 \cup \emptyset$
			q_0.
			$\{q_0, q_1\} \cup b$
			$\{q_0, b\} \cup \{q_1, b\}$
			$\{q_0, q_1\} \cup q_2$.

DFA construction:

- Initial state.

  The DFA starts in $\{q_0\}$, which is the initial state of the NFA

- Transitions from $\{q_0\}$:

  On input a, $\{q_0\}$ remains $\{q_0\}$

  On input b, $\{q_0\}$ moves to $\{q_0, q_1\}$

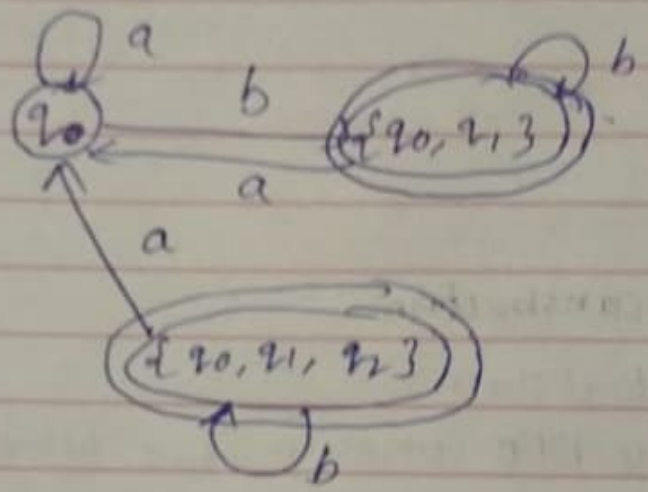
- Transitions from $\{q_0, q_1\}$

- On input a, $\{q_0, q_1\}$ moves to $\{q_0\}$.
- On input b, $\{q_0, q_1\}$ moves to $\{q_0, q_1, q_2\}$

Transitions from {q0, q1, q2}:

- on input a, {q0, q1, q2} moves to {q0}
  on input b, {q0, q1, q2} remains in {q0, q1, q2}

State Diagram



3. Explain the phases of compiler for the following example.

1; Position = Initial + rate * 60

Step1; position = Initial + rate * 60

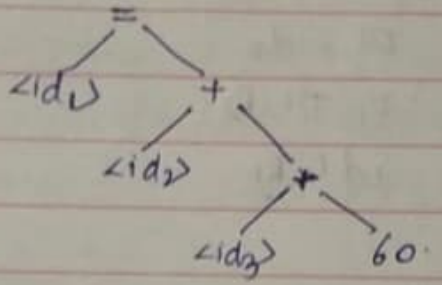
[Lexical Analyser] → In this phases the expression is divided into tokens

$id_1 := id_2 + id_3 * 60$

2) Syntax Analysis or parsing: Parses every token and constructs hierarchial structure format which is called as a Parse tree.

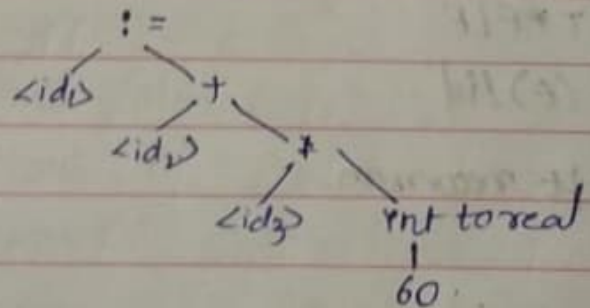
<id1> - <position>
<id2> - <initial>
<id3> - <rate>.

Syntax tree or Parse tree



3) Semantic Analyses:

Verifies the meaning of each statement



4) Intermediat code generator

Assigns to required number of temporary variable

t1 = int to real (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Called as 3 address code

5) code optimizer.

reduces the no of lines

temp1 := id3 * 60.0

id1 := id2 + temp1

6) Code Generator

```
LDF   R2 id3
MULF  R2 R2, #60
LDF   R1 , id2
ADDF  R1, R1, R2
STF   id1, R1
```

4) Construct the predictive parser for the following grammar

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

A)i) Eliminate left recursion

i) $E \rightarrow E+T/T$

$E \rightarrow TE'$

$E' \rightarrow +TE'/\varepsilon$

ii) $T \rightarrow T*F/F$

$T \rightarrow FT'$

$T' \rightarrow *FT'/\varepsilon$

iii) $F \rightarrow (E)/id$

After eliminating left recursion:

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$

Step2: left factoring is Not possible

Step3: First and follow functions

First(E) = First(T) = FIRST(F) = {(, id}

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

Follow function

Follow(E) = {$,)}

Follow(E') = {$,)}

Follow(T) = {+, $,)}

Follow(T') = {+, $,)}

Follow(F) = {*, +, $,)}

Predictive parsing Table

NTs	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	
E'	$E' \to +TE'$			$E' \to \varepsilon$		$E' \to \varepsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$		$T' \to \varepsilon$.
F			$F \to (E)$		$F \to id$	

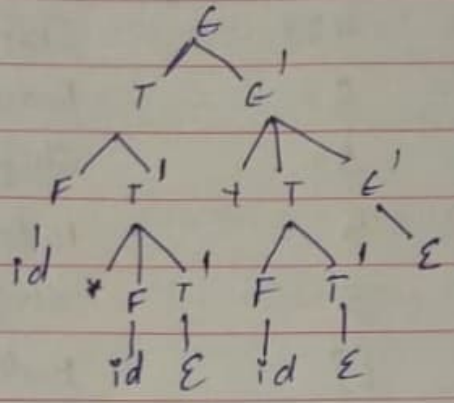
$w = id * id + id$

Stack	Input	output
$\$E$	$id * id + id\$$	
$\$E'T$	$id * id + id\$$	$E \to TE'$
$\$E'T'F$	$id * id + id\$$	$T \to FT'$
$\$E'T'id$	$id * id + id\$$	$F \to id$
$\$E'T'$	$* id + id\$$	
$\$E'T'F*$	$* id + id\$$	$T' \to * FT'$
$\$E'T'F$	$id + id\$$	
$\$E'T'id$	$id + id\$$	$F \to id$
$\$E'T'$	$+ id\$$	
$\$E'$	$+ id\$$	$T' \to \varepsilon$
$\$E'T+$	$+ id\$$	$E' \to +TE'$
$\$E'T$	$id\$$	
$\$E'T'F$	$id\$$	$T \to FT'$
$\$E'T'id$	$id\$$	$F \to id$
$\$E'T'$	$\$$	

$ E' $ $ E' \to \varepsilon $

$

Parse tree



id * id + id.

5) Consider the Grammar

$E \to 2E2$

$E \to 3E3$

$E \to 4$

Perform shift Reduce parsing for the input string

"32423"

Sol) Given Grammar rules

1. $E \to 2E2$
2. $E \to 3E3$
3. $E \to 4$

Shift - Reduce Parsing Table:

Step	Stack	Input	Action
1	[]	32423	Shift 3
2	[3]	2423	Shift 2
3	[3,2]	423	Shift 4
4	[3,2,4]	23	Reduce by $\epsilon \to 4$
5	[3,2,ϵ]	23	Shift 2
6	[3,2,ϵ,2]	3	Reduce by $\epsilon \to 2\epsilon 2$
7	[3,ϵ]	3	Shift 3
8	[3,ϵ,3]	[]	Reduce by $\epsilon \to 3\epsilon 3$
9	[ϵ]	[]	Accept

Explanation of the Parsing steps

1. Shift 3: We shift first symbol 3 from the input to the stack

2. Shift 2 : We shift '2' from the input to the stack.

3. Shift 4 : We shift '4' from the input to the stack

4. Reduce by $\epsilon \to 4$: The top of the stack has '4' which matches the rules $\epsilon \to 4$, so we reduce it to ϵ

5. Shift 2: Shift '2' from the input onto the stack

6. Reduce by $\epsilon \rightarrow 2\epsilon 2$: The stack now has '2 ε 2', which matches the rule $\epsilon \rightarrow 2\epsilon 2$, so we reduce it to 'ε'.

7. Shift 3: We shift '3' from the input to the stack.

8. Reduce by $\epsilon \rightarrow 3\epsilon 3$: The stack has '3 ε 3', which matches the rule $\epsilon \rightarrow 3\epsilon 3$, so we reduce it to 'ε'.

9. Accept: The stack has only 'ε' and the input is empty, so we accept the string.

The input string '32423' is successfully parsed using the provided grammar.