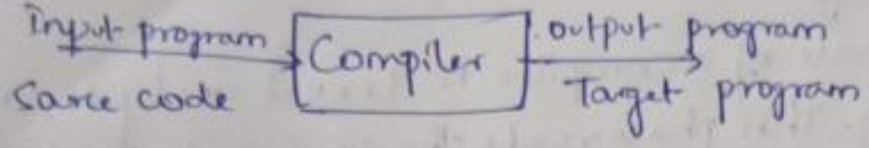
K. Mahathi
21BS1A3325 CSE-A

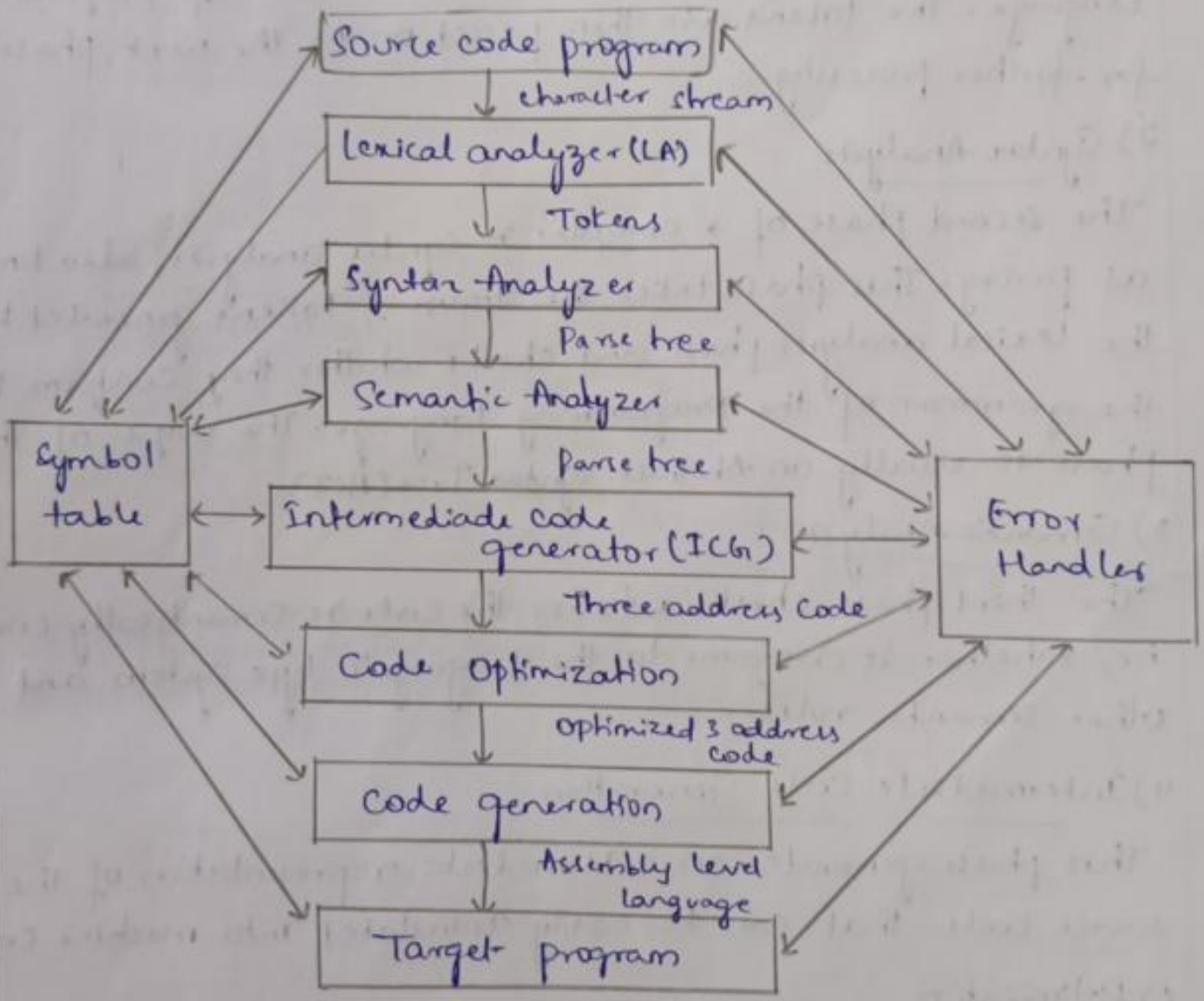
ACD Assignment-1

1) Define a Compiler. Explain briefly about the different phases of a compiler with example.

A compiler is a program that helps in translating the computer code from one programming language into another programming language.



Phases of a Compiler



There are 2 phases of compilers:

1) Analysis phase (front end of compiler)
2) Semantic phase (back end of compiler)

→ The analysis phase creates an intermediate representation from the given source code.

→ The synthesis phase creates an equivalent target program from the intermediate representation.

1) Lexical Analysis

The first phase of a compiler is lexical analysis, also known as scanning. This phase reads the source code and breaks it into a stream of tokens, which are basic units of programming language. The tokens are then passed on to the next phase for further processing.

2) Syntax Analysis

The second phase of a compiler is syntax analysis, also known as parsing. This phase takes the stream of tokens generated by the lexical analysis phase and checks whether they conform to the grammar of the programming language. The output of this phase is usually an Abstract Syntax Tree (AST).

3) Semantic Analysis

The third phase checks whether the code is semantically correct, i.e, whether it conforms to the language's type system and other semantic rules.

4) Intermediate Code Generation

This phase generates an intermediate representation of the source code that can be easily translated into machine code.

5) Optimization

This phase applies various optimization techniques to the intermediate code to improve the performance of the generated machine code.

6) Code Generation

This phase takes the optimized Intermediate code and generates the actual machine code that can be executed by the target hardware.

Ex:- position := initial + rate * 60

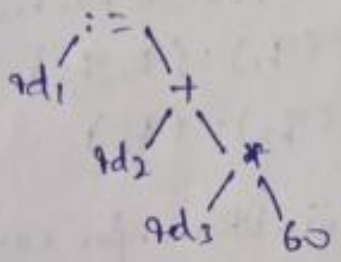
id1 := id2 + id3 * 60

↓

Lexical analyzer

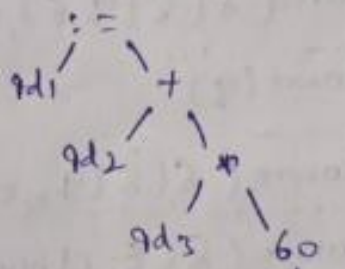
↓ id1, :=, id2, +, id3, *, 60

Syntax analyzer



↓

Semantic analyzer



int to float (or) int to real

↓

Intermediate Code generator

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

↓

Code optimizer

t1 = id3 * 60.0
t2 = id2 + t1
id1 = t2

(or)

t1 = id3 * 60.0
id1 = id2 + t1

↓

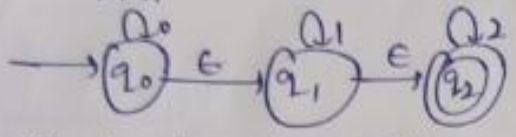
code generator

↓

MOVF id3, R2 // R2 = id3
MULF #60.0, R2 // R2 = R2 * 60.0

```
MOVF id2,R1 // R1 = id2
ADDF R2,R1 // R1 = R1+R2
MOVF R1,id1 // id1 = R1
```

2) Convert the given NFA with ε to NFA without ε and construct DFA



NFA with ε to NFA without ε

step1 :- ε - closure $(q_0) = \{q_0, q_1, q_2\}$ [∵ ε-closure(p) = {p} ∀ p ∈ Q, if ∃ ε-closure(p) = {q} & δ(q,ε) = r then ε-closure(p) = {q,r}]

ε - closure $(q_1) = \{q_1, q_2\}$,

ε - closure $(q_2) = \{q_2\}$

step 2 :- Find δ' transitions for each state

$\delta'(q,a) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q,\varepsilon), a))$ where

$\hat{\delta}(q,\varepsilon) = \varepsilon\text{-closure}(q)$

→ $\delta'(q_0,0) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_0,\varepsilon),0))$

$= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_0),0))$

$= \varepsilon\text{-closure}(\delta(q_0,q_1,q_2),0)$

$\delta'(q_0,0) = \varepsilon\text{-closure}(q_0) = \{q_0,q_1,q_2\}$

→ $\delta'(q_0,1) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_0,\varepsilon),1))$

$= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_0),1))$

$= \varepsilon\text{-closure}(\delta(q_0,q_1,q_2),1)$

$\delta'(q_0,1) = \varepsilon\text{-closure}(q_1) = \{q_1,q_2\}$

→ $\delta'(q_0,2) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_0,\varepsilon),2))$

$= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_0),2))$

$= \varepsilon\text{-closure}(\delta(q_0,q_1,q_2),2)$

$\delta'(q_0,2) = \varepsilon\text{-closure}(q_2) = \{q_2\}$

$\rightarrow \delta'(q_1,0) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_1,\epsilon),0))$

$= \epsilon\text{-closure}(\delta(q_1,q_2),0)$

$= \epsilon\text{-closure}(\phi) = \phi$

$\rightarrow \delta'(q_1,1) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_1,\epsilon),1))$

$= \epsilon\text{-closure}(\delta(q_1,q_2),1)$

$\delta'(q_1,1) = \epsilon\text{-closure}(q_1) = \{q_1,q_2\}$

$\rightarrow \delta'(q_1,2) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_2,\epsilon),2))$

$= \epsilon\text{-closure}(\delta(q_1,q_2),2)$

$= \epsilon\text{-closure}(q_2) = \{q_2\}$

$\rightarrow \delta'(q_2,0) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_2,\epsilon),0))$

$= \epsilon\text{-closure}(\delta(q_2),0)$

$= \epsilon\text{-closure}(\phi) = \phi$

$\rightarrow \delta'(q_2,1) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_2,\epsilon),1))$

$= \epsilon\text{-closure}(\delta(q_2),1)$

$= \phi$

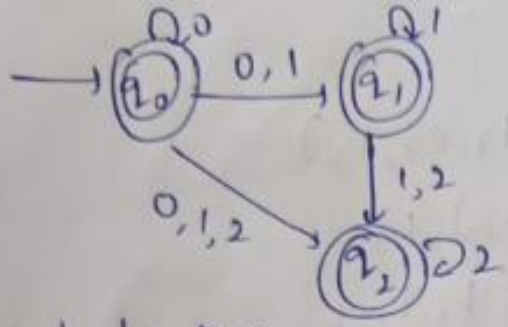
$\rightarrow \delta'(q_2,2) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_2,\epsilon),2))$

$= \epsilon\text{-closure}(\delta(q_2),2))$

$= \epsilon\text{-closure}(q_2) = \{q_2\}$

State \ Input	0	1	2
q_0	$\{q_0,q_1,q_2\}$	$\{q_1,q_2\}$	$\{q_2\}$
q_1	ϕ	$\{q_1,q_2\}$	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_2\}$

As ϵ-closure(q_0), ϵ-closure(q_1), ϵ-closure(q_2) contains final state q_2, hence the final states are q_0, q_1 and q_2



Convert to DFA

$\rightarrow \delta'([q_0],0) = \{q_0,q_1,q_2\}$

$\delta'([q_0],1) = \{q_1,q_2\}$

$\delta'([q_0],2) = \{q_2\}$

$\delta'([q_0,q_1,q_2],0) = \{q_0,q_1,q_2\}$

$\delta'([q_0,q_1,q_2],1) = \delta'([q_0],1) \cup \delta'([q_1],1) \cup \delta'([q_2],1)$

$= \{q_1,q_2\}$

$\delta'([q_1,q_2],0) = \delta'([q_1],0) \cup \delta'([q_2,0])$

$= -$

$\delta'([q_1,q_2],1) = \delta'([q_1],1) \cup \delta'([q_2],1)$

$= \{q_1,q_2\}$

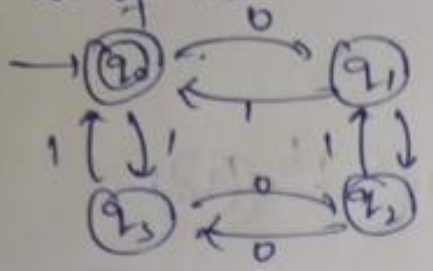
$\delta'([q_1,q_2],2) = \delta'([q_1,2]) \cup \delta'([q_2],2) = \{q_2\}$

$\delta'([q_2],0) = -, \delta'[[q_2],1) = -, \delta'([q_2],2) = \{q_2\}$

DFA

State \ I/P	0	1	2
q_0	A $\{q_0,q_1,q_2\}$	B $\{q_1,q_2\}$	q_2
A $\{q_0,q_1,q_2\}$	A $\{q_0,q_1,q_2\}$	B $\{q_1,q_2\}$	q_2
B $\{q_1,q_2\}$	-	B $\{q_1,q_2\}$	q_2
(q_2)	-	-	q_2

3) Design FA with $\Sigma = \{0,1\}$ accepts even number of 0's and even no. of 1's



$\{\{q_0, q_1, q_2, q_3\}, \{0,1\}, \delta, q_0, \{q_0\}\}$

	0	1
q_0	q_1	q_3
q_1	q_0	q_2
q_2	q_3	q_1
q_3	q_2	q_0

4) Construct ϵ-NFA for the given regular expression $(a+b)^*abb$

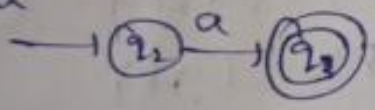
$r = (a+b)^* abb$

$r_1 \quad r_2$

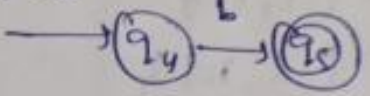
$r_1 = (a+b)^*$

$r_3 \quad r_4$

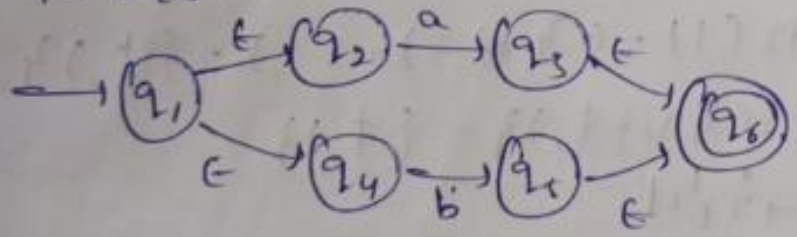
$r_3 = a$



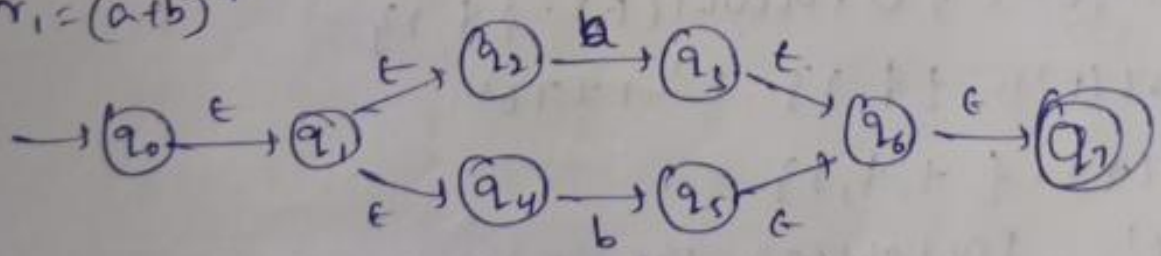
$r_4 = b$



$r_3 + r_4 = a+b$



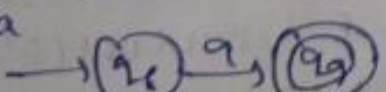
$r_1 = (a+b)^*$



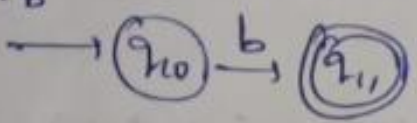
$r_2 = abb$

$r_5 \, r_6 \, r_7$

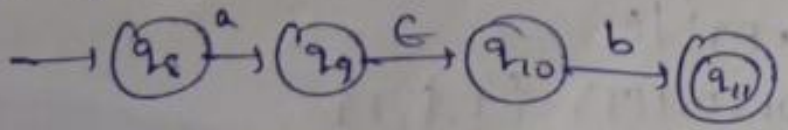
$r_5 = a$



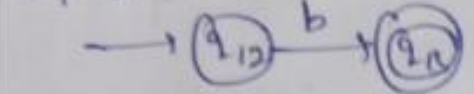
$r_6 = b$



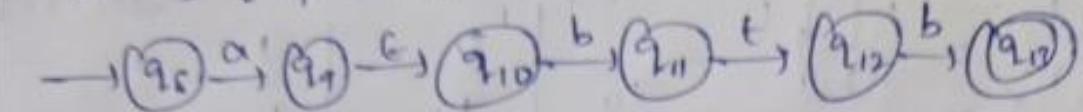
$r_5 \times r_6 = ab$



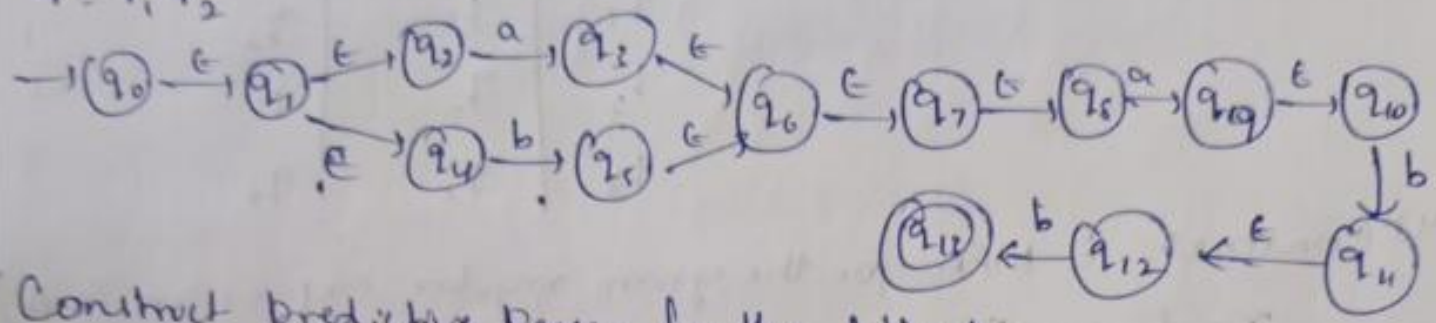
$r_7 : b$



$r_5 = r_5 r_6 r_7 = abb$



$r = r_1 r_2$



5) Construct predictive parser for the following grammar

$E \to TE'$ $E' \to +TE' / \epsilon$ $T \to FT'$ $T' \to *FT' / \epsilon$ $F \to id / (E)$

and parse the i/p string id+id*id

FIRST(E) = { (, id }

FIRST(E') = { +, ε }

FIRST(T) = { (, id }

FIRST(T') = { *, ε }

FIRST(F) = { (, id }

FOLLOW(E) = {FIRST ( ) − ε } ∪ =) − ε =) = { $,) }

FOLLOW(E) = { ε, ε } ∪ { $,) } = { $,) }

FOLLOW(E') $E \to \overset{\alpha}{T} \overset{B}{E'} \beta$

= { ε − ε } ∪ FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) } $E' \to +TE'$

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

$E \to TE'$ FOLLOW(T) = FIRST(E') = { +, ε } − ε ∪ FOLLOW(E)

= { +, $,) }

$E' \to \overset{\alpha}{+} \overset{\beta}{T} E'$ FOLLOW(T) = { +, ε } − ε ∪ FOLLOW(E')

= { +, $,) }

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

$T \to FT'$ $T' \to *FT' / \epsilon$

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

FOLLOW(F) = FIRST(T') - ε ∪ FOLLOW(T) = {*, +, $,)}

	FIRST(α)	Predictive parsing Element
E → TE'	(, id	M[E, (], M[E, id]
E' → +TE' | ε	+	M[E', +]
E' → ε	$,)	M[E', $], M[E',)]
T → FT'	(, id	M[T, (], M[T, id]
T' → *FT'	*	M[T', *]
T' → ε	+, $,)	M[T', +], M[T', $], M[T',)]
F → (E)	(	M[F, (]
F → id	id	M[F, id]

Terminals	+	*	(	)	$	id
E			E → TE'			E → TE'
E'	E' → +TE' | ε			E' → ε	E' → ε	
T			T → FT'			T → FT'
T'	T' → ε	T' → *FT'		T' → ε	T' → ε	
F			F → (E)			F → id

Stack	I/p	O/p
$E'	id + id * id $	E → TE'
$ E T	id + id * id $	T → FT'
$ E' T' F	id + id * id $	F → id
$ E' T' id	id + id * id $	T' → ε
$ E' T'	+ id * id $	E' → +TE'
$ E' T +	+ id * id $	T → FT'
$ E' T' F	id * id $	F → id

$E'T'id	id + id $	T' → * F T'
$E'T'F *	* id $	F → id
$E'T' id	id $	T' → ε
$E'	$	E' → ε
$	$	Successful

6) Construct SLR parsing table for the following grammar i/p string is aab

1) E → E+T 2) E → T, 3) T → TF 4) T → F 5) F → F*

6) F → a 7) F → b

Step1 :- E' → E

Step2 LR(0)

E' → .E (Augmented grammar)

I_0:
- E → .E+T
- E → .T
- T → .TF
- T → .F
- F → .F*
- F → .a
- F → .b

Step3

goto(I_0, E) → I_1:
- E' → E.
- E → E. +T

goto(I_0, T) → I_2:
- E → T.
- T → T.F
- F → .F*
- F → .a
- F → .b

goto(I_0, F) → I_3:
- T → F.
- F → F. *

goto(I_0, a) → I_4: F → a.

goto(I_0, b) → I_5: F → b.

goto(I_1, +) → I_6:
- E → E+.T
- T → .TF
- T → .F
- F → .F*
- F → .a
- F → .b

goto(I_2, F) → I_7:
- T → TF.
- F → F. *

goto(I_2, a) → I_4: F → a.

goto(I_2, b) → I_5: F → b.

goto(I_2, *) → I_8: F → F* .

goto(I_0, T)

$E \to E+T.$
$T \to T.F$
$F \to .F*$
$F \to .a$
$F \to .b$
} I_9

goto(I_0, F)

$T \to F.$
$F \to F.*$
} I_3

goto(I_0, a)

$F \to a.$ } I_4

goto(I_0, b)

$F \to b.$ } I_5

goto(I_1, *)

$F \to F*.$ } I_8

goto(I_9, F)

$T \to TF.*$
$F \to F.*$
} I_7

goto(I_9, a)

$F \to a.$ } I_4

goto(I_9, b)

$F \to b.$ } I_5

Step 4

FOLLOW(E) = { $, + }
FOLLOW(T) = { $, +, a, b }

FIRST(E) = {a, b}
FIRST(T) = {a, b}
FIRST(F) = {a, b}

$E \xrightarrow{E} E \xrightarrow{+} T \xrightarrow{T} F$ FOLLOW(F) = { $, + }

$A \xrightarrow{T} \alpha \xrightarrow{F} \beta$ FOLLOW(F) = { $, +, *, a, b }

$E' \to E.\ I_1$ ② $E \to T.\ I_2$ ④ $T \to F.\ I_3$ ⑥ $F \to a.\ I_4$

⑦ $F \to b.\ I_5$ ③ $T \to TF.\ I_7$ ⑧ $F \to F*.\ I_8$ ① $E \to E+T.\ I_9$

State	Action					goto		
	+	*	a	b	$	E	T	F
0			S_4	S_5		1	2	3
1	S_6			Accept	Accept			
2	r_2		S_4	S_5	r_2			7
3	r_4	S_8	r_4	r_4	r_4			
4	r_6	r_6	r_6	r_6	r_6			
5	r_7	r_7	r_7	r_7	r_7			
6			S_4	S_5		9	2	3
7	r_3	S_8	r_3	r_3	r_3			
8	r_5	r_5	r_5	r_5	r_5			
9	r_1		S_4	S_5	r_1			7

Stack	I/p	Remarks
0	aab$	S_4
0a4	ab$	r_1 F→a pop2
0F1	ab$	r_4 T→F pop2
0T2	ab$	S_4
0T2a4	b$	r_6 F→a pop2
0T2F3	b$	r_3 T→TF pop4
0T2	b$	S_5
0T2b5	$	r_7 F→b pop2
0T2F3	$	r_5 pop4 F→TF
0T2	$	r_2 E→T pop2
0E1	$	Accept

7) Construct CLR and LALR parsing table for following grammar
S→AA A→aA A→b I/p string is aab

CLR

first(E,$) = $

I) $S' \to .S, \$$
$S \to .AA, \$$
$A \to .aA, a/b$
$A \to .b, a/b$ } I_0

goto(I_0,S)
$S' \to S., \$$ } I_1

goto(I_0,A)
$S \to A.A, \$$
$A \to .aA, \$$
$A \to .b, \$$ } I_2

goto(I_0,a) first(ε,c/d) = c/d
$A \to a.A, a/b$
$A \to .aA, a/b$
$A \to .b, a/b$ } I_3

goto(I_0,b)
$A \to b., a/b$ } I_4

goto(I_2,a)
$A \to a.A, \$$
$A \to .aA, \$$
$A \to .b, \$$ } I_6

goto(I_3,b)
$A \to b. $ } I_4

goto(I_3,A)
$A \to aA., a/b$ } I_5

goto(I_3,b)
$A \to b., a/b$ } I_4

goto(I_3, a)

$A \to a.A, a/b$
$A \to .aA, a/b$
$A \to .b, a/b$ } I_3

goto(I_6, A)

$A \to aA., \$$ } I_9

goto(I_6, a)

$A \to a.A, \$$
$A \to .aA, \$$
$A \to .b, \$$ } I_6

goto(I_3, b)

$A \to b., \$$ } I_7

CLR Parsing table

State	Action			goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

(3) $A \to b., a/b$ I_4

(1) $S \to AA., \$$ I_5

(3) $A \to b., \$$ I_7

(2) $A \to aA., a/b$ I_8

(2) $A \to aA., \$$ I_9

State	I/p buffer	Remarks
0	aab$	S3
0a3	ab$	S3
0a3a3	b$	S4
0a3a3b4	$	Error

The given string is not accepted

LALR

	a	b	$	S	A
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_4		
89	r_2	r_2	r_2		

State	I/p buffer	Remarks
0	aab$	S_{36}
0a36	ab$	S36
0a36a36	b$	S47
0a36a36b47	$	r_3 A→b pop 2
0a36a36A89	$	r_2 A→aA pop 4
0A2	$	r_2 A→aA pop 4
	$	Error

Given string is not accepted

8) Discuss about S-attributed and L-attributed SDTs in syntax directed translation

S-attributed

→ It uses only synthesized attribute
→ Semantic actions are placed at right end of a production
Ex: $A \rightarrow BC\{\}$
→ Attributes are evaluated during bottom up parsing
→ It is based on LR grammar
→ These are implemented using LALR parser

L-Attributed

→ It uses both synthesized & inherited attribute
→ Semantic actions are placed anywhere
Ex: $A \rightarrow \{\}BC$, $A \rightarrow BC\{\}$, $A \rightarrow B\{\}C$
→ Attributes are evaluated by translating parse tree to depth first
→ It is based on LL grammar
→ Implemented using predictive parser