CO - Chapter 1

Compiler :

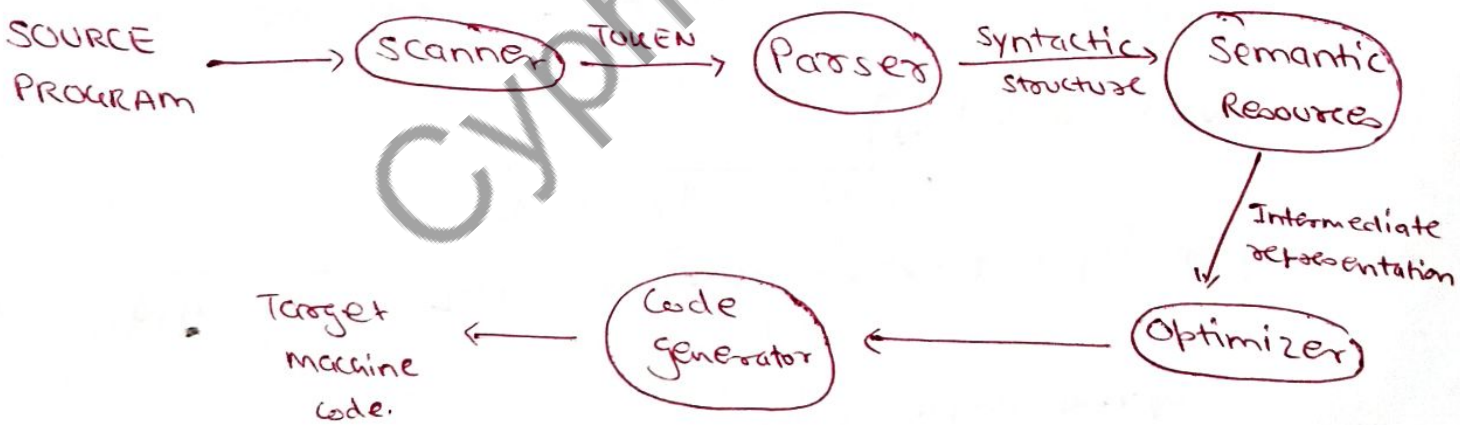
→ translator

Programming Language → Machine Language

Main task

1. Analysis of source program
2. Synthesis of machine lanuage program

Structure: (Phases of CO)



Steps :

1. **Scanner :** It begins the analysis of source code. and group characters into individual words & symbols (tokens)

② Parser: Given a formal syntax, PARSER reads the tokens and group them into units as specified by the productions of CFG

③ Semantic Routines: Performs 2 fxns

Main

↳ Checks static semantics of each construct

↳ Do actual translation.

④ Optimizer: The IR code generated by SR in above step is optimized.

[Peephole optimization]

⑤ Code Generator: Interpretive code generation

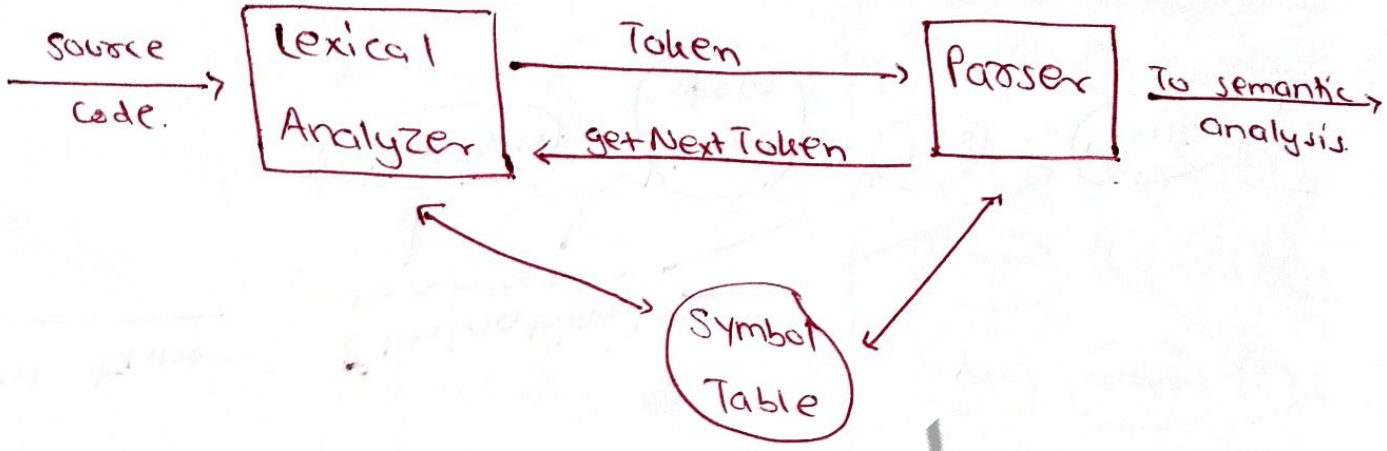
Generating code from tree.

Application of compiler:

- Command language of OS.
- Query languages of DBMS
- Text formatting languages.

Annexure No :

Lexical Analysis:



Tasks:

(i) helps to identify token into the symbol value.

(ii) remove white spaces and comments from source code.

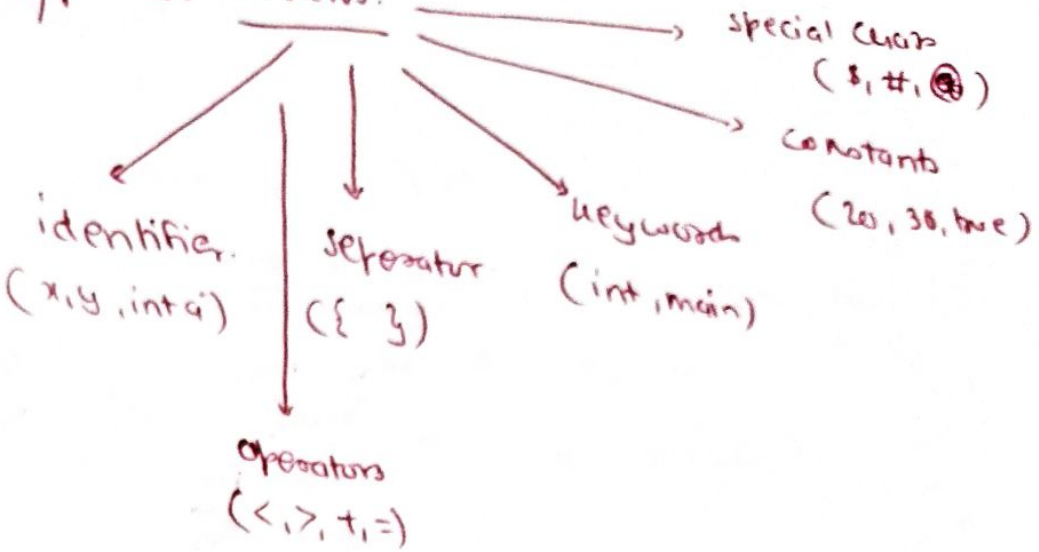
(iii) read i/p characters from source program.

Token
↓
A group of symbols

Pattern
↓
Description of the form that lexemes of token may take.

Lexeme
↓
Sequence of characters in the source program that matches pattern for a token.

Lexical Analyzer takes i/p as stream of characters and gives out o/p as tokens.



inputs of code are taken as lexeme.

Second task: Give Error Messages

- Exceeding length
- Unmatched string
- illegal character.

Third tasks:

Elimination:

- Comments
- White spaces
- tabs
- newline

Annexure No :

eg. (i)

```
int main ()
{
/* find max of a & b */
int a=20, b=30;
  if (a<b)
     return (b);
  else
     return (a);
}
```

Count tokens :

int main ( ) { — Comment removed —

int a = 20 , b = 30 ;

if (a < b)

return (b) ;

else return (a) ;

}

32 tokens.

(ii) `printf("i = %d, &i = %x", i &i);`

NOTE: tokens in inverted commas are considered to be one.

printf (" i = %d, &i = %x " , i & i) ;

8 tokens

(iii) `a = b+++---++ +==;`

a = b ++ + - -- ++ += = ;

11 token.

First & Follow:

First(A) → Contains all terminals present in first place of every string derived by A.

(1) S → abc | def | ghi

first(S) => a, d, g

(2) S → ABC | ghi | jkl

A → a | b | c

B → b

D → d

f(D) => d

f(B) => b

f(A) = a, b, c

f(S) = A B C | g hi | jkl

↓ ↓ ↓

a b c g hi jkl

(3) S → ABC

A → a | b | ε

B → c | d | ε

C → e | f | ε

f(C) = e, f, ε

f(B) = c, d, ε

f(A) = a, b, ε

f(S) = A B C

↓ ↓ ↓

f(A) f(B) f(C)

a b c d e f ε

∴ f(S) = a, b, c, d, e, f, ε

(4) $E \rightarrow TE'$

$E' \rightarrow *TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow \varepsilon \mid +FT'$

$F \rightarrow id \mid (E$

$f(F) = id, ($

$f(T') = \varepsilon, +$

$f(T) = f(F)$

$\Downarrow$

$id, ($

$f(E') = *, \varepsilon$

$f(E) = f(T)$

$\Downarrow$

$id, ($

Follow ( )

↳ Contains set of all terminals present immediate in right of 'A'

Rules:

① follow of start symbol is "$"

② follow of ε in left side goes to the follow of start symbol / follow of right character.

Eg: $S \rightarrow ACD$

$C \rightarrow a \mid b$

$fo(A) = first(C) = \{a, b\}$

$fo(D) = follow(S) = \{\$\}$.

To find follow, we need to know the first of the upcoming terminal.

Q. (2) $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

Note: follow never contains ε

$Fo(S) \Rightarrow (S) \rightarrow a(S)b(S) \mid b(S)a(S)$

- $\Downarrow$ → $ (already done)
- $\Downarrow$ → first(b) → $\Downarrow$ → b
- $\Downarrow$ → first(a) → $\Downarrow$ → a

$\therefore Fo(S) \Rightarrow \$, b, a$

(3) $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

$Fo(A) \Rightarrow a, b$

$Fo(B) \Rightarrow b, a$

(4) $S \rightarrow ABC$

$A \rightarrow DEF$

$B \rightarrow \varepsilon$

$C \rightarrow \varepsilon$

$D \rightarrow \varepsilon$

$E \rightarrow \varepsilon$

$F \rightarrow \varepsilon$

$Fo(A) = first(B)$ but it ε

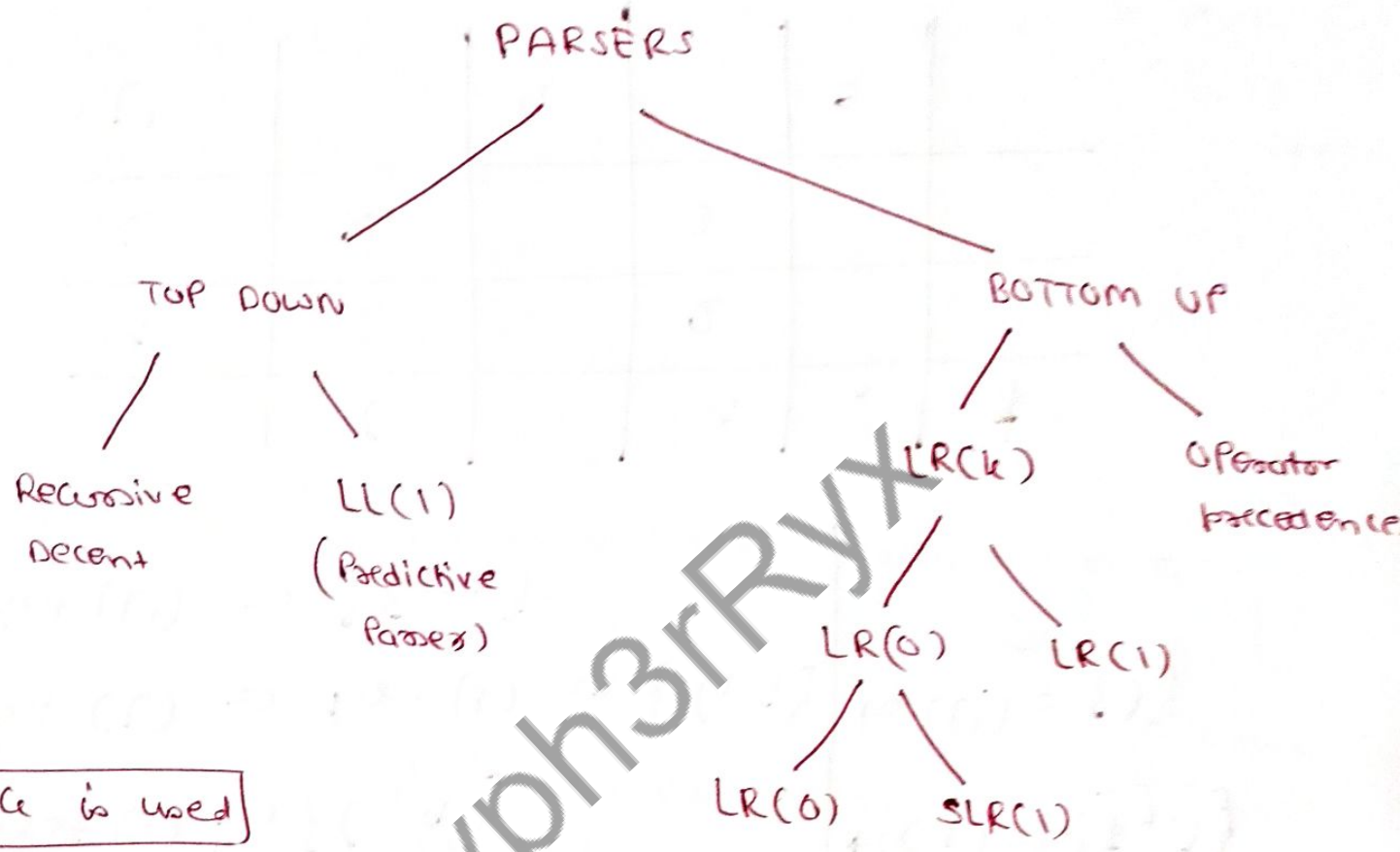
$\therefore first(C)$

but it ε

$\therefore follow(S)$

$\Rightarrow \$$

Parsing: Process of deriving string from a given grammar

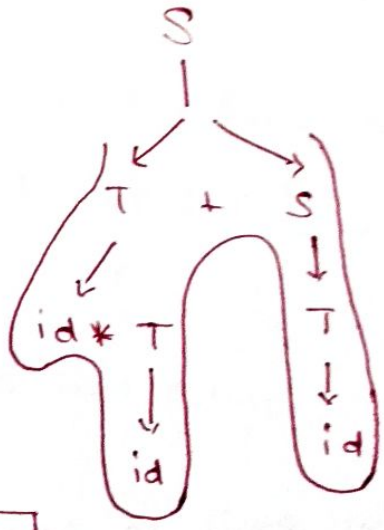


CFG is used

$S \rightarrow T + S \mid T$

$T \rightarrow id * T \mid id \mid S$

$w = (?)$



$w = id * id + id$

How to checck that particular grammar is LL1?

See that no more than 1 value is there in the parsing table.

PARSING TABLE:

eg. $S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow \varepsilon \mid ,SL'$

first (S) $\Rightarrow \{(, a\}$

first (L) $\Rightarrow$ first (S) $\Rightarrow \{(, a\}$

first (L') $\Rightarrow \{\varepsilon, ,\}$

fo(L) = {)}

fo(S) $\Rightarrow \{\$, ,,)\}$

fo (L') = {)}

	(	)	a	,	$
S	1		2		
L	3		3		
L'		4		5	

Prodxn ① $S \to L$ ⟹ RHS ka first nikalo i.e (

∴ mark ① in "(" column.

" ② $S \to a$

" ③ $L \to SL'$

" ④ $L' \to \varepsilon$

" ⑤ $L' \to ,SL'$

for $S \to a$ ⇒ RHS ka first nikalo. i.e a

∴ mark ② in "a" column

for $L \to SL'$ ⇒ SL' ka first nikalo.

i.e "S" ka ⟶ $S \Rightarrow \{(, a\}$ ∴ mark ③ in "(" & "a"

for $L' \to \varepsilon$ ⇒ in case of "ε" do the follow of LHS

∴ fo(L) = {)} ∴ mark ④ in {)}

for $L' \to ,SL'$ ⇒ ,SL' ka first nikalo.

i.e "," ∴ mark ⑤ in {,}

As there is single single entry in each box

∴ Grammar is LL1 ✓

Eg. ② $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

$f(S) = \{a, b, \varepsilon\}$

$fo(S) = \{a, \$, b\}$.

	a	b	$
S	1/3	2/3	3

Prodxn ① $\Rightarrow$ $S \rightarrow aSbS$

$\therefore$ find first of aSbS i.e "a" [mark ① in a]

Prodxn ② $\Rightarrow$ $S \rightarrow bSaS$

$\therefore$ find first of bSaS i.e "b" [mark ② in b]

Prodxn ③ $\Rightarrow$ $S \rightarrow \varepsilon$

in case of ε ; find follow of LHS

i.e $fo(S) \Rightarrow \{a, b, \varepsilon\}$

[$\because$ mark ③ in (a, b, ε)]

Here 2 values harbour in same box thus it doesn't follow the rule and that is why it is not LL'.

LL(1) Parser:

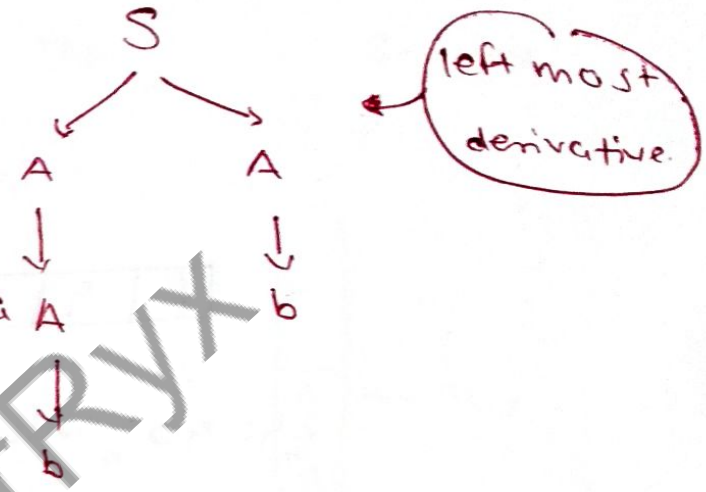
- Top down Parser
- STACK structure

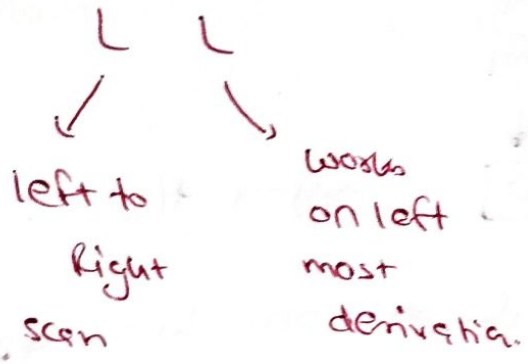
eg.

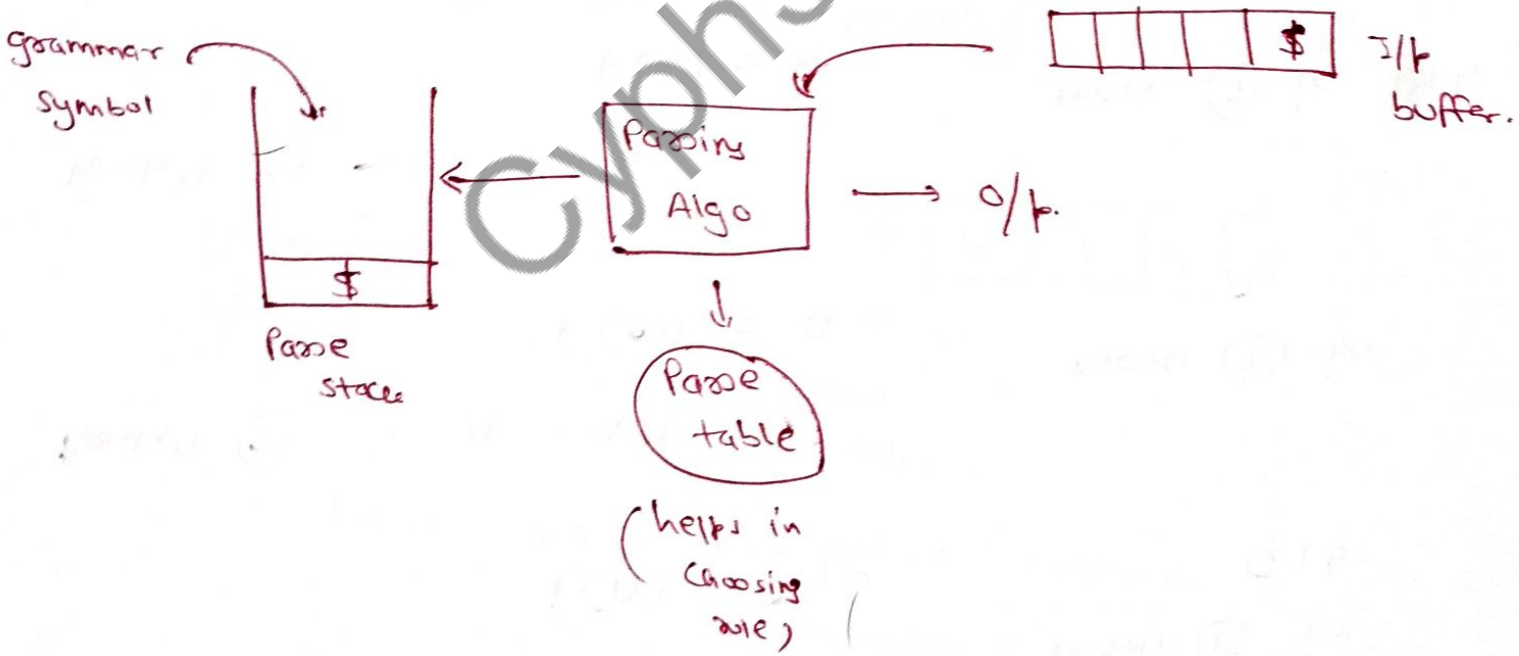
$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$







eg. $S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

	a	b	$
S	1	1	
A	2	3	

Prodxn (1) → $S \rightarrow AA$

$f(A) = a, b$

mark (1) in a, b

Prodxn (2) → $A \rightarrow aA$

$f(aA) = a$

mark (2) in a

Prodxn (3) → $A \rightarrow b$

$f(b) = b$

mark (3) in b

let say we are given a string "abab"

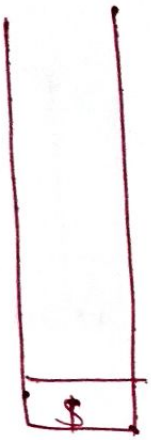
i/p buffer

a	b	a	b	$

look ahead

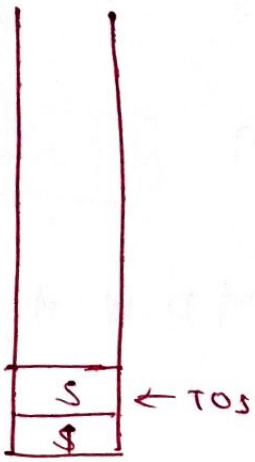
Annexure No :

S-1 =



By default

S-2 =

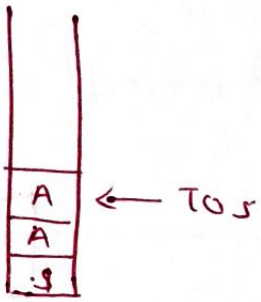


Put S in TOS

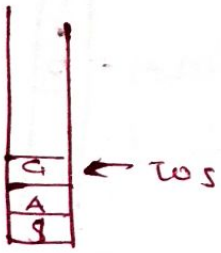
S-3 = see the grammar for S & A in ∴ parse table & ~~any~~ grammar is

S → AA

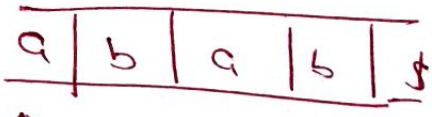
∴



S-4 = Now for A & a we have prodxn (2) A → aA



So now

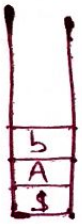


Look ahead

Now TOS & look ahead matches

∴ remove them & move look ahead "+1".

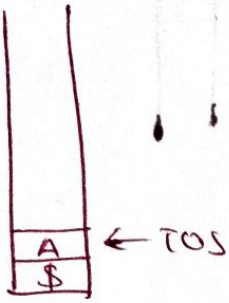
S-5 =

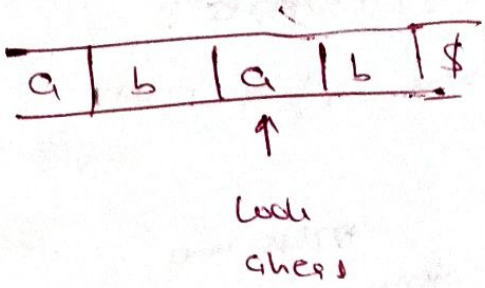


Now b & b matches

∴ repeat (S-4).

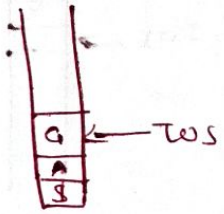
S-6





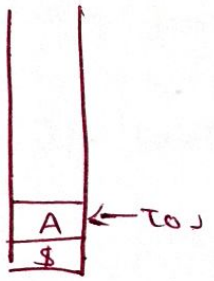
A G : Prodxn (2).

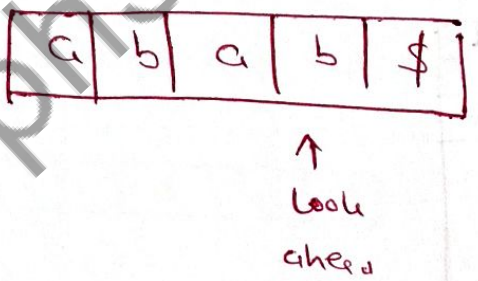
A → a A



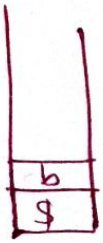
a a → matches : go ahead.

S-7



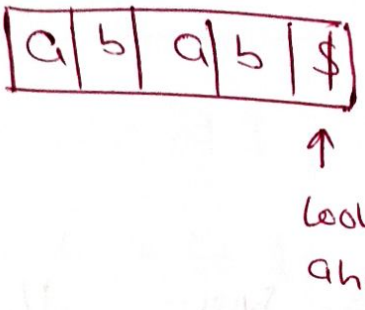


for next look ahead pop A & push b



Look ahead matches : pop.

S-8



Stack & look ahead matches.

∴ The string given is perfect for LL' parsing

Eg.2 Construct LL1 parsing table

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

- first remove left recursion
- left factoring.

	first	follow
$S \rightarrow AaAb \mid BbBa$	a, b	$
$A \rightarrow \varepsilon$	ε	a, b
$B \rightarrow \varepsilon$	ε	b, a

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	
B	$B \rightarrow \varepsilon$	$B \rightarrow \varepsilon$	

eg.

	First	Follow
$E \to TE'$	{id, (}	{$,)}
$E' \to \varepsilon \mid +TE'$	{ε, +}	{$,)}
$T \to FT'$	{id, (}	{+, $,)}
$T' \to \varepsilon \mid *FT'$	{ε, *}	{+, $,)}
$F \to id \mid (E)$	{id, (}	{*, +, $,)}

	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'			$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

let the i/p string be "id + id $"

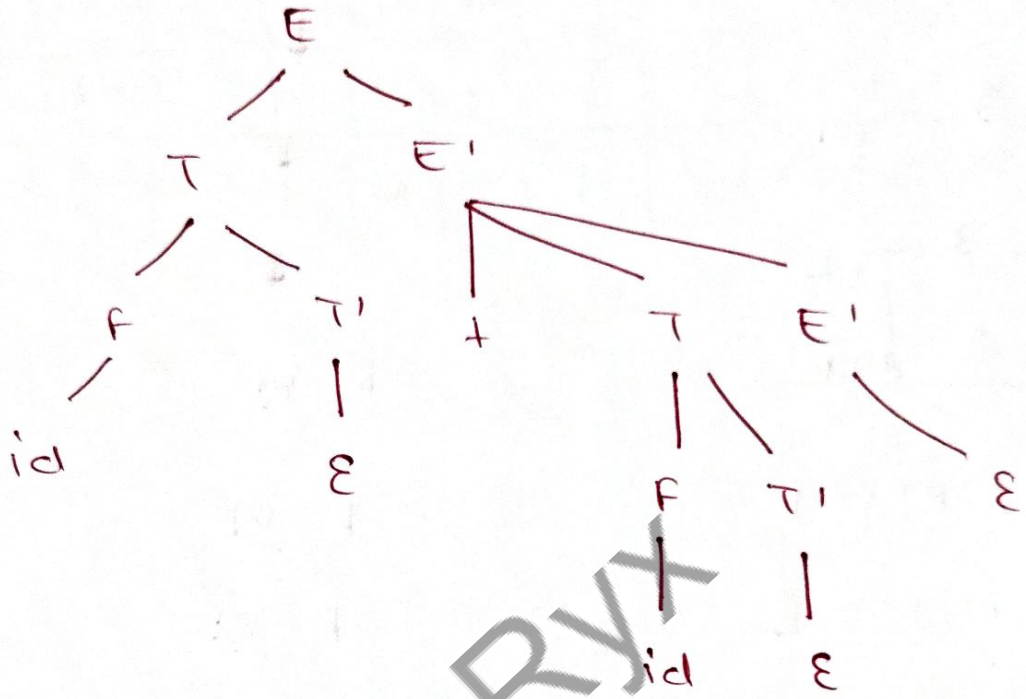
Annexure No :

Stack	i/p	Prodxn.
$ E	id + id $	E → TE'
$ E' T	id + id $	T → FT'
$ E' T' F	id + id $	F → id
$ E' T' id	id + id $	POP
$ E' T'	+ id $	T' → ε
$ E'	+ id $	E' → +TE'
$ E' T +	+ id $	POP
$ E' T	id $	T → FT'
$ E' T' F	id $	F → id
$ E' T' id	id $	POP
$ E' T'	$	T' → ε
$ E'	$	E' → ε
$	$	

Accepted.

Construct

Parse tree.



eg (2)

	first	follow
S → aBDh	{a}	{$}
B → cC	{c}	{g, f, h}
C → bC | ε	{b, ε}	{g, f, h}
D → EF	{g, f, ε}	{h}
E → g | ε	{g, ε}	{f, h}
F → f | ε	{f, ε}	{h}

	a	b	c	f	g	h	$
S	S→aBDh						
B			B→c, ε				
C		C→bC		C→ε	C→ε	C→ε	
D				D→EF	D→EF	D→EF	
E				E→ε	E→g	E→ε	
F				f→f		F→ε	

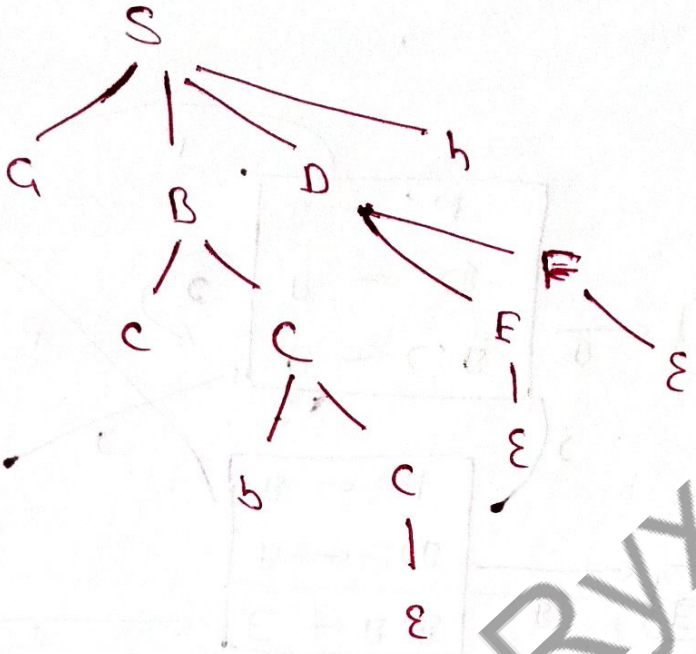
let i/p string be " acbh $ "

stack	i/p	Production
$ S	acbh $	S → aBDh
$ hDBa	acbh $	POP
$ hDB	cbh $	B → cC
$ hDCc	cbh $	POP
$ hDC	bh $	C → bC
$ hDCb	bh $	POP
$ hDC	h $	C → ε
$ hD	h $	D → EF
$ hFE	h $	E → ε
$ hF	h $	F → ε
$ h	h $	POP
$	$	

Accepted

Annexure No :

Parse tree:



Accepted

LR(0) Parser:

S-1 = Augment Grammer.

S-2 = Draw Conical Collection of LR(0)

S-3 = No. of prodxn

S-4 = Create Parsing table

S-5 = Stack implementation

S-6 = Draw Parsing tree.

eg. $E \rightarrow BB$

$B \rightarrow cB \mid d$

String : ccdd $

∴ (1) $E \rightarrow BB$

$E' \rightarrow E$

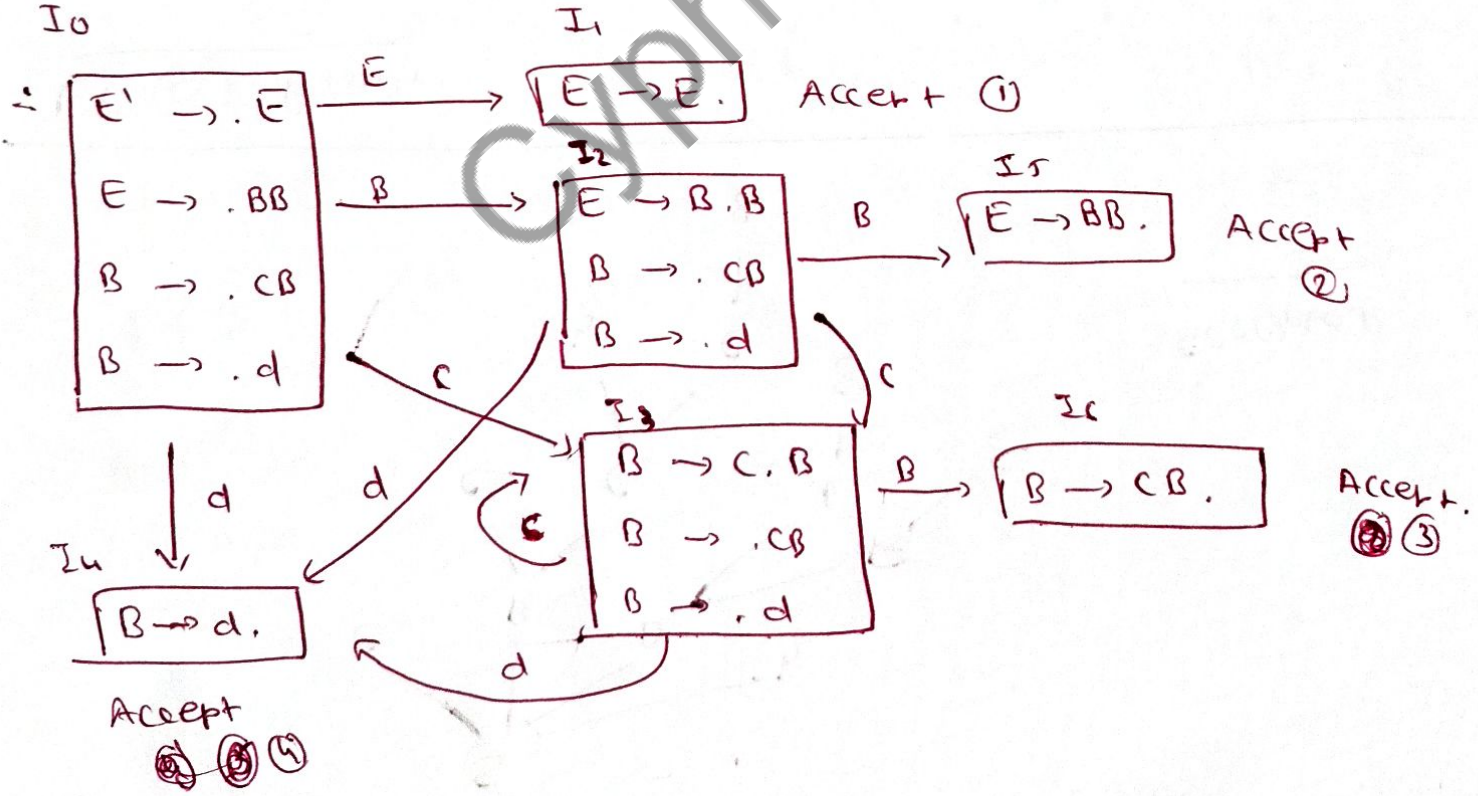
$B \rightarrow cB \mid d$

(2) $E' \rightarrow .E$

$E \rightarrow .BB$

$B \rightarrow .cB$

$B \rightarrow .d$



(3) $E' \rightarrow E$

$E \rightarrow BB$

$B \rightarrow CB$

$B \rightarrow d$

(4) Passing table:

State	Action			GoTo	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Based on this make a state and parse table.

Stack	i/p	Action.
$ 0	CCdd$	Shift C to stack & goto S3.
$0C3	Cdd$	Shift C and S3.
$0C3C3	dd$	Shift d and S4
$0C3C3d4	d$	Reduce r3, B→d
$0C3C3B6	d$	Reduce r2, B→CB
$0C3B6	d$	Reduce r2, B→CB
$0B2	d$	Shift d, S4
$0B2d4	$	Reduce r3, B→D
$0B2B5	$	Reduce r1, E→BB
$0E1	$	Accept

Annexure No :

SLR-(1)

Do everything same as LR(0) but what we need to change here is that we don't need to write reduce form (r_1, r_2) in every column, but instead we need to write the reduce only in the follow of that LHS.

Previous eg.

State	Action			GOTO	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

LR(0) table.

SLR(1) table:

State	Action			GoTo	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5			r_1		
I_6	r_2	r_2	r_2		

$E \rightarrow BB$

$B \rightarrow cB \mid d$

$I_4 \Rightarrow B \rightarrow d$ (r_3)

$fo(B) = \{c, d, \$\}$

$I_5 \Rightarrow E \rightarrow BB$ (r_1)

$fo(E) = \{\$\}$

$I_6 \Rightarrow$ (r_2) $\rightarrow B \rightarrow cB$

$fo(B) = \{c, d, \$\}$

Conflict :

Reduce - Reduce conflict

LR(0) => ✓

SLR(1) => X

Shift - reduce conflict

LR(0) => ✓

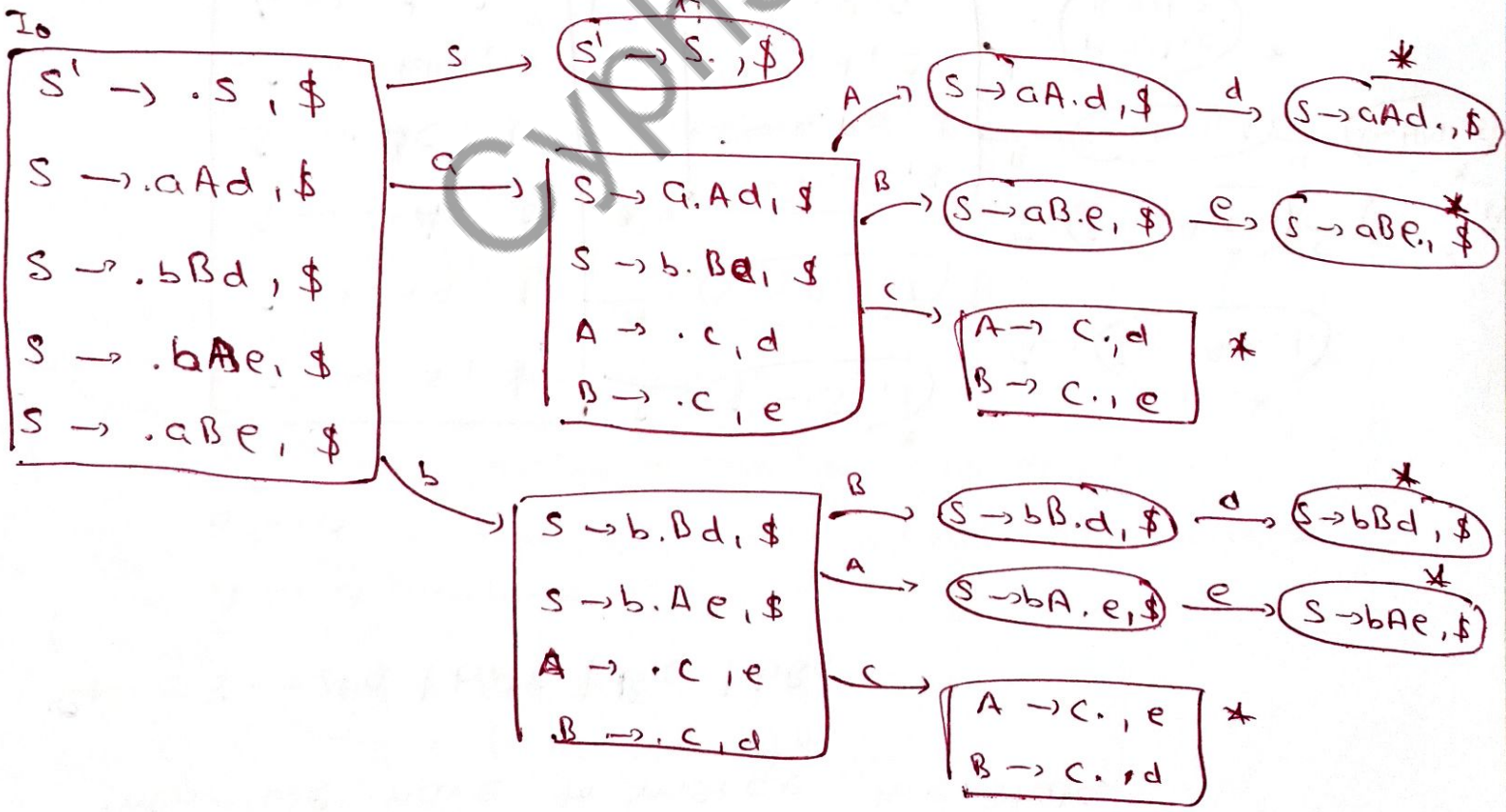
SLR(1) => X

CLR

S → aAd | bBd | aBe | bAe

A → c

B → c



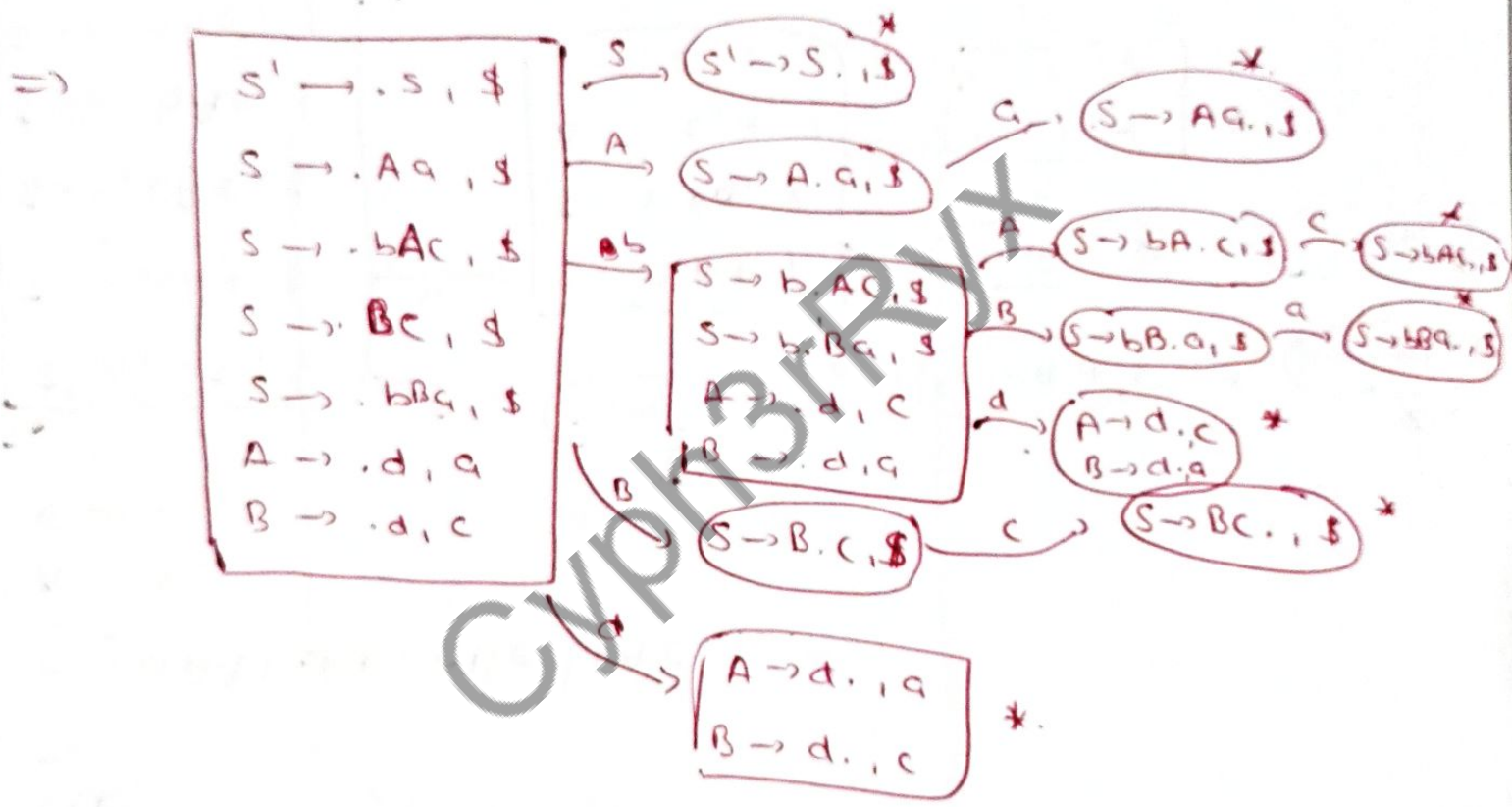
For LALR, we have to the same as CLR

then we have to merge the states

eg $S \to Aa \mid bAc \mid Bc \mid bBa$

$A \to d$

$B \to d$

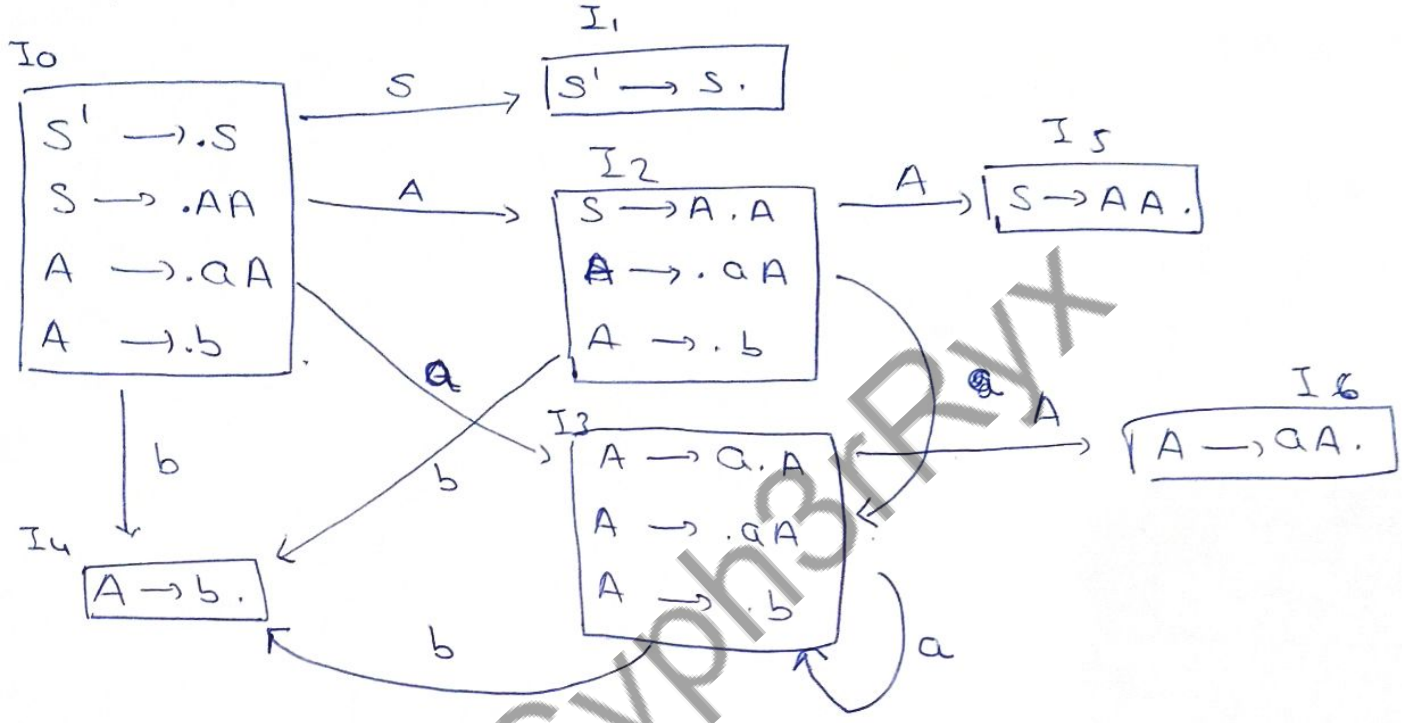


Now can we join them? We can but it will create multiple entries in the table and that causes a confusion and if multiple entries are seen that means it is not LALR(1) but CLR grammar.

LR(0)

$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

$r_1 \Rightarrow S \rightarrow AA$
$r_2 \Rightarrow A \rightarrow aA$
$r_3 \Rightarrow A \rightarrow b$



	ACTION			GOTO	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

Annexure No :

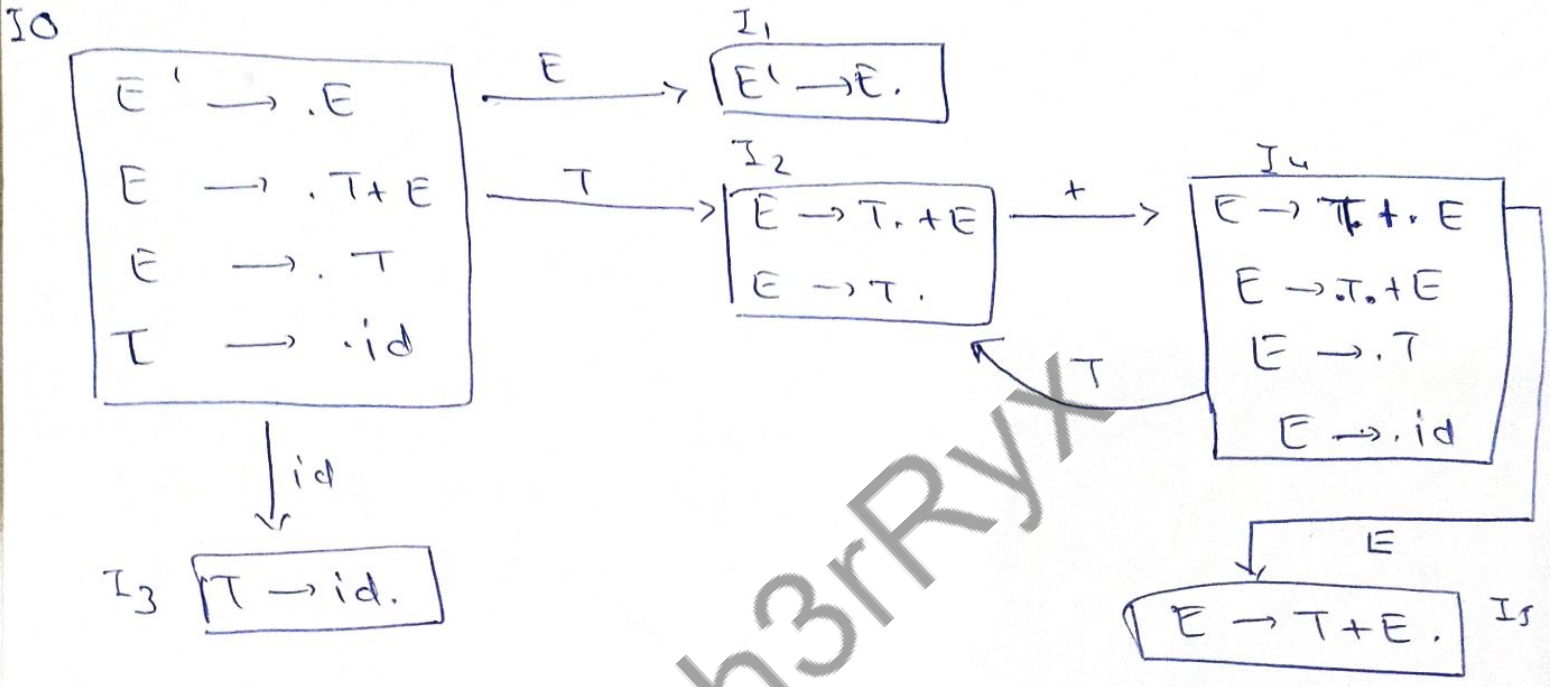
SLR(1) [Calculate the follow of LHS of $r_1, r_2, r_3 \ldots$]

$E \rightarrow T+E \mid T$

$T \rightarrow id$

$FO(E) \rightarrow \{\$\}$

$FO(T) \rightarrow \{\$, +\}$



	ACTION			GO TO	
	id	+	$	E	T
0	S_3			1	2
1			Accept		
2		S_4	r_2		
3		r_3	r_3		
4				5	2
5			r_1		

lookahead → (First)

result → (Follow)

Parul® University

Faculty of Engineering & Technology
Subject Name:
Subject Code:
B.Tech. ___Year ____ Semester____

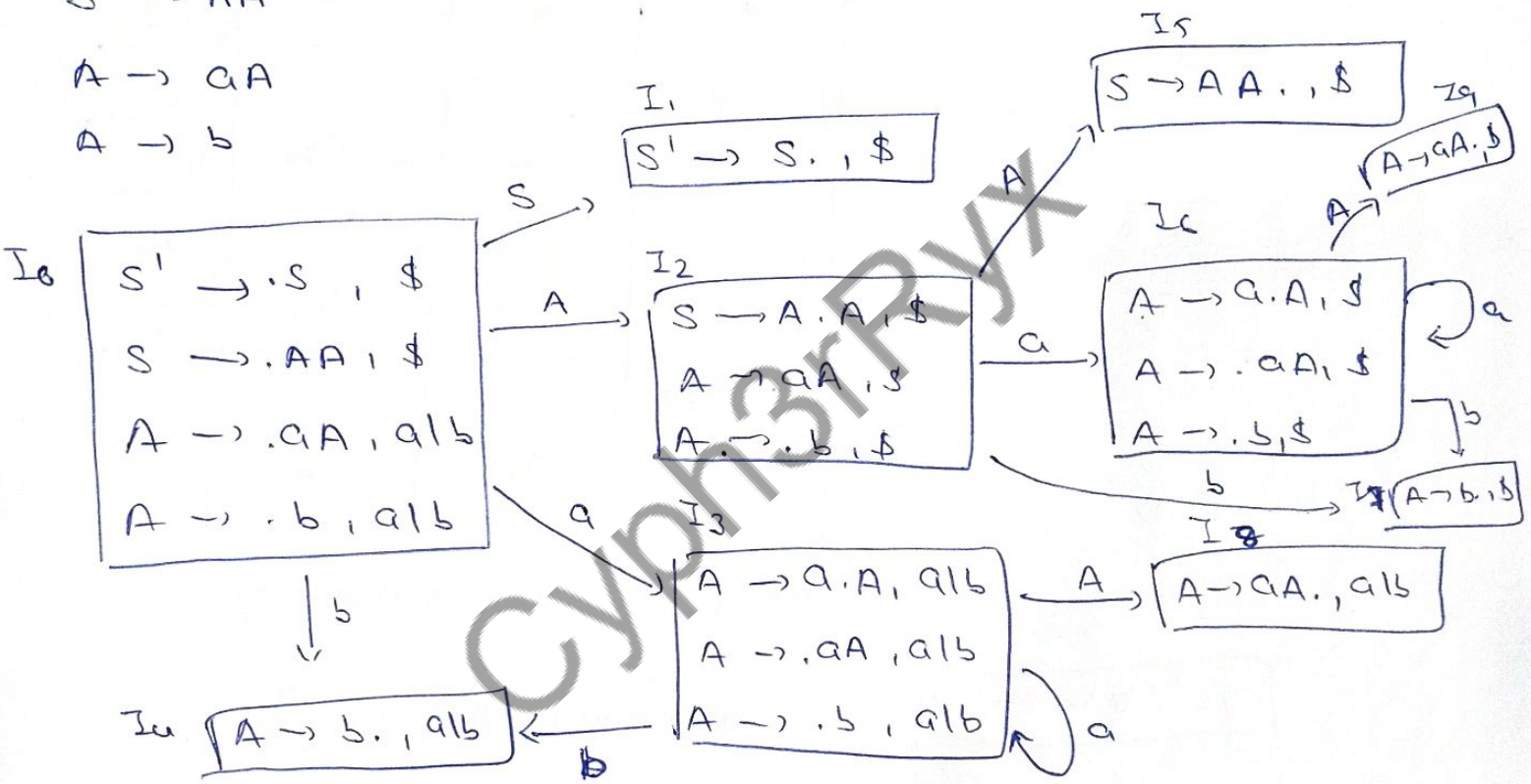
Annexure No :

CLR(1)

$S \rightarrow AA$
$A \rightarrow aA \mid b$

$f(A) = \{a, b\}$
$f(S) = \{\$\}$ } → lookahead.

$S' \rightarrow S$
$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$



	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		

LALR(1) (Merge 2 entries with same rules & diff. lookaheads)

In last problem I_3 & I_6 were same.

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

∴ merge them

same goes for I_4 & I_7 ; I_8 & I_9 ⟶ $A \to aA., a/b$

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

∴ merge all in Parsing table

	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

SDD → Syntax Directed Definition.

Prodxn	semantic rule
E → E + T	E.val = E.val + T.val
E → T	E.val = T.val

- Attributes are associated w/ grammar symbols & semantic rules are associated with prodxns.
- Attributes maybe numbers, strings, reference datatypes, etc.

eg.

Prodxn	Rule
E → E # T	E.val = E.val * T.val
E → T	E.val = T.val
T → T & F	T.val = T.val + F.val
T → F	T.val = F.val
F → digit	F.val = digit.val

Q. ⇒ 10 # 8 & 6 # 9 & 4 # 5 & 2 = ?

A. ⇒ 10 × 8 + 6 × 9 + 4 × 5 + 2

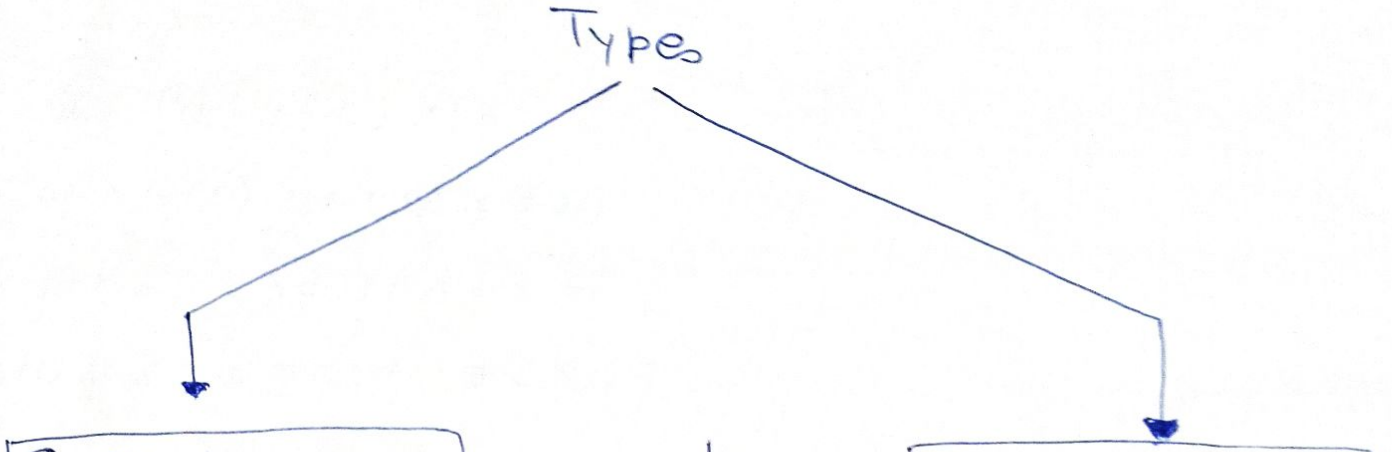
∴ 10 × (8+6) × (9 + 4) × (5 + 2)

∴ 10 × (14) × (13) × (7)

= 140 × 13 × 7

= 12740

SDD Types:



S-Attributed SDT

- Based on synthesized attribute
- Use Bottom Up passing.
- Semantic rules always written at Rightmost Position in R.H.S.

L-Attributed SDT

- Based on Both synthesized & inherited attribute (Parent, Left sibling)
- Top down Passing
- Semantic rules anywhere in R.H.S.

$A \rightarrow XYZW$

if A is taking value from XYZW
then A is synthesized attribute
but if vice versa then it is inherited
even Y takes value from X then it is
considered as inherited [∵ sibling]

SDD:

if grammar is (L-) or (S-)

(L-) → Synthesized & inherited

(S-) → Synthesized

eg $A \rightarrow PQ$ $\{P = A+Z, Q = P+A, A = P+Q\}$

Here A = Parent ; PQ = Child

P = A + Z ; Child ; from parent (inherited)

Q = P + A ; ; from Child (inherited)

A = P + Q ; Child ; from parent (synthesized)

Here A = Parent P & Q = Child

: P = A + Z ; Child taking val. from parent (inherited)

: Q = P + A ; " (inherited)

: A = P + Q ; Parent taking val. from Child (synthesized)

inherited + synthesized = L - Attribute

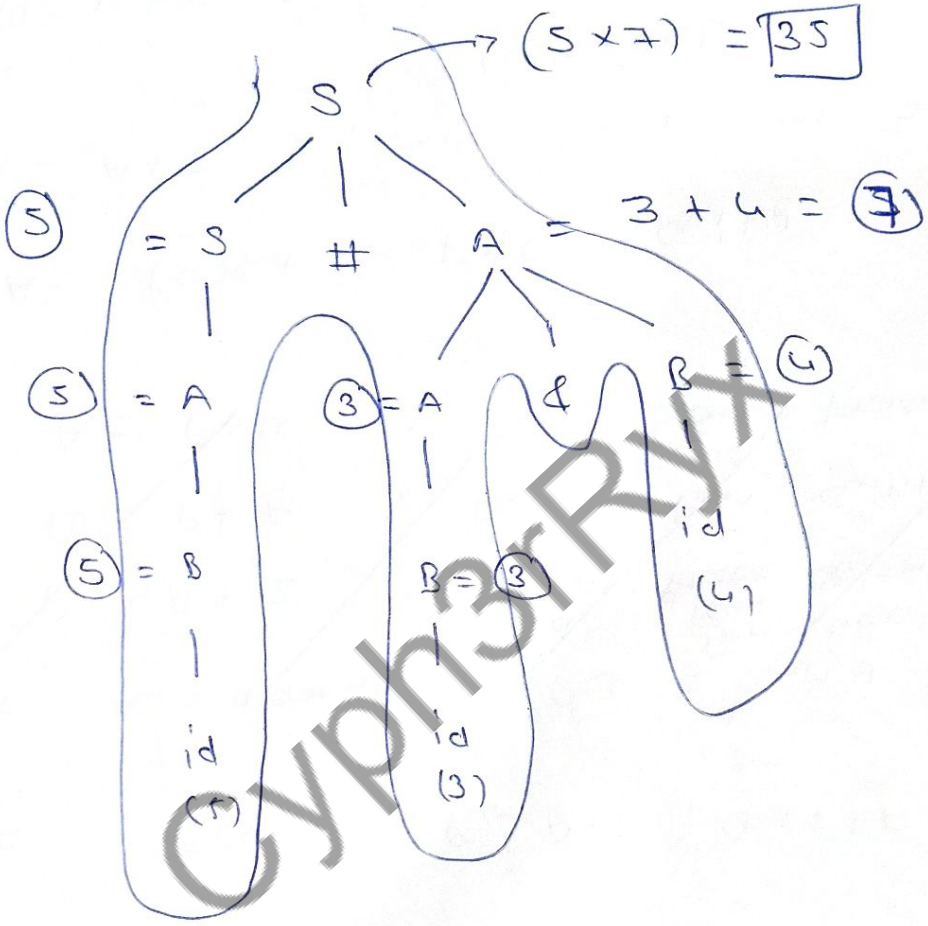
Enrollment No :

$S \longrightarrow S \# A \mid A$ { S.val = S.val * A.val ; }

$A \longrightarrow A \& B \mid B$ { A.val = A.val + B.val ; }

$B \longrightarrow id$ { B.val = id', }

grammar S # 3 & 4



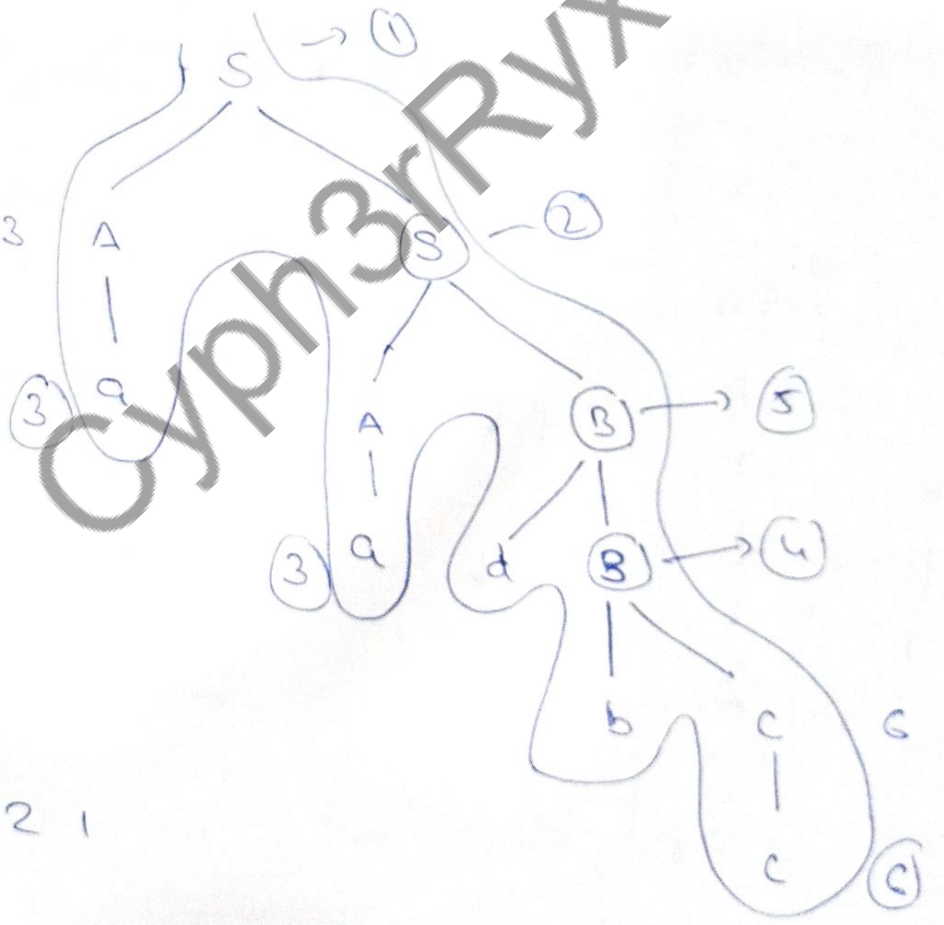
Here it goes according to DMAS.

Q.2

=

$S \rightarrow AS$	{printf(1);}
$S \rightarrow AB$	{ " (2);}
$A \rightarrow a$	{ " (3);}
$B \rightarrow bC$	{ " (4);}
$B \rightarrow dB$	{ " (5);}
$C \rightarrow c$	{ " (6);}

i/p = aadbc



o/p. 3 3 6 4 5 2 1

Dependency graph:

$L \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

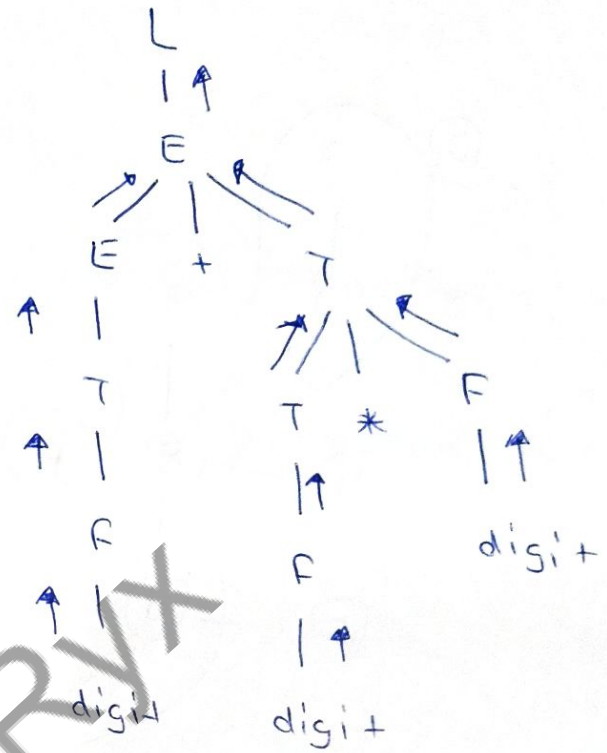
$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow (digit)$

i/p = digit + digit * digit



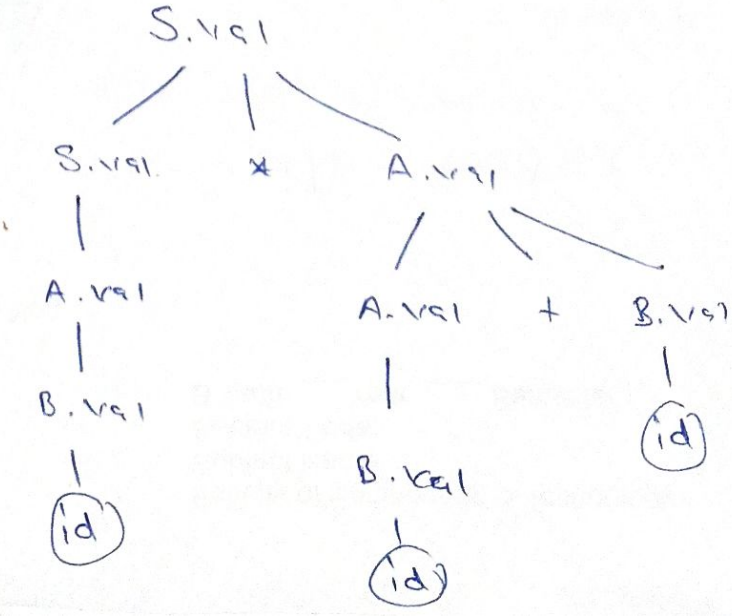
Annoted Parse tree :

$S \rightarrow S\#A \mid A$ {S.val = S.val * A.val ; S.val = A.val ;}

$A \rightarrow A\&B \mid B$ {A.val = A.val + B.val ; A.val = B.val ;}

$B \rightarrow id$ {B.val = id}

i/p ⇒ id # id & id



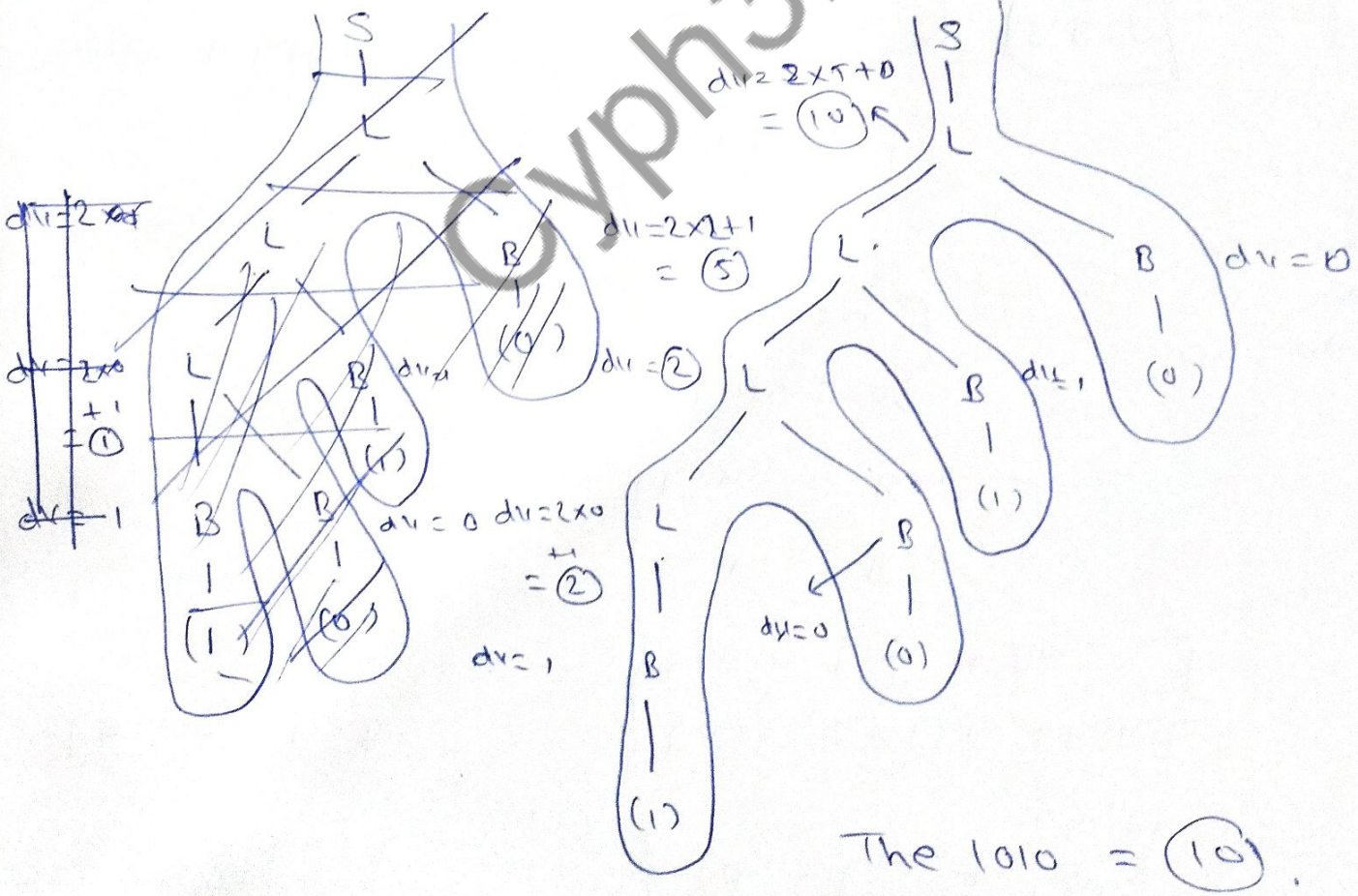
SDD Binary to decimal:

Prodxn	Rule.
$S \rightarrow L$	$S.dv = L_{dv}$
$L \rightarrow LB$	$L_{dv} = 2 \times L_{dv} + B_{dv}$
$L \rightarrow B$	$L_{dv} = B_{dv}$
$B \rightarrow 0$	$B_{dv} = 0$
$B \rightarrow 1$	$B_{dv} = 1$

eg. take

1010

& convert it into D.V (decimal value)



The 1010 = (10).

Parul® University

Faculty of Engineering & Technology
Subject Name:
Subject Code:
B.Tech. ___Year ____ Semester___

Annexure No :

Intermediate Code Generation :

Peephole Optimization :

By 5 methods we can optimize the code . ,

(1) Redundant Load & Store :

↳ Remove the unnecessary loading & storing ofn.

eg. $a = b + c$
$d = a + e$

```
mov  b, R0
Add  c, R0
mov  R0, a   ]
mov  a, R0   ]  → used 2 times
                  ∴ remove one line
Add  e, R0
mov  R0, d
```

(2) Strength Reduction :

↳ reduce the strength of code.

eg. $x^2 \Rightarrow x * x$

$x * 2 \Rightarrow$ left shift

$x / 2 \Rightarrow$ Right shift .

(3) Simplify Algebraic Expressions:

```
a = a+0
a = a*1
a = a/1
a = a-0
```

} This all are unnecessary codes returning the same value

(4) Replace slower instruction w/ faster:

Add #1 , R => INC R

Sub #1 , R => DEC R

(5) Deadcode Elimination:

↳ Any code that is not participating in the o/p → remove it

```
int dead (void)
{
    int a = 10;
    int b = 20;
    int c;
    c = a*5;
    return c;

    b = 30;
    b = b*10;
    return 0;
}
```

→ working

→ not working
= remove it.

CODE OPTIMIZATION

Platform Dependent Techniques

1. Peephole optimization
2. Instruction level parallelization
3. Data level parallelism
4. Cache optimization
5. Redundant resources

Platform Independent Techniques

1. Loop optimization
   - loop unrolling
   - loop jamming
   - code movement
   - frequency reduction
2. Constant folding
3. Constant propagation
4. Common sub-expression elimination

LOOP OPTIMIZATION:

loop takes too much time while in the code running : optimizing it is necessary.

(1) Code motion (frequency reduction):

```
a = 100;
while (a>0);
{   x = y+2;
    if (a % x == 0);
    printf("%d", a);
}
```

x = y+2 is a constant term so taking it out of loop will make it easy to not go on for 99 times in the code.

(2) Loop fusion:

↳ combining the loops in one loop to optimize the outcome. [just don't change the meaning of code]

eg.

```
int i, a[100], b[100]
for (i=0; i<100; i++)
    a[i]=1;
for (i=0; i<100; i++)
    b[i]=2;
```

$O(n^2)$

→

```
int i, a[100], b[100]
for (i=0; i<100; i++)
    a[i]=1;
    b[i]=2
```

$O(n)$

(3) Loop unrolling:

↳ lower the no. of times a statement is executing

```
for (i=0; i<5; i++)
printf("varun");
```

$O(n)$

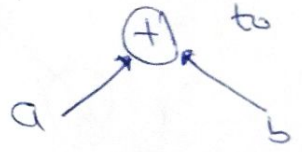
```
Printf("varun");
Printf("varun");
Printf("varun");
Printf("varun");
Printf("varun");
```

$O(1)$

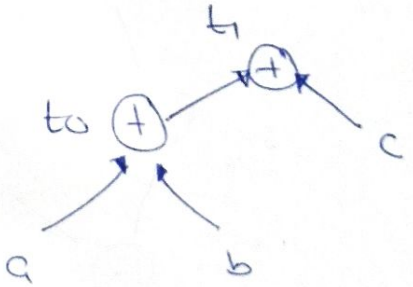
DAG (Directed Acyclic Graph):

- shows the structure of basic blocks
- helpful to understand the flow of value in basic blocks.
- provides optimization.
- leaf node = identifier, variable name, constant.
  intermediate node = operators

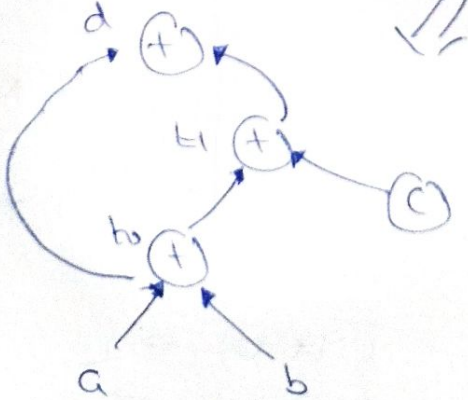
$t_0 = a + b$,
$t_1 = t_0 + c$
$d = t_0 + t_1$



$[t_0 = a + b]$



$[t_1 = t_0 + c]$



$[d = t_0 + t_1]$

Construct a DAG for expression

$T = (a * b) + (a * b) / (a * b)$

$T_1 = a * b$

$T_2 = a * b$

$T_3 = T_1 / T_2$

$T_4 = a * b$

$T_5 = T_4 + T_3$.

$T_5 = (T_4) + (T_3)$

$T = (a * b) + [(a * b) / (a * b)]$

(T_1) (T_2)

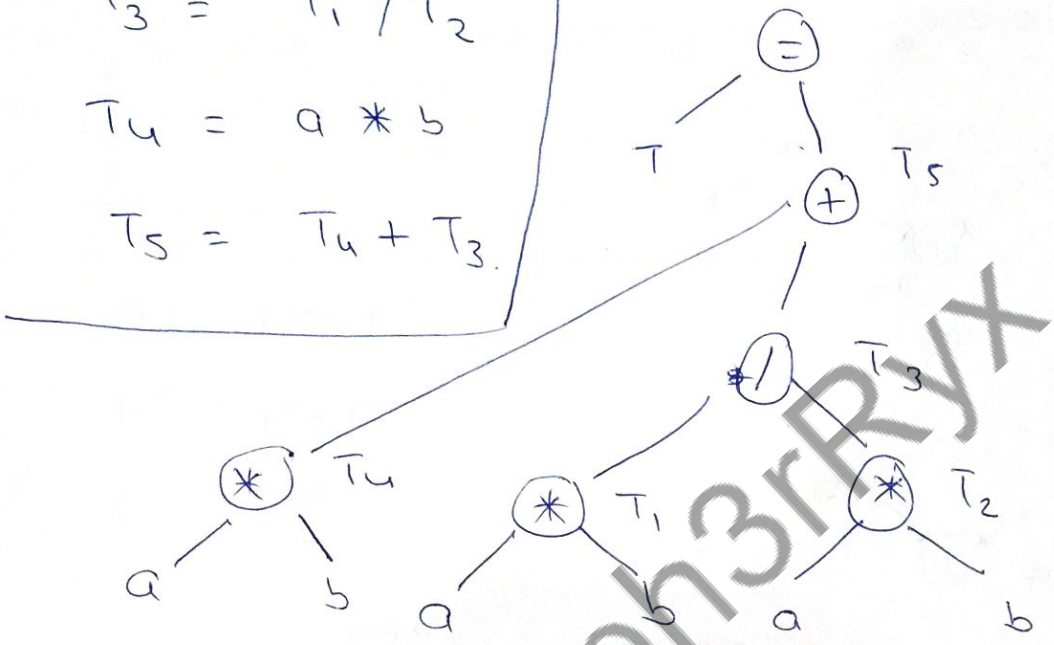
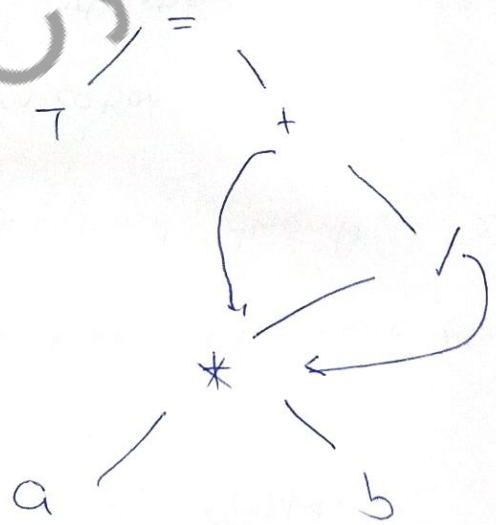


Table:



MCQ & FITB:

1. YACC = Yet Another Compiler Compiler.
2. Lexical & Syntax Analyzer = front end of compiler.
3. Grammar has > 1 Parse tree = Ambigious
4. P, Q, & R are total no. of states in LR(0), SLR(1) & CLR(1)
   then P < Q < R

- Operator grammar has no null prodxn.
- Top down parser is not applicable for left recursive grammar.
- Lexical Analysis can eliminate white space.

Compiler	Interpreter
(1) Scans whole code.	(1) Translates one statement at a time
(2) Converts source code → object code	(2) dont converts but scan it line by line
(3) don't require source code for execution	(3) requires source code for execution
(4) machine code stored in disk	(4) machine code is not stored anywhere
(5) CPU utilization = high	(5) CPU utilization = less
(6) C, C++, C#	(6) Python, Ruby, Perl
(7) more efficient.	(7) less efficient
(8) fast execution	(8) slow execution

Cross compiler : Runs on one platform ⟶ Produces executable code on other platform.

eg. Compiler on windows produces .exe code for linux.

Left factoring : used in grammar transformation

why?

to eliminate common prefixes in alternative prodxn of a non terminal

Error Handler : resolving & detecting errors.

→ error (detected) → takes action to eliminate it.

Ambigious grammar : can generate more than one parse tree for i/p string.

→ can create errors ∴ left factoring & operator precedence is used

---

Syntax Analysis = checks grammar of program

DAG = Directed Acyclic Graph

- LL(1) = TOP DOWN
- LALR > SLR
- Ambig. can't be LR(1) Grammar

LR = left to right & rightmost derivation in reverse

yylex() is automatically generated by FLEX.

A synthesized attribute is a attribute whose value at a parse tree depends on the values of its children in parse tree.

Shift reduce conflict = state don't know if it will make a shift opn or redn for terminal

I/p to lexical Analyzer = source code being compiled

SDT => Syntax Directed Translation.

Augmented Grammar = Additional symbol added to grammar to improve parsing by parser :

S- Attributed Grammar = CFG that associates attribute with grammar symbols & rules

Bottom up | Synthesized attributes

SYNTAX TREE = Tree like DS → represents syntactic structures of a string

nodes => Parent, Children

TOKEN = Group of symbols

PATTERN = sequence of lexeme / set of rules & instruction

Lexeme = sequence of characters in the source program.

Finite Automata is used for recognition of tokens.

mul = 4 * 3
→ identifier.

Canonical LR = most powerful parser.

Data items grouped together = Record

Code, Procedures, Variable = managed by Run Time Env.

Lexical Analyzer reads source program & breaks it into tokens

LR(1) is look ahead LR Parser.

Inherited → get value from siblings or parents

Synthesized → get value from children to parent

Common subexpression elimination → elimination takes place in intermediate code generation

Recursive Decent Parser = Top-Down Parser

Compiler can check SYNTAX Errors

Part of string matches → Rhs of any prodxn = handle

Semantic Analyzer → grammar checking are tokens together into semantic structures.

Pf("i = %d , &i = %x , i , &i"); = 21 tokens [∵ exp. is very long]

elimination of left recursion is done via TOP DOWN PARSER

Lexical Analyzer skips white spaces

DAG is used to represent common subexpressions → optimizes code by reducing redundancy.

Error recovery by panic mode, phrase level, error prodxn