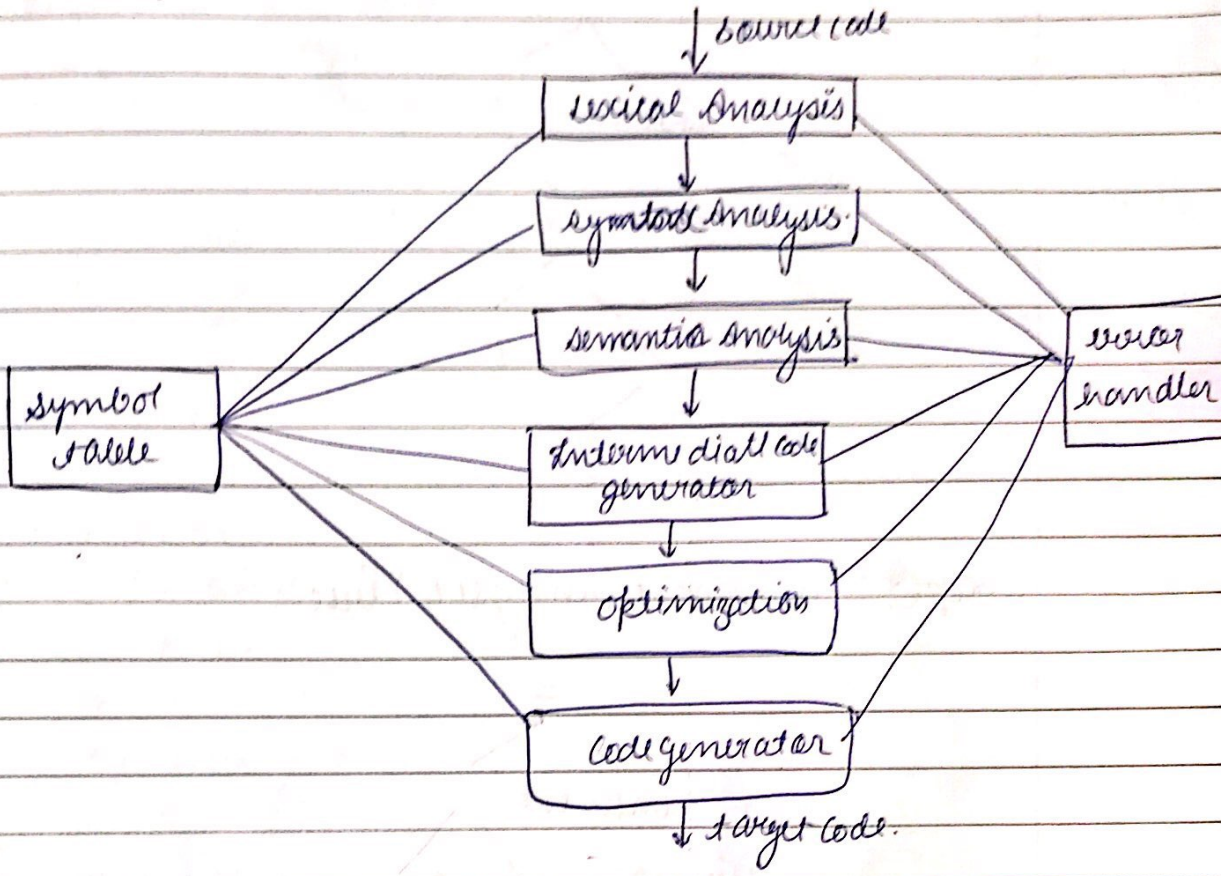
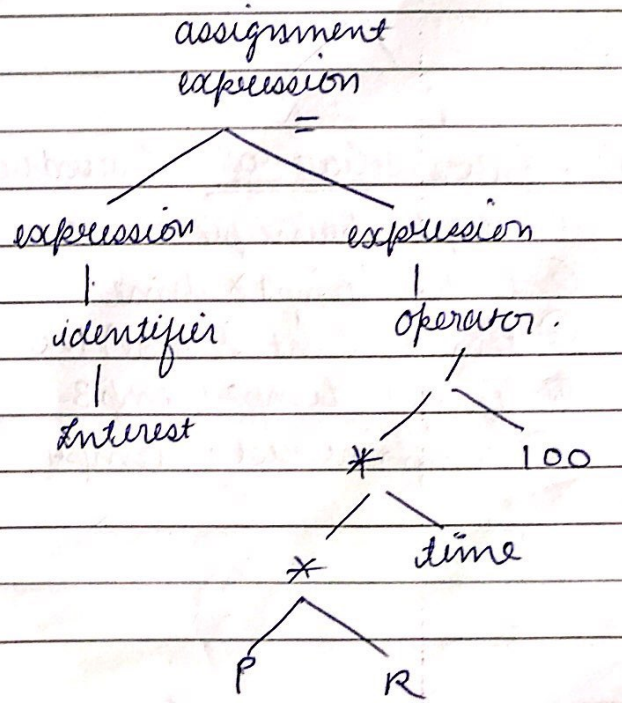
Page No.:
Date 19/7/19

Phases of compiler :-

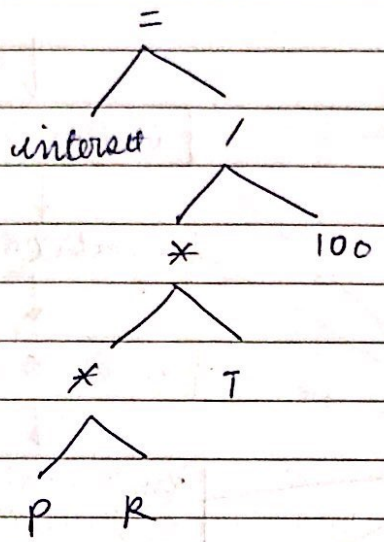


Step ① Parse tree

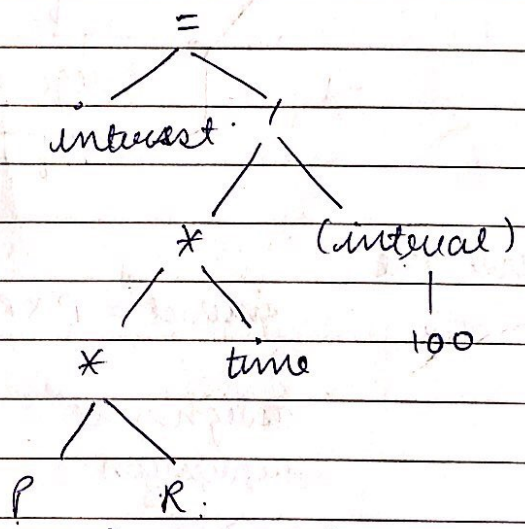
interest = P X R X T / 100



Step ② - Symbol tree



Step ③ - Semantic (analyser) Tree



Step ④ - Intermediate code (Three address code)

① temp1 = principle * rate
② temp2 = temp1 * time
③ temp3 = (int to real) 100
④ temp4 = temp2 / temp3
⑤ ~~temp~~ interest = temp4

Page No.:
Date:

Step (5) - Optimization

(1) temp1 = P x R.
(2) temp2 = temp1 * time
(3) int. = temp2/100.0
(4) temp4 = temp2/temp3
(5) w

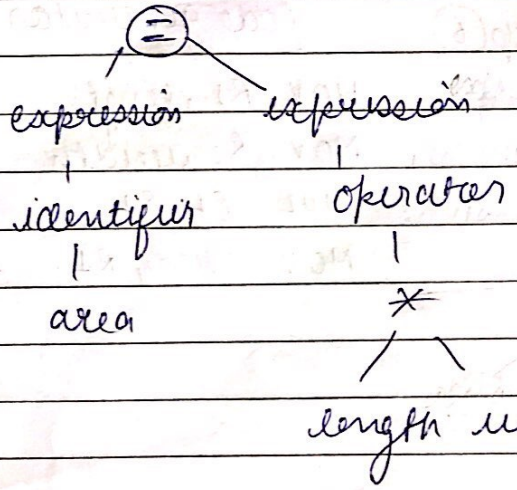
code
Step (6) Generation.

1.) MOV R1 P.
2.) MOV R2 R
3.) MUL R1 R2
4.) MOV R3 time
5.) MUL R1 R2
6.) DIV R1 100.0
7.) MOV Interest R1

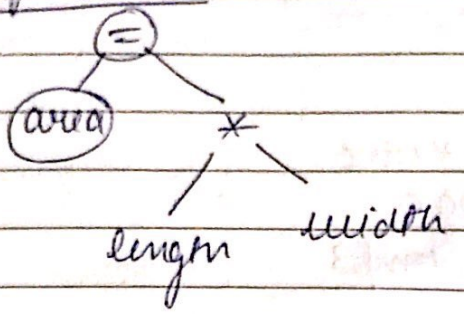
Perform all phases of the compiler.

area = length x width.

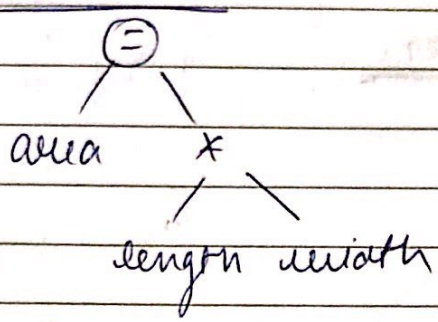
Step (1) parse tree.



Step (2) symbol tree



Step (3) Semantic Tree



Step (4) - Intermediate code.

temp1 = length * width.
temp2 = temp1.
area = temp2.

Step (5) code optimization

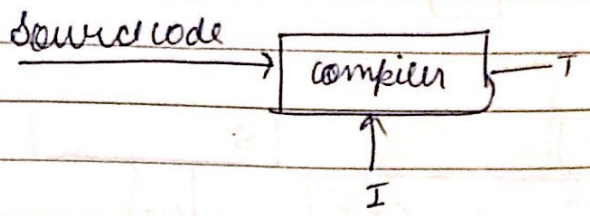
temp1 = length * width
area = temp.

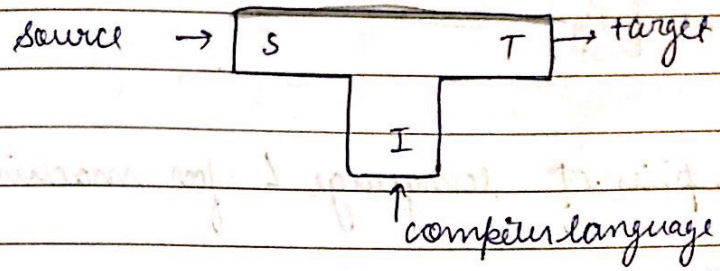
Step (6) Code generator.

MOV R1, length
MOV R2, width
MUL R1, R2
MOV area, R1.

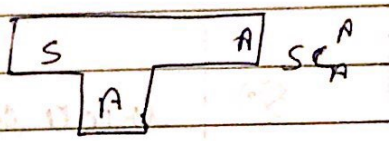
Bootstraping :

T Diagram -



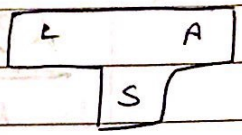


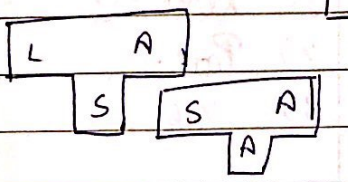
$S \subseteq L$

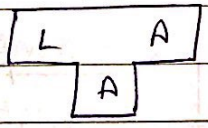
1.) 

$S \subseteq_A^A$

2.) $L \subseteq_S^A$

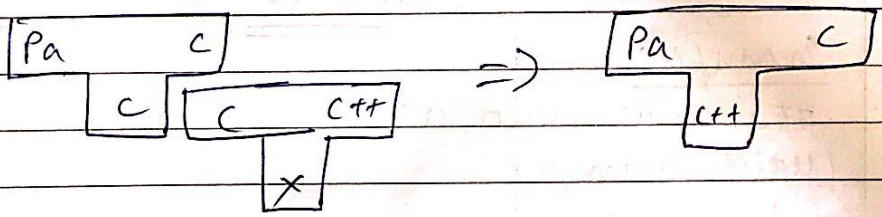


3.) 

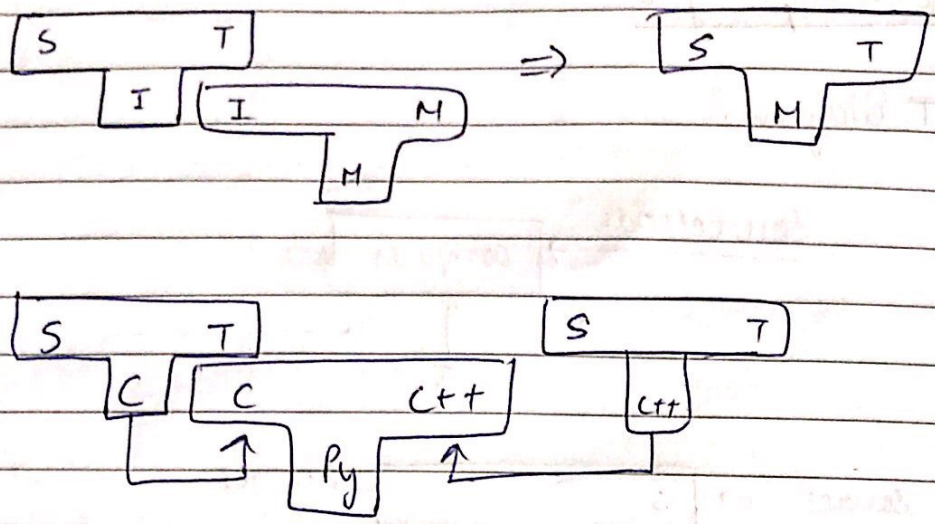
$\Rightarrow$ 

Q.) Design a compiler for language L.

Q.) ~~Design~~ a we have a pascal translator written in C language that takes pascal code & produce C as output. Create pascal translator in C++ or same



for ex :- X = Java



Q) Create a compiler of language L for machine B.

* Lexical Analyser :-

1.) Lexeme
2.) Pattern
3.) Token

Example:

```
int max(int x, int y)        (7)
  int x, y                   (5)
{                            (1)
  return (x > y ? x : y);    (11)
}                            (1)
```

Total :- 25

Example:

```
#include <stdio.h>
void main()
{
}
```

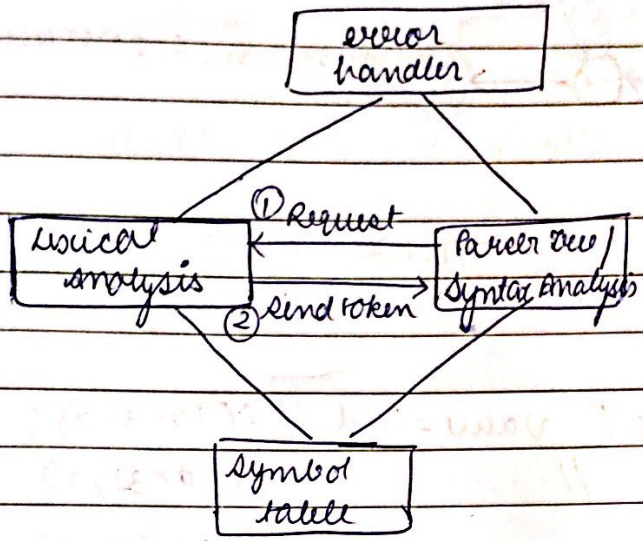
Total :- 6 | not defined

Page No.:
Date:
YOUVA

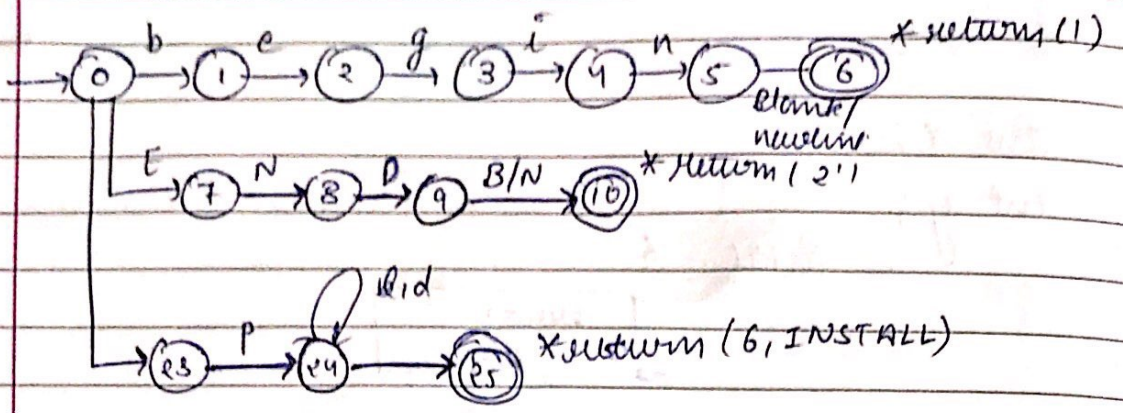
Example -

int x;
int y;

Total :- 6



Token	code	Value
begin	1	–
end	2	–
if	3	–
then	4	–
else	5	–
identifier	6	Pointer to symbol table
const	7	— " —
<	8	1
<=	8	2
==	8	3
!=	8	4
>	8	5
>=	8	6




```
#define f(x,y) x/y+x
main()
{
    printf("value = %d", f(10+3,3));
                         f((10+3),3);
}
```

output =>

$10 + 3/3 + 10 + 3$

$= 10 + 1 + 10 + 3$

$= 24$

output =>

$(10+3)/3 + (10+3)$

$= 4 + 13$

$= 17$

Buffering

- One Buffer scheme
- Two Buffer scheme

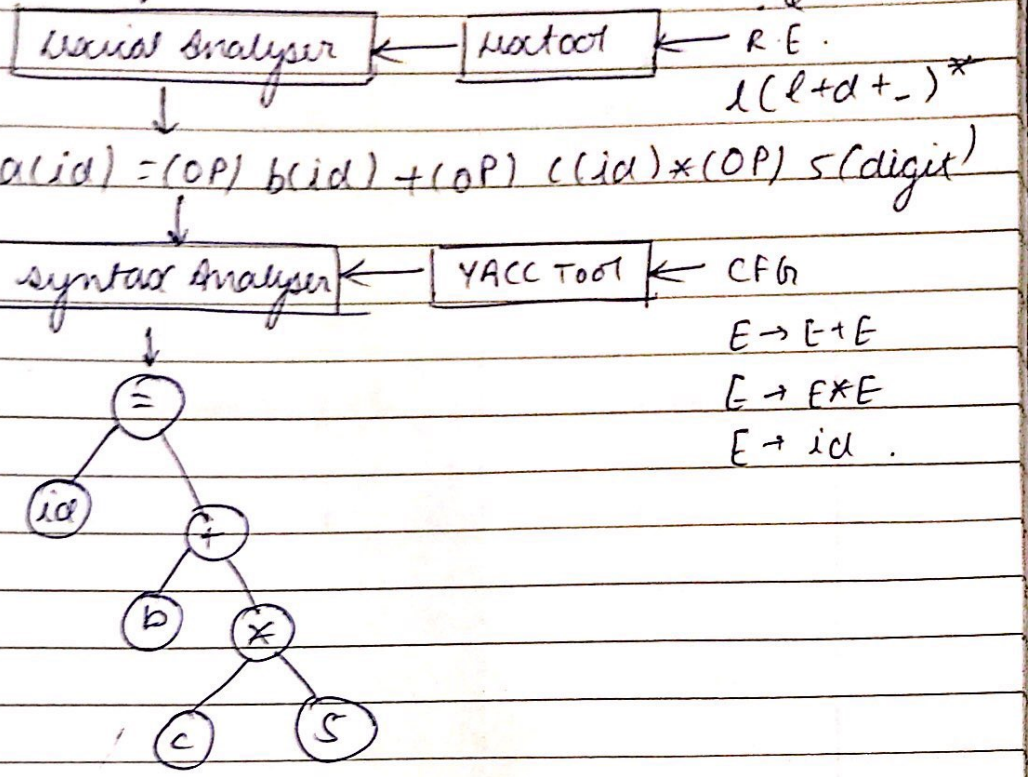
Lex & YACC Tool

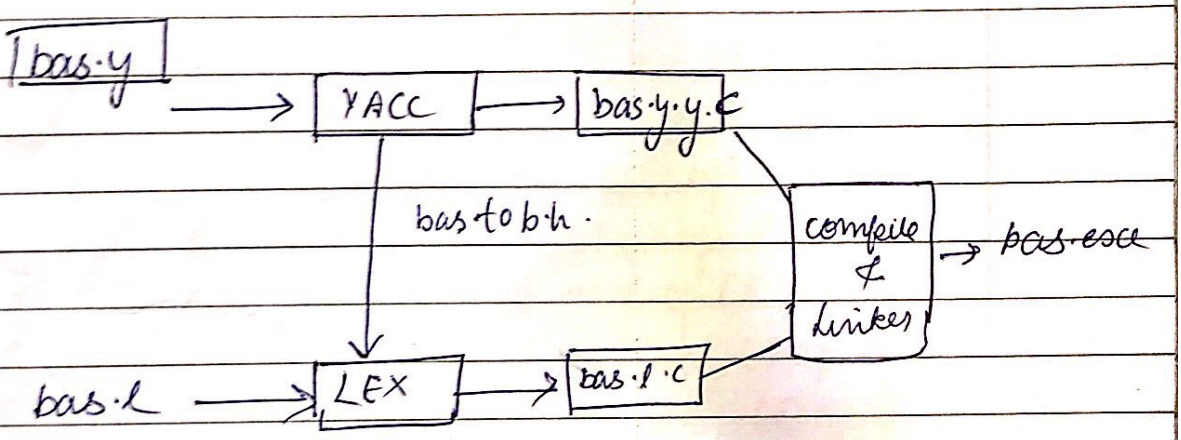
↓

yet another compiler compiler

a = b + c x 5

↓





Structure of Lex :-

Declaration

%%

Rules

%%

Subroutines → optional.

%.%.

Letter [A-z a-z]

digit [0-9]

id [letter & letter | digit }*]

%.%.

.l

Unit-2

Syntax Analyzer

Context free grammar:-

$G = (V, T, \xi, P, S)$

$E \to E+E$

$E \to E*E$

$E \to id$

$V = \{E\}$

$S = \{E\}$

$\Sigma = \{id\}$

rule :- $vn \to n$

$n \in V_n$

$n \in (V_n \cup \Sigma)^*$

$E \to E+E \Rightarrow E \to id+E \Rightarrow E \to id+id$ LMD

$E \to E+E \Rightarrow E \to E+id \Rightarrow E \to id+id$ RMD

$a = bx(c+d) | E$

$E \to E\,op\,E \mid (E) \mid -E \mid D$

$op \to + \mid - \mid * \mid / \mid \% \mid \uparrow$

$D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$-7+5$

$E \to E\,op\,E$

$\to -E\,op\,E$

$\to -D\,op\,D$

$\to -7\,op\,5$

$\to -7+5$

$8*(3/5)-7$

$E \to E\,op\,E$

$\to D\,op\,E$

$\to D\,op\,E\,op\,E$

$\to D\,op\,(E)\,op\,E$

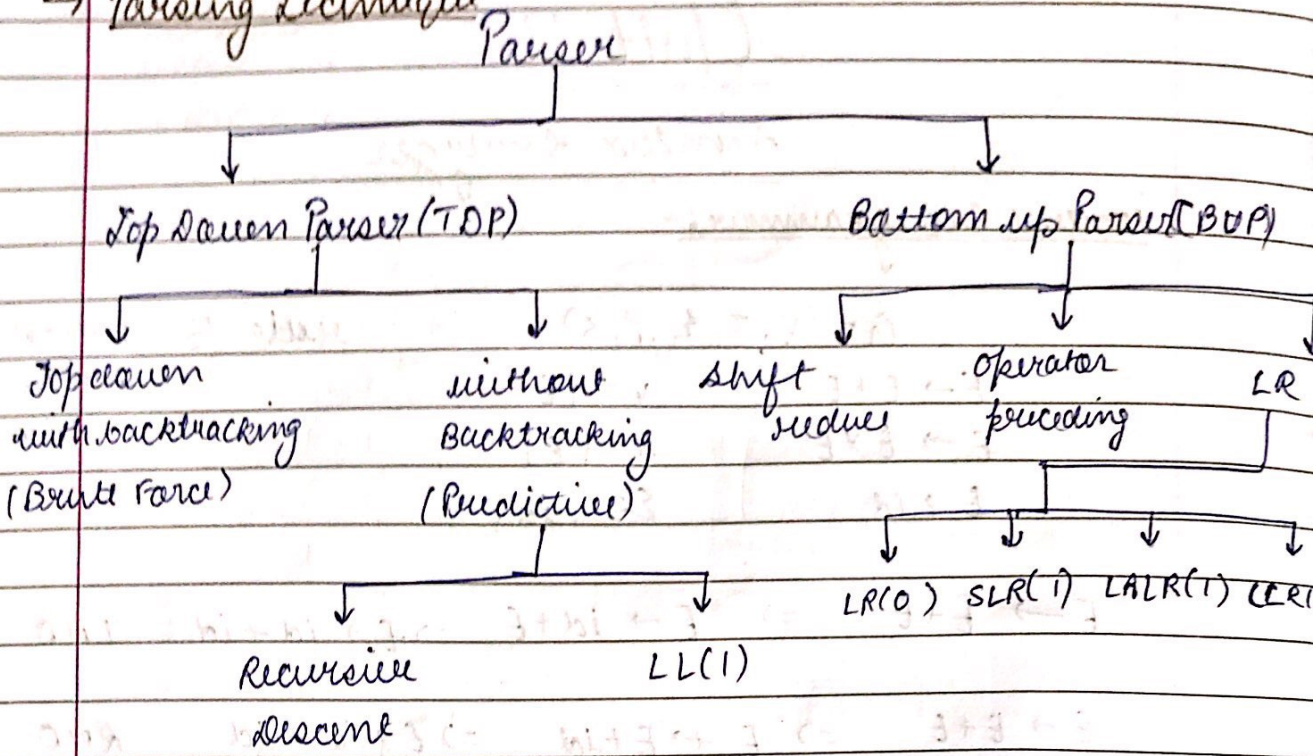
$\to D\,op\,(E\,op\,E)\,op\,E$

$\to D\,op\,(D\,op\,D)\,op\,D$

$\to 8\,op\,(3\,op\,5)\,op\,7$

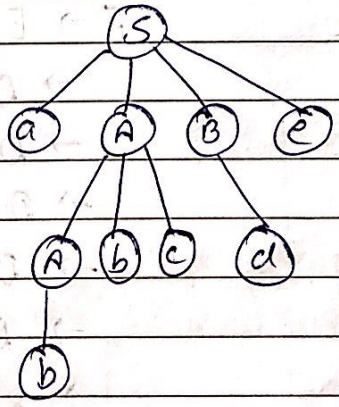
$\to 8*(3/5)-7$

→ Parsing Technique

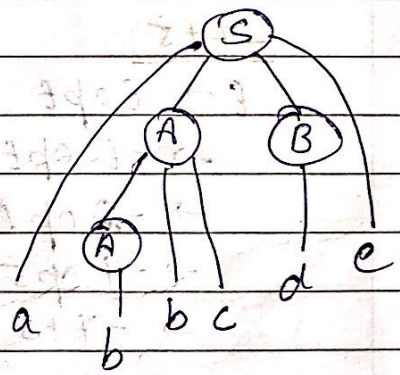


S → aABe
A → Abc | b
B → d
input → abbcde

TDP

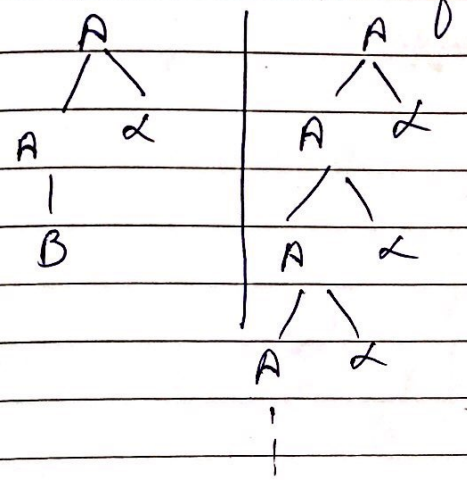


BUP



Types of Grammar	TDP	BUP
1.) Ambiguous	✗	✗
2.) Unambiguous	✓	✓
3.) Left Recursive	✗	✓
4.) Right Recursive	✓	✓
5.) Non Deterministic	✗	✓
6.) Deterministic	✓	✓

$A \rightarrow A\alpha \mid \beta$ Left recursive. Function:




```
A()
{
  A()
   α
}
```

$A \rightarrow A\alpha \mid \beta$

↓ conversion

$A \rightarrow \beta A'$

$A' \rightarrow \in \mid \alpha A'$

$A \rightarrow \alpha A \mid B \rightarrow$ Right Recursive.

$A \rightarrow aBB \mid aB\in \mid ab$

$B \rightarrow d$

$C \rightarrow c$

→ Example :-

$E \to E+T \mid T$
$T \to T+F \mid F$
$F \to id$

$E \to E+T$
$E \to E+F$
$E \to T+T$
$E \to F+F$
$E \to id$

$E \to E+T \mid T$
(A) (A α) (β)

$T \to T+F \mid F$
(A) (A α) (β)

$F \to id$

$A \to A\alpha \mid B$
$A \to \beta A'$
$A' \to \in \mid \alpha A'$

$E \to E+T \mid T.$
$E \to TE'$
$E' \to \in \mid +TE'$
$F \to id.$
$T \to T+F \mid F.$
$T \to TT'$
$T' \to \in \mid +FT'$

Non Deterministic Grammar :-

$A \to \alpha\beta_1 S \mid \alpha\beta_2 A \mid \alpha\beta_3 S \mid \alpha\beta_3 B.$

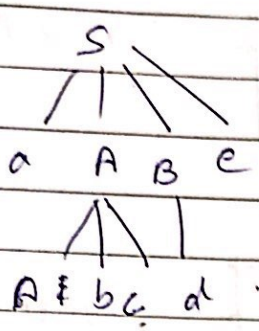
→ if a part of grammar is same it is non-deterministic
→ if unique it is deterministic.

$A \to \alpha A'$
$A' \to \beta_1 S \mid \beta_2 A \mid \beta_3 S \mid \beta_3 B$
$A' \to \beta_1 S \mid \beta_2 A \mid \beta_3 S'$
$S' \to S \mid B$

$S \rightarrow aABe$

$A \rightarrow Abc / b$

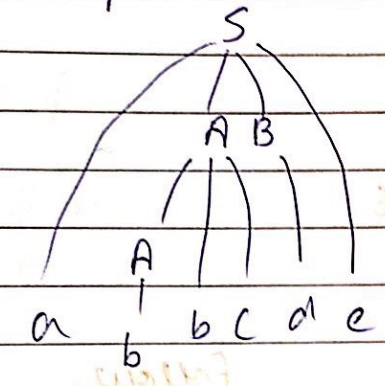
Top Down [abbcde]



↓

Top down approach is also left most derivation (LMD)

Bottom up.



Bottom up approach is also right most derivation (RMD)

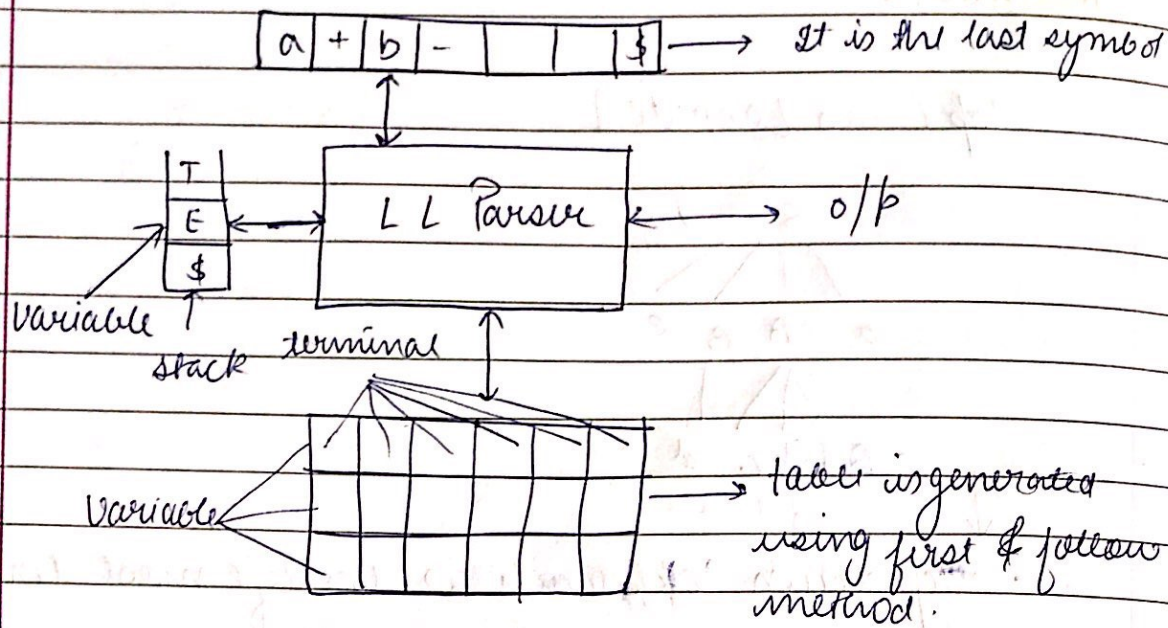
* Top Down Parsing :-

1.) Backtracking
2.) Recursive
3.) LL(1)

LL(1) → reads only 1 i/p symbol at a time

- scan from left to right
- LMD

* LL(1)



example :- $E \to TE'$

$E' \to +TE' / \epsilon$

$T \to FT'$

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

$F \to (E) / id$

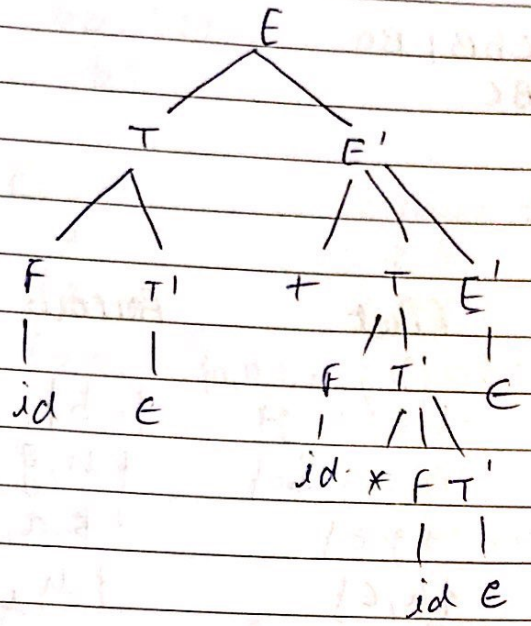
	first (from bottom)	follow (from top)
E	{(, id}	{$,)}
E'	{+, ε}	{$,)}
T	{(, id}	{+, $,)}
T'	{*, ε}	{+, $,)}
F	{(, id}	{*, +, $,)}

because $E' = E$ so $E' = E$. Follow of T = first of E' and E' is equal to ε then take follow

	id	+	*	(	)	ε	$
E	$E \to TE'$	error	error	$E \to TE'$	error	error	error
E'	error	$E' \to +TE'$	"	error	$E' \to \epsilon$	"	$E' \to \epsilon$
T	$T \to FT'$	error	"	$T \to FT'$	error	"	error
T'	error	$T' \to \epsilon$	$T' \to *FT'$	error	$T' \to \epsilon$	"	$T' \to \epsilon$
F	$F \to id$	error	error	$F \to (E)$	error	"	error

Page No.:
Date:

id + id * id $



Q:) Find first & follow.

S → ABCDE
A → a | ε
B → b | ε
C → c
D → d | ε
E → e | ε

	First	follow
S	{a, b, c}	{$}
A	{a, ε}	{b, c}
B	{b, ε}	{c}
C	{c}	{d, e, $}
D	{d, ε}	{e, $}
E	{e, ε}	{$}

Page No.:
Date:

Example :-

$S \rightarrow ACB \mid CbB \mid Ba$
$A \rightarrow da \mid BC$
$B \rightarrow g \mid \epsilon$
$C \rightarrow h \mid \epsilon$

	First	Follow
	{d, g, h, ϵ, a, b}	
S	{d, g, h, ϵ, a, b}	{$}
A	{d, g, h, ϵ}	{h, g, $}
B	{g, ϵ}	{$, a, h, g}
C	{h, ϵ}	{h, g, $, b}

Table of LL:

	a	b	g	h	ϵ	d	$
S	S→ACB S→Ba	S→CbB	S→ACB CbB|Ba	S→ACB CbB	~~S→ACB~~ error	S→ACB	~~S→ACB~~ S→ACB
A	error	error	A→BC	A→BC	"	A→da	A→BC
B	B→ϵ	"	B→g	B→ϵ	"	error	B→ϵ
C	error	C→ϵ	C→ϵ	C→h C→ϵ	"	"	C→ϵ

Q.) $S' \rightarrow S\#$
$S \rightarrow ABC$
$A \rightarrow a \mid bbD$
$B \rightarrow a \mid \epsilon$
$C \rightarrow b \mid \epsilon$
$D \rightarrow C \mid \epsilon$

ip :- bbcabb#
Construct parse tree using LL(1)

S #

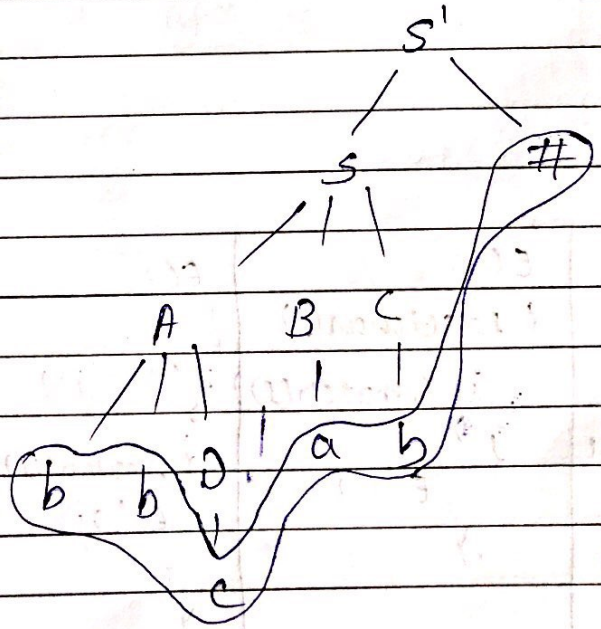
·ABC#

bbC·BC#

bbC

	First	Follow
S'	{a,b}	{$}
S	{a,b}	{#}
A	{a,b}	{a,b,#}
B	{a,ϵ}	{b, #}
C	{b,ϵ}	{#}
D	{c,ϵ}	~~{#}~~ {a,b,#}

	a	b	c	#	ϵ	$
S'	~~S'~~→S#	~~S'~~→S#	error	error	error	error
S	S→ABC	S→ABC	"	"	"	"
A	A→a/~~bbD~~	A→bbD	"	"	"	"
B	B→a	B→ϵ	"	B→ϵ	"	"
C	error	C→b	"	C→ϵ	"	"
D	D→ϵ	D→ϵ	D→c	D→ϵ	"	"

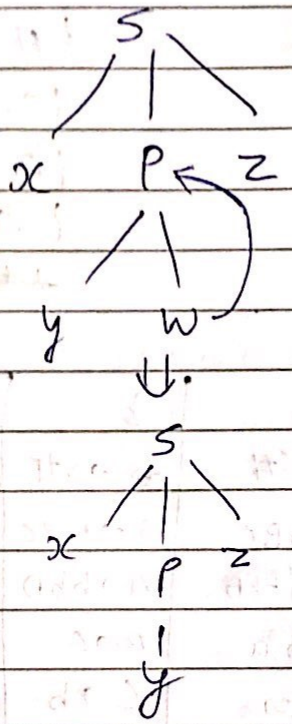


* Backtracking :-

$S \rightarrow xPz$

$P \rightarrow yw \mid y$

i/p string : xyz



2.) Recursive Descent :-

$E \rightarrow iE'$

$E' \rightarrow +iE' \mid \epsilon$

input i+i$

```
main()
{   E()
    if (l=='$')
    print complete
    else
    print error
}
```

```
E()
{ l=getchar()
  if (match(l))
  {
    E'()
  }
```

```
E()
{
if(i=='i')
{ match('i');
  E'();
}
}
```

```
matchchar(t)
{
 if(l==t)
{
 l=getchar();
} else print error
}
```

```
E'()
{ if (l=='t')
   { match('+');
     match('i');
     E'();
   } return;
}
```

Ex example:- $S \rightarrow cAd.$

$A \rightarrow ab|a.$

```
main
{ S()
  if(l=='$')
    printcompete.
  else
    print error.
}
```

```
S()
{ if(l=='c')
  { match(c);
    A()
    match(d);
  }
  else return;
  print error.
```

```
A()
{ if(l=='a')
  } match(a);
  if(l=='b')
  { match(b);
  }
  else return;
  Print error:
```

```
match(char t)
{ if (l==t)
  {
    l=getchar();
  }
  else
  print
  error.
```

t)

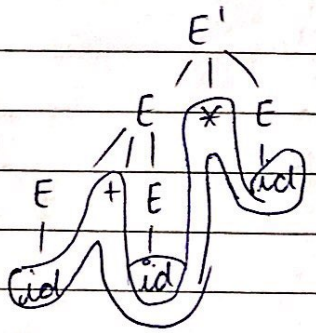
* Bottom up Parsing :-

→ Shift Reduce Parsing :-

- Shift
- Reduce
- Accept
- Error

example :

STACK	Tokens	operation Name
$	id + id * id	shift
$id	+ id * id	Reduce
$E	+ id * id	shift
$E+	id * id	shift
$E+id	* id	reduce
$E+E	* id	reduce
$E	* id	shift
$E*	id	shift
$E*id	-	reduce
$ E*E	-	reduce
$E	-	accept.



bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false
input string :- not (true or false)
shift reduce algorithm:

bexpr
bterm
bfactor
not bfactor
not (bexpr)
not (bexpr or bterm)
not (bterm or bterm)
not (bfactor or bfactor)
not (true or false)

STACK	Tokens	operation name
$	not (true or false)	shift
$not	(true or false)	~~reduce~~ shift
$not(	true or false)	shift
$not(true	or false)	Reduce
$not(bfactor	or false)	Shift-reduce
$not(bfactor or	false)	shift
$not(bfactor or false	)	Reduce
$not(bfactor or bfactor	)	Reduce
$not(bterm or bterm	)	Reduce
$not(bexpr or bterm	)	~~Reduce~~ shift
$not(bexpr)	-	~~or~~ Reduce
$ not bfactor		"
$ bfactor		"
$ bterm		"
$ bexpr		"

Not
Sum

Stack	i/p	Operation
$	not(true or false)	shift
$ not	(true or false)	shift
$ not(	true or false)	shift
$ not(true	or false)	Reduce
$ not(bfactor	or false)	Reduce
$ not(bterm	or false)	Reduce
$ not(bexpr	or false)	shift
$ not(bexpr or	false)	shift
$ not(bexpr or false	)	~~shift~~ reduce
$ not(bexpr or bfactor	)	reduce
$ not(bexpr or bterm	)	reduce
$ not(bexpr	)	shift
$ not(bexpr)	-	Reduce
$ not bfactor	-	Reduce
$ bfactor	-	Reduce
$ bterm	-	Reduce
$ bexpr	-	Accept

Q- S → T L ;
T → int | float
L → L, id | id
I/p → int id, id ;

S → T L ;
S → int L ;
S → int L, id ;
S → int id, id

Stack	i/p	Operation
$	int id, id ;	shift
$ int	id, id ;	Reduce
$ T	id, id ;	reduce
$ T, id	, id ;	reduce
$ T L	, id ;	shift
$ T L, id	id ;	shift
$ T L, id	;	Reduce
$ T L	;	shift
$ T L ;		Accept
$ S		

Page No.:
Date:
YOUVA

Operator Precedence Parsing:-

→ For operator grammar → it should not have null production
→ it should not have two consecutive non-terminal symbol.

eg $A \to \epsilon$ ✗

$B \to AAb$ ✗

eg: $E \to EAE \mid (E) \mid -E \mid id$

$A \to + \mid - \mid * \mid \uparrow$

⇓

$E \to E+E \mid E-E \mid E*E \mid E \uparrow E \mid (E) \mid -E \mid id$

eg: $S \to SAS \mid a$

$A \to bSb \mid b.$

⇓

$S \to SbSbS \mid SbS \mid a$

eg: $E \to E+E$

$E \to E*E$

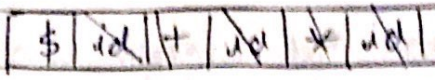
$E \to id$

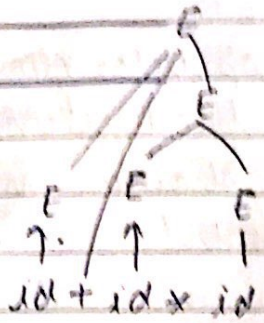
→ First generate operator precedence table.

f ↓ \ g →	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	-

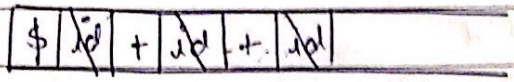
* if top of stack priority is less lookahead. use push
pop

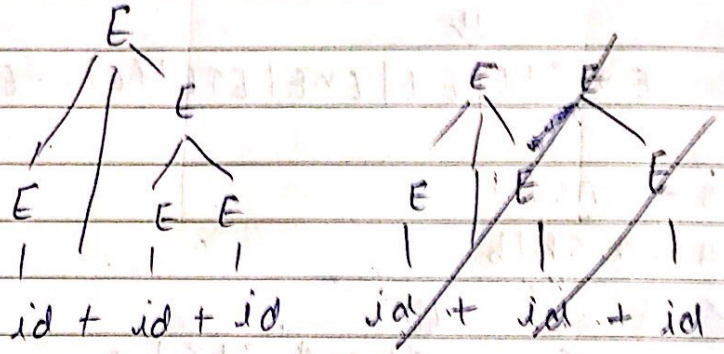
i/p :- id + id * id $





i/p :- id + id + id $



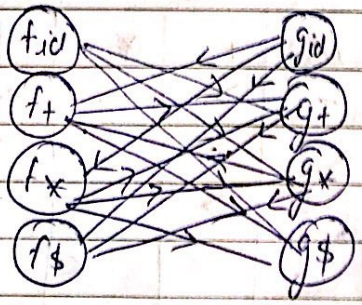


Operator function :-

Time complexity.

given terminal symbols = n + 1

complexity = $O(n^2)$ (for previous table)



→ It should not have any cycle.

→ From this we can not generate the priority

$id \to g_* \to f_+ \to g_+ \to f_\$$

$g_{id} \to f_* \to g_* \to f_+ \to g_+ \to f_\$$

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

$2(n+1)$

$= O(n)$

eg: $g \to id + id * id \$$

$\to f$ | $ | ~~id~~ |

E

|

id + id * id

Toc	$P \to S.R \mid S$
	$R \to bSR \mid bS$
	$S \to Wbs \mid W$
	$W \to L*W \mid L$
	$L \to id$

construct operator table.

$P \to W\ SbSR \mid SbS \mid S$

$P \to SbP \mid SbS \mid S$

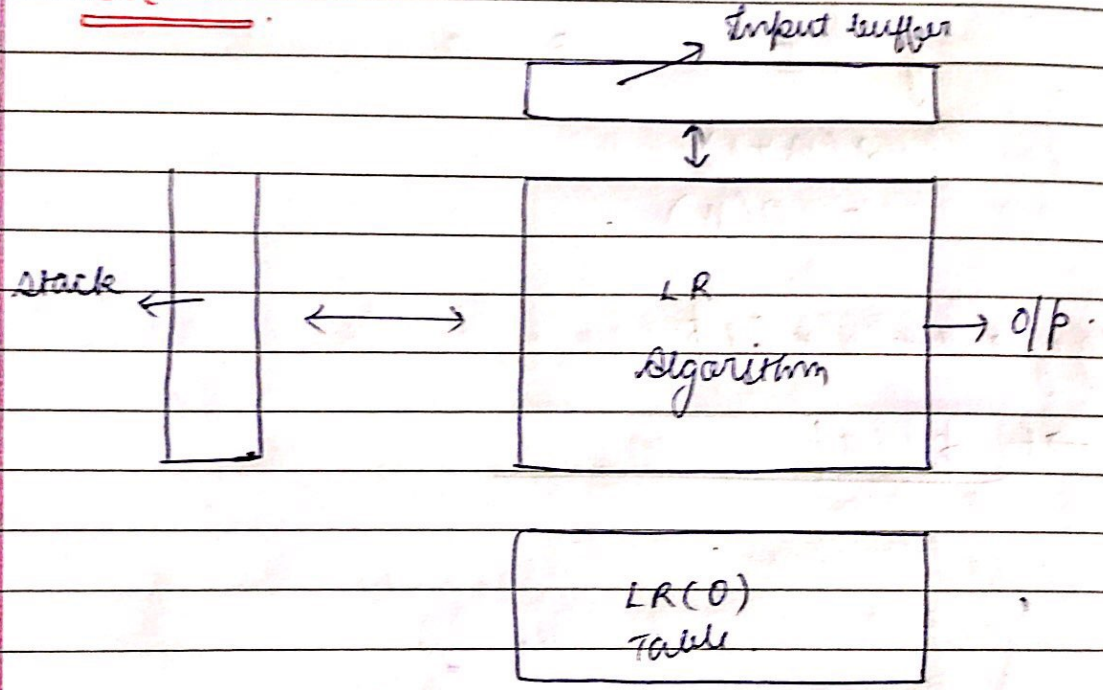
$S \to WbS \mid W$

$W \to L*W \mid L$

$L \to id$.

	id	*	b	$
id	-	>	>	>
*	<	<	>	>
b	<	<	<	>
$	<	<	<	-

* LR(0)



E → BB

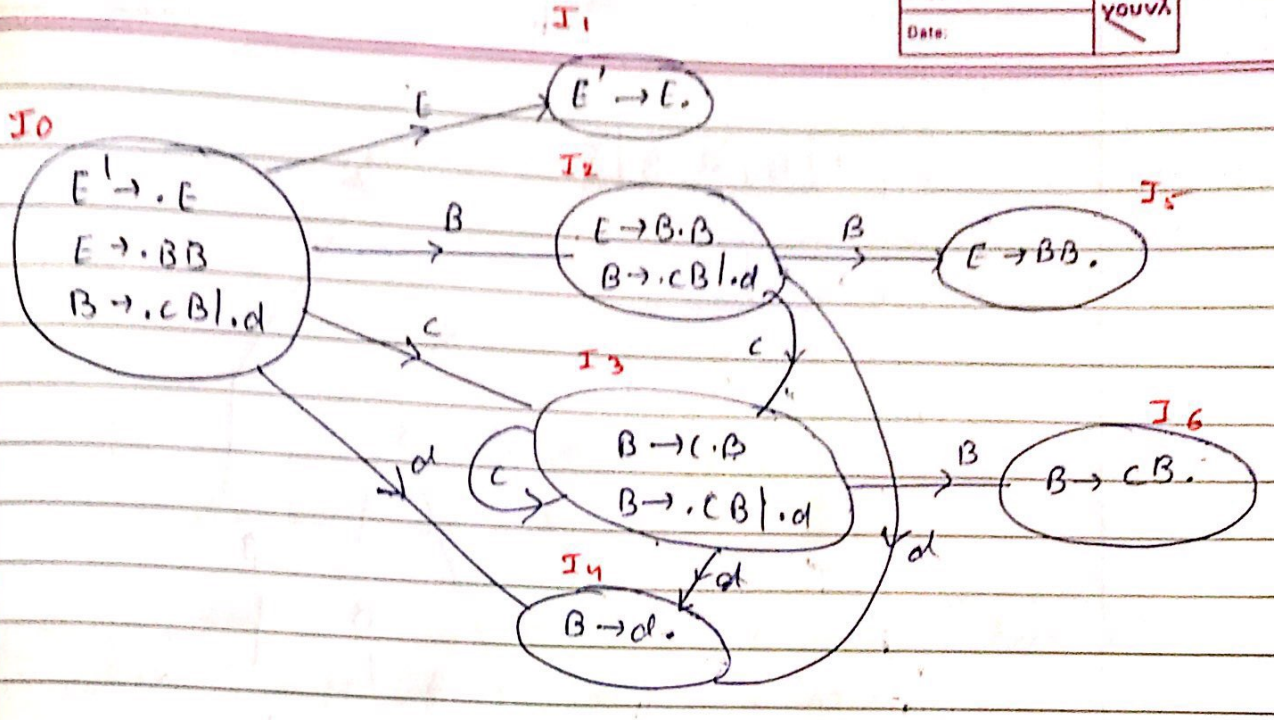
B → CB|d

generate augment grammar.

E' → .E

E → .BB

B → .CB|.d



	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		

S → shift
r → reduce

for parse true -

i/p :- ccdd$

0	c	3	c	3	d	4

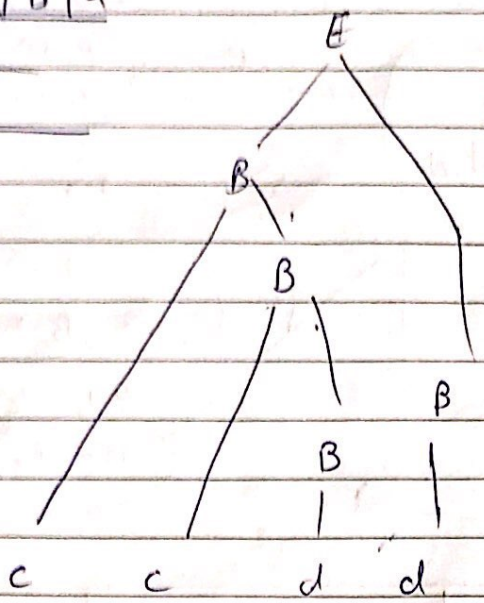
0	c	3	c	3	B	6

0	c	3	B	6

0	B	2	d	4

0	B	B	B	$

0	E	1



Example $E \to T+E \mid T$

$T \to id$

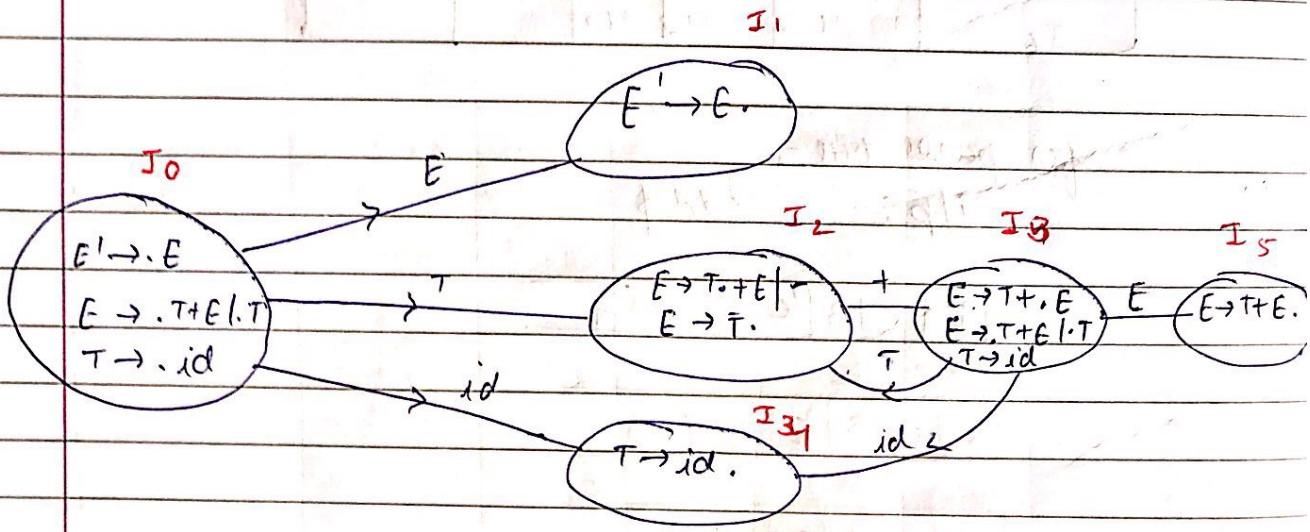
Construct LR(0) parser table ?

augment grammar.

$E' \to .E$

$E \to .T+E \mid .T$

$T \to .id$



	Action			Goto	
	id	+	$	S	T
I_0	S_4			1	2
I_1			Accpt		
I_2	R_0	R_3	R_2		
I_3	S_4				
I_4					

Shift / reduce conflicting :- For a single terminal there is both shift & reduce operation.

* SLR(1) :-

$E \to BB$;
$B \to cB \mid d$

$E \to BB$
$B \to cB$
$B \to d$

I_4 : $B \to d.$
I_5 : $E \to BB.$
I_6 : $B \to cB.$

I_4 :-
: $B \to d$
follow(B)
$F(E) = \$$
$F(B) = \{c, d, \$\}$

	Action			Goto	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accpt		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	R_3	R_3	R_3		
I_5					
I_6	R_2	R_2	R_2		

CLR(1) & LALR(1)

↳ LR(0) items + lookahead.

Ex:- $S \to AA$

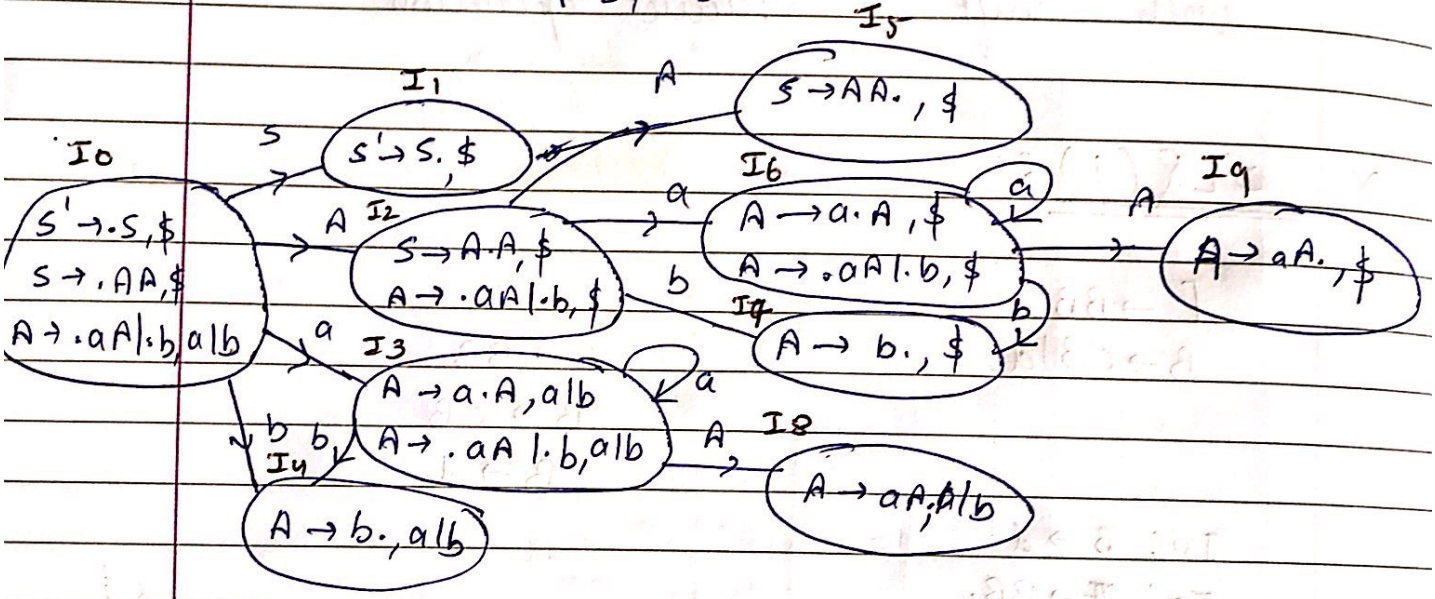
$A \to aA \mid b$

Augmented grammar:-

$S' \to .S, \$$

$S \to .AA, \$$

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



	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

for LALR(1):

$I_3 + I_6 = I_{36}$

$I_4 + I_7 = I_{47}$

$I_8 + I_9 = I_{89}$

	Action			Goto	
	a	b	$	S	A
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	R_3	R_3	R_3		
I_5			R_1		
~~I_6~~	S_{36}	S_{47}			89
~~I_7~~			R_3		
I_{89}	R_2	R_2	R_2		
~~I_9~~			R_2		

Ex: $S \to Aa \mid dAb \mid dca \mid cb$

$A \to c$

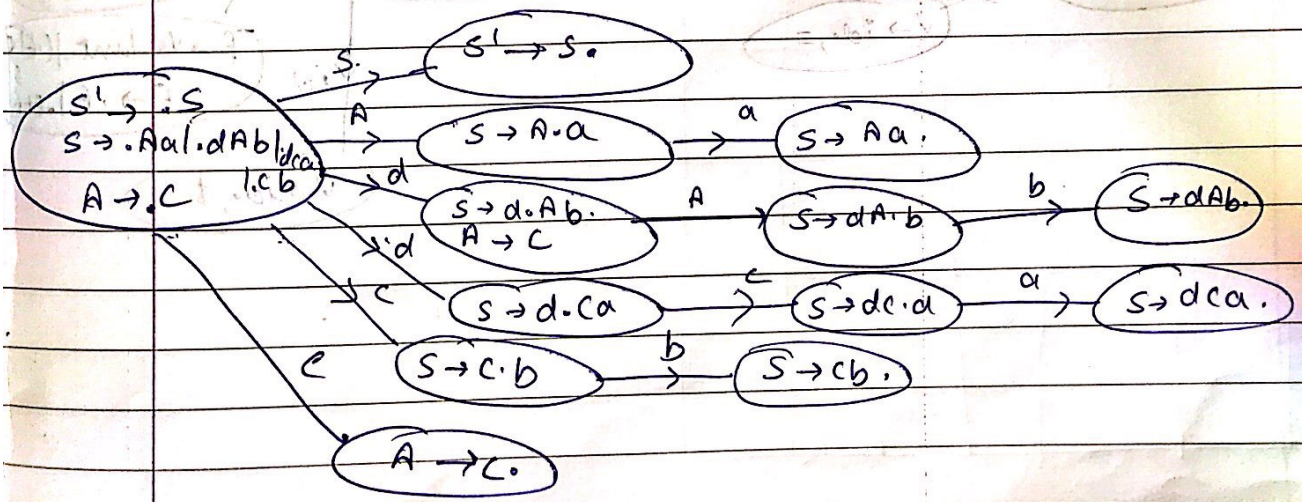
SLR(1) & LALR(1) table?

Augmented grammar:

$S' \to .S$

$S \to .Aa \mid .dAb \mid .dca \mid .cb$

$A \to .c$

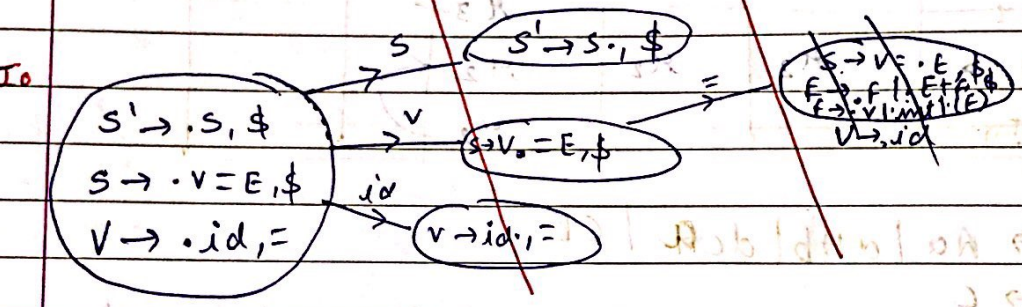


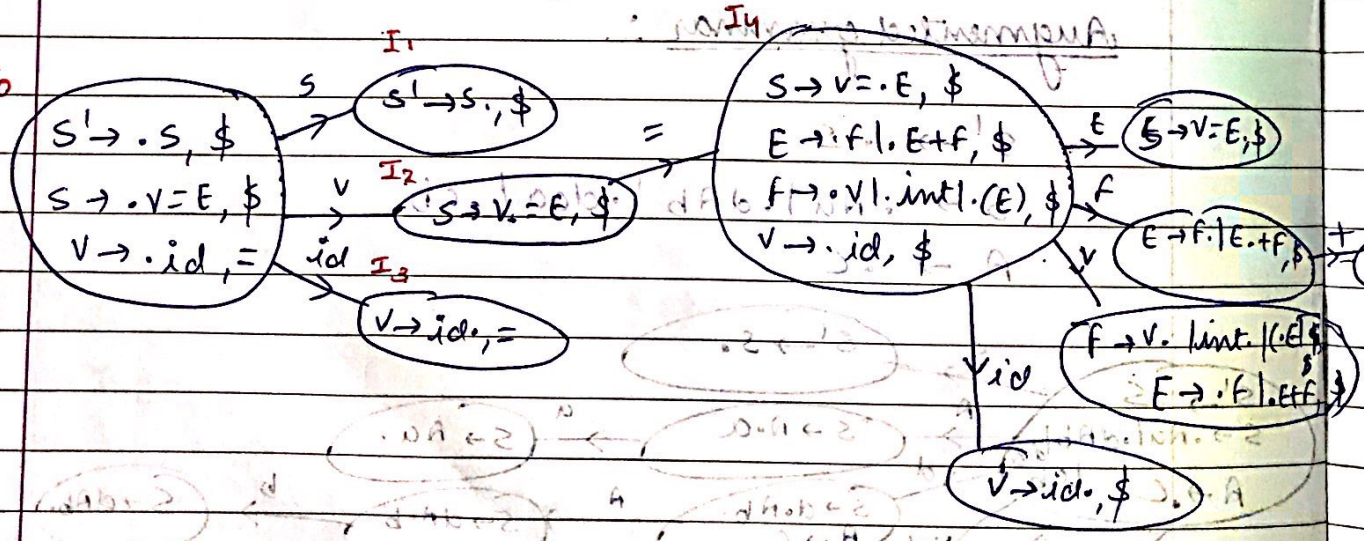
LR(0)
SLR(1) → LR(0) + Follow
CLR(1) → LR(0) + Lookahead
LALR(1) → — " —

Ex:
$S \to V = E$
$E \to F \mid E+F$
$F \to V \mid int \mid (E)$
$V \to id$

construct LALR(1) parsing table?
Also construct parse tree for i/p id = (id) + (id)

$S' \to .S, \$$
$S \to .V = E, \$$
$E \to .F \mid .E+F$
$F \to .V \mid .int \mid .(E)$
$V \to .id.$





Unit-3

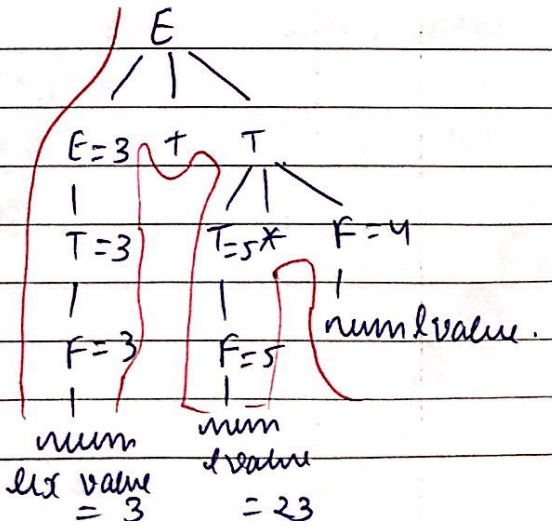
* Syntax Direct Translation :-

SDT = Grammar + Semantic Rules.

- Syntax Direct Definition
  - Synthesized
  - Inherited.
- Syntax Direct translation.

Grammar	Semantic rules
$E \rightarrow E+T$	E.value = E.value + T.value
$E \rightarrow T$	E.value = T.value
$T \rightarrow T*F$	T.value = T.value * F.value
$T \rightarrow F$	T.value = F.value
$F \rightarrow num$	F.value = num.lexvalue.

Ip :- 3+5*4=23



Page No.:
Date:

Semantic rules

{ printf("+"); }

{ printf("*"); }

{ }

{ printf("num"); }

I/p = 3 + 5 * 4

→ 354 * + (Infix to postfix)

Q.) S → xxW { printf("1"); }

S → y { printf("2"); }

W → Sz { printf("3"); }

i/p :- xxxxyzz

→ ip = xxxxyzz

-23131

Page No.:
Date:

8) $E \rightarrow E * T$ { E.value = E.value * T.value }

$E \rightarrow T$ { E.value = T.value }

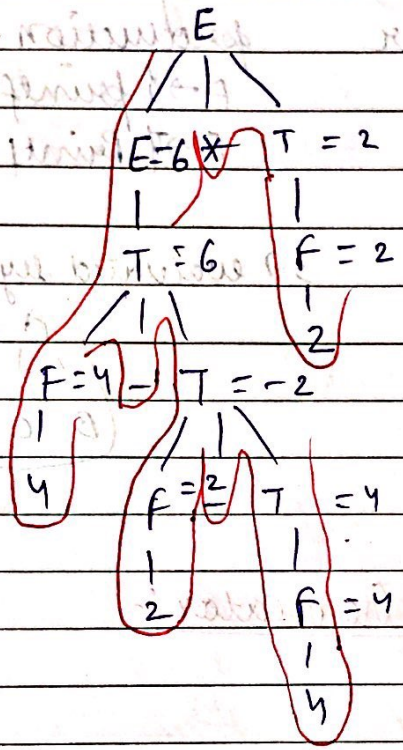
$T \rightarrow F - T$ { T.value = F.value - T.value }

$T \rightarrow F$ { T.value = F.value }

$F \rightarrow 2$ { F.value = 2 }

$F \rightarrow 4$ { F.value = 4 }

i/p = 4 - 2 * 4 * 2



i/p :- (4-(2-4)) * 2

(4-(-2)) * 2

= (6) * 2

= 12

9.) $E \rightarrow E \# T$ { E.value = E.value * T.value }

$E \rightarrow T$ { E.value = T.value }

$T \rightarrow T \$ F$ { T.value = F.value - T.value }

$T \rightarrow F$ { T.value = F.value }

$F \rightarrow num$ { F.value = num.lvalue }

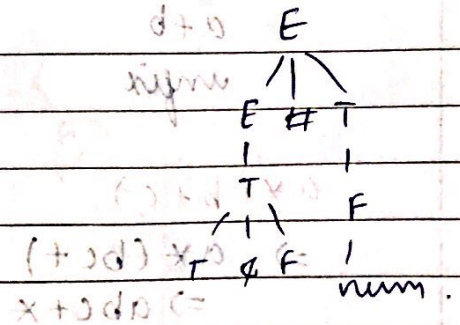
2 # 3 $ 5 # 6 $ 4.

2 * 3 - 5 * 6 - 4

2 * (-2) * 2

= -4 * 2

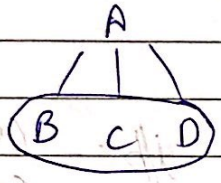
= -8.

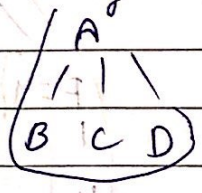


Page No.:
Date:
YOUVA

SDT : (Syntax Direct Translation) :-

S-attributed	L-attributed
1.) only synthesized allow	1.) Both synthesized & inherited allow but in inherited only left sibling. B.id → A.id ✓ B.id → C.id ✗ C.id → B.id ✓
2.) Semantic rules place at right side of grammar production rule.	2.) anywhere in right of production. E → { printf("+"); } T+F. E → T { Printf("+"); } +F.
3.) evaluated BUP manner	3.) evaluated by DF & right do left.





* Intermediate Code Generator :-

1.) Postfix Notation.
2.) Syntax Tree
3.) Three address code → quadruples & tuples.

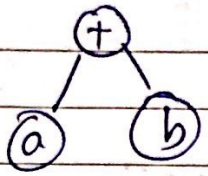
1.) Postfix Notation :-

a+b	+ab	ab+
infix	prefix	postfix

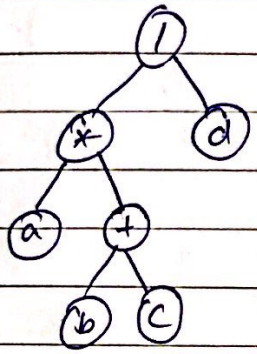
a * (b + c)
⇒ a * (bc+)
⇒ abc+*

2.) Syntax tree:-

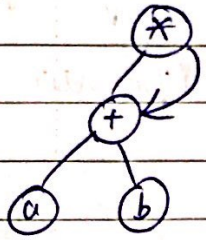
(1) a+b



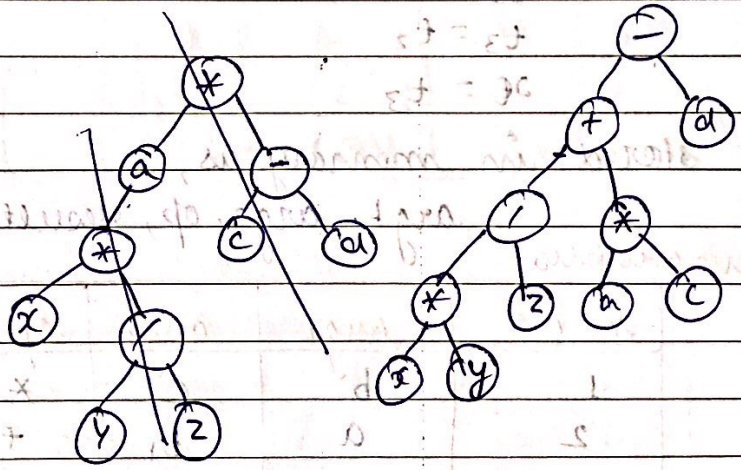
(2) a*(b+c)/d



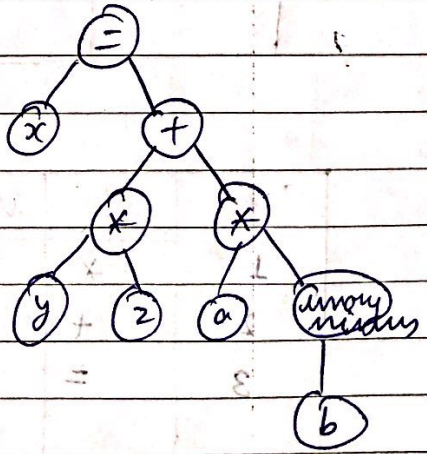
(3) (a+b)*(a+b)



(4) x*y/z+a*c-d.



(5) x=y*z+a*-b



1.) Binary op :- x op y
2.) Unary op :- x = op y
3.) Copy statement :- x = y
4.) Procedure call :- Param x
Param y.
5.) Index Arrangment :- x = y[i]
y[i] = x
6.) Conditional jump if x relop y goto label.
7.) Address & pointer - x = &y
y = *x

Example :-

$x = a + b * c$

$t_1 = b * c$

$t_2 = a + t_1$

$t_3 = t_2$

$x = t_3$

stored in memory as,

arg1, arg2, op, result.

Quadruples

Sr. No	arg1	arg2	op	result.
1	b	c	*	t_1
2	a	t_1	+	t_2
3.	t_2	-	=	x

Triples

Sr No.	op	arg1	arg2
1	*	b	c
2	+	a	(1)
3	=	(2)	

Page No.:
Date:

Example:- $a = b * -c + b * -c$

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = t_2 + t_2$
$a = t_3$

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = t_2 + t_2$
$a = t_3$

Quadruples :-

Sr. No.	arg 1	arg 2	O/p	result.
1	c		-	t_1
2	b	t_1	*	t_2
3.	t_2	t_2	+	t_3
4.	t_3		=	a

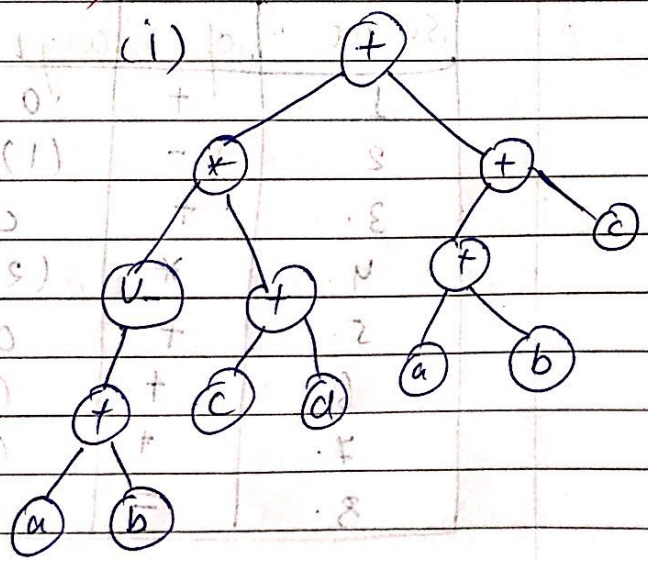
Triples.

Sr no.	op	arg 1	arg 2.
1	-	c	
2.	*	b	(1)
3.	+	(2)	(2)
4.	=	(3)	

Q.) Translate the given expression :-
$x = -(a+b) * (c+d) + (a+b+c)$

(i) Syntax tree.
(ii) Three address code
(iii) Quadruple.
(iv) Triple.
(v) Indirect triple.

(i)



(ii)

$t_1 = a + b$.

$t_2 = -t_1$

$t_3 = c + d$.

$t_4 = t_2 * t_3$.

$t_5 = a + b + c$.

~~$t_6 = t_5 + t_4$.~~

$t_6 = t_5 + c$.

$t_7 = t_4 + t_6$

$x = t_7$.

Quadruple:-

Sr No.	arg 1	arg 2	op	result
1.	a	b	+	t_1
2.	t_1		u-	t_2
3.	c	d	+	t_3
4.	t_2	t_3	*	t_4
5.	a	b	+	t_5
6.	t_5	c	+	t_6
7.	t_4	t_6	+	t_7
8.	t_7	=	=	x

Triple:-

Sr. No	op	arg 1	arg 2
1	+	a	b
2	u-	(1)	
3.	+	c	d
4.	*	(2)	(3)
5.	+	a	b
6	+	(5)	c
7.	+	(4)	(6)
8.	=	(7)	x

Indirect Triple :-

Indirect	Address
1	101
2	102
3	103
4	104
5	105
6	106
7	107
8	108

Address	arg 1	arg 2	op.
101	a	b	+
102	(101)		-
103	c	d	+
104	(102)	(103)	*
105	a	b	+
106	(105)	c	+
107	(104)	(106)	+
108	x	(107)	=

Example :-

```
int i;
i = 1
while (i<10) do
if x<y then
i = x+y
else
i = x-y
```

3 address code.

```
1.) i = 1
2.) if i<10 goto(4)
3.) goto (10)
4.) if x>y goto(6)
5.) goto(8)
6.) i = x+y
7.) goto(2)
8.) i = x-y
9.) goto(2)
10.) exit
```

Example :-

```
c=0
do
{
   if(a<b) then
   x++
   else
   x--
   c++
} while(c<5)
```

1.) c=0
3.) goto(8)
4.) if a<b goto(6)
5.) goto(7)
6.) x++
7.) x--
8.) c++
9.) if (c<5) goto(4)
10.) goto(9)
11.) exit

1.) c=0
2.) if a<b goto(4)
3.) goto(9)
4.) x=x+1
5.) c=c+1
6.) goto(9)
7.) x=x-1
8.) c=c+1
9.) if (c<5) goto(2.)
10.) goto(11)
11.) stop.

Example :-

```
int i,x=0;
for(i=1; i<=5; i++)
   {
      x=x+i;
   }
```

write 3 address code of the given code ?

Page No.:
Date:

3 address code:-

1.) ~~i=0~~, x=0, i=1
2.) if $i \leq 5$ goto (4)
3.) goto (2)
4.) x = x + i
5.) ~~exit~~ goto (2)

1.) oc = 0
2.) i = 1
3.) if $i \leq 5$ goto (5)
4.) goto (8)
5.) x = x + i
6.) i = i + 1
7.) goto (3)
8.) stop.

Example:-

```
int i = 0, a[10];
while i < 10 do
    a[i] = 0;
    i = i + 1;
```

3 address code:

1.) i = 0
2.) if i < 10 goto (4)
3.) goto (8)
4.) t = 4 * i;
5.) a[t] = 0
6.) i = i + 1
7.) goto (2)
8.) stop.

$A[i] = B.A + w * (i - lB)$
$= B.A + 4 * (i - 0)$
$= B.A. + 4 * i$

Example:-

```
int i;
int a[10][10];
  i = 0;
while (i < 10)
{ a[i][i] = 1;
  i++
```

Page No.:
Date:
YOUVA

3 address code

1.) i = 0
2.) if (i < 10) goto (3)
3.) goto (8).
4.) t = 44 * i
5.) a[t] = 1
6.) i = i + 1
7.) goto (2)
8.) stop.

= Base (A) + w [M(K-1) + (J-1)]
= of or
B.A. + w x (m x (j-1B) + (j-1B
= B.A + w (+0 x [+ 1)
= 44 x i .

Example :-

```
int i;
int a[10][10];
for(i=0; i<10; i++)
    {
        for(j=0; j<10; j++)
        {
            a[i][j] = 1;
        }
    }.
```

3 address code :-

1.) if (i<10) goto (4)
2.) goto (4).
3.) goto ( )
4.) if (j<10) goto ( )
5.) goto ( )
6.) t = 44 x i
    a[t] = 1.
    j = j i = i + 1
    j = goto

1.) i = 0
2.) if i < 10 goto(4)
3.) goto(15)
4.) j = 0
5.) if j < 10 goto (7)
6.) goto (13)
7.) $t_1 = 10 * i$
8.) $t_2 = t_1 + j$
9.) $t_3 = 4 * t_2$
10.) a[t_3] = 1
11.) j = j + 1
12.) goto(5)
13.) i = i + 1
14.) goto (2)
15.) stop

Example:

```
int a=512;
char *ptr;
ptr = (char*) a.
ptr[0] = 1
ptr[i] = 2
printf("%d", a);
```

Three address code:-

1.) a = 512
2.) ptr =

Page No.:
Date:

Generate three address code:-

① a > b and x < y or r < s.

1.) if a > b goto (4)
2.) $t_1 = 0$
3.) goto (5)
4.) $t_1 = 1$
5.) if x < y goto (8)
6.) $t_2 = 0$.
7.) goto (9)
8.) $t_2 = 1$
9.) if r < s goto (12)
10.) $t_3 = 0$
11.) goto (13)
12.) $t_3 = 1$.
13.) $t_4 = t_1$ and t_2
14.) $t_5 = t_4$ or t_3.

② x and Y or not z.

1.) if x and Y goto (4)
2 $t_1 = 0$
3. goto (5)
4 $t_1 = 1$.

. t_1 = Not z
t_2 = x and Y
$t_3 = t_1$ or t_2

Page No.:
Date:

Unit - 4

Storage organization :-

→ Procedures

```
main()
{
  —
  —
procedure1()
procedure2()
procedure3()
  —
  —
}
```

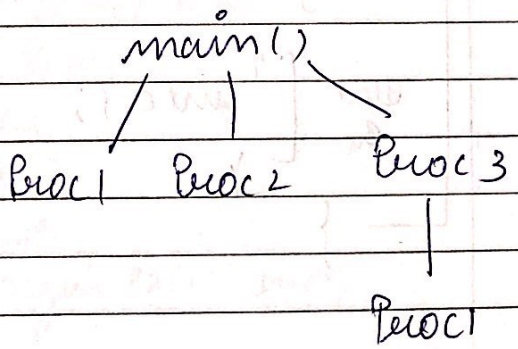
```
procedure1()
{
  —
  —
}
```

```
procedure3()
{
  —
  —
  —
}
```

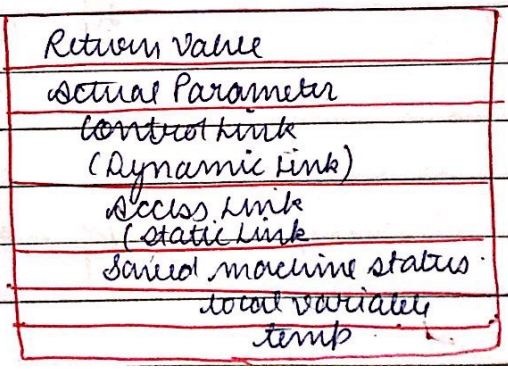
```
procedure2()
{
  —
  —
}
```

Procedure Activation Tree :-

```
proc1()
 terminate
proc2()
 terminate
proc3()
 proc1()
  terminate proc1
 terminate proc3()
```



Activation Record :-



Page No.:
Date:

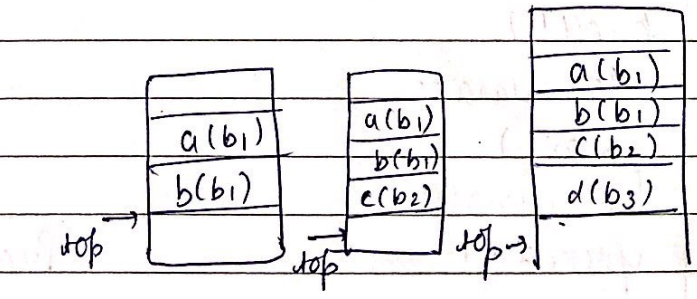
* Accessing local or non-local names in a block structured

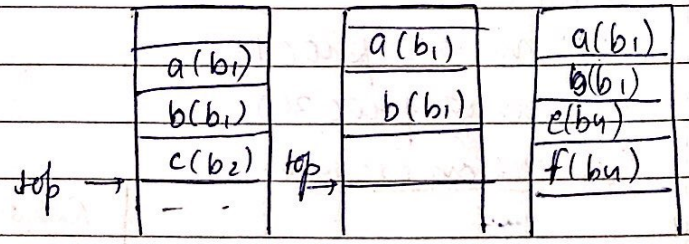
→ Static scope or lexical scope

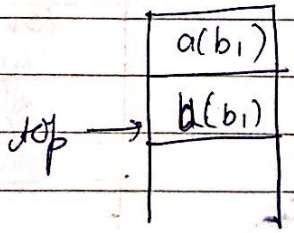
```
main()
{
  int x = 15
    {
      int x = 10;
      print x;
    }
}
```

X — X

```
Scope_obtain()
{
  int a, b;
    { int c;
      Block B2 [ Block B3 [ { int d; } ] ]
    }
  Block B1
  Block B4 [ { int e, f; } ]
}
```







Page No.:
Date:

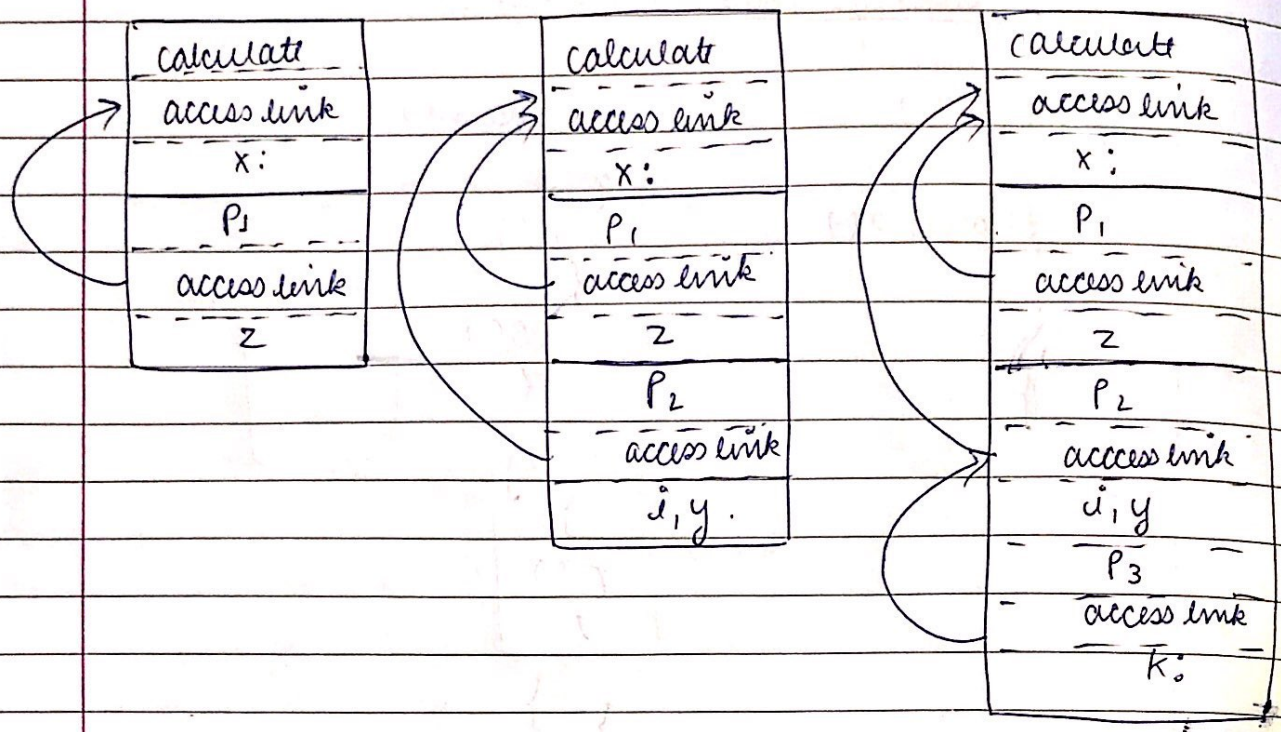
Lexical scope for Nested Function :-

```
process1()
{
   process 2()
   {
   }
}
```

```
main()
{
  A()
  {
    B()
    {
    }
  }
  C()
  {
  }
}
```

```
Program calculate;
var x : int;
procedure P1;
  var z : int;
  begin
    x := 1
  end;
procedure P2 (i : int);
  var y : int;
    procedure P3;
      var k : int;
      begin P1; end
```

```
begin
if (i<>0) then
  P2(i-1)
else
  P3
end
begin P2(1); end;
```



* Parameter Passing :-

→ Pass by value
→ Pass by reference
→ Pass by copy restore
→ Pass by name

```
{
    y: int;
    procedure P(x: int)
    {
        x = x+1;
        x = x+y;
    }
    y=2;
    P(y)
    write(y)
}
```

output :-

① call by value
o/p = 2.

② call by reference
o/p = 6.

③ call by copy-restore
o/p = 5

Page No.:
Date:
YOUVA

Q.)
```
begin:
int a, b.
procedure P(int w)
  begin                    | a = 10
  ② write(w)               | ① write (a)
     w = w * 3             |   P(a)
  ③ write (a)              | ④ write (a)
  end.                     |   end.
```

o/p :

call by value → ① 10 ② 10 ③ 10 ④ 10

call by reference ① 10 ② 10 ③ 30 ④ 30

call by copy restore ① 10 ② 10 ③ 10 ④ 30

* Pass by name :-

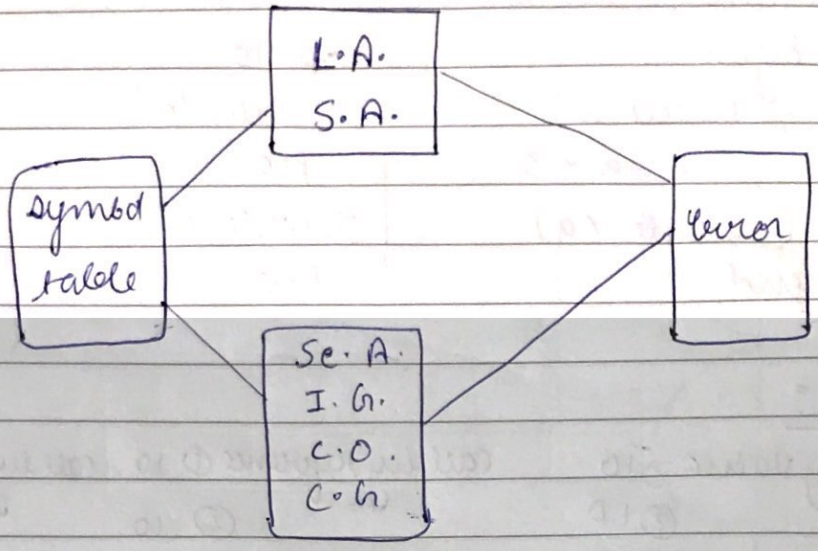
```
{ C: array [1 -- 10] of int.
  m, n: int.
  procedure x(k, j: int)
  begin
    k = k+1  ⟶  m = m+1
    j = j+2  ⟶  C[2] = C[2]+2
    end x.
    m = 2
    x(m, C[m])
    write m.
}
```

o/p = 3

C[] = {1, 2, 3, --- 10}

C[] = {1, 4, 3, 4 -- 10}

Symbol Table :-



Symbol table entries :-

1.) variable
2.) constant
3.) Function
4.) Labels
5.) Temporary variable.

```
static int a;
int b;
```

Sr.No.	Name	type	Attributes
1	a	int	static
2	b	int	auto

Type of Symbol table :-

1.) Fixed Length.
2.) variable Length.

ex:

```
int calculate;
int sum;
int x, y;
```

Page No.:	YOUVA
Date:	

c	a	l	c	u	l	a	t	e										int
s	u	m																int
x																		int
y																		int

c	a	l	c	u	l	a	t	e	$	s	u	m	$	x	$	y	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Starting	length	type
0	10	int
10	4	int
14	2	int
16	2	int

* Function of Symbol table :-

1.) Insert
   syntax :- insert (var, type)
2.) Search
   syntax :- search (var)
3.) Delete
   syntax :- delete (var)
4.) Scope management.

Eg :-
→ int value;
→ int one()
{ int a, b;
{ int c, d;
} }

→
```
int two()
{
    int e, f;
    {
        int x, y;
    }
}
```

Value	var	int	
one	proced.	int	-
two	proced.	int	

a	var	int
b	var	int

c	var	int
d	var	int

e	var	int
f	var	int

x	var	int
y	var	int

* Implementation of symbol table :-

1.) Array
2.) Linked list
3.) Binary search tree.
4.) Hash table.

1.) Array :-

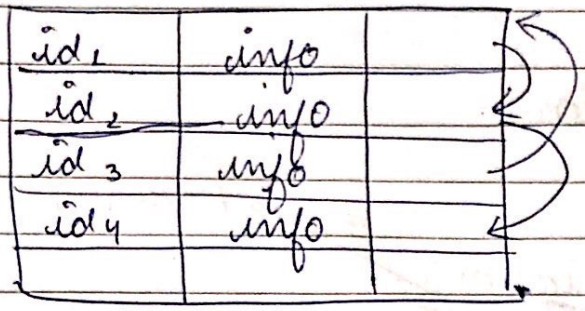
Name 1	info
Name 2	info
Name 3	info

available pointer

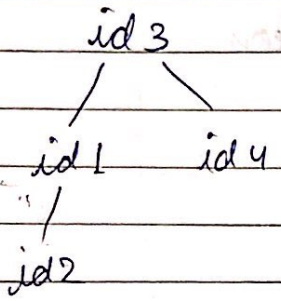
O(n)

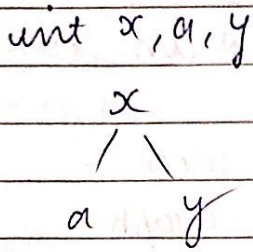
id3 > id1 > id2 > id4

2.) Linked list :-

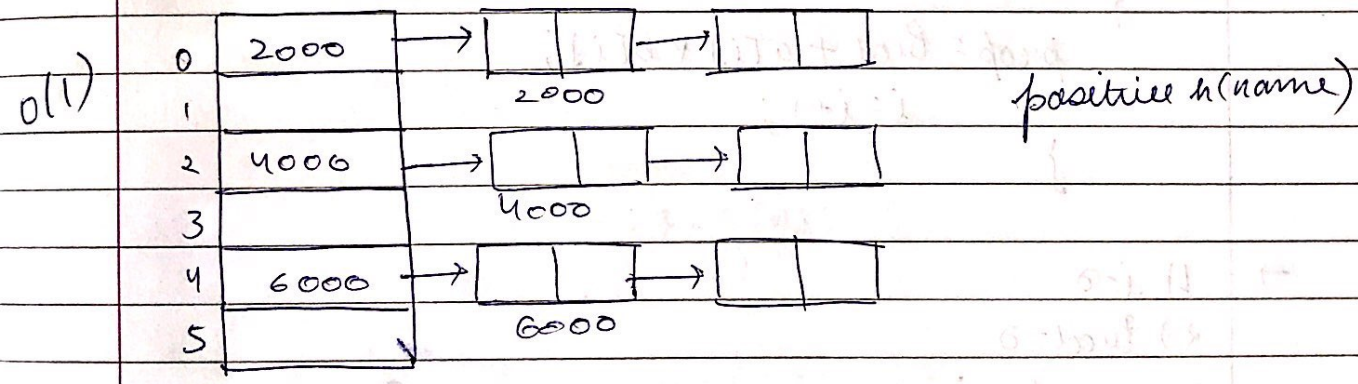


3.) Binary Search Tree





4.) Hash Table



Unit - 5

code optimization :-

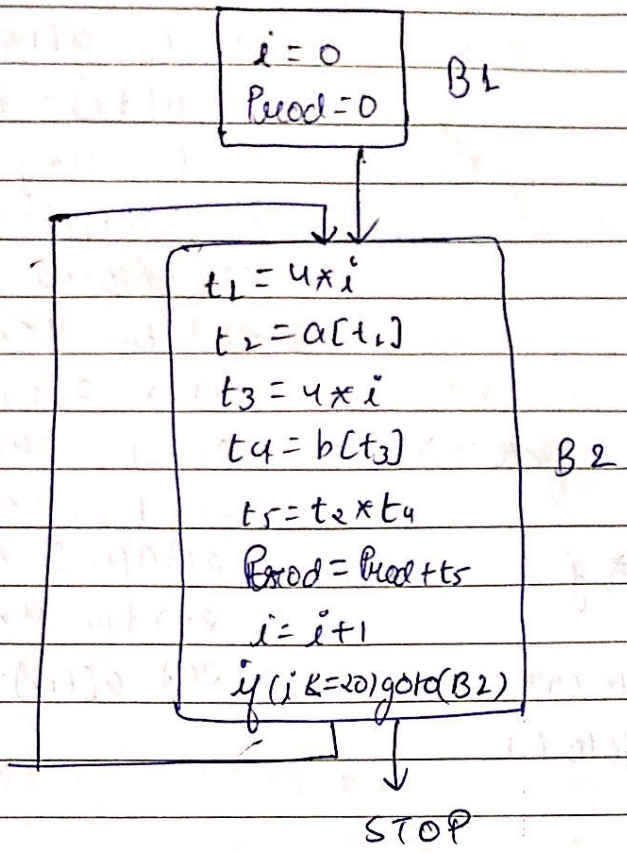
code optimization → Machine independent
code optimization → Machine dependent

Term used in code optimization.

1.) Basic block
2.) Flow graph.

```
int i=1, prod=0;
while (i<=20)
{
    prop = prod + a[i] * b[i];
        i = i+1;
}
```

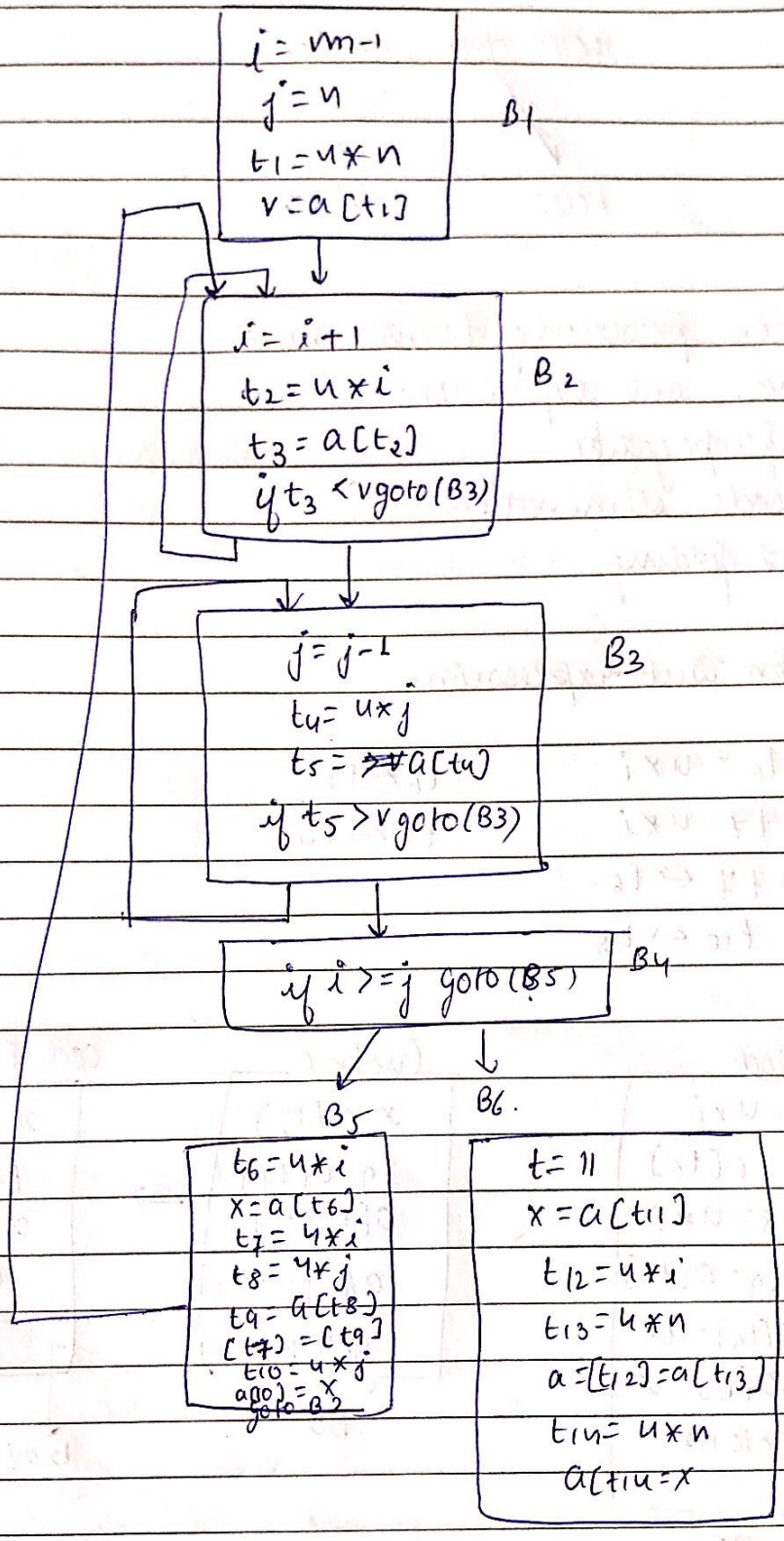
→ 1) i=0
2.) Prod=0
→ 3.) $t_1 = 4 * i$
4.) $t_2 = a[t_1]$
5.) $t_3 = 4 * i$
6.) $t_4 = b[t_3]$
7.) $t_5 = t_2 * t_4$
8) Prod = Prod + t_5
9.) i = i+1
10.) if (i<=20) goto (3)
11.) STOP.



Void quicksort :- (m,n)

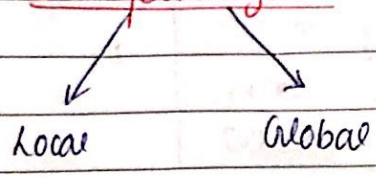
```
{
  int m,n;
  int i,j;
  int v,x;
  if (n<=m) return;
   i= m-1; j=n; v=a[n];
  while(1)
  {
    do i=i+1; while(a[i]<v);
    do j=j-1; while(a[j]>v);
    if i>=j break;
      x=a[i]; a[i]=a[n]; a[n]=x;
  }
    x=a[i]; a[i]=a[n]; a[n]=x;
    quicksort(m,j); quicksort(i+1,n)
}
```

1.) $i = m-1$
2.) $j = n$
3.) $t_1 = 4*n$
4.) $v = a[t_1]$
5.) $i = j+1$
6.) $t_2 = 4*i$
7.) $t_3 = a[t_2]$
8.) if $t_3 < v$ goto(5)
9.) $j = j-1$
10.) $t_4 = 4*j$
11.) $t_5 = a[t_4]$
12.) if $t_5 > v$ goto(9)
13.) if $i >= j$ goto( )
14.) $t_6 = 4*i$
15.) $x = a[t_6]$
16.) $t_7 = 4*i$
17.) $t_8 = 4*j$
18.) $t_9 = a[t_8]$
19.) $a[t_7] = t_9$
20.) $t_{10} = 4*j$
21.) $a[t_{10}] = x$
22.) goto(5)
23.) $t_{11} = 4*i$
24.) $x = a[t_{11}]$
25.) $t_{12} = 4*i$
26.) $t_{13} = 4*n$
27.) $a[t_{12}] = a[t_{13}]$
28.) $t_{14} = 4*n$
29.) $a[t_{14}] = x$



Page No.:
Date:
YOUVA

Code optimization



Function preserving transformation:-

1.) common sub expressions
2.) copy propagation
3.) head code elimination
4.) Constant folding

1.) Common Sub expression -

B5

$t_6 = 4 * i$ $\quad t_6 \leftrightarrow t_2$

$t_7 = 4 * i$ $\quad t_8 \leftrightarrow t_4$

$t_7 \leftrightarrow t_6$

$t_{10} \leftrightarrow t_8$

Local Block

$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * i$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B2

B5

→

Global

$x = a[t_2]$
$t_9 = a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B2

B5

→

Copy Propagation

$x = a[t_2]$
~~$t_9 = a[t_4]$~~
$a[t_2] = a[t_4]$
$a[t_4] = x$
goto B2

Dead code elimination

B6

$t_{12} \rightarrow t_{11}$

$t_{14} \rightarrow t_{13}$

```
t11 = 4*i
x = a[t11]
t13 = 4*n
a[t11] = a[t13]
a[t13] = x
```

→

```
x = a[t2]
a[t2] = a[t1]
a[t1] = x
```

→

```
x = a(t2)
a[t2] = a(t1)
a(t1) = x
```

B6

$t_{11} \leftrightarrow t_{a2}$

$t_{13} \leftrightarrow t_1$

* Loop optimization:-

1.) Code motion

2.) Induction & variable elimination

3.) Reduce in strength.

1.) Code motion

```
while (i <= max-1)
{
  ==
  i = i+1;
}
```

```
n = max-1
while (i<=n)
{
  ==
  i = i+1;
}
```

B2

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3 < v goto B2
```

$t_2 = 4 * i$

```
t2 = t2+4
t3 = a[t2]
if t3 < v goto B2
```

$t_4 = 4 * i$ → code motion

```
t4 = t4-4
t5 = a[t4]
if t5 > v goto B3
```

Induction variable elimination.

Page No.:
Date:

* Directed Acyclic Graph (DAG)

Implementing transformation of basic block of the address code.

1.) Leaf nodes are labeled as unique identifier or variable const.
2.) Interior nodes are labeled as operator.
3.) Interior nodes are provided with the identifier for labels.

Ex:-

```
sum=0
for(i=0; i<=10; i++)
    sum = sum+a[i].
```

Three address code:-

1.) sum = 0
2.) i = 0

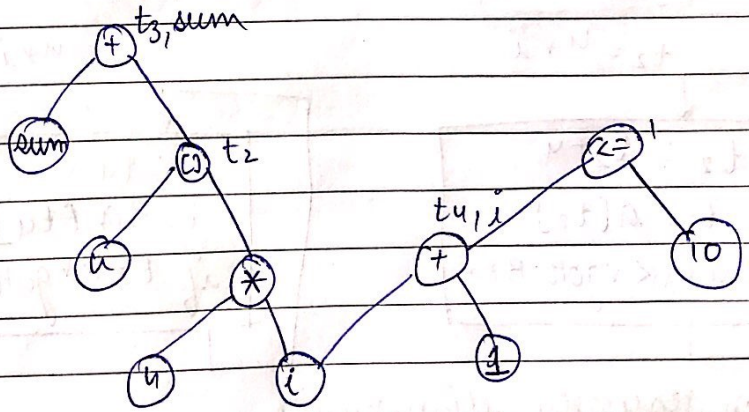
Block B1.

3.) $t_1 = 4 * i$
4.) $t_2 = a[t_1]$
5.) $t_3 = sum + t_2$
6.) $sum = t_3$
7.) $t_4 = i + 1$
8.) $i = t_4$
9.) if $i <= 10$ goto (3)

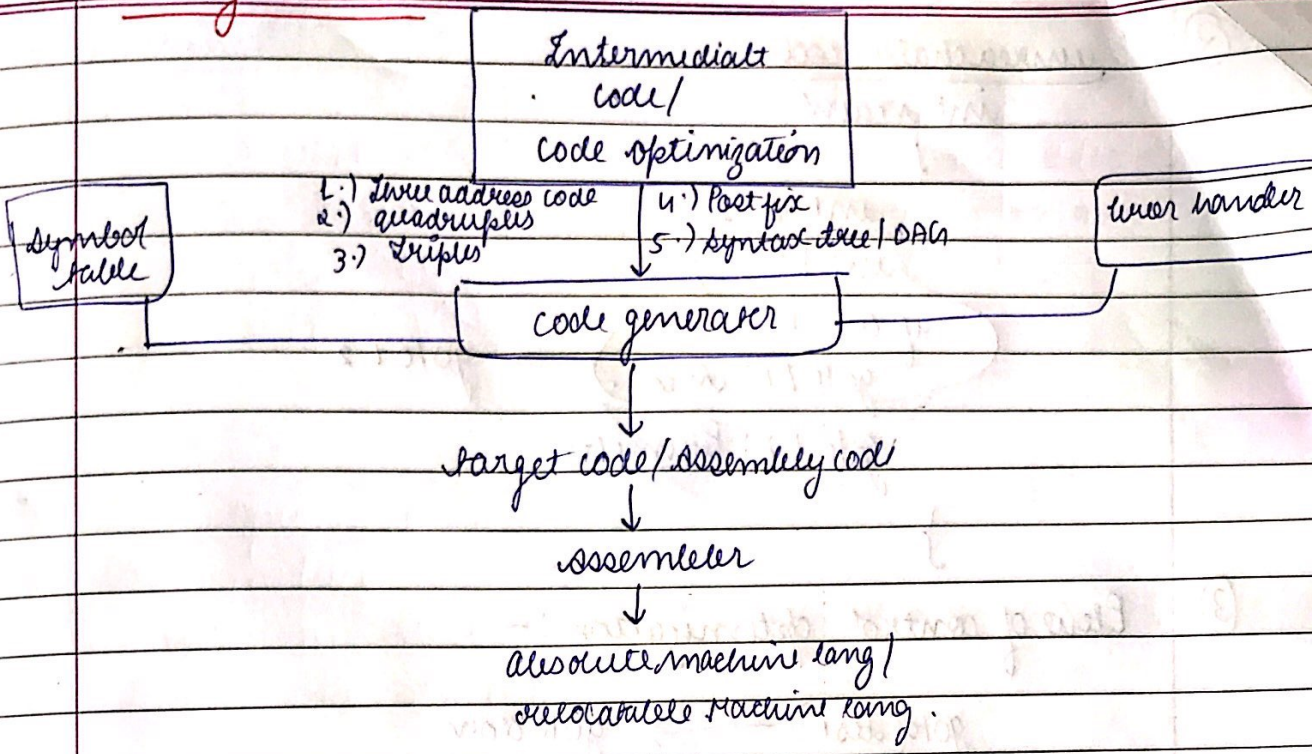
Block B2.

$sum = sum + t_2$.

$i = i + 1$



Code generator :-



Major Properties :

1.) correctness
2.) High quality
3.) Efficiently. Resources of target machine.
4.) Faster code generator.

Issues in code generation :-

1.) Input to the code generator
2.) Target code / Program.
3.) Memory Management.
4.) Instruction selection
5.) Register allocation.

* Peephole optimization :-

characteristics :

① Redundant instruction :-

MOV x, R0
MOV R0, x / x

Page No.:
Date:

(2) unreachable code

```
int main()
{
    int a=1;
    int b=2;
    if a==b
    goto L1: Area  ---> goto L2
    goto L2: Perimeter
}
```

(3) Flow of control optimization :-

goto test ⟶ goto done

```
    ---
    ---
    ---
test : goto done
    ---
    ---
    ---
done :
    ---
    ---
    ---
```

(4) Algebraic simplification :-

$x = x + 0$

OR

$a = 1 * b$

(5) Reduction in strength :-

~~Effect~~ + (add) ⟵ * (multiplication)

* - (sub) ⟵ / (Division)

Page No.:
Date:

(6) Machine Idioms :-

```
a = a+1 ;
b = b-1 ;
```

] auto increment & auto decrement will be applied on a, b.