LALR :-

① I_3 I_6 = I_{36}

I_4 I_7 = I_{47}

I_8 I_9 = I_{89}

State	Action			Goto	
	c	d	$	E	B
I_0	S_{36}	S_{47}		1	2
I_1			A		
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		

Syntax directed defination :- (SDD)

SDD = Grammar + Semantic rules

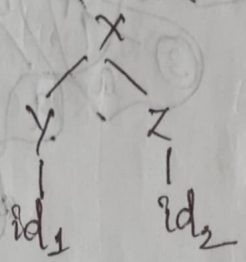
SDD = $E \rightarrow E+T$ + $E.val = E.val + T.val$

→ A SDD is a CFG, together with semantic rules.

→ Attributes are associated with grammar symbols & semantic rules are associated with productions

→ If x is a symbol & a is one of its attribute, then x.a denotes value at node x.

Ex :- $X \rightarrow YZ$
$Y \rightarrow id_1$
$Z \rightarrow id_2$



Parse tree → Semantic → Anotated tree.

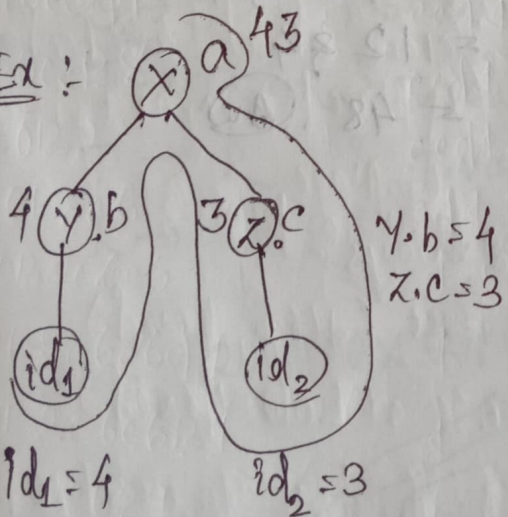
Anotated tree :-

→ A parse tree with attribute value at each node is annotated parse tree.

Synthesized Attributes :-

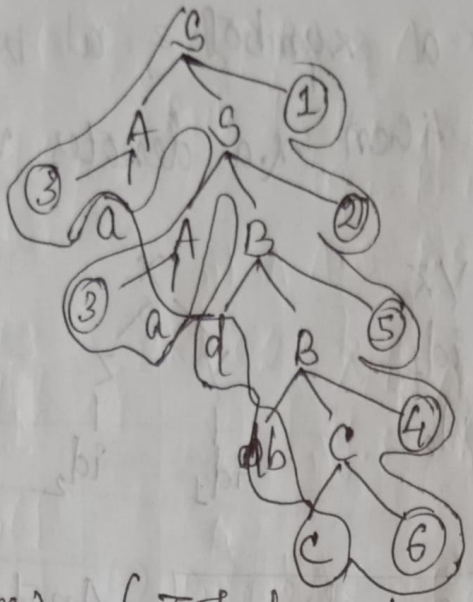
→ If its value at the parse tree node is determined from the attributes value of the children of the node, then it is called synthesized attributes.

Ex :-



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

I/P :- aadbc



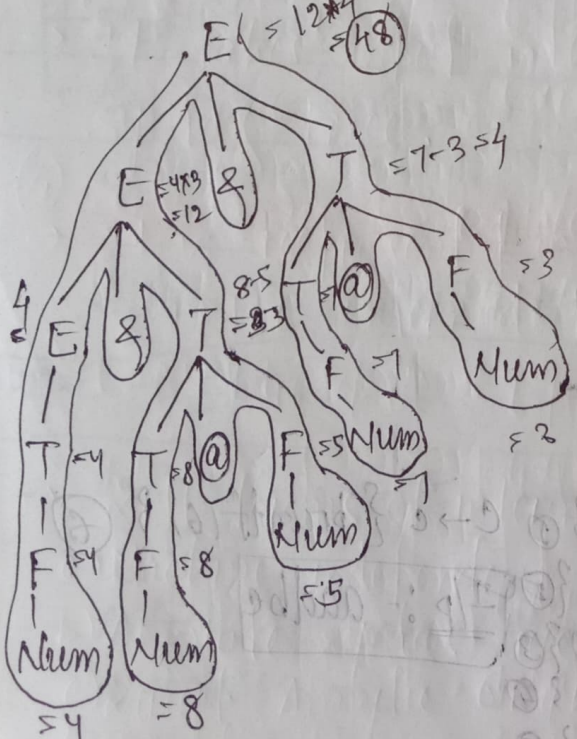
O/p :-
3364521

(Bottom-up parsing) (Top-down parsing)

O/p :- 3364521

Ex :- E → E & T { E.val = E.val * T.val }
/T { E.val = T.val }
T → T @ F { T.val = T.val - F.val }
/F { T.val = F.val }
F → num { F.val = num }

I/p :- 4 & 8 @ 5 & 7 @ 3 = 4 & (8-5) & (7-3)
= 4 & 3 & 4
= 12 & 4
= 48 . Ans



Types of SDT :- (Semantic Directed Translation)

SDT

S-attribute L-attribute

S-attribute :-

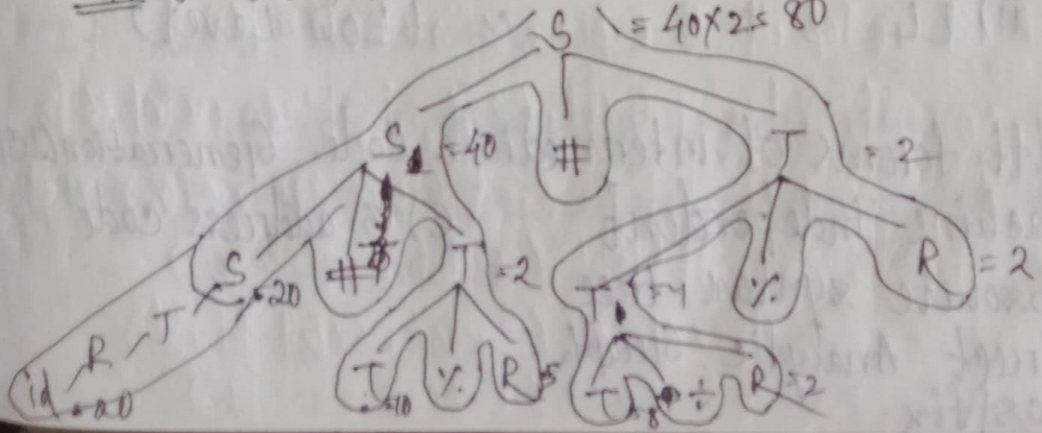
→ Based on synthesized attributes
→ uses bottom-up parsing
→ Semantic rule always written at rightmost posn in right hand side.

L-attribute :-

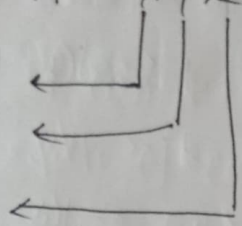
→ Based on both synthesized & inherited attribute
→ It uses top-down parsing.
→ Semantic rules are written any where in the RHS.

Ex :- $S \to S_1 \# T$ { S.val = S_1.val * T.val }
$S \to T$ { S.val = T.val }
$T \to T_1 \% R$ { T.val = T_1.val ÷ R.val }
$T \to R$ { T.val = R.val }
$R \to id$ { R.val = id.val }

I/P :- 20 # 10 % 5 # 8 % 2 % 2



$A \to XYZ$



derived value from its left side only not from right side.

Ex:- $A \to PQ$ & $A \to XY$

{Rule 1: $P.i = A.i + 2$ (inherit)
$Q.i = P.i + A.i$ (both)
$A.S = P.S + Q.S$} (synthesize)

{Rule 2: $X.i = A.i + Y.S$ (does not possible)
$Y.i = X.S + A.i$} (both)

⇒ Rule 2 is not defined & Rule 1 is L-attribute STD.

Intermediate code generation :- (ICG)

→ It is used to translate the source code into machine code & it lies b/w HLL & LLL.

→ If the compiler directly translate source code & destination code without generating the intermediate code then a full native compiler is required for each new machine.

→ It takes input in the form of an anotated syntax tree.

→ It is represented in 2 ways -
i) HL Intermediate code (High Level)
ii) LL " " (Low Level)

→ Diff. forms of intermediate code Generation (ICG)
a) Machine independent
b) Absolute syntax tree
c) Direct Analytic Graph
d) postfix
e) 3-address code

Ex :- $x = \underbrace{(a+b)}_{t_1} * \underbrace{(c+d)}_{t_2}$

$t_1 = (a+b)$

$t_2 = (c+d)$

$t_3 = t_1 * t_2$

$x = t_3$

3-address code :-

op - operator

1) Assignment
   - $x = y$ op $z \longrightarrow x = y + z$
   - $x =$ op $z \longrightarrow x =$ op z
   - $x = y \longrightarrow x = y$

2) Jump
   - conditional — if x relop z goto L
   - unconditional — goto L

   (relop - relational operator
   L - level)

3) Array Assignment → $x = y[i]$
   $x[i] = y$

4) Pointer & address Assignment → $x = \&y$
   $x = *y$

:- Boolean Expression :-

→ It has 2 primary purposes
   i) used for computing logical values
   ii) used as conditional expression using if-then-else or while-do

Ex:- 1) $E \rightarrow E$ OR E ~~{E → E1 OR E2}~~

2) $E \rightarrow E$ AND E

3) $E \rightarrow$ NOT E

4) $E \rightarrow (E)$

5) $E \rightarrow$ id relop id

6) $E \rightarrow$ True

7) $E \rightarrow$ False

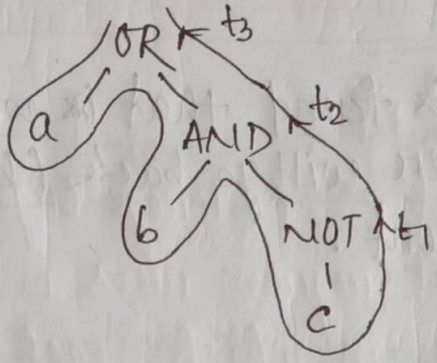
Production Rule	Semantic Rules
① $E \rightarrow E_1$ OR E_2	{ E.place := 'new temp(); Exit (E.place := 'E1.place 'OR' E2.place) }
② $E \rightarrow E_1$ AND E_2	{ E.place := 'new temp(); Exit (E.place := 'E1.place 'AND' E2.place) }
③ $E \rightarrow$ NOT E_1	{ E.place := 'new temp(); Exit (E.place ':=' 'NOT' E1.place }
④ $E \rightarrow (E_1)$	{ E.place := ~~'new temp();~~ E1.place } ~~Exit (E.place := (E1.place~~
⑤ $E \rightarrow$ id relop id	{ E.place := 'new temp(); Exit ('if id1.place relop id2.place 'goto' nextstate +3) Exit (E.place := '0') Exit ('goto' nextstate +2) } Exit (E.place := '1')

⑥ E → True { E.place := 'new temp();
{ Exit(E.place := '1')

⑦ E → false { E.place := 'new temp();
{ Exit(E.place := '0')

Ex :- a OR b AND NOT c

Sol^{n} :-



t_1 = NOT c
t_2 = b AND t_1
t_3 = a OR t_2

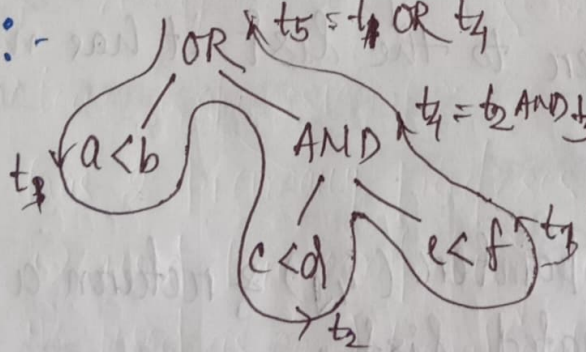
Ex :- if a<b then 1 else 0

Sol^{n} :-

100: if a<b goto 103
101: t_1 = 0
102: goto 104
103: t_1 = 1

Ex :- a<b or c<d and e<f

Sol^{n} :-



108: if e<f goto 111
109: t_3 = 0
110: goto 112
111: t_3 = 1

104: if c<d goto 107
105: t_2 = 0
106: goto 108
107: t_2 = 1

100: if a<b goto 103
101: t_3 = 0
102: goto 104
103: t_3 = 1

Back Patching :-

→ It comes into play in the intermediate code generating state of compiler.

→ There are times when the compiler has to execute the jump instruction, but it does not know where it so it will fill in some kind of filler value at this point & remembered that this happens.

→ then once the value is found that will actually be used the compiler will go back (back patch) & fill in the current.

Operations :-

i) Makelist(i)
ii) Merge(i,j)
iii) BackPatch(p,i)

i) Makelist(i) :-

→ It will create a new list containing only i, & inserts into the array of quadruplee & return a pointer to the list it has made.

ii) Merge(i,j) :-

→ It concatinates pointer (i,j) & return a pointer to the concatenated list.

iii) Backpatch(p,i) :- at the target

→ Insert i level for each of the list pointed by p.

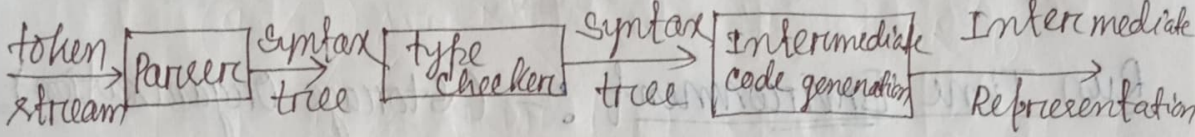
Type checking :- (TC)

→ It is a module of a compiler denoted to time checking task.

→ It is of 2 type ┬ static (compile time)
                  └ Dynamic (Run time)

→ The design of ~~time~~ TC depends on syntacting structure of language construction.

Position of type checker while compilation :-



Type Expression & Type systems :-

→ It will denote the type of language construction.

→ It is of 2 type ┬ Basic type (int, float, —)
                  └ Type name (Array, String, —)

→ Type system :-

→ It is a collection of rule for assigning the type

→ The components ┬ Basic type
                 ├ type construction
                 └ type equivalence ┬ name equivalence
                                    └ Structural equivalence

Type conversion or Type casting :-

→ It is a conversion of one type to another.

→ There are 2 type of conversion ┬ Implicit
                                 └ Explicit

Implicit :-

→ If compiler converts one data type to another automatically, it is Implicit type conversion.

→ There is no data loss.

Explicit :-

→ When data of one type is converted explicitely to another type with the help of predefined function.

→ There is a data loss.

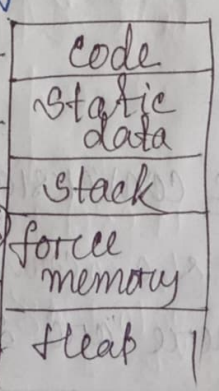
Run Time Environment :- (Storage Organization)

→ The executing target program runs in its own logical address in which each program value has its own address.

→ The management & organization of this logical address region is shared b/w the compile ~~org~~ operating system & target machine.

→ The OS maps the logical address into physical address which are usually spread throughout memory.

Subdivision of Run Time Memory :-



Memory loc^{n} for code are determine at compile time

Loc^{n} are determined at compile time

Data objects allocated at run time

Other dynamically allocated (Activation records) object at run time (Ex: Malloc area of C)

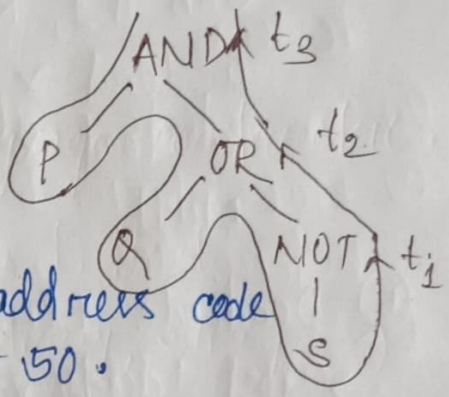
→ Run time storage comes into proex, where the byte is used to show the smallest unit of addressible memory.

→ Run time storage can be subdivided to all the diff. components of an executible program.

→ It generated
→ Static data object
→ Dynamic " " (heap).
→ Automatic " " (stack)

Q :- P AND Q OR NOT S

Sol^n :- t_1 = NOT S
t_2 = Q OR t_1
t_3 = P AND t_2



Q : Translate a<b into 3-address code sequence & start numbers at 50.

Sol^n :- 50 : if a<b goto 53
51 : t_1 = 0
52 : goto 54
53 : t_1 = 1

Q :- S → AS/b
A → SA/a] CLR

S → bS'
S' → ∈/ASS'/aS'

Sol^n :- S' → .S, $
S → . AS, $
S → . b, $
A → .SA, b
A → . a, b

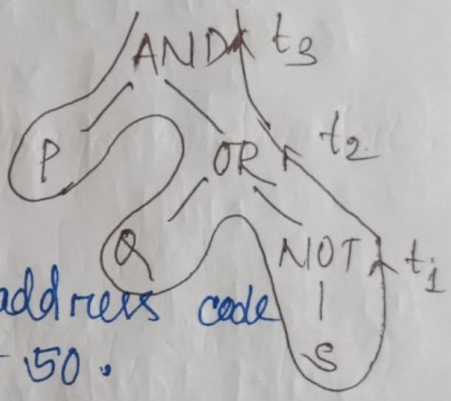
→ Run time storage comes into picx, where the byte is used to show the smallest unit of addressible memory.

→ Run time storage can be subdivided to all the diff. components of an executible program.

→ It generated
→ Static data object
→ Dynamic " " (heap).
→ Automatic " " (stack)

Q :- P AND Q OR NOT S

Sol^n :- t_1 = NOT S
t_2 = Q OR t_1
t_3 = P AND t_2



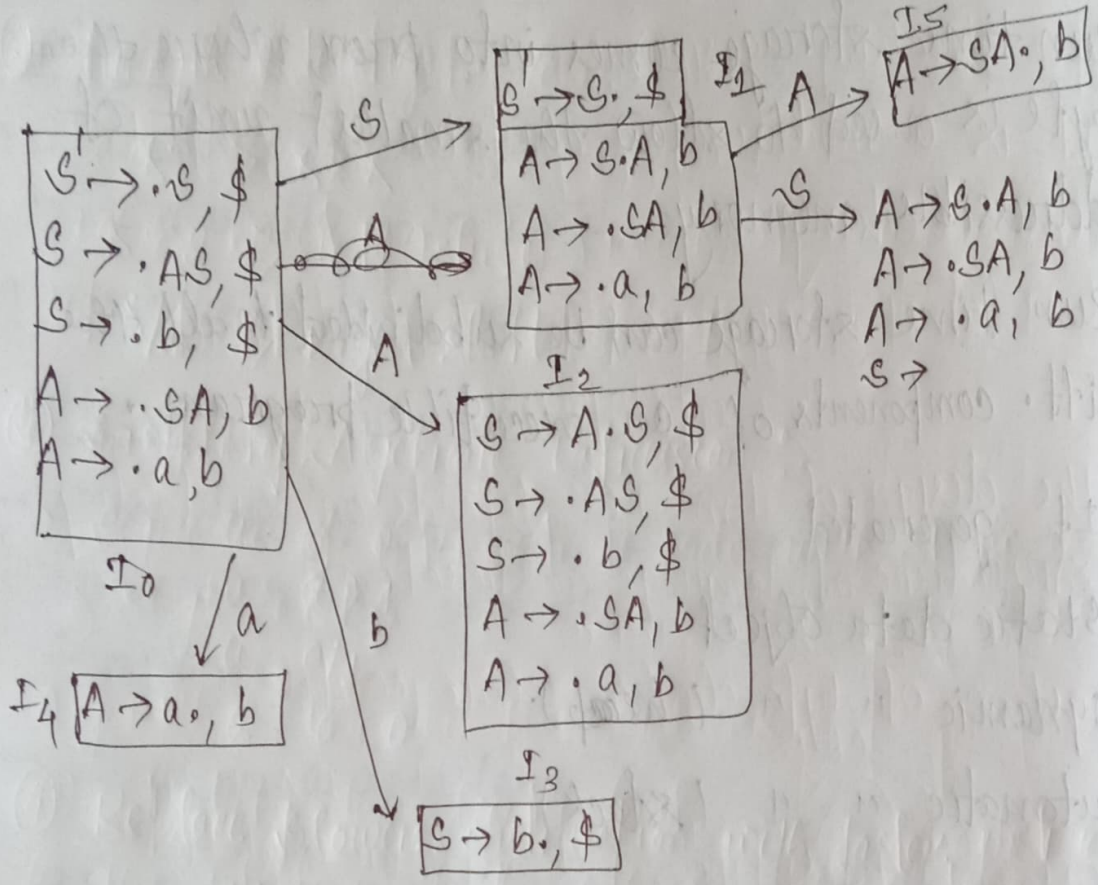
Q : Translate a<b into 3-address code sequence & start numbers at 50.

Sol^n :- 50 : if a<b goto 53
51 : t_1 = 0
52 : goto 54
53 : t_1 = 1

Q :- S → AS | b
A → SA | a] CLR

S → bS'
S' → ∈ | ASS' | aS'

Sol^n :- S' → .S, $
S → . AS, $
S → . b, $
A → .SA, b
A → . a, b


S' → .S, $
S → .AS, $
S → .b, $
A → .SA, b
A → .a, b
I0
S
A
b
a
S' → S., $
I1
A → S.A, b
A → .SA, b
A → .a, b
A
S
A → SA., b
I5
A → S.A, b
A → .SA, b
A → .a, b
S →
A → a., b
I4
S → A.S, $
S → .AS, $
S → .b, $
A → .SA, b
A → .a, b
I2
S → b., $
I3

Code Optimization :-

⇒ It is divided into 2 type-

- Platform dependent technique (PDT)
- Platform independent technique (PIT)

PDT :-

- Peephole optimization
- Instruction level parallelism
- Data level parallelism
- Code optimization
- Redundant Resources

PIT :-

- Loop optimization
- Constant folding
- Constant propagation
- common subexpression Elimination

Peephole optimization :-

① Redundant load & store :-

```
a = b+c
d = a+e
MOV b, R0
Add c, R0
MOV R0, a      } Redundant load & store
MOV a, R0
Add e, R0
MOV R0, d
```

② Strength Reduction :-

$x^2 \Rightarrow x*x$

$x*2 \Rightarrow$ left shift

$x/2 \Rightarrow$ Right shift

③ Simplify Algebraic Expression :-

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

④ Replace slower instruction with faster instruction;

Add #1, R ⇒ INC R
Sub #1, R ⇒ DEC R

a load x
a load x
Mull

⇒

a load x
dup
Mul

⑤ Deadcode Elimination :-

```
int dead(void)
{ int a = 20;
  int b = 30;
  int c;
    c = a*10;
  return c;
  b = 30;
  b = b*10;
  return 0;
}
```

⇒ dead code

Loop Optimization :-

① Code Motion (frequency Reduction) :-

```
a = 100                          a = 100
while (a>0)                      x = y+z;
{                                while (a>0)
    x = y+z;                     {
    if (a%x == 0)        =>          if (a%x == 0)
        Pf (" ")                         Pf (" ")
}                                }
```

② Loop fusion (Jamming) :-

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

③ Loop unrolling :-

```
for (i=0; i<5; i++)              Pf ("Hanza");
{                                Pf ("Hanza");
    Pf ("Hanza");           =>   Pf ("Hanza");
}                                Pf ("Hanza");
                                 Pf ("Hanza");
```

Principle Source of Optimization :-

→ It is of 2 types - + Machine independent
+ " dependent

Machine Independent :-

→ These are program transformation that improve target code without taking into consideration any

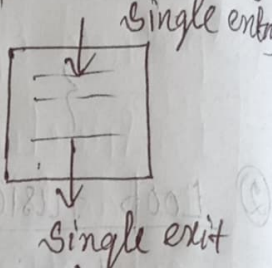
properties of target M/c.

M/c dependent :-

→ It is based on register allocation or utilization of special M/c instruction sequence.

Basic block :-

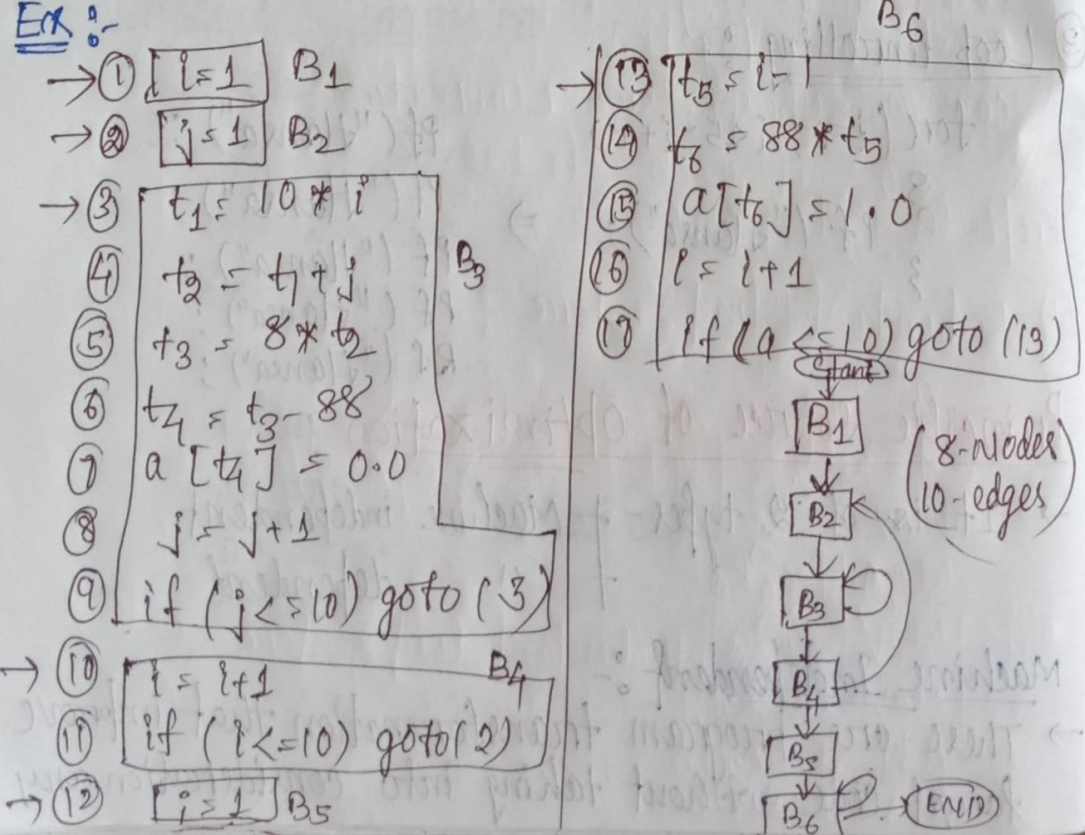
→ Sequence of intermediate code with single entry & single exit.



Algorithms to make Basic Blocks :-

① Identify the leaders first
② the 1st line of the code is the leader
③ address of conditional, unconditional goto are leaders
④ Immediate next line of goto are leaders
⑤ Make basic block from leaders to line before the next leader.

Ex :-



CLR ~~SLR LALR~~ :- (canonical LR)

→ first 3 steps same.

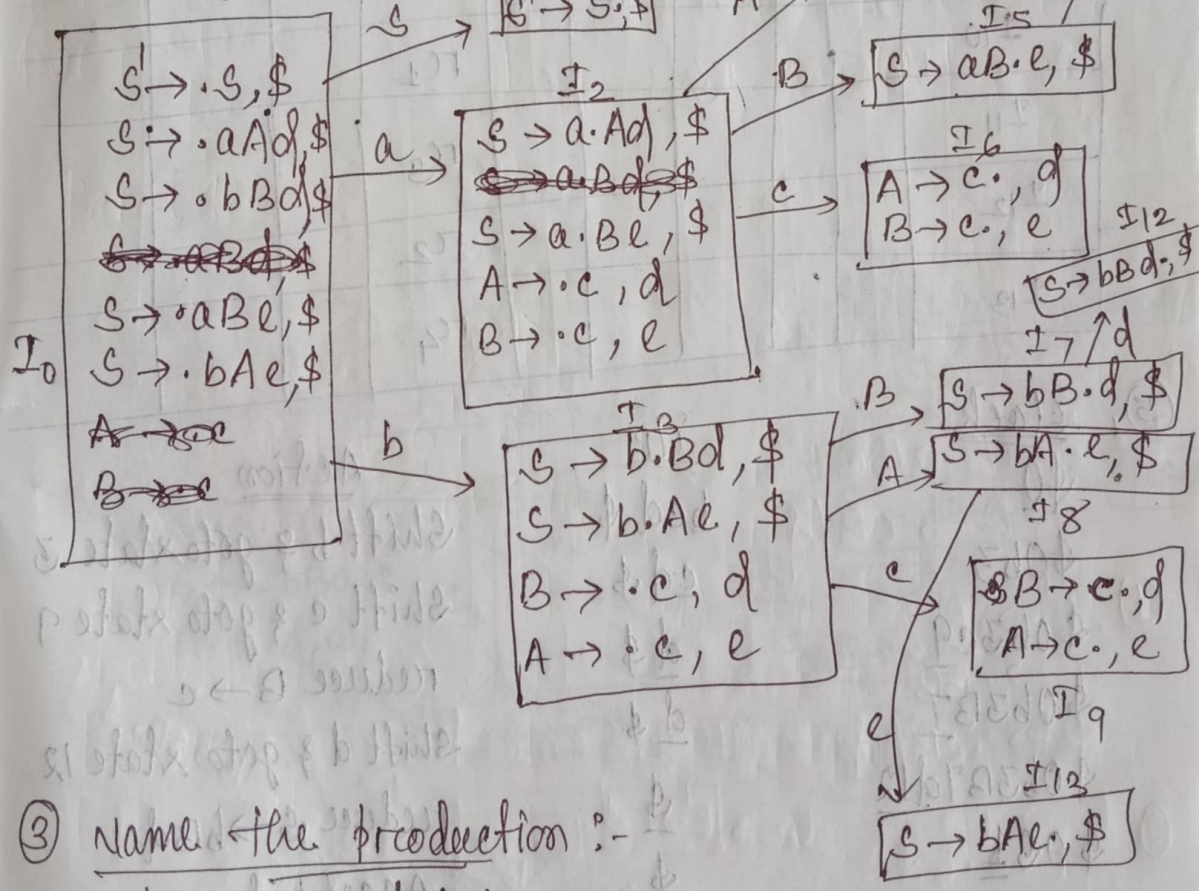
LR(0) - put reduce in every row
SLR - put reduce in follow
CLR - " " " look ahead
LALR -

$S \to aAd / bBd / aBe / bAe$
$A \to c$, $B \to c$

① Augmented grammar :-

$S' \to .S$
$S \to .aAd$
$S \to .bBd$
$S \to .aBe$
$S \to .bAe$
$A \to .c$
$B \to .c$

② canonical form :-



③ Name the production :-

$S \to aAd$ (1)
$S \to bBd$ (2)
$S \to aBe$ (3)
$S \to bAe$ (4)
$A \to c$ (5)
$B \to c$ (6)

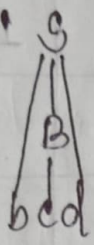
④

State	Action						Goto		
	a	b	c	d	e	$	S	A	B
I_0	S_2	S_3					1		
I_1						A			
I_2			S_6					4	5
I_3			S_9					8	7
I_4				S_{10}					
I_5					S_{11}				
I_6				r_5	r_6				
I_7				S_{12}					
I_8					S_{13}				
I_9				r_6	r_5				
I_{10}						r_1			
I_{11}						r_3			
I_{12}						r_2			
I_{13}						r_4			

⑤

Stack	i/p	Action
$0	bcd$	Shift b & goto state 3
$0b3	cd$	Shift c & goto state 9
$0b3c9	d$	reduce B → c
$0b3B7	d$	Shift d & goto state 12
$0b3B7d12	$	reduce S → bBd
$0S1	$	Accepted.

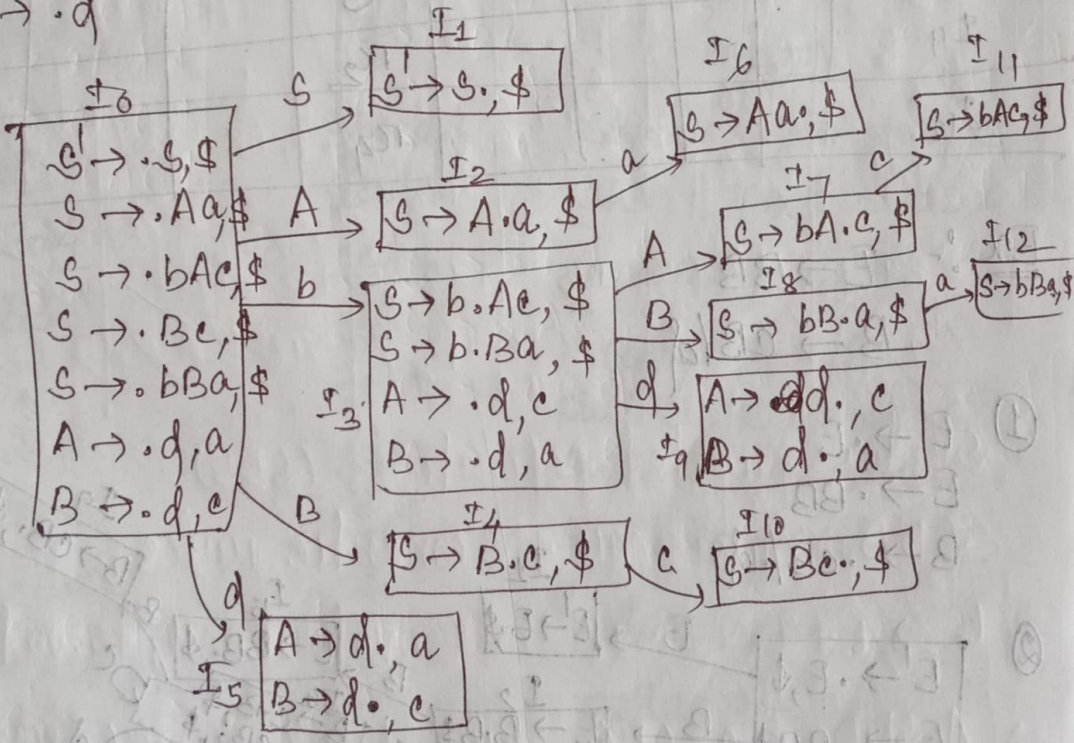
Parse tree :-



Ex :- $S \to Aa \mid bAc \mid Bc \mid bBa$
$A \to d$
$B \to d$

① $S' \to .S$
$S \to .Aa$
$S \to .bAc$
$S \to .Bc$
$S \to .bBa$
$A \to .d$
$B \to .d$

②



③ $S \to Aa$ (1) $A \to d$ (5)
$S \to bAc$ (2) $B \to d$ (6)
$S \to Bc$ (3)
$S \to bBa$ (4)

④ State

	Action					Goto		
	a	b	c	d	$	S	A	B
I_0		S_3		S_5		1	2	4
I_1					A			
I_2	S_6							
I_3				S_9			7	8
I_4			S_{10}					
I_5	r_5		r_6					
I_6					r_1			
I_7			S_{11}					
I_8	S_{12}							
I_9	r_6		r_5					
I_{10}					r_3			
I_{11}					r_2			
I_{12}					r_4			

Ex :- $E \to BB$

$B \to cB \mid d$

① $E' \to .E$

$E \to .BB$

$B \to .cB \mid .d$

② I_0: $E' \to .E, \$$; $E \to .BB, \$$; $B \to .cB, c/d$; $B \to .d, c/d$

I_1: $E' \to E., \$$

I_2: $E \to B.B, \$$; $B \to .cB, \$$; $B \to .d, \$$

I_3: $B \to c.B, c/d$; $B \to .cB, c/d$; $B \to .d, c/d$

I_4: $B \to d., c/d$

I_5: $E \to BB., \$$

I_6: $B \to c.B, \$$; $B \to .cB, \$$; $B \to .d, \$$

I_7: $B \to d., \$$

I_8: $B \to cB., c/d$

I_9: $B \to cB., \$$

I_{10}: $B \to c.B, c/d$; $B \to .cB, c/d$; $B \to .d, c/d$

I_{11}: $B \to d., c/d$

I_{12}: $B \to cB., c/d$

③ $E \to BB$ ①
$B \to cB$ ②
$B \to d$ ③

④ State

	Action			Goto	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			A		
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	r_2		
I_9			r_2		

⑤ State

State	i/p	Action
$0	cdccd$	Shift c & goto state 3
$0c3	dccd$	Shift d & goto state 4
$0c3d4	ccd$	reduce $B \to d$
$0c3B8	ccd$	reduce $B \to cB$
$0B2	ccd$	Shift c & goto state 6
$0B2c6	cd$	Shift c & goto state 6
$0B2c6c6	d$	Shift d & goto state 7.
$0B2c6c6d7	$	reduce $B \to d$
$0B2c6c6B9	$	reduce $B \to cB$
$0B2c6B9	$	reduce $B \to cB$
$0B2B5	$	reduce $E \to BB$
$0E1	$	Accepted

LALR :-

① I_3 I_6 = I_{36}

I_4 I_7 = I_{47}

I_8 I_9 = I_{89}

State	Action			Goto	
	c	d	$	E	B
I_0	S_{36}	S_{47}		1	2
I_1			A		
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		

Syntax directed defination :- (SDD)

SDD = Grammar + Semantic rules

SDD = $E \rightarrow E+T$ + $E.val = E.val + T.val$

→ A SDD is a CFG, together with semantic rules.

→ Attributes are associated with grammar symbols & semantic rules are associated with productions