Date
13-3-13
Wednesday

• CD

Ques: If (a<b and c>d)
{ if (e<f)
s = s+1;
else
s = s-1;
}
else
p = p+1;

Solve:
```
0   if a<b goto 2
1     goto 12
2   if c>d goto 4
3     goto 12
4   if e<f goto 6
5     goto 9
6   t1 = s+1
7     s = t1
8     goto 12
9   t2 = s-1
10    s = t2
      goto 12
11  t3 = p+1
12
13    p = t3
```

Q

Ques: for (i=0; i<20; i++)
s = s+1;

Solve:
0 i=0
1 if i<20 goto 3
2 goto 8
3 t1 = s+1
4 s = t1
5 t2 = i+1
6 i = t2
7 goto 1

Ques:
while (a<b)
{ if (c>d)
s = s+1;
else
s = s-1;
}

Solve:
0 if a<b goto 2
1 goto 10
2 if c>d goto 4
3 goto 7
4 t1 = s+1
5 s = t1
6 goto 0
7 t2 = s-1
8 s = t2
9 goto 0
10) stop

• EXAMINATION QUESTIONS:

Ques: While (A<C and B>0) do
    if A==1 then
     C=C+1
    else
     while A<=0 do
      A=A+3

Solve:
```
0  if A<C goto 2
1    goto 12
2  if B>0 goto 4
3    goto 12
4  if A==1 goto 6
5    goto 8
6    t1=C+1
7    C=t1
     goto 0
8  if A<=0 goto 10
9    goto 0
10   t2=A+3
11   A=t2
12   goto 8
13   goto 0
14
```

Ques: while (a<c and b<d) do
    if a==1 then
     s=s+1
    else
     c=c+2
    end do

Solve:

```
0  if a<c goto 2
1    goto 12
2  if b<d goto 4
3    goto 12
4  if a==1 goto 6
5    goto 8
6    t1 = S+1
7    S = t1
     goto 0
8    t2 = C+2
9    C = t2
10   goto 0
11
```

Ques:

```
begin
  while (a>b) do
  begin
    z = y+z
    a = a-b
  end
  z = y-z
end
```

Solve:

```
0  if a>b goto 2
1    goto 7
2    t1 = y+z
3    z = t1
4    t2 = a-b
5    a = t2
6    goto 0
```

7 t3 = y - x

8 x = t3

Date: 14-5-13 Thursday

•CD

Ques Write a three address code for the following expression and also represent it by quadruple, triple and Indirect triple.

A[i] := B

Solve: A[i] := B

t1 := A[i]

t1 := B

(i) Quadruple Representation :

S.No	Operator	Op1	Op2	Result
1	=[ ]	A	i	t1
2	=	B	–	t1

(ii) Triple Representation:

	Operator	Op1	Op2
(1)	=[ ]	A	i
(2)	=	(1)	B

(iii) Indirect Triple Representation:

Pointer to triple

(1) (11)

(2) (12)

Operator	Op1	Op2
=[]	A	C
=	(11)	B

Ques: $y = a+b$

$x[i] := y$

$z = z+2$

$y[j] = z$

Solve:

$t1 := a+b$

$y := t1$

$t2 := x[i]$

$t2 := y$

$t3 := z+2$

$z := t3$

$t4 := y[j]$

$t4 := z$

(i) Quadruple representation:

S.No	Operator	Op1	Op2	result
(1)	+	a	b	t1
(2)	=	y t1	-	y
(3)	=[]	z	i	t2

	Operator	Op1	Op2	result
(4)	=	y	—	t2
(5)	+	x	2	t3
(6)	=	*3	—	x
(7)	=[ ]	y	i	t4
(8)	=	z	—	t4

(ii) Triple representation

S.No	Operator	Op1	Op2
(1)	+	a	b
(2)	=	y	(1)
(3)	=[ ]	z	i
(4)	=	(2)	y
(5)	+	x	2
(6)	=	x	(5)
(7)	=[ ]	y	i
(8)	=	(4)	z

(iii) Indirect Triple representation

	Statement
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)
(7)	(17)
(8)	(18)

S.No	Operator	op1	op2
(11)	+	a	b
(12)	=	y	(11)
(13)	=[ ]	z	i
(14)	=	(13)	y
(15)	+	x	2
(16)	=	z	(15)
(17)	=[ ]	y	i
(18)	=	(17)	z

Ques. while (a>b or b<c and a>20) do
if (d>20 or not b>c)
then
a = a+b
else
do
b = b-20
while (x>y and y>x)

Solve:

```
0   if a>b goto 6
1     goto 2
2   if b<c goto 4
3     goto 20
4   if a>20 goto 6
5     goto 20
6   if d>20 goto 10
7     goto 8
8   if b>c goto 13
9     goto 10
10    t1 = a+b
11    a = t1
12    goto 0
13    t2 = b-20
14    b = t2
15    if (x>y) goto 17
16      goto 0
17    if (y>x) goto 13
18      goto 0
19      goto 0
```

Date
19-3-13
Tuesday

CD

- SWITCH CASE:

```
Switch (a+b)
{
   case 1: a = b+c;
   case 2: b = b+3;
   default: c=3;
}
```

Ans:

```
0   t1 = a+b
1   goto 10
2   t2 = b+3
3   a = t2
4   goto next
5   t3 = b+3
6   b = t3
7   goto next
8   c=3
9   goto next
10  if t1 == 1 goto 2
11  if t1 == 2 goto 5
12  goto 8
```

Ques:
```
switch (a+b)
{
  case 2: {x = y; break;}
  case 5: { switch (x)
            {
              case 0: {a = b+1; break;}
              case 1: {a = b+3; break;}
              default: {a = 2; break;}
            }
          break;
        }
  case 3: {x = y-1; break;}
  default: {a = 2; break;}
```

Ans:

```
0   t1 = a+b
1   goto 2
2   t2 = y
3   x = t2
4   goto next
5   t3 = b+1
6   a = t3
7   goto next
8   t4 = b+3
9   a = t4
10  goto next
11  t5 = 2   a = t5
12  goto next
13  if (x == 0) goto 5
```

```
14   if x==1 goto 8
15   goto 11
16   t5 = y-1
17   x = t5
18   goto next
19   a = 2
20   goto next
21   if t3 == 2 goto 2
22   if t1 == 5 goto 13
23   if t1 == 3 goto 18
24   goto 20
25
```

Ques:

```
void math()
{  int i;
   int a[20];
 for(i=1; i<=20; i++)
     { a[i] = i;
       i = i+1;
     }
}
```

Ans:

// address of a[i] will be,

baseaddress + w(i-1)

a + w(i-1)

a + w*i - w

a - w + w*i

$t_1 = a - w$

$t_2 = w * i$

// $\therefore t_1[t_2]$

a[i] = *(a+i)

// Assume that bpw = 4.

Ans:
0 i = 1
1 if i <= 20 goto 3
2 goto 9
3 t1 = addr(a) - 4
4 t2 = 4 * i
5 t1[t2] = i
6 t3 = i + 1
7 i = t3
8 goto 1

Ques:
```
for (i=1; i<=20; i++)
    sum = sum + a[i] + b[i];
```

Ans:
0 i = 1
1 if i <= 20 goto 3
2 goto 13.
3 t1 = addr(a) - 4
4 t2 = 4 * i
5 t3 = t1[t2]
6 t4 = addr(b) - 4
7 t5 = 4 * i
8 t6 = t4[t5]
9 t7 = t3 + t6
10 sum = t7 + sum
11 t8 = sum
12 t9 = i + 1
13 i = t9
14 goto 1

Date
20-3-13
Wednesday

SYNTAX DIRECTED DEFINITION (SDD)

→ A SDD is a generalisation of a context free grammer in which each grammer symbol has an associated set of attributes, partitioned into two subsets called the synthesised and inherited attributes of that grammer symbol.

→ With each production in CFG, it also associates a set of semantic rules to compute values of the attributes of the grammer symbol which appears in that production.

→ Hence, the grammer & set of semantic rules constitutes syntax directed definition.

→ In the SDD, each grammer production $A \to \alpha$ has associated with it a set of semantic rules of the form $b := f(c_1, c_2 \ldots c_k)$ where f is a function and either
(i) b is a synthesised attribute of A and $c_1, c_2 \ldots c_k$ are attributes belonging to the grammer symbols of the production, or
(ii) b is a inherited attribute of one of the grammer symbols on the RHS of the production and $c_1, c_2 \ldots c_k$ are attributes belonging to the grammer symbol of the production.

→ For ex: The SDD for a desk calculator program:

SDD of a simple desk calculator.

$L \to E\,n$	{ Print (E.val) }
$E \to E_1 + T$	{E.val := E_1.val + T.val }
$E_1 \to T$	{E.val := T.val}
$T \to T_1 * F$	{ T.val := T_1.val * F.val }
$T_1 \to F$	{T.val := F.val}
$F \to (E)$	{F.val := E.val}
$F \to$ digit	{F.val := digit.lexval. }

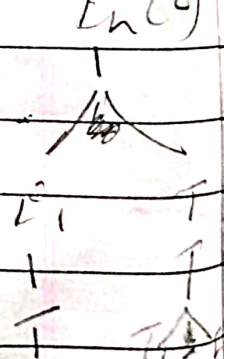
- **SYNTHESISED ATTRIBUTE**

→ An attribute is said to be synthesised, if its value at a parse tree node is determined by the attribute values at the child nodes.

→ A synthesised attribute has a desirable property; it can be evaluated during a single bottom-up traversal of the parse tree.

→ SDD that only use synthesised attributes are :

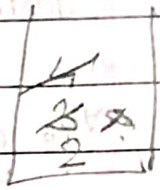
$L \to E\,n$	{ Print (E.val) }
$E \to E_1 + T$	{ E.val := E_1.val + T.val }
$E_1 \to T$	{ E.val := T.val }
$T \to T_1 * F$	{ T.val := T_1.val * F.val }
$T_1 \to F$	{ T.val := F.val }
$F \to (E)$	{ F.val := E.val }
$F \to$ digit	{ F.val := digit.lexval }



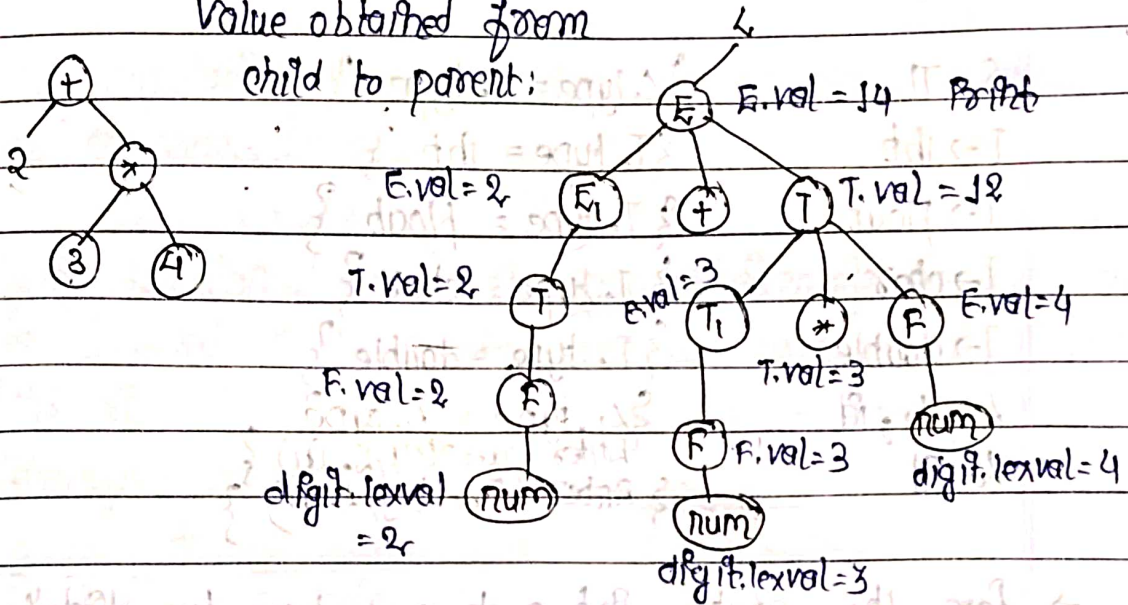
→ A parse tree along with the values of the attributes at the node for an expression 2+3*4.

⇒ Postfix notation = 234 * +

234*+



Value obtained from child to parent:



→ SDD that only used synthesised attributes are known as "S-attributed" definition.

V.Imp → If translation are specified using S-attribute definitions, then the semantic rules can be easily evaluated by the LR parser itself during the parsing, thereby making translation more efficient.

- **INHERITED ATTRIBUTE:**

→ The value of inherited attribute at a node in a parse tree is defined using the attribute values at the parents or siblings

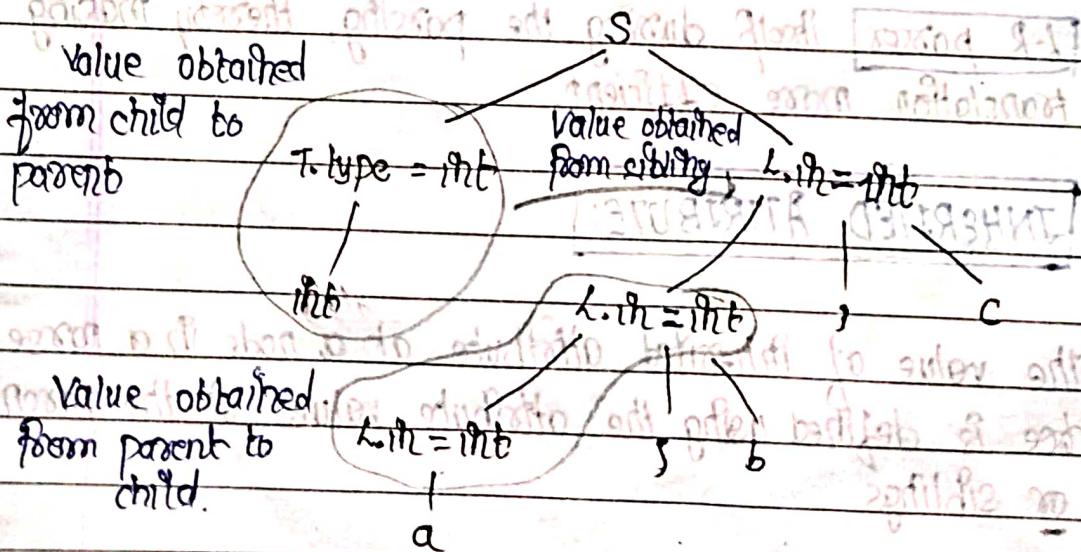
→ Consider an example, and let us compute the inherited attributes,

→ The grammer is and their SDD is:

S → TL	{L.type = T.type}
T → int	{T.type = int}
T → float	{T.type = float}
T → char	{T.type = char}
T → double	{T.type = double}
L → L1, id	{L1.in = L.in; Enter(id.entry, L.in)}
L → id	{Enter(id.entry, L.in)}

→ For the string int a, b, c we have to distribute the data type int to all the identifiers a, b and c; such that a becomes integer, b becomes integer and c becomes integer.

→ The parse tree will be,



→ The value of L nodes is first obtained from T.type (sibling). The T.type is basically lexical value obtained as int or float or char or double. Then the L nodes give the type of the identifiers a, b, c.

→ The computation of type is done in top-down manner or in pre-order traversal.

→ Using function 'Enter', the type of identifiers a, b and c is inserted in the symbol table at corresponding id.entry (the id. entry is the address of corresponding identifier in symbol table.)

- **DEPENDENCY GRAPH:**

→ If an attribute 'b' at a node in a parse tree depends on an attribute c, then the semantic rule for b at node that node must be evaluated after the semantic rule that defines c.

→ The interdependency among the inherited and synthesized attributes at the node in a parse tree can be depicted by a directed graph called as dependency graph..

→ Before constructing a dependency graph, for a parse tree, first we have to put each semantic rule into the form $b = f(c_1, c_2 \ldots c_k)$ by introducing a dummy synthesized attribute b for each semantic rule that

consists of a procedure call.

→ The graph has a node for each attribute and an edge to the node for b from the node for c attribute. b depends on attribute c

→ For ex: (1) Suppose $A.a := f(X.x, Y.y)$ is a semantic rule for the production $A \to XY$.
This rule defines a synthesized attribute A, a that depends on the attributes X, x & Y, y.

(2) → If the production $A \to XY$ has the semantic rule $X.i = g(A.a, Y.y)$ associated with it, then there will on edge to $X.i$ from $A.a$ and also an edge to $X.i$ from $Y.y$, since $X.i$ depends on both $A.a$ and $Y.y$

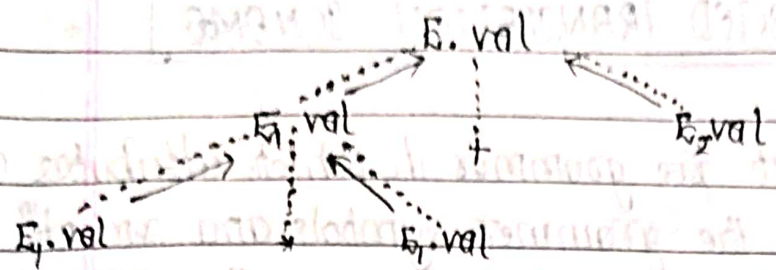
→ ~~For ex:~~ Dependency graph for the following grammer

$E \to E_1 + E_2$

$E \to E_1 * E_2$

~~The semantic~~

Production Rule	Semantic Rule.
$E \to E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E \to E_1 * E_2$	$E.val = E_1.val * E_2.val$

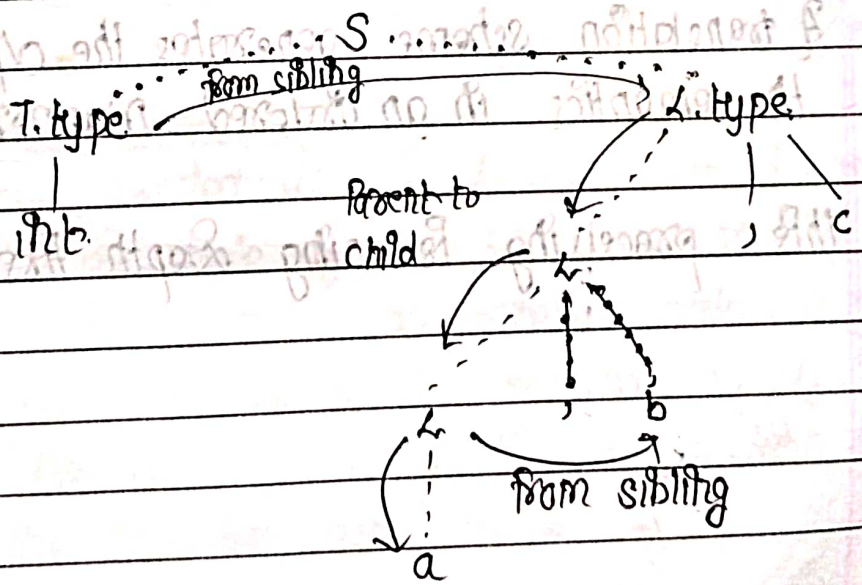


Dependency Graph

(→) Suppose grammer and semantic action is:

$S \to TL$	{ L.type = T.type }
$T \to$ int	{ T.type = int }
$T \to$ float	{ T.type = float }
$T \to$ char	{ T.type = char }
$T \to$ double	{ T.type = double }
$L \to L_1$, id	{ L_1.in = L.in }
	Enter (id.entry, L.in) }
$L \to$ id	{ Enter (id.entry, L.in) }

∴ Dependency graph will be.



• SYNTAX DIRECTED TRANSLATION SCHEME

→ It is a context free grammer in which attributes are associated with the grammer symbols and semantic actions, enclosed within { }, are inserted in the right side of the productions.

→ During the process of parsing, the evaluation of attributes takes places by consulting the semantic action enclosed in { } at the right of the grammer symbol.

→ This process of execution of code fragment, semantic actions from the syntax directed definition is called syntax directed translation.
Thus, the execution of syntax directed definition can be done by syntax directed translation scheme.

→ A translation scheme generates the o/p by executing the semantics in an ordered manner.

→ This processing is using depth first traversal.

- **S-ATTRIBUTED DEFINITION & L-ATTRIBUTED DEFINITION :**

→ A syntax directed definition that uses synthesised attributes exclusively is said to be an S-attributed definition.

→ A parse tree showing the values of attributes in each node is called an annotated or decorated parse tree. Synthesised attributes can be evaluated by bottom-up parser as the i/p is being parsed.

→ The parser can keep a value of synthesised attributes associated with the grammar symbol on its stack. Whenever, a reduction is made, the values of the new synthesised attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the Reducing producing.

- **L-attributed definition:**

→ A syntax directed defn is L-attributed if each inherited attribute of X_j, $1 \le j \le n$ on the right side of $A \to X_1, X_2 \ldots X_n$ depends only on :

(i) the attributes of the symbol $X_1, X_2, \ldots X_{j-1}$ to the left of X_j in the production and

(ii) the inherited attributes of A,

$S \to TL$	$\{L.type = T.type\}$
$T \to int$	$\{T.type = int\}$
$L \to L_1, id$	$\{L_1.in = L.in\}$ Enter (id.entry, L.in) $\}$
$L \to id$	$\{$ Enter (id.entry, L.in) $\}$

→ The SDD given above is an example of the L-attributed def^n, bcz the inherited attribute L.type depends on T.type, and T is to the left of L in the production $S \to TL$.

→ Similarly, the inherited attribute L_1.type depends on the inherited attribute L.type and L is parent of L_1 in the production $L \to L_1, id$.

- Note: Note that every S-attributed def^n is L-attributed.

- CD

Date 29-3-13 Friday

- **POSTFIX NOTATION**

The Ordinary (infix) way of writing the sum of A & B is with the operator in the middle A+B.

The postfix notation for the same expression places the operator at the right end as, AB+.

In general, if even e_1 and e_2 are any postfix expression and $\oplus$ is any binary operator, the result

of applying θ to the values denoted by e_1 and e_2 is indicated in postfix notation by $e_1 e_2 \theta$

No parenthesis are needed in postfix notation because the position and no. of arguments of the operators permits only 1 way to decode a postfix expression.

For Ex:

(1) (a+b) *c
⇒ ab+c*

(2) a *(b+c)
⇒ abc+*

(3) (a+b) * (c+d)
⇒ ab+cd+*

- **IMPORTANT TERMS**

Goto	Jump
jlt	jump less than
jleqz	jump less than equal to zero
jeqz	jump equals zero.

Ques 1 if e then
S_1

Solve: Postfix notation:
e l_1 jeqz S_1 l_1: ..

Ques 2 if e then
S_1
else
S_2

Solve: Postfix notation:
e l_1 jeqz S_1 l_2 jump l_1: S_2 l_2:

Ques 3 if e_1 then
if e_2 then
S_1
else
S_2
else
S_3

Solve: Postfix notation:
e_1 l_1 jeqz e_2 l_2 jeqz S_1 l_2: S_2 l_1: S_3 l_3:

Ques: 4 if e_1 then
S_1
else
if e_2 then
S_2
else
S_3

Solve: Postfix notation:

$e_1\ l_1 jeqz\ S_1\ l_3 jump\ l_1: e_2\ l_2 jeqz\ S_2\ l_3 jump\ l_2: S_3\ l_3:$

Ques: 5 while e do
$a = b + c$

Solve: Postfix notation:

$l_1: e\ l_2 jeqz\ bc+\ l_1 jump\ l_2:$

Ques: 6 if a then if c-d then a+c else a*c else a+b

Solve: $a\ l_2 jeqz\ cd-\ l_1 jeqz\ ac+\ l_3 jump\ l_1: ac*\ l_3 jump$

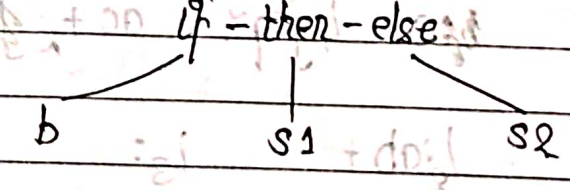
$l_2: ab+\ l_3:$

- **SYNTAX TREE**

- **SYNTAX TREE**

→ An abstract syntax tree is a condensed form of parse tree useful for representing language construct.

→ The production $S_1 \rightarrow$ if b then S_1 else S_2 appear in a syntax tree as:



→ In a syntax tree, operators and keywords do not appear as leaves but rather are associated with the interior node that would be the parent of those leaves.

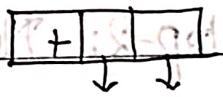
in the parse tree.

- **CONSTRUCTION OF SYNTAX TREE FOR EXPRESSION**

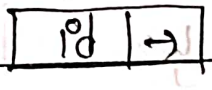
→ Constructing syntax tree for an expression means translation of expression into postfix form. The nodes for each operand & operator is created.

→ Each node can be implemented as a record with multiple fields.

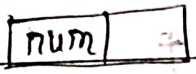
Following are the functions used in syntax tree for expression:

1- mknode (op, left, right): 

→ This function creates a node with the field operator having operator as label and the two pointers to left and right.

2. mkleaf (id, entry): 

→ This function creates an identifier node with label id and a pointer to symbol table is given by entry.

3. mkleaf (num, val): 

→ This function creates node for number with label num and val is for value of that number.

Ques: Construct the syntax tree for the expression.

$x + y - 5 + z$

and the grammar production is $E \to E_1 + T$

$E \to E_1 - T$

$E \to E_1 * T$

$E \to T$

$T \to id$

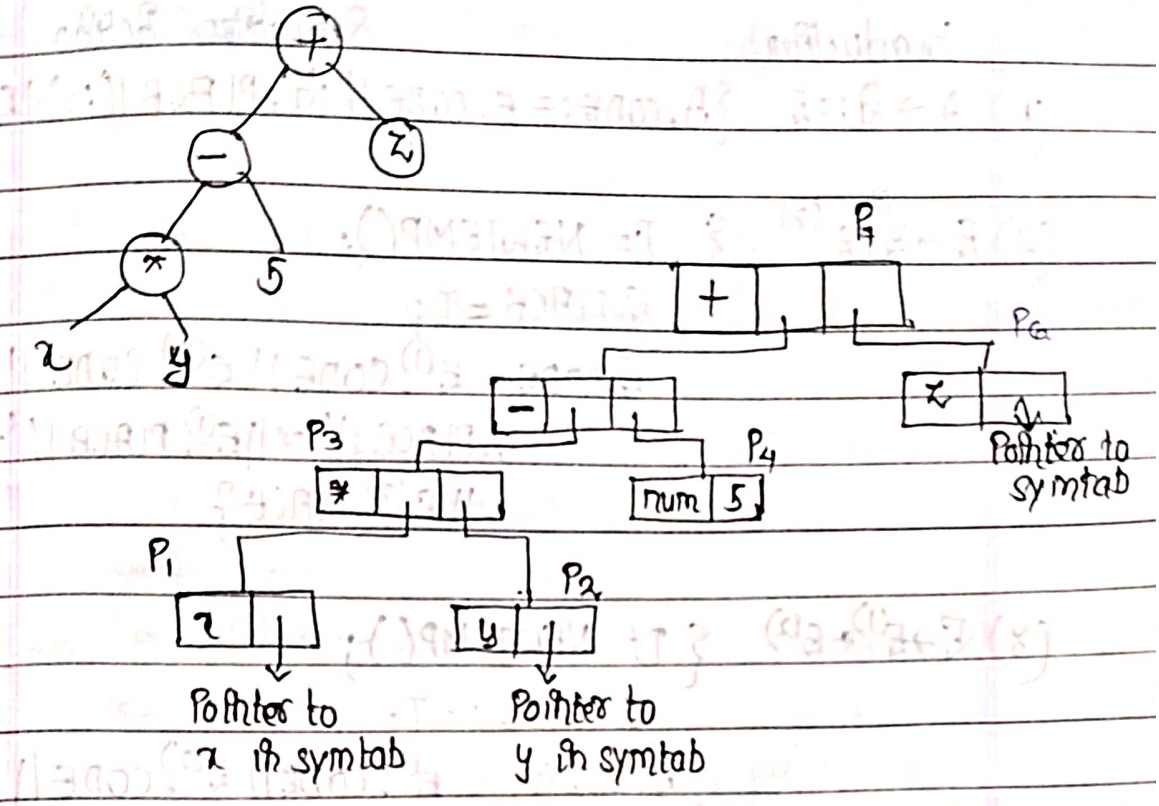
$T \to num$

Solve: Step-1: Convert the expression from infix to postfix:

$\therefore \quad xy * 5 - z +$

Step-2: Make use of functions mknode(), mkleaf(id, p) and mkleaf(num, val)

Symbol	Operation
x	P_1 = mkleaf(id, Pointer to x in sym tab)
y	P_2 = mkleaf(id, Pointer to y in sym tab)
*	P_3 = mknode(*, P_1, P_2)
5	P_4 = mkleaf(num, 5)
-	P_5 = mknode(-, P_3, P_4)
z	P_6 = mkleaf(id, Pointer to z in symtab)
+	P_7 = mknode(+, P_5, P_6)



Date 1-4-13 Monday

- CD
- ~~SDD BOR SYNTAX TREE~~

- SYNTAX DIRECTED TRANSLATION SCHEME (SDTS) FOR ASSIGNMENT STATEMENTS WITH INTEGER TYPE

	~~Production~~	~~Semantic Action~~
1)	$A \rightarrow id := E$	~~A.id := E~~
2)	$E \rightarrow E + E$	
3)	$E \rightarrow E * E$	
4)	$E \rightarrow -E$	
5)	$E \rightarrow (E)$	
6)	$E \rightarrow id$	

code CODE → Sequence
PLACE → Actual storage.

Production	Semantic Action
(1) $A \rightarrow id := E$	{A.CODE := E.CODE || id.PLACE || ':=' || E.PLACE
(2) $E \rightarrow E^{(1)} + E^{(2)}$	{ T := NEWTEMP(); E.PLACE = T; E.CODE = $E^{(1)}$.CODE || $E^{(2)}$.CODE || E.PLACE || ':=' || $E^{(1)}$.PLACE || '+' || $E^{(2)}$.PLACE }
(3) $E \rightarrow E^{(1)} * E^{(2)}$	{ T := NEWTEMP(); E.PLACE := T; E.CODE := $E^{(1)}$.CODE || $E^{(2)}$.CODE || E.PLACE || ':=' || $E^{(1)}$.PLACE || '*' || $E^{(2)}$.PLACE
(4) $E \rightarrow -E^{(1)}$	{ T := NEWTEMP(); E.PLACE := T; E.CODE := $E^{(1)}$.CODE || E.PLACE || ':=' || '-' || $E^{(1)}$.PLACE }
(5) $E \rightarrow (E^{(1)})$	{ E.PLACE := $E^{(1)}$.PLACE; E.CODE := $E^{(1)}$.CODE }
(6) $E \rightarrow id$	{ E.PLACE := id.PLACE; E.CODE := null }

- **ASSIGNMENT STATEMENTS WITH MIXED TYPE**

(for same previous grammer)

```
T := NEWTEMP()
if E(1).MODE := INTEGER and E(2).MODE = INTEGER then
  begin
    GEN(T := E(1).PLACE int op E(2).PLACE);
    E.MODE := INTEGER
  end
else if E(1).MODE := REAL and E(2).MODE := REAL then
  begin
    GEN(T := E(1).PLACE real op E(2).PLACE);
    E.MODE := REAL
  end
else if E(1).MODE := INTEGER and E(2).MODE := REAL then
  begin
    U := NEWTEMP();
    GEN(U := int to real E(1).PLACE);
    GEN(T := U real op E(2).PLACE);
    E.MODE := REAL
  end
else /* if E(1).MODE = REAL and E(2).MODE = INTEGER */
  begin
    U := NEWTEMP();
    GEN(U := int to real E(2).PLACE);
    GEN(T := E(1).PLACE real OP U);
    E.MODE := REAL
  end
E.PLACE := T;
```

$ME^{(2)}$ → M stores the address location of $E^{(2)}$

*VV. imp

SDTS FOR BOOLEAN EXPRESSION.

CF Grammer:

$E \rightarrow E$ OR E

$E \rightarrow E$ and E

$E \rightarrow$ not E

$E \rightarrow (E)$

$E \rightarrow$ id

$E \rightarrow$ id relop id

The revised grammer is:

$E \rightarrow E^{(1)}$ or $ME^{(2)}$

$\rightarrow E^{(1)}$ and $ME^{(2)}$

$\rightarrow$ not $E^{(1)}$

$\rightarrow (E^{(1)})$

$\rightarrow$ id

$\rightarrow$ id$^{(1)}$ relop id$^{(2)}$

$M \rightarrow \epsilon$.

(i) $E \rightarrow E^{(1)}$ or $ME^{(2)}$

{BACKPATCH ($E^{(1)}$.FALSE, M.QUAD);

E.TRUE := MERGE ($E^{(1)}$.TRUE, $E^{(2)}$.TRUE);

E.FALSE := $E^{(2)}$.FALSE}

(ii) $E \rightarrow E^{(1)}$ and $ME^{(2)}$

{ BACKPATCH($E^{(1)}$.TRUE, M.QUAD);
E.TRUE := $E^{(2)}$.TRUE
E.FALSE := MERGE($E^{(1)}$.FALSE, $E^{(2)}$.FALSE)
}

(iii) $E \rightarrow$ not $E^{(1)}$

{
E.TRUE := $E^{(1)}$.FALSE
E.FALSE := $E^{(1)}$.TRUE
}

(iv) $E \rightarrow (E^{(1)})$

{
E.TRUE := $E^{(1)}$.TRUE
E.FALSE := $E^{(1)}$.FALSE.
}

(v) $E \rightarrow$ id

{
E.TRUE := MAKELIST(NEXTQUAD);
E.FALSE := MAKELIST(NEXTQUAD+1);
GEN(if id.PLACE goto ___);
GEN(goto ___); }

(vi) $E \rightarrow id^{(1)}$ relop $id^{(2)}$

{ E.TRUE := MAKELIST (NEXTQUAD)
E.FALSE := MAKELIST (NEXTQUAD + 1)
GEN(if $id^{(1)}$.PLACE relop $id^{(2)}$.PLACE goto _
GEN(goto _)
}

(vii) $M \rightarrow \epsilon$

{ M.QUAD := NEXTQUAD }

Ex: P<Q or R<S and T<U

100 if P<Q goto _
101 goto _
102 if R<S goto _
103 goto _
104 if T<U goto _
105 goto _

2-4-13
Tuesday

• SDTS FOR CONTROL FLOW.

Grammer is:

1) $S \rightarrow$ if E then S
2. $\rightarrow$ if E then S else S
3. $\rightarrow$ while E do S
4. $\rightarrow$ begin L end
5. $\rightarrow A$
6. $L \rightarrow L ; S$
7. $L \rightarrow S$

Where

S

- S :→ Statement.
- L :→ Statement List.

A :→ Assignment Statement

E :→ Boolean valued expression.

1) $S \rightarrow$ if E then $MS^{(1)}$

{ BACKPATCH (E.TRUE, M.QUAD);
S.NEXT := MERGE (E.FALSE, $S^{(1)}$.NEXT) }

2) $S \rightarrow$ if E then $M^{(1)} S^{(1)} N$ else $M^{(2)} S^{(2)}$

{ BACKPATCH (E.TRUE, $M^{(1)}$.QUAD);
BACKPATCH (E.FALSE, $M^{(2)}$.QUAD);
S.NEXT := MERGE ($S^{(1)}$.NEXT, N.NEXT, $S^{(2)}$.NEXT) }

3) S → while $M^{(1)} E$ do $M^{(2)} S^{(1)}$

{
BACKPATCH ($S^{(1)}$.NEXT, $M^{(1)}$.QUAD);
BACKPATCH (E.TRUE, $M^{(2)}$.QUAD);
S.NEXT := E.FALSE
GEN(goto $M^{(1)}$.QUAD) }

4) S → begin L end
{ S.NEXT = L.NEXT }

5) S → A
{ S.NEXT := MAKELIST() }

6) $L \rightarrow L^{(1)}$; MS
{BACKPATCH ($L^{(1)}$.NEXT, M.QUAD);
L.NEXT := S.NEXT }

7) L → S
{ L.NEXT := S.NEXT }

- FOR LOOP :

$S \rightarrow FOR (E^{(1)} ; M^{(1)} E^{(2)} ; M^{(2)} E^{(3)}) M^{(3)} S^{(1)}$
{ BACKPATCH ($E^{(2)}$.TRUE, $M^{(3)}$.QUAD);
BACKPATCH ($E^{(3)}$.NEXT $M^{(1)}$.QUAD);
BACKPATCH ($S^{(1)}$.NEXT, $M^{(2)}$.QUAD);
GEN (goto $M^{(2)}$.QUAD);
S.NEXT = $E^{(2)}$.FALSE }

- **DO- WHILE LOOP:**

$S \rightarrow$ do $M^{(1)} S^{(1)}$ while $M^{(2)} E$
{ BACKPATCH ($E.TRUE$, $M^{(1)}.QUAD$);
BACKPATCH ($S^{(1)}.NEXT$, $M^{(2)}.QUAD$);
$S.NEXT := E.FALSE$ }

Date 8-3-13 Wednesday

- **3- ADDRESS CODE FOR PROCEDURES.**

V. imp

① int g-a;
void increment()
{
g-a = g-a+1;
}

int main()
{ int V_1, V_2, V_3;
$V_1 = 10$;
$V_2 = 20$;
increment();
}

Three address code
Enter 6
$t_1 = g - a + 1$
$g - a = t_1$
Exit 6
Enter 10
$V_1 = 10$
$V_2 = 20$
call increment
exit 10.

Three address code

② int add(int a, int b)
{ int c;
c = a+b;
return(c);
}
int main()
{ int V_1, V_2, V_3;
$V_1 = 10$;
$V_2 = 3$;
V_3 = add(V_1, V_2);
}

Enter 20
$t_1 = a+b$
$c = t_1$
return(c)
exit 20
enter 40
$V_1 = 10$
$V_2 = 3$
Param V_1
Param V_2
call add
retrieve t_2
$V_3 = t_2$
exit 40

- **SDD SCHEME FOR PROCEDURAL CALL**

Grammer is:

1. S → call id (elist)
2. elist → elist, E
3. elist → E

1. S → call id (elist)
{ for each item P on Queue do
GEN (Param P)
GEN (call id.Place)

elist → arguments

2. elist → elist, E
{
append E.PLACE to the end of QUEUE
}

3. elist → E
{ Initialize QUEUE to obtain only E.PLACE
}

Ques: Write SDD for the grammer:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

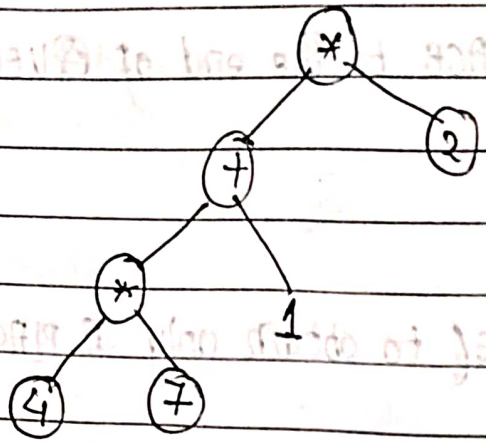
and construct decorated parse tree for the expression

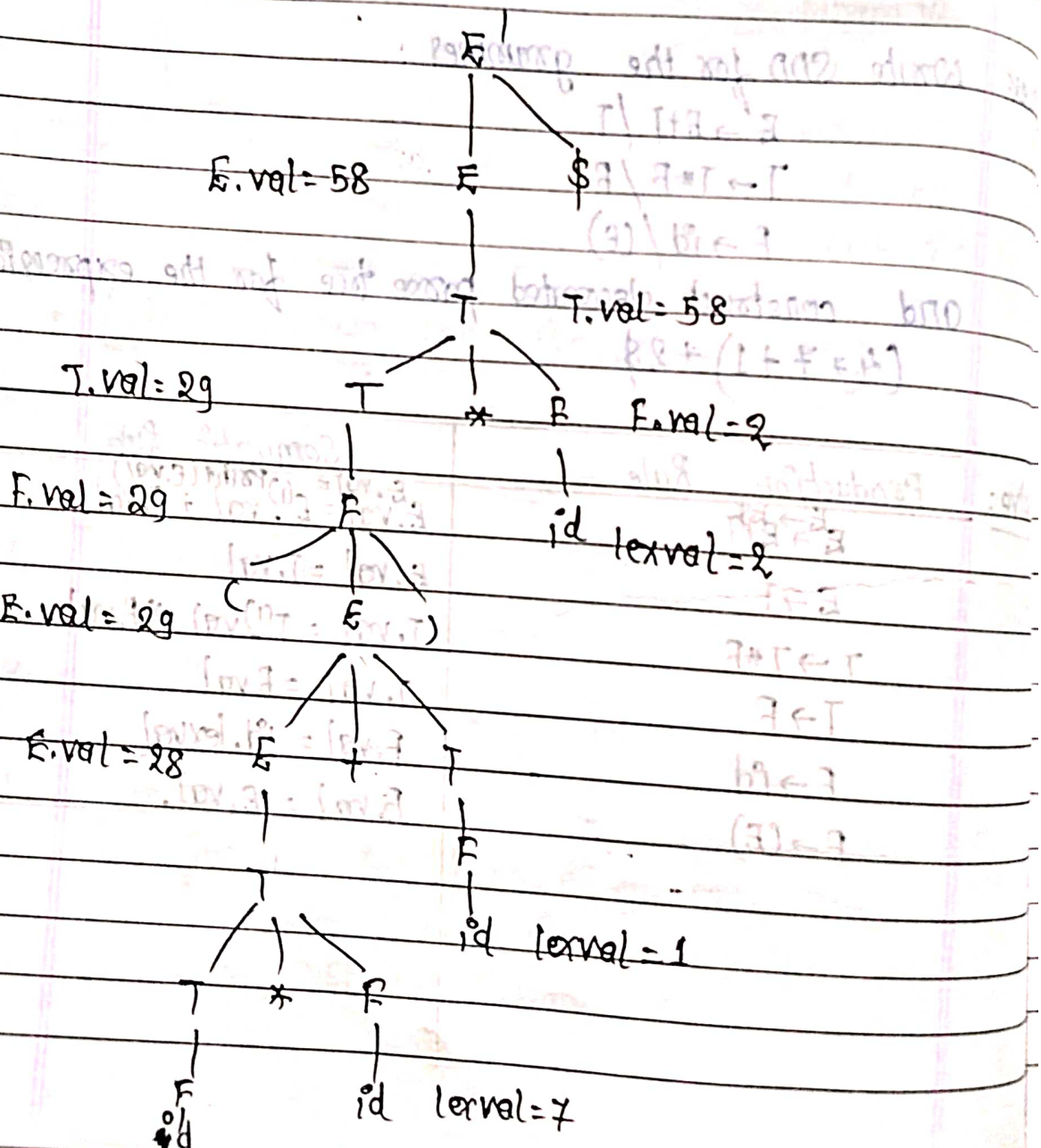
$(4*7+1)*2\$$

Solve:

Production Rule	Semantic Rule
$E \rightarrow E+T$	$E.val = E^{(1)}.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T*F$	$T.val = T^{(1)}.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow id$	$F.val = id.lexval$
$F \rightarrow (E)$	$F.val = E.val$

- Postfix notation will be, 47*1+2*





Ques: Using the given grammer write the SDD to evaluate an expression. construct the annotated parse tree for the sentence 2+3*7 $

$E \rightarrow E+T / T$

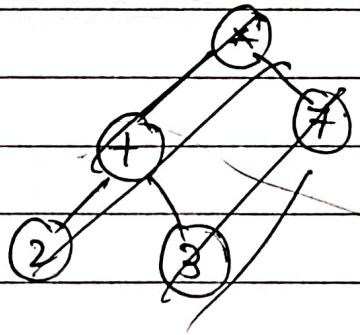
$T \rightarrow T * F / F$

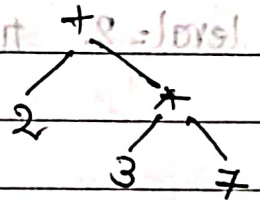
$F \rightarrow num / (E)$

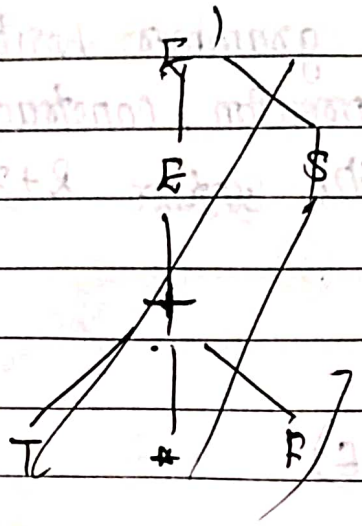
Solve:

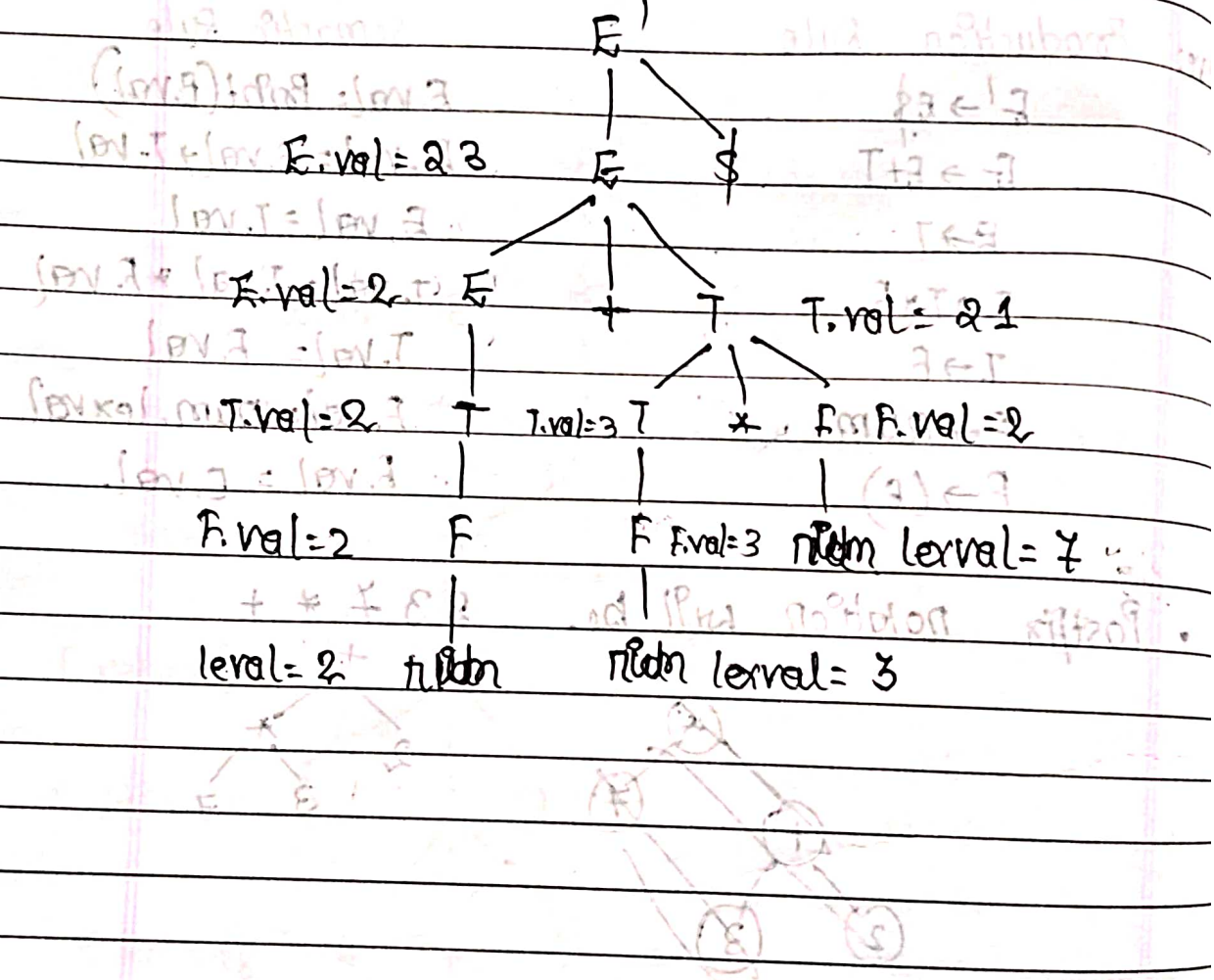
Production Rule	Semantic Rule
$E' \rightarrow E\$$	E.val = Print(E.val)
$E \rightarrow E+T$	E.val = E.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T*F$	T.val = T.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow num$	F.val = num.lexval
$F \rightarrow (E)$	F.val = E.val

- Postfix notation will be. 2 3 7 * +






E'
E
$
+
T
*
F


E'
E.val=23
E
$
E.val=2
E
+
T
T.val=21
T.val=2
T
T.val=3
T
*
F.val=2
F.val=2
F
F
F.val=3
lexval=2
lexval=3
lexval=7

Date 5-4-13 Friday

CD

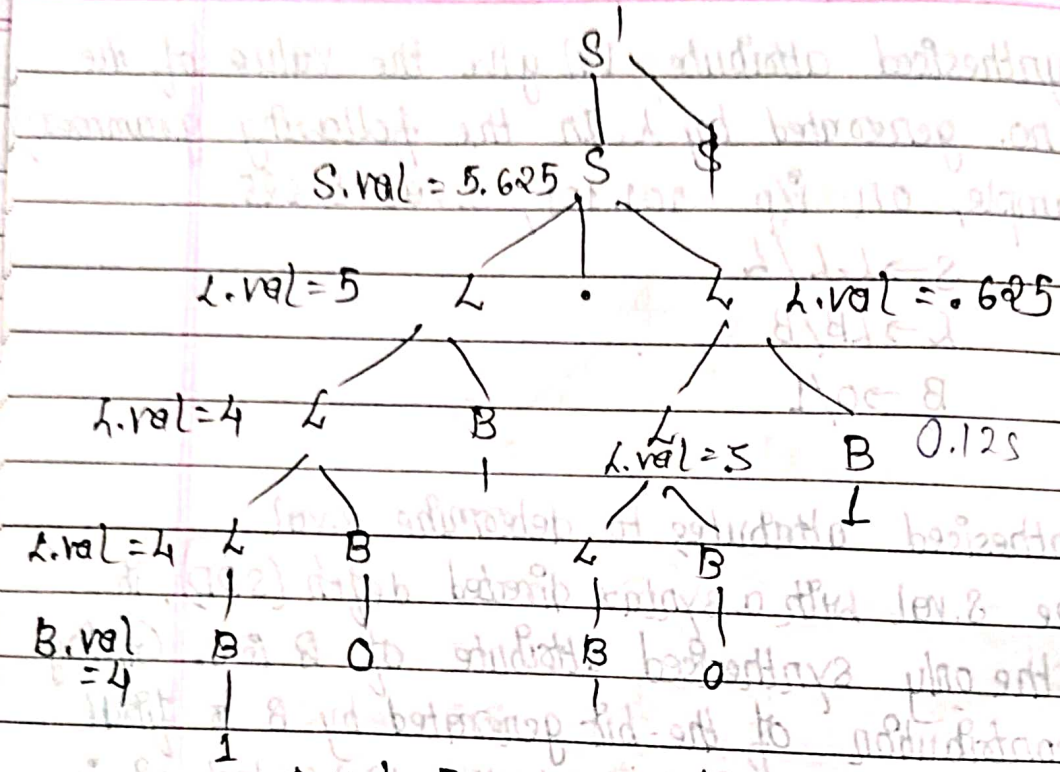
Ques: Let synthesised attribute val give the value of the binary no. generated by L. In the following grammer, for example, on i/p 101.101, S.val = 5.625

S → L.L / L
L → LB / B
B → 0 / 1

(i) Use synthesised attributes to determine S.val.
(ii) Determine S.val with a syntax directed defn (SDD) in which the only synthesised attribute of B is c. Giving the contribution of the bit generated by B to final value. For ex: the contribution of the first & last bit is in 101.101 to the value 5.625 is 4 and .125 respectively.

Solve:

Production	Semantic Rule
$S' \to S\$$	Print(S.val)
$S \to L.L$	$S.val = L.val \cdot L^{(1)}.val$
$S \to L$	S.val = L.val
$L \to LB$	$L.val = L^{(1)}.val + B.val$
$L \to B$	L.val = B.val
$B \to 0$	B.val = B.c
$B \to 1$	B.val = B.c



B.val = B.C = 1*4 = 4

L.val = B.val = 4

L.val = L.val + B.val = 4+0 = 4

L.val = L.val + B.val = 4+1 = 5

B.val = B.C = .5*1 = .5

Ques: $S \rightarrow E\$$ — 23 * 5 + 4 $

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow I$

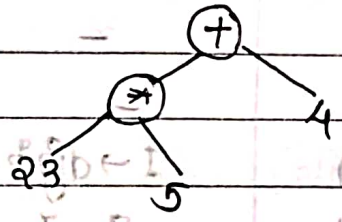
$I \rightarrow I$ digit

$I \rightarrow$ digit

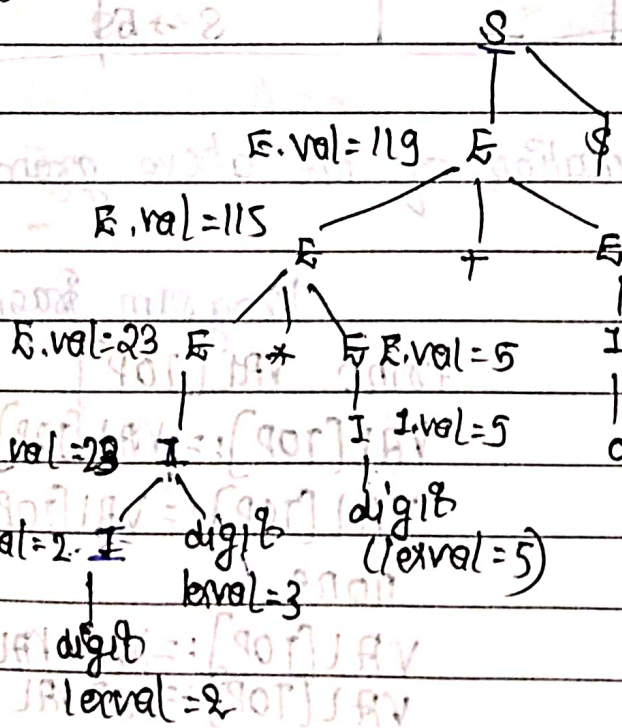
Production	Semantic Rule
$S \to E\$$	Print $E.val$
$E \to E+E$	$E.val = E^{(1)}.val + E^{(2)}.val$
$E \to E*E$	$E.val = E^{(1)}.val * E^{(2)}.val$
$E \to (E)$	$E.val = E^{(1)}.val$
$E \to I$	$E.val = I.val$
$I \to I\ digit$	$I.val = 10 * I^{(1)}.val + lexval$
$I \to digit$	$I.val = lexval$

- Postfix notation will be,

$235 * 4 +$



- Syntax tree will be,



- Bottom-up average of above grammer

S.NO	Input	State	val	Production used
1.	23*5+4$	—	—	—
2.	3*5+4$	2	-	—
3.	3*5+4$	I	2	I→digit
4.	*5+4$	I3	2_	—
5.	*5+4$	I	23	I→Idigit
6.	*5+4$	E	23	E→I
7.	5+4$	E*	23_	—
8.	+4$	E*5	23__	—
9.	+4$	E*I	23__	I→digit
10.	+4$	E*I	23_5	E→I
11.	+4$	E	115	E→E*E
12.	4$	E+	115_	—
13.	$	E+4	115__	—
14.	$	E+I	115_4	I→digit
15.	$	E+E	115_4	E→I
16.	$	E	119	E→E+E
17.	—	E$	119	—
18.	—	S	—	S→E$

- Stack Implementation of the above grammer

Production	Program Fragment
S→E$	Print VAL[TOP]
E→E+E	VAL[TOP]:=VAL[TOP]+VAL[TOP
E→E*E	VAL[TOP]:=VAL[TOP-1]
E→I	none
I→Idigit	VAL[TOP]:=10*VAL[TOP]+LEX
I→digit	VAL[TOP]:=LEX VAL

• UNIT-5

CODE OPTIMIZATION & CODE GENERATION

• BASIC BLOCKS

Prog 1:
```
begin
  PROD:=0
  I:=1
  do
    begin
      PROD:= PROD + A[I] * B[I]
      I:= I+1
      end
    while (I<=20)
end.
```

Assume bpw (bits per w) = 4

1. PROD = 0
2. I := 1
3. $T_1 = 4 * I$
4. $T_2 = addr(A) - 4$
5. $T_3 = T_2[T_1]$
6. $T_4 = addr(B) - 4$
7. $T_5 = T_4[T_1]$
8. $T_6 = T_3 * T_5$
9. PROD := PROD + T_6
10. I := I + 1
11. if I ≤ 20 goto 3

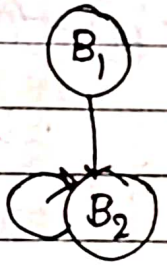
B_1

```
PROD = 0
I = 1
```

B_2

```
T1 = 4 * I
T2 = addr(A) - 4
T3 = T2 [T1]
T4 = addr(B) - 4
T5 = T4 [T1]
T6 = T3 * T5
PROD = PROD + T6
I = I + 1
if I ≤ 20 goto 3
```

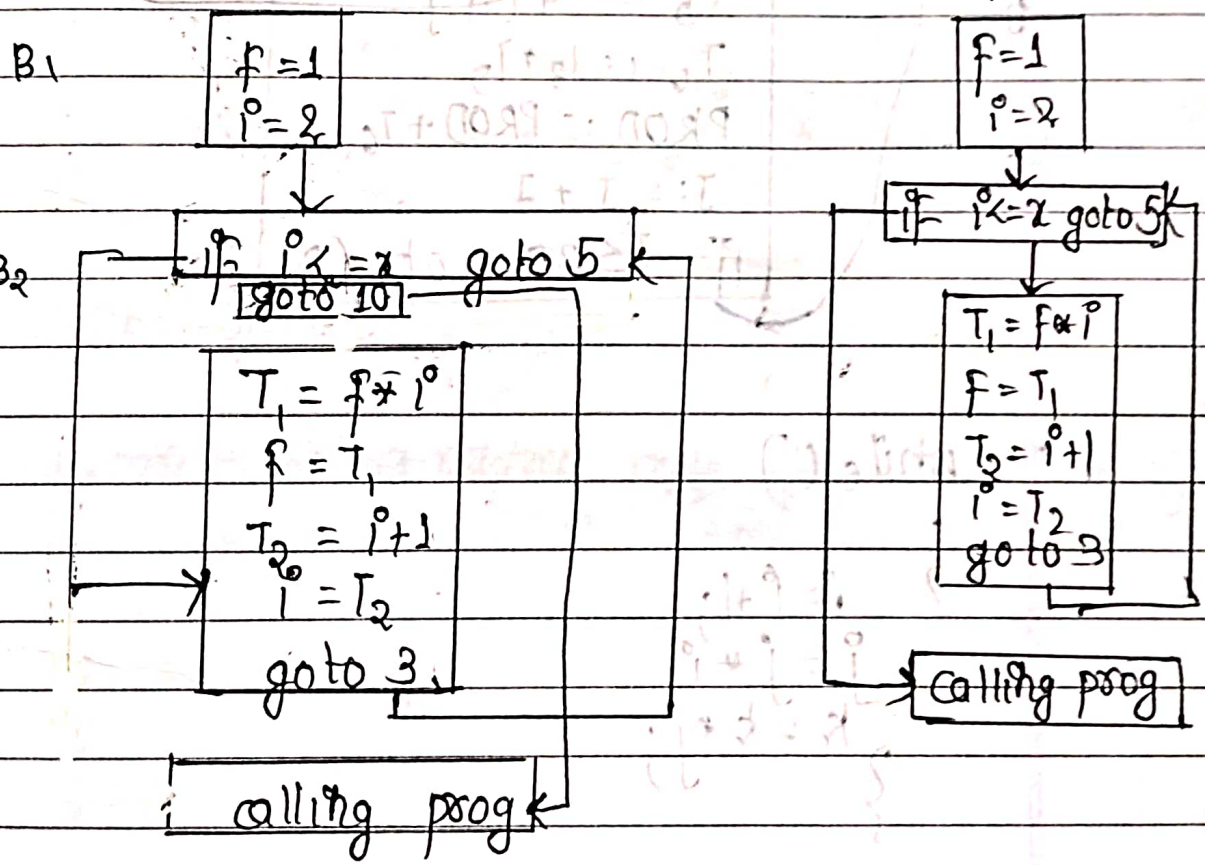
- Flow graph:



Prog2:

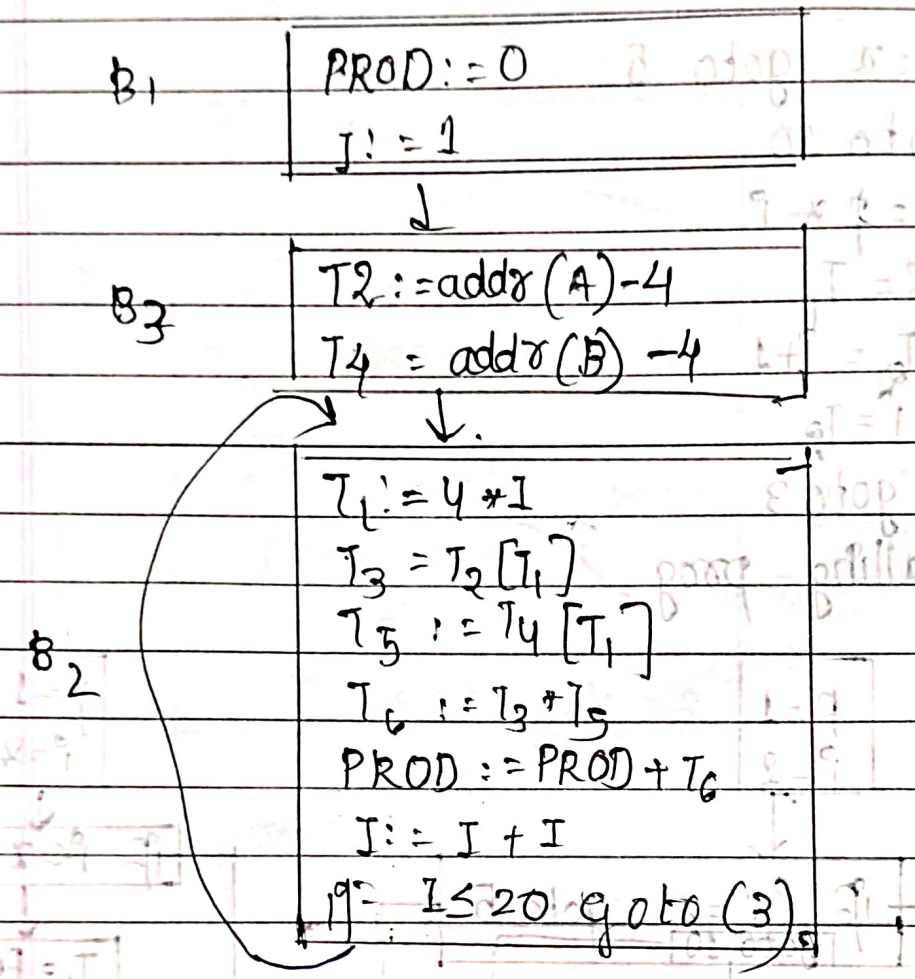
```
fact(x)
{ int f=1;
  for(i=2; i<=x; i++)
      f = f * i;
  return f;
}
```

1. f = 1
2. i = 2
3. if i <= x goto 5
4. goto 10
5. $T_1 = f * i$
6. $f = T_1$
7. $T_2 = i + 1$
8. $i = T_2$
9. goto 3
10. calling prog

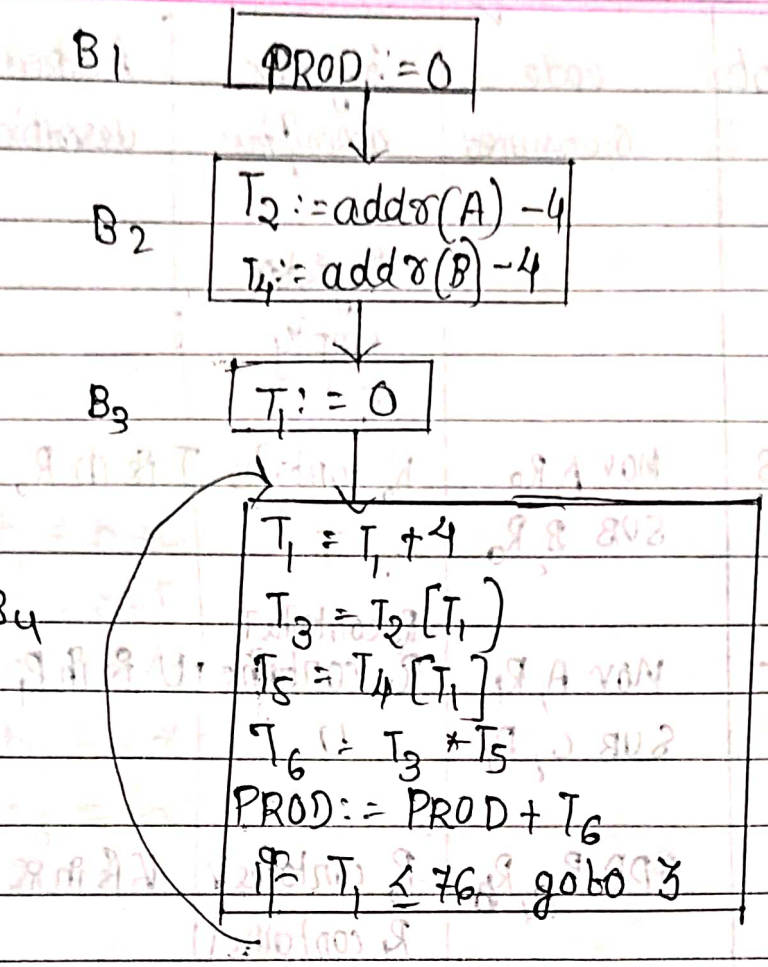


1) Code motion - no change
2) Induction variables
3) Reduction in strength

- Continued from program 1...



- while ()
{
i = i + 1;
j = j * i;
k = k + j;
}



- CODE GENERATION

Q:- Generate the 3 address code & target code of

$$\omega = (A-B) + (A-C) + (A*C)$$

Solve:

$T = A - B$

$U = A - C$

$V = T + U$

$\omega = V + U$

This is not Sunday :)

Statements	Code Generated	Register descriptor	Address descriptor	Cost
→		Register empty		-
→ T = A - B	MOV A, R_0 SUB B, R_0	R_0 contains T	T is in R_0	2 2
→ U = A - C	MOV A, R_1 SUB C, R_1	R_0 contains T R_1 contains U	U is in R_1	2 2
→ V = T + U	ADD R_1, R_0	R_0 contains V R_1 contains U	V is in R_0	1
→ W = V + U	ADD R_0, R_1	R_0 contains V R_1 contains W	W is in R_1	1
	MOV R_1, W	R_1 contains W	W is in R_1 and memory	2
		COST		12

MOV R_0, R_1 = 1

MOV R_0, A = 2

MOV A, R_0 = 2

MOV A, B = 3

MOV #1, A = 3

MOV #1, R_1 = 2

A = B[i]	MOV i, R_0	2
	MOV B[R_0], A	3
	⋮ ⋮ ⋮	5
	MOV B[R_0], R_0	2

Ques: $x = a/(b+c) - d*(e+f)$

Solⁿ:

$t_1 = b+c$

$t_2 = e+f$

$t_3 = a/t_1$

$t_4 = d*t_2$

$t_5 = t_3 - t_4$

$x = t_5$

Statements	Code Generated	Register descriptor Register empty	Address descriptor	Cost
$t_1 = b+c$	MOV b, R_0	R_0 contains t_1	t_1 is in R_0	2
	ADD c, R_0			2
$t_2 = e+f$	MOV e, R_1	R_0 contains t_1	t_2 is in R_1	2
	ADD f, R_1	R_1 contains t_2		2
	MOV a, R_2			2
$t_3 = a/t_1$	DIV R_1, R_2	R_2 contains t_3	t_3 is in R_2	2
$t_4 = d*t_2$	MUL d, R_1	R_1 contains t_4	t_4 is in R_1	2
$t_5 = t_3 - t_4$	SUB R_2, R_1	R_1 contains t_5	t_5 is in R_1	2
$x = t_5$	MOV t_5, x	R_1 contains	t_5 is in R_1 and memory	2
				15

Ques: Generate code for the following C statements for the target machine assuming all variables are static. Assume registers are available. Calculate cost of the instruction in each case

(i) $x = a[i] + 1$

(ii) $x = a + b + c$.

Solve: $x = a[i] + 1$

$t_1 = a[i]$

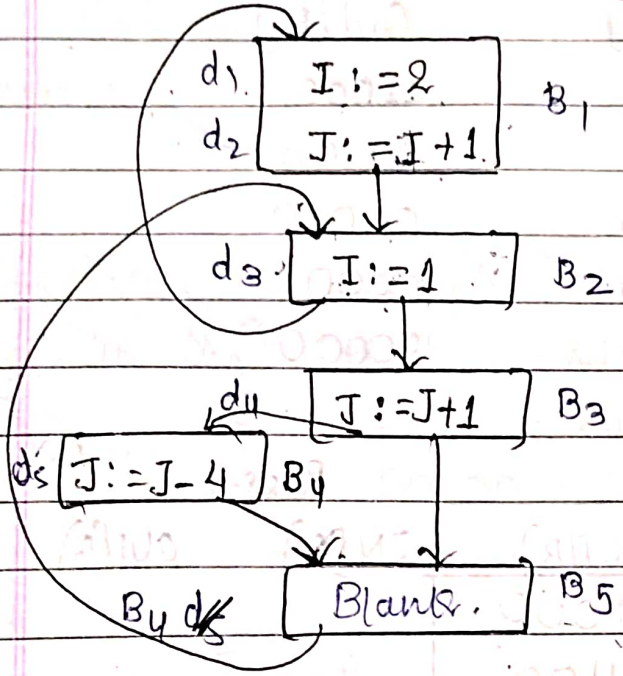
$t_2 = t_1 + 1$

$x = t_2$

Date: 6-4-13 Saturday

- **GLOBAL OPTIMIZATION**

- **DATA FLOW EQUATION**



$IN[B] = \cup\ OUT[P]$

P a predecessor of B

$OUT[B] = IN[B] - KILL[B] \cup GEN[B]$

Calculation of GEN and KILL

BLOCK B	GEN [B]	bit vector $\{d_1, d_2, d_3, d_4, d_5\}$	KILL[B]	bit VECTOR $\{d_1, d_2, d_3, d_4, d_5\}$
B_1	$\{d_1\ d_2\}$	11000	$\{d_3, d_4, d_5\}$	00111
B_2	$\{d_3\}$	00100	$\{d_1\}$	10000
B_3	$\{d_4\}$	00010	$\{d_2, d_5\}$	01001
B_4	$\{d_5\}$	00001	$\{d_2, d_4\}$	01010
B_5	$\{\phi\}$	00000	$\{\phi\}$	00000

8286643541

28-4-13

• Calculation of IN and OUT.

Initial.

Block	IN[B]	OUT[B]
B_1	00000	11000
B_2	00000	00100
B_3	00000	00010
B_4	00000	00001
B_5	00000	00000

Pass-1

Block	IN[B]	OUT[B]
B_1	00100	11000
B_2	11000	01100
B_3	01100	00110
B_4	00110	00101
B_5	00111	00111

→ same.

$IN[B_1] = OUT[B_2]$

$OUT[B_1] = IN[B_1] - KILL[B_1] \cup GEN[B_1]$

$= 00100 - 00111 \cup 11000$

$= 00100 \cup 11000$

$IN[B_2] = OUT[B_1] \cup OUT[B_3]$

$= 11000 \cup 00000$

$= 11000$

$OUT[B_2] = IN[B_2] - KILL[B_2] \cup GEN[B_2]$

$= 11000 - 10000 \cup 00100$

$= 01000 \cup 00100$

$= 01100$

$IN[B_3] = OUT[B_2]$

$= 01100$

$OUT[B_3] = IN[B_3] - KILL[B_3] \cup GEN[B_3]$

$= 01100 - 01001 \cup 00010$

$= 00110$

$IN[B_4] = OUT[B_3] = 00110$

$OUT[B_4] = IN[B_4] - KILL[B_4] \cup GEN[B_4]$

$= 00110 - 01010 \cup 00001$

$= 00100 \cup 00001$

$= 00101$

Pass - 2

Blocks	IN[B]	OUT[B]
B_1	01100	11000
B_2	11111	01111
B_3	01111	01100 ∨ 00110
B_4	00110	00101
B_5	00111	00111

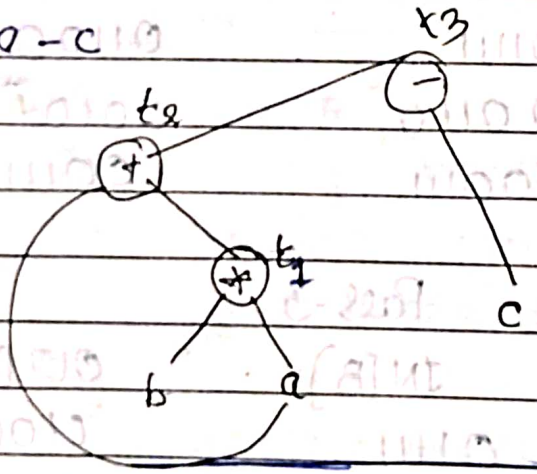
Pass - 3

Blocks	IN[B]	OUT[B]
B_1	01111	11000
B_2	11111	01111
B_3	01111	01100 ∨ 00110
B_4	00110	00101
B_5	00111	00111

- **CODE GENERATION ALGORITHM AND FUNCTION**
  GetReg

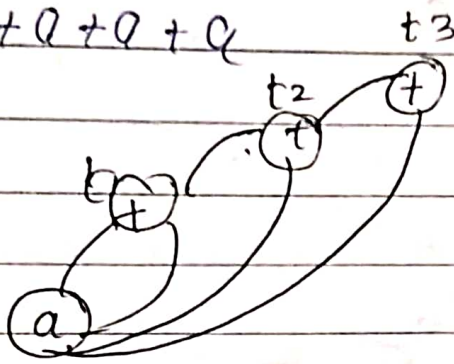
- **DAG (Directed Acyclic Graph)** (local optimization)

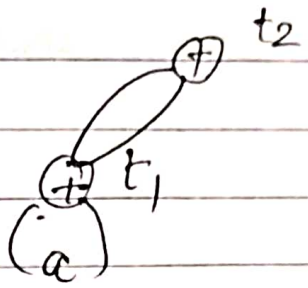
(i) a + b * a - c



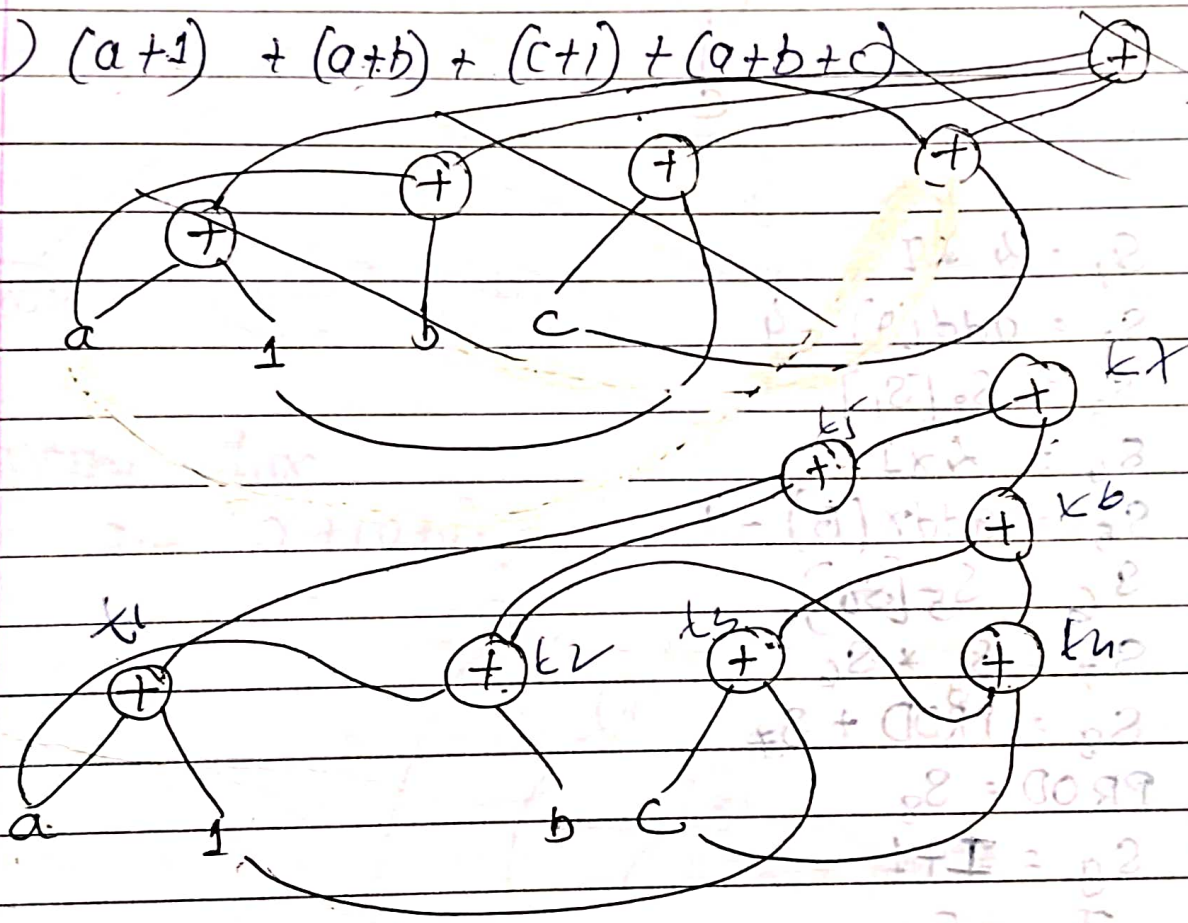
t1 = b * a
t2 = a + t1
t3 = t2 - c

(ii) $a+a+a+a$





(iii) $(a+1)+(a+b)+(c+1)+(a+b+c)$



Ques: 1. Construct a syntax tree and DAG for the following expression:

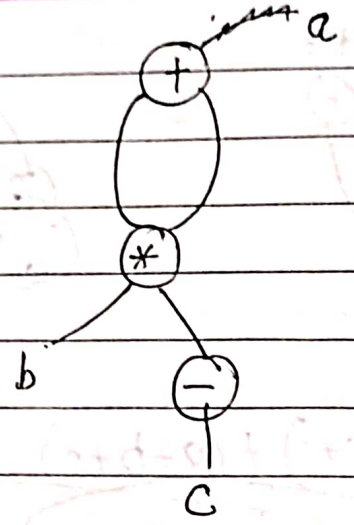
(i) $a = b * -c + b * -c$

(ii) $x = a + (a+a) + ((a+a+a) + (a+a+a+a))$

(iii) $y = a + (b * c) - \{-(b+c)\}$

(iv) $z = (a * -b) + (c - (d+e))$

Solve. (i)



Ques: $S_1 = 4 * I$

$S_2 = addr(A) - 4$

$S_3 = S_2[S_1]$

$S_4 = 4 * I$

$S_5 = addr(B) - 4$

$S_6 = S_5[S_4]$

$S_7 = S_3 * S_6$

$S_8 = PROD + S_7$

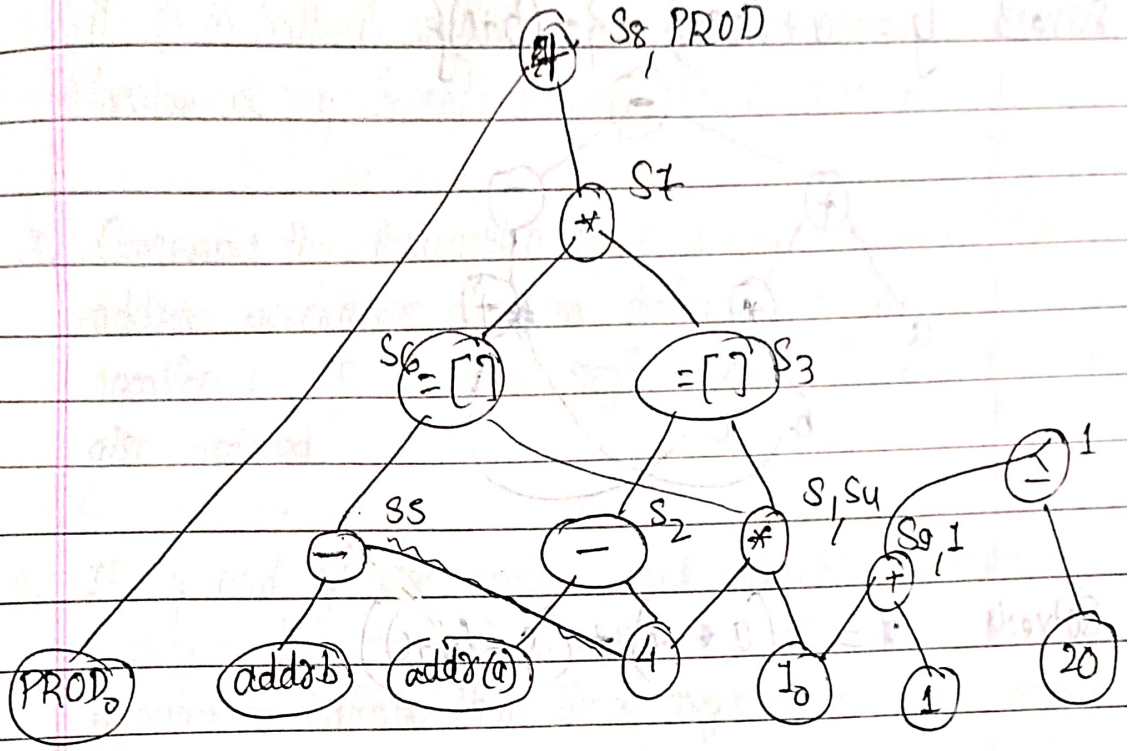
$PROD = S_8$

$S_9 = I + 1$

$I = S_9$

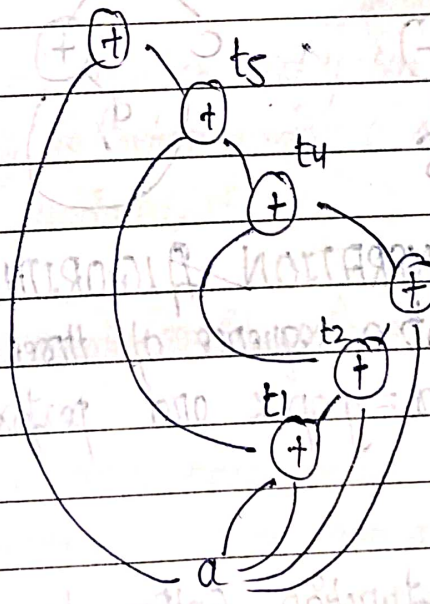
If $I \leq 20$ goto (1)

Q.T.O

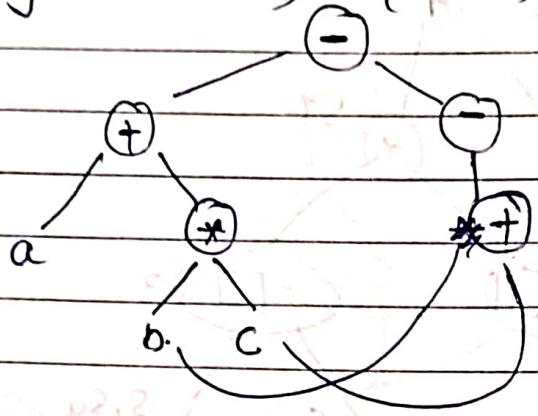


Solve (ii) DAG for

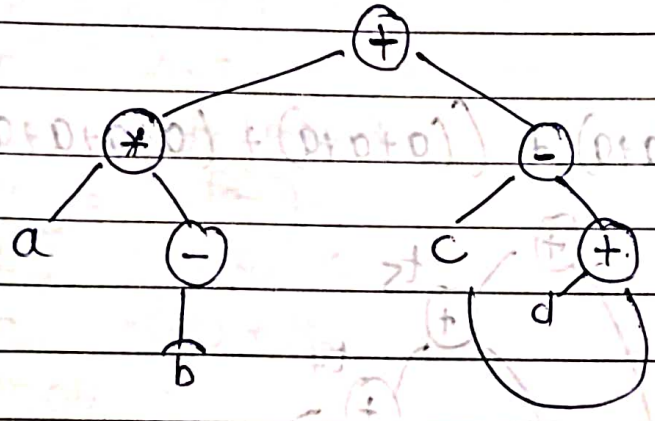
$x = a + (a+a) + ((a+a+a) + (a+a+a+a))$



Solve:3 $y = a + (b * c) - \{-(b + c)\}$



Solve:4 $x = (a * -b) + (c - (d + c))$



- [CODE GENERATION ALGORITHM:]
  It takes input a sequence of three address statements of form $x = y$ op z and performs following action:

1. Calls the function GetReg to determine location L for x.

2. Consult the address descriptor for y to determine y' (current location of y). Prefers the register for y

-[-(2.13)]

ACE
Page No.
Date: / /

if it is both in register and memory location: if value of y is not already in L then mov y, L.

3. Generates the instruction op x', L ; and updates the address descriptor of x to indicate that x is in location L, if L is a register then its descriptor is also updated.

4. If y and y' are register and current values of y and x have no next use then register descriptor is updated to indicate that those register will no longer contain y or x.

- [Function GetReg: for x = y op z]

1. If y is in a register and y has no next use then register of y is returned for L and address descriptor is updated.

2. Failing 1 an empty register is returned for L.

3. Failing (2) if x has a next use or op is an indexing operator requiring register then a suitable occupied register whose datum is referenced furthest in future, is selected, and its value is stored in memory by mov.

4. If x is not used in the block and no suitable register is found the L is selected as a m/y location.

- Elimination of left Recursion from any grammer is required because if the left recursive RD is made as input to the top-down parser for the parsing then the parser may undergo into an infinite loop.

- Elimination of left factoring reduces the amount of backtracking.

- DYNAMIC STORAGE ALLOCATION. Unit-4

The data under the program control can be allocated dynamically using the space of the heap memory.

→ There are two strategies for the dynamic allocation of the data they are:

(i) Explicit allocation
(ii) Implicit allocation.

These techniques are explained as follows:

(1) Explicit allocation:

→ The explicit allocation of the data is done using some procedures or functions as -alloc. In PASCAL allocation of memory is done using the function 'new' and the de-allocation is done using the function 'dispose'.

→ The explicit allocation can be performed for both variable sized as well as fixed sized blocks as explained below:

(a) Explicit allocation of fixed sized blocks:

→ The simplest form of dynamic allocation involves blocks of a fixed size. It is done by maintaining a 'free list' of blocks.

Thus, allocation and deallocation can be done quickly with little or no overhead. The pointer that points to the first block of free list is called the AVAILABLE.

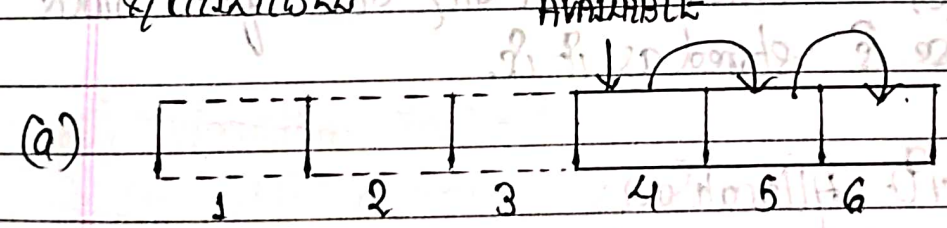


Fig: Deallocated block is added to the list of available blocks.

(ii) Explicit Allocation of Variable-Sized blocks:
During such allocation, due to allocation and deallocation of var blocks, storage can become fragmented i.e the heap may consists of alternated free blocks.

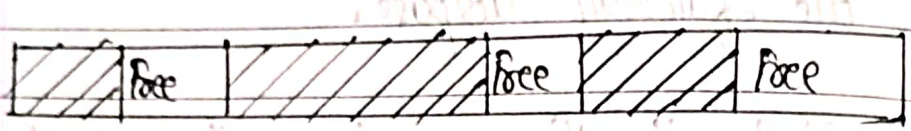


Fig: free and used blocks in Heap.

→ During such case, the allocation can be done using various method one of which is called first fit method, In this method, the first free block with size greater than the required size S is searched, say f > S, then it is divided into used block of size S and free block of size f-S.

Similarly, when a block is de-allocated it is either combined with any other free block or else is stored as it is.

(2) Implicit Allocation:
The Implicit allocation of the blocks requires cooperation between user program and run-time package which is implemented by fixing the format of storage blocks.

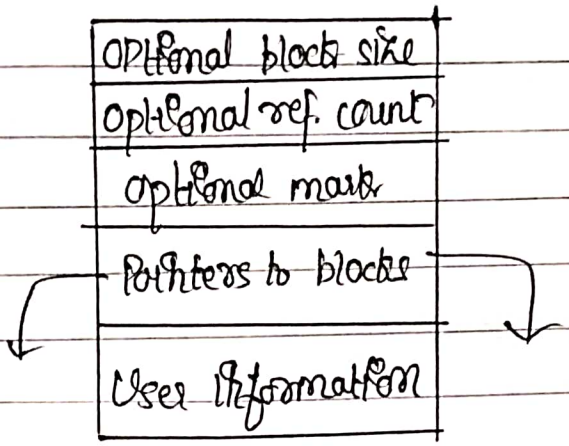
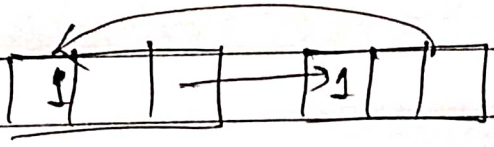


Fig: Format of block.

There are two strategies of implicit allocation as explained below:

(i) Reference Count:

→ In reference count we keep track of the number of blocks that points directly to the present block thus signifying its used and if ever this count drops to zero, then the block can be deallocated as it is a garbage.



(ii) Marking Technique:

→ In this technique, the user program is suspended temporarily and the frozen pointers are used to mark the or determine the blocks that are in use.

→ Thus the unused blocks in the heap can be collected.