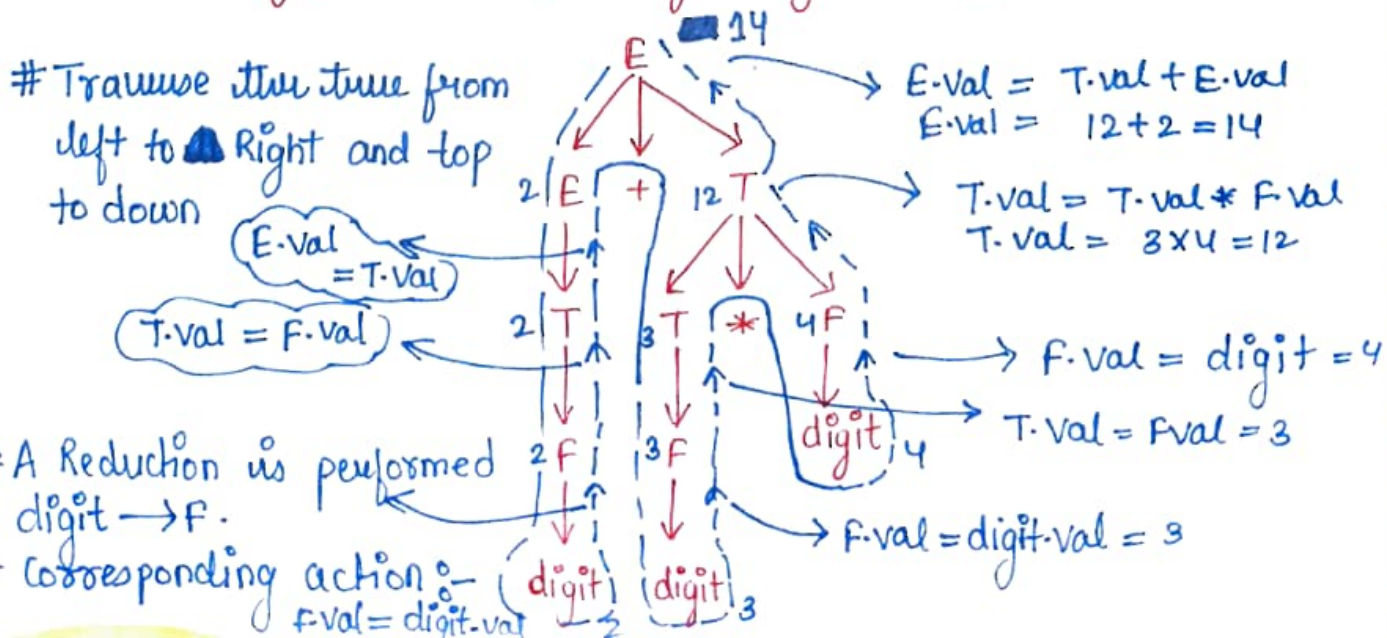
Compiler design
Syntax directed definition (SDD)

SDD = grammar + semantic Rules
similar to syntax Analyzer, semantic Analyzer's output is also 'parse tree'.
But the 'Parse tree' generated in syntax Analyzer is verified semantically in the semantic Analyzer phase. In this phase, we check if the parse tree generated is meaningful or not.
Semantically Correct means according to the Rule decided for each Operations.

Reduction Rules # Production	Action Corresponding to Reduction. # Semantic Rule
$E \to E+T$	E.val = Eval + T.val
$E \to T$	E.val = T.val
$T \to T*f$	T.val = T.val * F.val
$T \to F$	T.val = F.val
$f \to digit$	F.val = digit.val

⇒ Here, Each production is defined semantically in the corresponding Semantic Rule. This is also called Syntax directed definition.

⇒ Corresponding parse tree for the given grammar :-



When traversing from top to down and left to Right ⇒ Not first time but when the variable is visited second time while going up during reduction ⇒ then corresponding semantic Rule (Action) is performed.

In (T.val) ⇒ 'val' → attribute of variable 'T'.

Attributes are associated with grammar symbols and semantic rules are associated with Productions

Attributes may be number, strings, memory (Address), data types etc.

[Context free grammar + Semantic Rule = SDD]

(Q.)

Production	Semantic Rule
E → E # T	E.val = Eval * T.val
E → T	E.val = T.val
T → T $ F	T.val = ———
T → F	T.val = F.val
F → digit	F.val = digit.val

If Expression 8 # 12 $ 4 # 16 $ 12 # 4 $ 2 is evaluated as 512 Then which of the following is correct Replacement for blank.

∴ Here ⇒ two operations ⇒ '#' and '$'

(E → E # T) → shows '#' is left associative as this is a left Recursive grammar.

(T → T $ F) → shows '$' is left associative.

precedence of '$' > precedence of '#'.

Note :- Operator which is closer to start Variable ⇒ the precedence of that operator is lower than variable which far from start variable.

using ⇒ (T.val = T.val − f.val) corresponding to ——

→ '$' → − } → here, precedence of '−' is greater corresponding to this grammar than '*'.
'#' → * }

⇒ [redacted] 8#1284#1681 2#482

⇒ (8)*(12−4)*(16−12)*(4−2)

⇒ $2^3 * 2^3 * 2^2 * 2^1$

⇒ 2^9 = (512)

Types of SDD

S-attributed SDD

1) A SDD that uses only synthesized attribute is called S-Attributed SDD.

Example:- A → BCD
$A_i = B_i$
$A_i = C_i$
$A_i = D_i$

2) Semantic actions are always placed at the Right end of the production. (Postfix SDD)

A → BCD { }
↳ semantic actions

3) attributes are evaluated with Bottom up parser.

L-attributed SDD

1) A SDD that uses both synthesied and inherited attributes is called as L-attributed SDD but each inherited attribute is restricted to inherit from parent or left sibling only.

Example:- A → BCD
$[C_i = A_i$
$C_i = B_i]$

2) Semantic actions are placed anywhere on R.hs. of the Production.

A → { }BCD, A → B{ }CD
↑ Semantic actions

3) Attributes are evaluated by traversing Parse tree depth first left to Right.

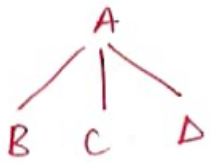
Synthesized attribute :-

$A \rightarrow BCD$

$A_i^0 = B_i$

$A_i^0 = C_i$

$A_i^0 = D_i$



The value of Node A in the "Parse tree" corresponding to attribute - "i" can only be calculated by the attribute value of the children of the node then it is called Synthesized attributes.

Inherited attribute :-

$A \rightarrow BCD$

$C_i^0 = A_i$

$C_i^0 = B_i$

$C_i^0 = D_i$

⇒ If the value of C Node's attribute "i" is calculated with the attribute value of its parent's attribute or its sibling's attribute, then its is called Inherited attribute

Relation between S-attributed and L-attributed SDD :-

⇒ (S) L

⇒ L-attributed SDD is a superset of S-attributed SDD.

⇒ S-attributed SDD is a subset of L-attributed SDD.

⇒ Every S-attributed SDD is a L-attributed SDD.

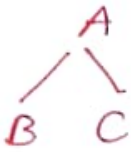
given Productions :-

$A \rightarrow BC \{ B^i = A^i, C^i = B^i, A^i = C^i \}$

$A \rightarrow QR \{ R^i = A^i, Q^i = R^i, A^i = Q^i \}$

$A \rightarrow PQ \{ A^i = P^i, A^i = Q^i \}$

First Production :-



$B^i = A^i$ → Parent

$C^i = B^i$ → left sibling

$A^i = C^i$ → Parent / child } S-attribut

$A \rightarrow BC$ → L-attributed only

Second Production :-

$R^i = A^i$ → Parent

$Q^i = R^i$ → Right sibling

→ Neither S or L attributed SDD

third Production :-

$A^i = P^i$ → children

$A^i = Q^i$ → children

both

L-attributed SDD ← S-attributed SDD

An SDD to Convert Binary to Decimal :-

decimal value

$S \rightarrow L \quad \{ S.dv = L.dv \}$

$L \rightarrow LB \quad \{ L.dv = 2 * L.dv + B.dv \}$

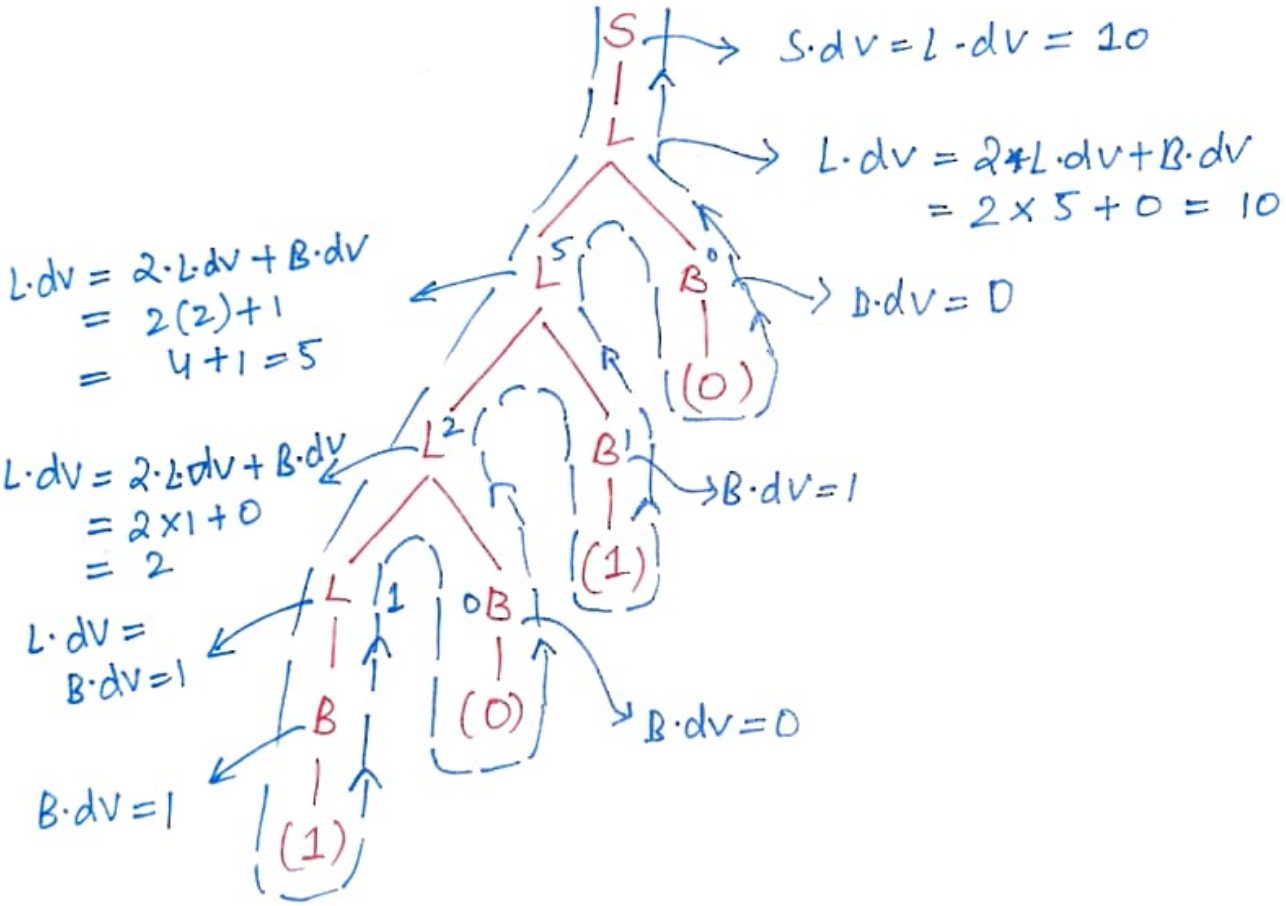
$L \rightarrow B \quad \{ L.dv = B.dv \}$

$B \rightarrow 0 \quad \{ B.dv = 0 \}$

$B \rightarrow 1 \quad \{ B.dv = 1 \}$

Productions | grammar

Parse tree :- [For Binary = 1010] ⇒ decimal = 10



S.dv = 10

SSD to Binary to Decimal with fraction :-

:: Binary = 101.101

↓ Decimal Conversion

$\Rightarrow 1\times2^{2}+0\times2^{1}+1\times2^{0}+1\times2^{-1}+0\times2^{-2}\times1\times2^{-3}$

$\Rightarrow 4+0+1+\frac{1}{2}+\frac{1}{8}$

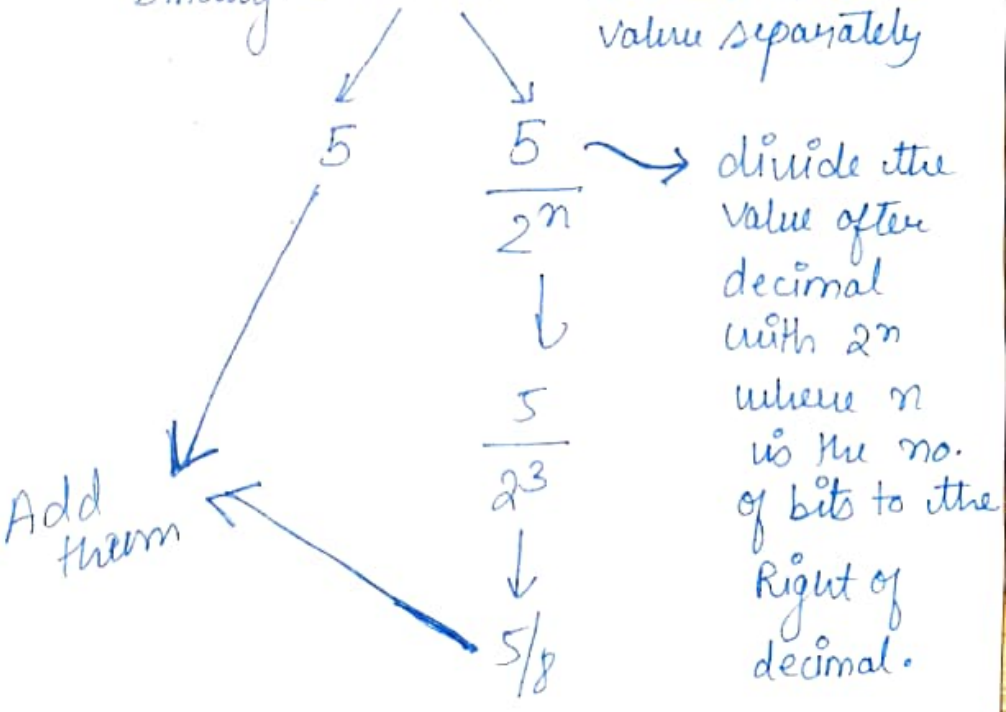
$\Rightarrow 5+\frac{1}{2}+\frac{1}{8}$

$\Rightarrow \frac{40+4+1}{8} = \left(\frac{45}{8}\right) = 5.625$

---

method-2 :-

Binary = 101.101 Calculate decimal value separately



$5+\frac{5}{8} = \frac{45}{8} = 5.625$

SSD for Binary to Decimal with fraction :-

$$S \rightarrow L_1 \cdot L_2 \quad \{ S.dv = L_1.dv + \frac{L_2.dv}{2^{L_2.nb}} \}$$

$$L \rightarrow L \cdot B \quad \{ L.dv = 2 * L.dv + B.dv \;\; L.nb = L.nb + B.nb \}$$

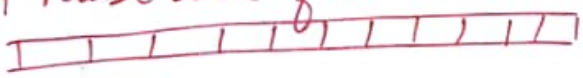
$$L \rightarrow B \quad \{ L.dv = B.dv \;\; L.nb = B.nb \}$$

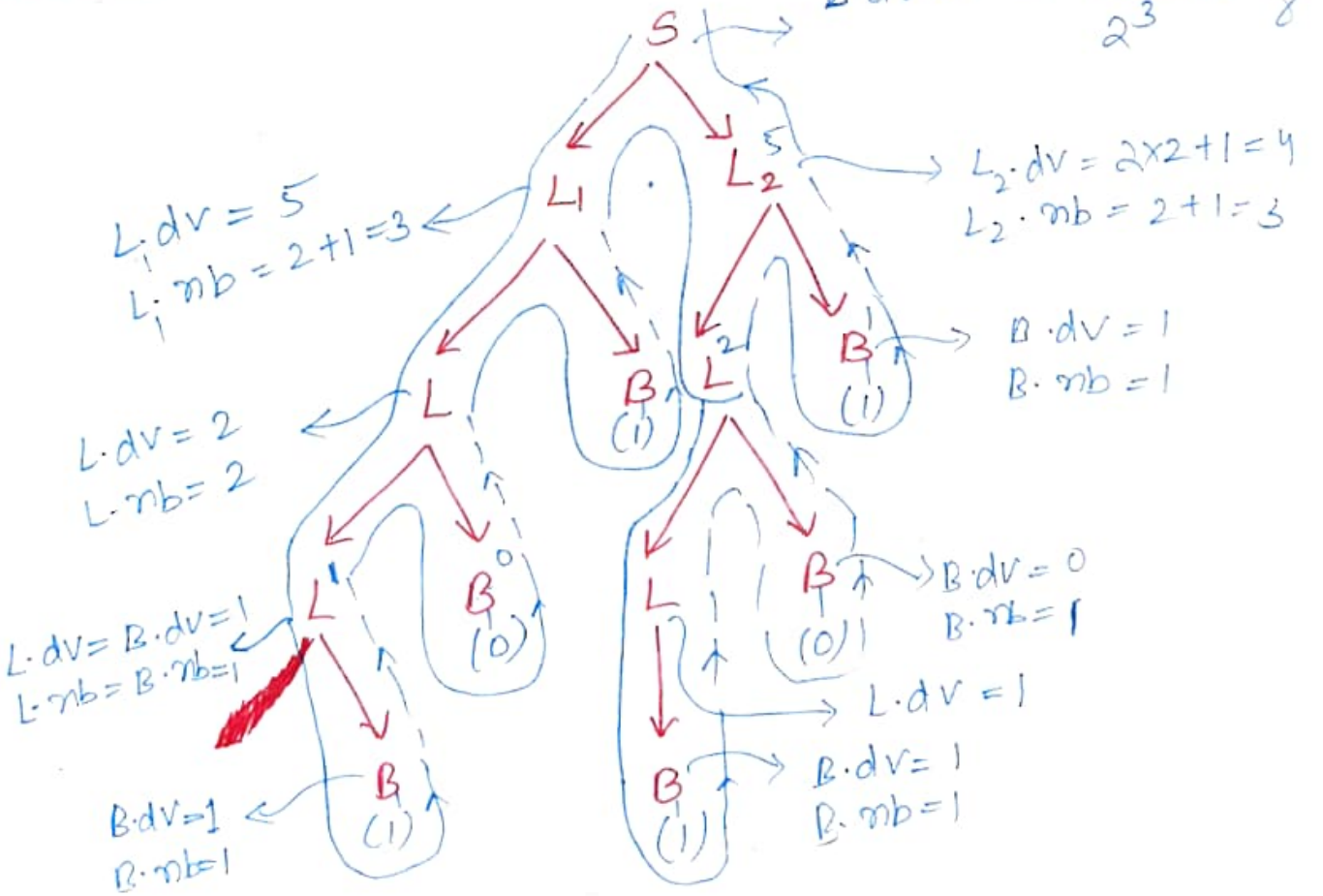
$$B \rightarrow 0 \quad \{ B.dv = 0 \;\; B.nb = 1 \}$$

($B.dv$ → decimal value; $B.nb$ → number of bits)

$$B \rightarrow 1 \quad \{ B.dv = 1 \;\; B.nb = 0 \}$$

Parse tree for 101.101 :-





SDD to Construct Syntax Tree :-

given SDD :-

$E \rightarrow E+T \quad \{ E.nptr = mknode(E.nptr, +, T.nptr) \}$

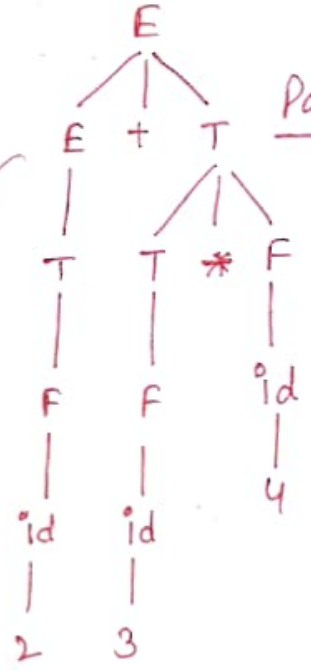
$E \rightarrow T \quad \{ E.nptr = T.nptr \}$

$T \rightarrow T*F \quad \{ T.nptr = mknode(T.nptr, *, F.nptr) \}$

$T \rightarrow F \quad \{ T.nptr = F.nptr \}$

$F \rightarrow id \quad \{ F.nptr = mknode(Null, id, Null) \}$

Parse Tree Corresponding to given for Expression :- (2+3x4)



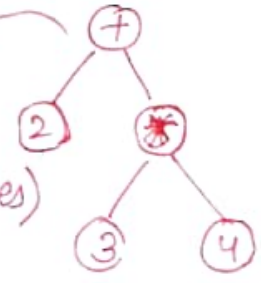
Parse Tree

Parse Tree is also called Concrete syntax tree

[uses Terminals and Variables both]

syntax tree :- If a tree is Constructed only using terminals then that tree is syntax tree (Variables are hidden).

syntax tree is also called abstract syntax tree as some details (variables) are hidden.



syntax Tree

The given SDD corresponds to a syntax tree.

Syntax tree is also a representation of Intermidiate Code in Intermidiate Code generation phase of Compiler.

given SDD :- To Construct a syntax tree

$E \rightarrow E+T$ { E.nptr = mknode (E.nptr, +, T.nptr)

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

$T \rightarrow T*F$ { T.nptr = mknode(T.nptr, *, F.nptr)

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

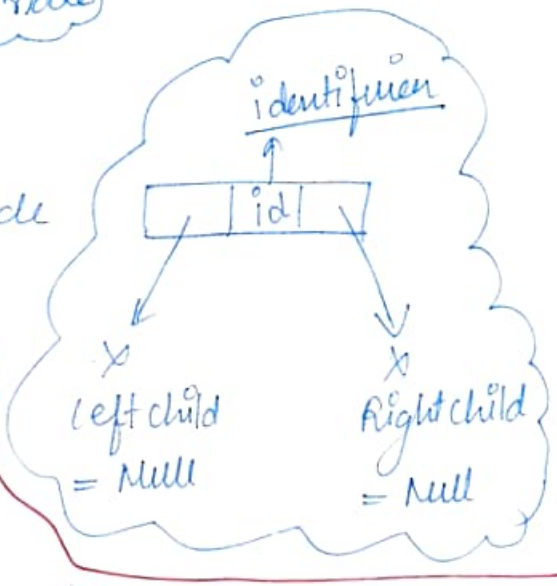
$F \rightarrow id$ { F.nptr = mk (Null, id, Null) }

F.nptr → reference address of node

mk → mak node function()

:: mknode(Null, id, Null)

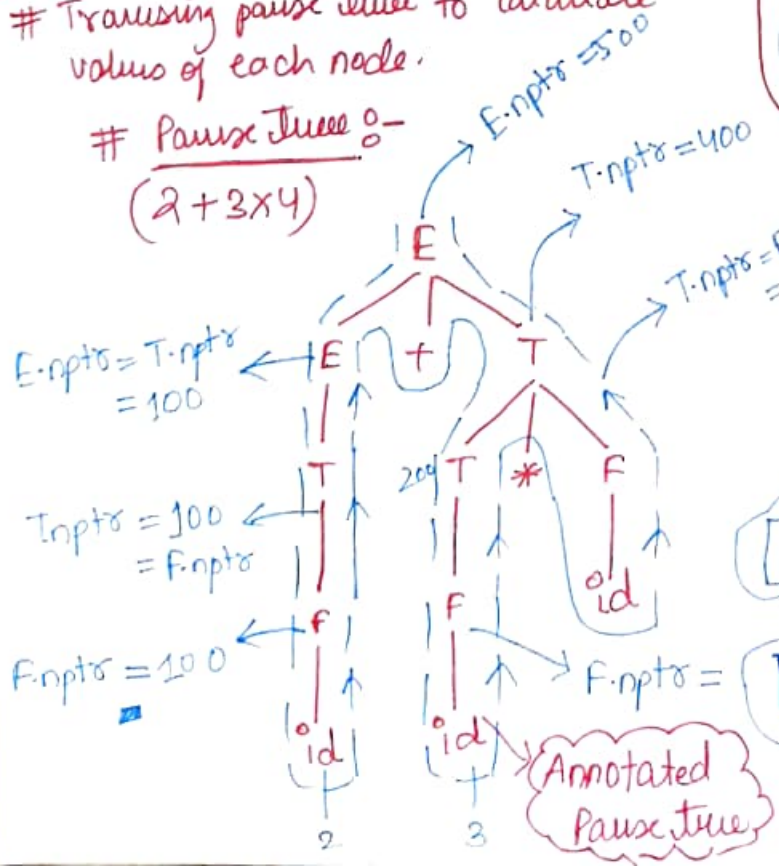
↳ It creates a node

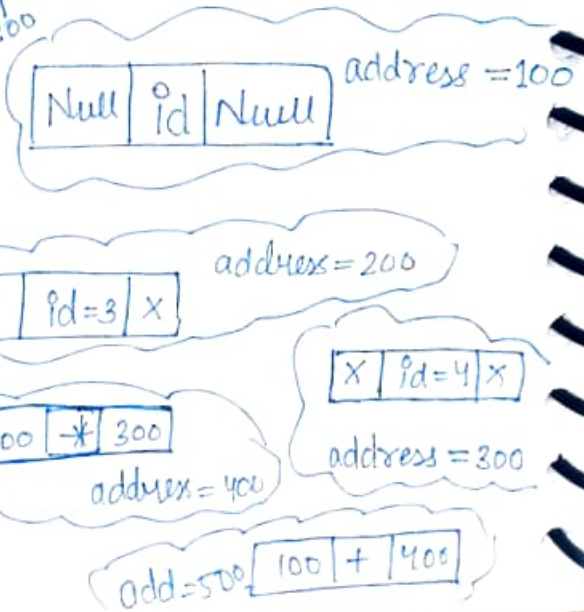


Traversing parse tree to calculate value of each node.

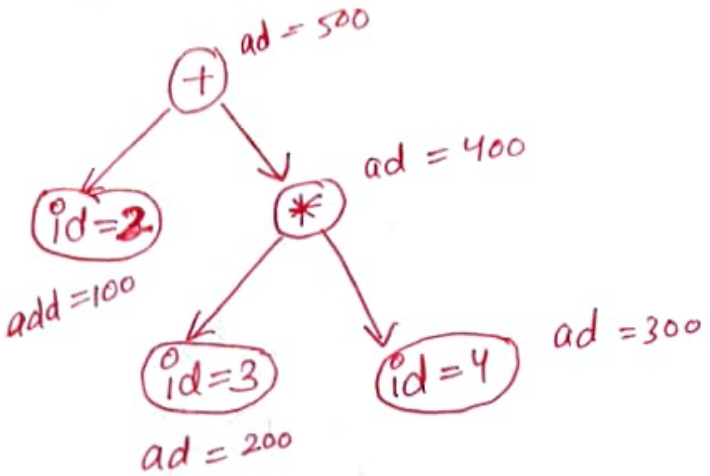
Parse Tree :-

(2+3*4)





Corresponding syntax Tree :-



SDD to store Type information into symbol table corresponding to a variable :-

Symbol table :- int x, y, z;

Variable name	Variable type
y	int
z	int
x	int

SDD :-

$D \longrightarrow D_1, id \quad \{ \text{Add type}(id, D_1.type);\ D.type = D_1.type \}$

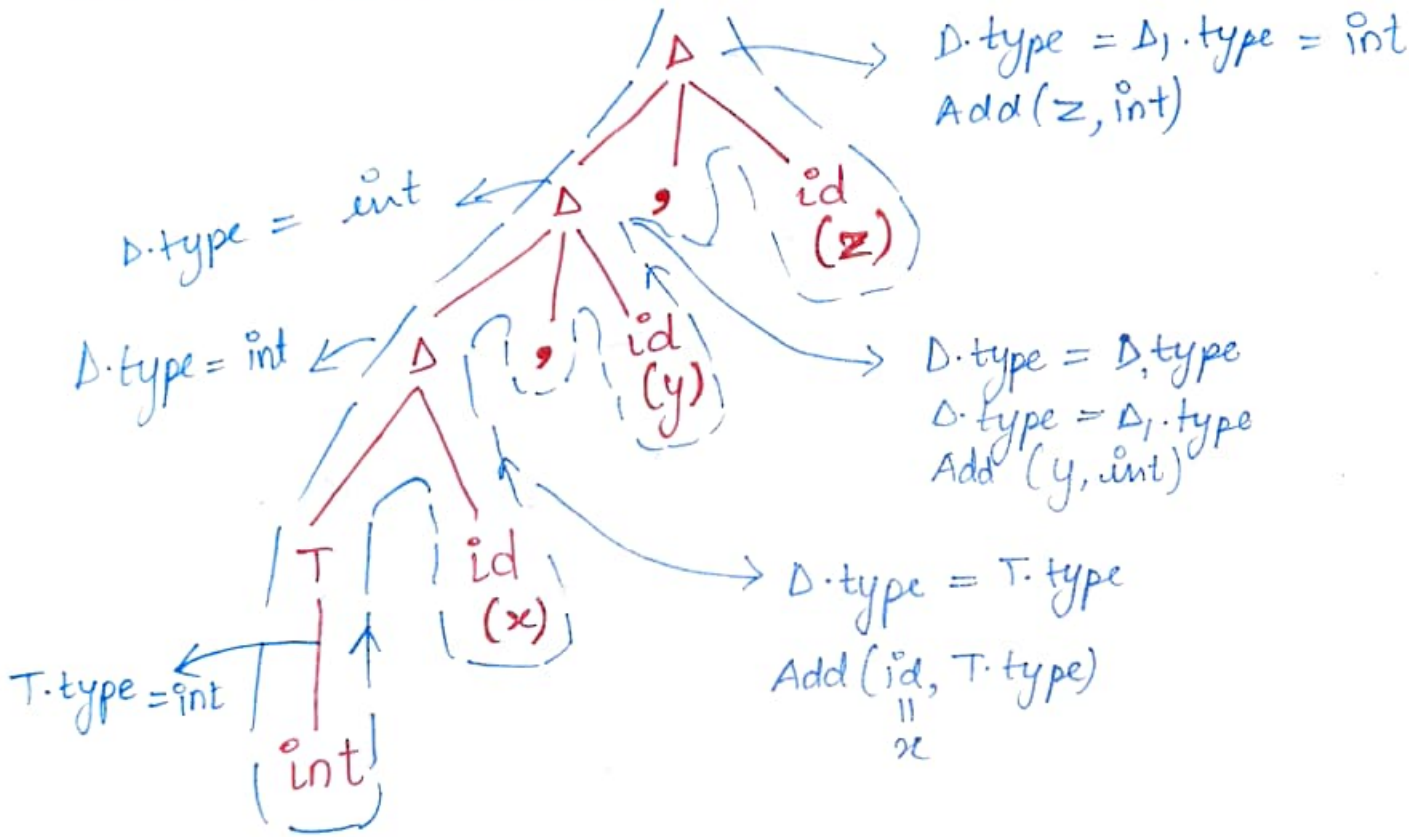
$D \longrightarrow T\ id \quad \{ \text{Add type}(id, T.type);\ D.type = T.Type \}$

$T \rightarrow int \{ T.type = int \}$

$T \rightarrow char \{ T.type = char \}$

$T \rightarrow float \{ T.type = float \}$

Parse tree for (int x, y, z):-



Symbol table

V.N	T.N
x	int
y	int
z	int

SDD to generate three address Code :-

A input Expression is Converted into three - address Code.

$S \rightarrow id = E$ { gen (id.name = E.place) }

$E \rightarrow E+T$ { E.place = new temp(); gen(E.place = E.place + T.place); }

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

$T \rightarrow T*F$ { T.place = new temp() gen (T.place = T.place * F.place); }

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

$F \rightarrow id$ { F.place = id.name }

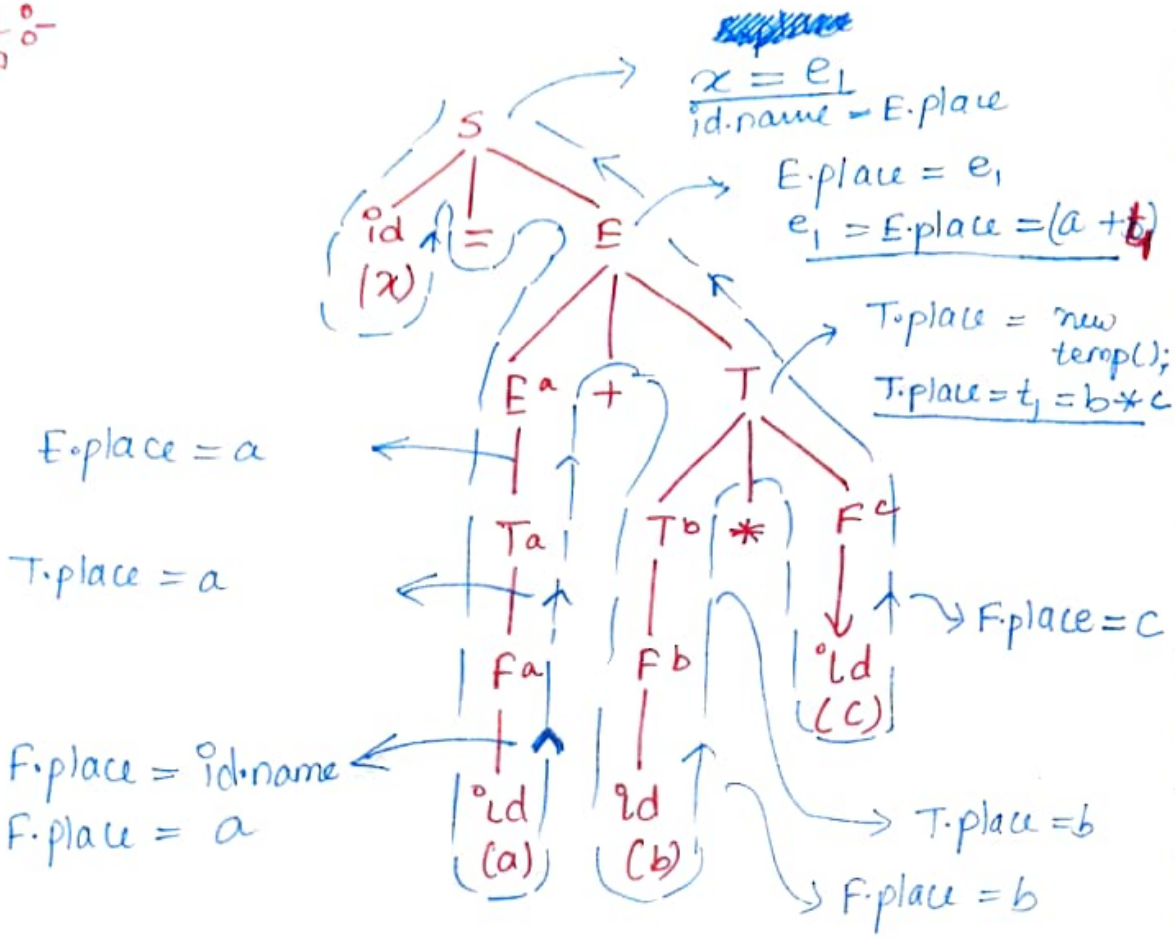
Three address Code is a representation of a intermidiate Code.

For eg :- Expression ⇒ $x = a+b*c$

This expression when stored in memory must contain 4 addresses for, x, a, b, c ⇒ we need to convert this into a three address Code where only three address Code is required.

For example :- $x = a+b*C$

Parse Tree :-



new temp() → creates a temporary variable t_1 = T.place

∴ Three address Code generated are :-

$t_1 = b * c$

$e_1 = a + t_1$

$x = e_1$

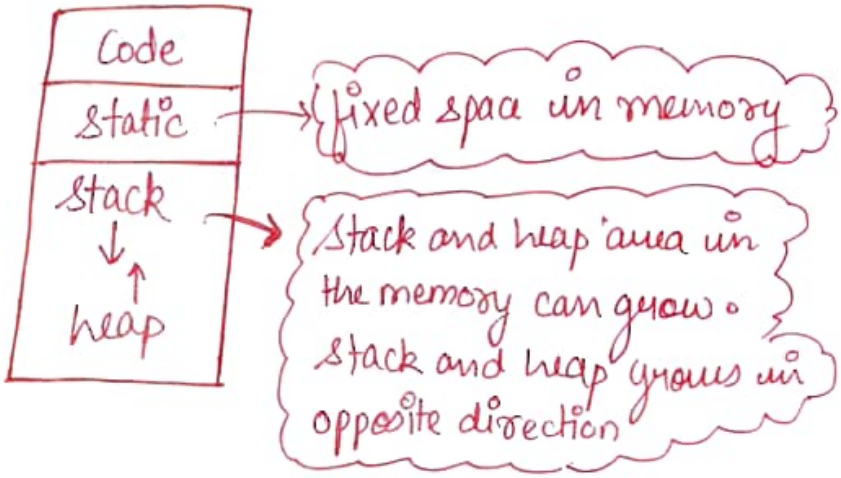
Run Time Environment

Runtime is the time during which program is executing.

Compiler does not execute the program ⇒ processor does.

Compiler compiles the program and creates a runtime environment for smooth execution of program.

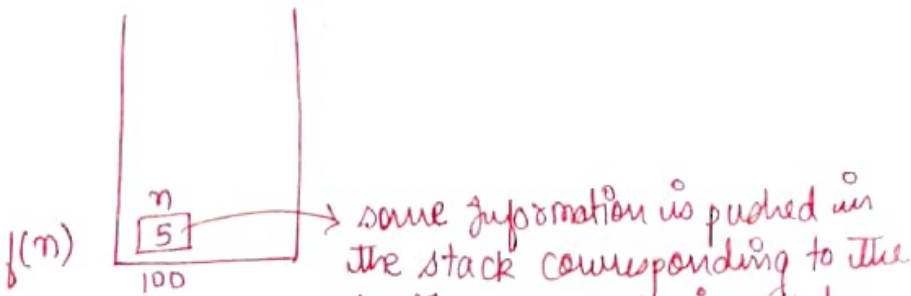
Main memory is divided into some parts :-



Consider a function :-

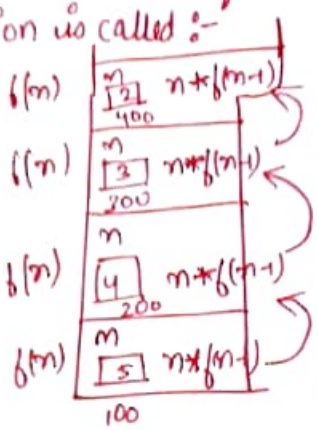
```
fact (int n){
    if(n<=1) return 1;
    Else
        return (n*fact(n-1));
}
```

As soon as the function is called fact(5) ⇒ a stack is created in a main memory :-



Activation record is the all the information related to a function call.

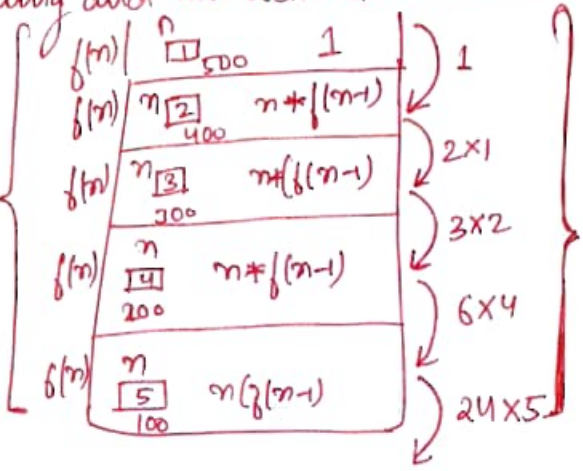
When recursion function is called :-



All the variables – 'n's are created at runtime as the functions are called.

functions keeps calling and new activation Record keeps inserting into stack.

variable 'n' in each activation record acts as a local variable for the functions.



This all happens in runtime. Insertion and popping from stack.

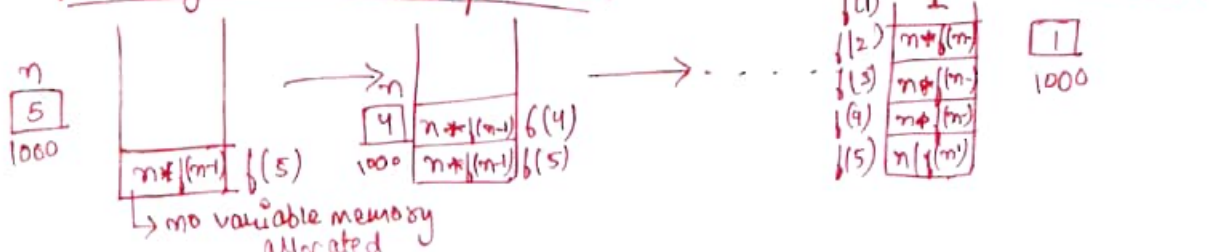
Now returning from each function pops the activation Record.

Now considering the function :-

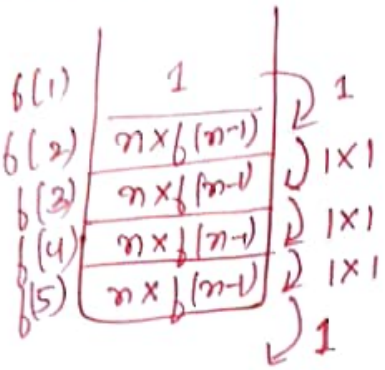
```
fact (static int n){
    if (n<=1) return 1;
    Else
        return (n*fact(n-1));
}.
```

A variable 'n' is initialized at compile time with value = 0

following runtime stack operations :-



while returning ⇒ only static int value (n=1) is now considered for all functions because static variable survives during runtime and function calls.



Input = 5
output = 1

⇒ Here, for each function ⇒ A local variable is not created, all functions used a single variable which was initialised at compiler time.

Dynamic memory allocation is of two types :-

→ system allocates memory :- For E.g, when a function is called and that function call is stored in stack and memory is allocated by system. This memory is allocated in stack area. user has no control over stack area.

→ user allocates memory :- user allocates memory in C++ using new and delete operator. This memory is allocated in heap area. user only has control over heap memory.

Memory allocation techniques :-

[redacted] static Allocation :-

(I) Memory allocation is done at Compile time
(II) Binding do not change runtime. Variable address do not change during runtime.
(III) Recursion is not supported
(IV) size of data objects must be known as Compile time
(V) Dynamic data structure not supported.

stack Allocation :- System Controlled

(I) Recursion supported
(II) Local Variable belongs to new activation record.
(III) Dynamic data structure is not supported.

Heap allocation :- User Controlled

(I) Allocation and Deallocation will be done at anytime based on user requirement.
(II) Recursion supported
(III) Dynamic data structure supported.

Activation Record :-

⇒ Whenever a function is called → Corresponding Activation Record is pushed into stack.

⇒ When a function's Role is finished and it returns value then the activation Record for that function is popped from the stack.

An activation Record for a function Call Includes :-

(I) Local Variable :- holds the data that is local to the execution of the function.

(II) Temporary values :- stores the values that arises in the intrim evaluation of expression.

(III) Machine Status :- Holds the info about the status of the machine just before the function call.

(IV) Access link :- It is used to refer to non-local variable/data held in other activation record.

(V) Control link :- Store the address of activation record of the caller function.

(VI) Actual parameters :- Store Aactual parameters that are used to send input to the caller functions.

(VII) Return Values :- To store the Result of function Call.

Before the function calling ⇒ All the values of these 7 parameters of activation Record is static/Null and default values. Once the function is called ⇒ all these 7 parameters keeps changing, does not remain static.

parameters are of two types :-

→ actual parameters :- parameters passed in function call.

→ formal parameters :- parameter passed in function definition

---

Foreg :- Cosider the function calling chain :- Main → A1 → A2 → A21 → A1

```
Main()
{
  A1()
  {
    A2();
  }
  A2(){
    A21(){
      A1();
    }
    A21();
  }
  A1();
}
```

Access link	(current fn)	control link
Null	Main	A1()
main	A1	A2()
main	A2	A21()
A2	A21	A1()
main	A1	A2()

→ just check that the current function is defined in which bigger function

just check which func^{n} is called in current function.

Intermidiate Code generation

This is a phase of Compiler where Intermediate Code is generated.

Compiler has two ends → frontend
→ backend

frontend of Compiler has the following phase :-

- → syntax Analyzer
- → Lexical Aanalyzer
- → semantic Analyzer
- → Intermediate Code generation.

backend of Compiler has :-

- → Code optimization
- → Target Code generator

All the phases of the Compiler in frontend are same in all platforms like (linux, windows).

All phases of backed are need to changed when travelling across platforms.

Intermediate Code generation phase is a phase of Compiler which is frontend and is same for all platforms.

Intermidiate Code generation

Linear form

Postfix Notation

ab+ab+c+*

↳ operator written after operands

Three address Code

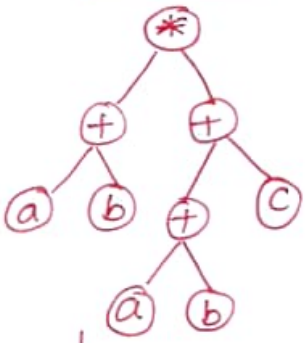
$t_1 = a+b$
$t_2 = a+b$
$t_3 = t_2+c$
$t_4 = t_1*t_3$

↳ only three variables are there in every instruction

↳ Any high level language can be converted into three address Code.

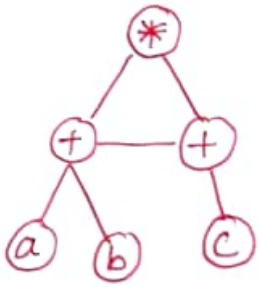
Tree form

syntax Tree



↳ formed of only terminals

DAGs



↳ first syntax tree is converted in DAGs.

↳ All the nodes in DAGs are unique with no repetition

↳ values are reused by making circles or loops are allowed

Types of three address Code :-

⇒ All instructions in the high level Code is given in the following syntax :-

All above syntax are allowed in 3-address Code.

(i) $x = y \ op \ z$ ⟶ an instruction with three variable.

(ii) $x = op \ z$ ⟶ op is unary operator.

(iii) $x = y$ ⟶ assignment.

(iv) if (x relational opy) goto L ⟶ Conditional goto statement

(v) goto L → unconditional goto statement

(vi) $A[i] = x$, $y = A[i]$ } Assigning an element to array / storing the value of Array.

(vii) $x = *p$ ⟶ representation of pointer. pointer's Address value is stored in x

$y = \&x$ ⟶ storing the address of x in y.

Representation of three address Code in the memory :-

Expression in C : $[-(a*b) + (c*d+e)]$

three address Code

$t_1 = a*b$
$t_2 = -t_1$
$t_3 = c*d$
$t_4 = t_3 + e$
$t_5 = t_2 + t_4$

} This three address Code can be represented/ stored in memory in following 3 representations :-

(i) quadruples
(ii) Triples
(iii) Indirect Triples

Quadruple Representation :- There will be total 4 columns
Each expression / instruction represents rows

		operator	OP1	OP2	result
5 expression	0	*	a	b	t_1
	1	-	t_1		t_2
	2	*	c	d	t_3
	3	+	t_3	e	t_4
	4	+	t_2	t_4	t_5

(4 columns)

Adv :-
⇒ statements can be moved or rearranged.

dis Adv :-
⇒ One extra column so more space

Triples Representation :- # There will be 3 column
result column is eliminated

sequence/add	operator	op1	op2
0	*	a	b
1	-	(0)	
2	*	c	d
3	+	(2)	e
4	+	(1)	(3)

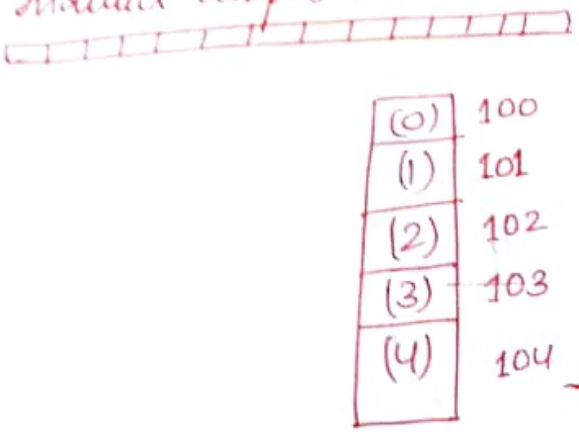
→ (a*b) is stored in a Temporary Register
→ sequence of the temporary register is used to denote value.

Indexing is turned into address basically

Adv :-
⇒ less space is occupied due to one lesser column.

dis Adv :-
⇒ Instructions / statements / rows cannot be rearranged or moved.

Indirect Triples Instruction :-



Here, 100 is the address of a block where (0) is stored. (0) is a reference to the three address code triple instruction which is to be Computed / Executed.

Two memory references are used, required.

more space is required

instructions / statements can be moved or rearranged.

Here, reference is stored in the table and sequentially we run each row.

For each row, we go the stored reference instruction and run that instruction which is stored in another triples representation.

Converting Program into three address Code :-

① If (a<b) then t=1 else t=0
↳ I ↳ III ↳ IV
↳ II

⇒ There are four address used to execute this Code, so this instruction need to be Converted into three address Code.

⇒ If already three Address Code ⇒ current statement is the answer.

⇒ Three address Code is written using backpatching :-

↳ Meaning the labels are empty and filling them later is called backpatching.

↳ Conditional / unConditional goto statements are left blank initially.

↳ After writing whole Three address Code ⇒ labels which are filled later :-

Three address Code :-

1) if (a<b) then goto (4)
2) t=0
3) goto (5)
4) t=1
5) → outside If/Else

(4) filled later after writing whole 3 address code

This process is called "back-patching".

---

(11) if (a<b) && (c<d) then t=1 else t=0.

(Ans) (I) (II) (III) (IV) (V) (VI)

Three address Code :-

1) if (a<b) goto 4
2) t=0
3) goto 7
4) if (c<d) goto 6
5) goto 12
6) t=1
7)

Example – (III) :- backpatching :-

```
for(int i = 1; i<=n; i++)
  {
    x = a+b*c;
  }
```

Three address code :-

1) i=1
2) if (i<=n) goto (4)
3) goto (9)
4) $t_1 = b*c$
5) $t_2 = a+t_1$
6) $x = t_2$
7) i = i+1
8) goto (2)
9)

1) i=1
2) if (i>n) goto 8
3) $t_1 = b*c$
4) $t_2 = a+t_1$
5) $x = t_2$
6) i = i+1
7) goto 2
8)

Example – (IV) Code :-

```
Switch(i)
{
 case-1:
        x1 = a1+b1*c1;
        break;
 Case-2:
        x2 = a2+b2*c;
        break;
 default:
        x3 = a3+b3*c3;
        break;
}
```

Three address Code :-

1) if (i==1) goto 7
2) if (i==2) goto 11
3) $t_1 = b_3 * c_3$
4) $t_2 = a_3 * t_1$
5) $x_3 = t_2$
6) ⟶ END

Case-1 :-

7) $t_1 = b_1 * c_1$
8) $t_2 = a_1 * t_1$
9) $x = t_2$
10) goto 6

Case-2 :-

11) $t_1 = b_2 * c_2$
12) $t_2 = a_2 + t_2$
13) $x = t_2$
14) goto 6

#

Example-5 :- Code :-

```
int A[10], B[10]
int x=0, i;
for(i=0 ; i<10 ; i++)
   {
     x = x + A[i]*B[i]
   }
```

Three address Code

1) $x = 0$
2) $i = 0$
3) if ($i \geq 10$) goto 15
4) t_1 = Base address of A
5) $t_2 = (i*2)$ [2 → size of int (datatype)]
6) $t_3 = t_1[t_2]$ ⟶ A[i] value
7) t_4 = Base address of B
8) $t_5 = i*2$
9) $t_6 = t_4[t_5]$ ⟶ B[i] value
10) $t_7 = t_3 * t_6$
11) $t_8 = x + t_7$
12) $x = t_8$
13) $i = i+1$
14) goto 3
15) ⟶ End

Note :-

$$A[i] = \left[\text{Base address}\right] + \underbrace{(i - \text{lower bound}) \times \text{size of data type}}_{\text{offset}}$$

$A[i] = \left[\text{Base address} + \text{off-set}\right]$

Three address Code Corresponding to 2D-Array :-

For 2-D array ⇒ It is Implemented with respect to addresses in a 1-D array only, in the memory.

Visualization :- A[4][4] :-

00	01	10	11
10	11	12	13
20	21	22	23
30	31	32	33

Representation in Memory :- (programming follows row major order.

Base address

00	01	02	03	10	11	12	13	20	21	22	23	...	33

∴ Address of i^{th} row j^{th} column :-

$$A[i][j] = \text{Base address} + \left[\left(i - \text{lower bound}\right) \times \left(\text{No. of column}\right) + j\right] \times \text{size of each element}$$

size of datatype

lower bound = 0

$$A[i][j] = \text{Base address} + \left[\left(i \times N_c\right) + j\right] \times \left(\text{size of each element}\right)$$

$X = A[i][j]$:- A : 10 × 15 (r, c)

Three address code :-

1) $t_1 = i * 15$
2) $t_2 = t_1 + j$
3) $t_3 = t_2 * 2$ → size of data type
4) $t_4 =$ base Addrss of Array A
5) $t_5 = t_4[t_3] \longrightarrow A[i][j]$
6) $X = t_5$

To find the address of 3-D Array :-

int x[m][n][k]

$x[i][j][k] =$ [base address] + [i × n × k_1 + j × k_1 + k] × (size of data type)

→ for Row major order

$x[i][j][k] = x[$ [i × n × k_1 + j × k_1 + k] × (size) $]$

Colum major Order :- int x[m][n][k_1]

$x[i][j][k] = x[(k \times m \times n + j \times m + i) \times size]$

$x[i][j][k] = x[(k \times (m \times n) + (j \times m) + i) \times size]$

Syntax tree and DAG Conversion :-

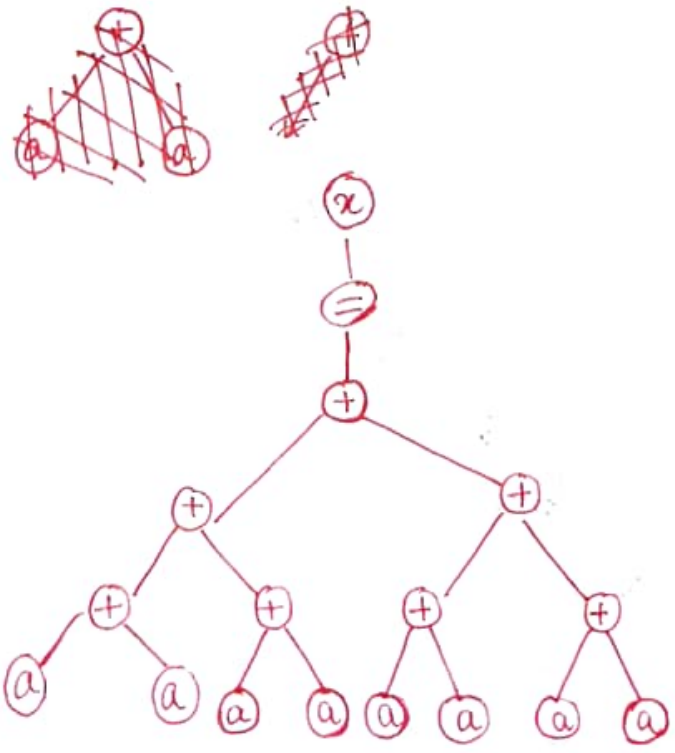
given Expression :-

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

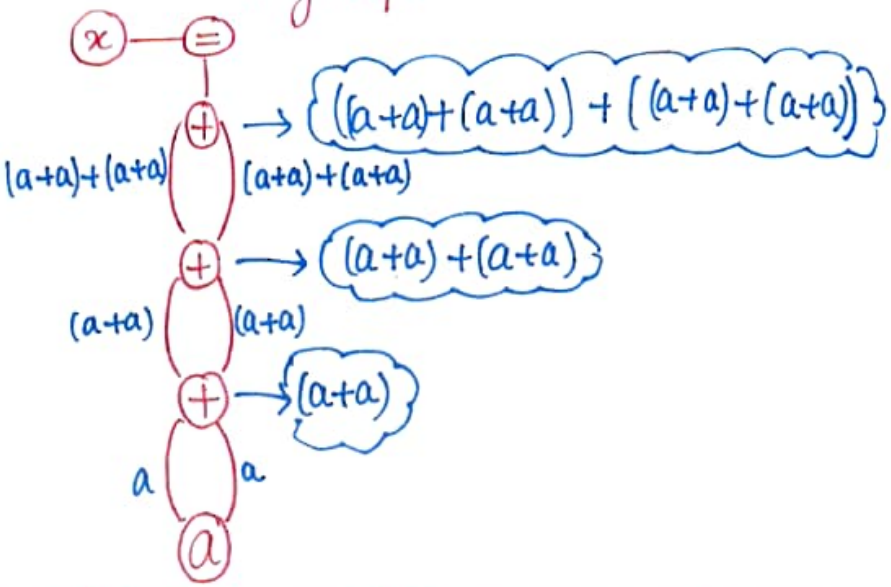
Three address code :-

1) $t_1 = a+a$
2) $t_2 = a+a$
3) $t_3 = t_2+t_1$
4) $t_4 = a+a$
5) $t_5 = a+a$
6) $t_6 = t_4+t_5$
7) $t_7 = t_3+t_6$
8) $x = t_7$

Syntax tree :- only terminals are to be considered :-



Directed A-cyclic graph :- No repetion of terminals are allowed. Compressed form of syntax Tree containing loops.

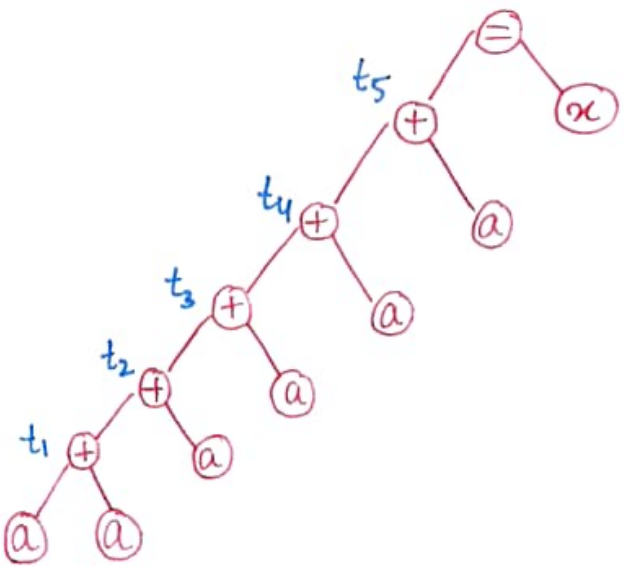


Given Expression :- $x=a+a+a+a+a+a$

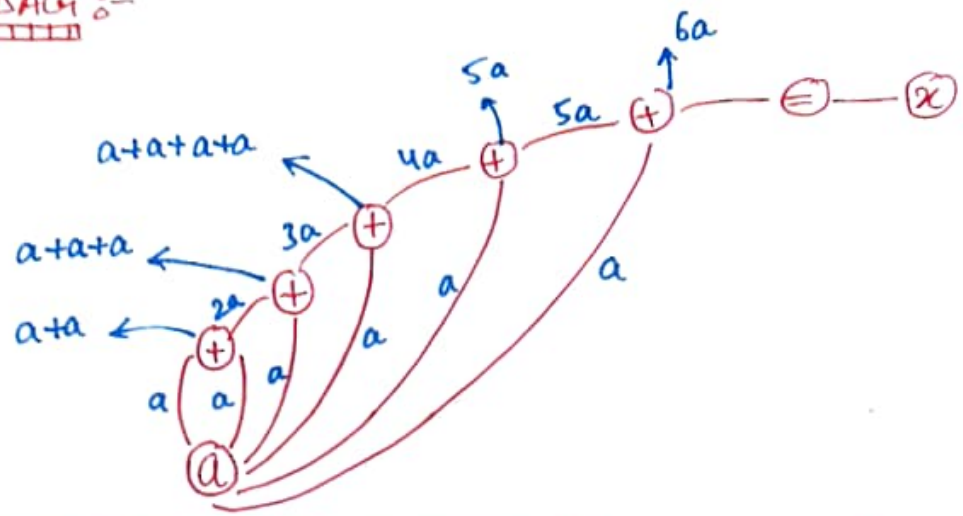
Three address Code :-

1) $t_1 = a+a$
2) $t_2 = t_1+a$
3) $t_3 = t_2+a$
4) $t_4 = t_3+a$
5) $t_5 = t_4+a$
6) $x = t_5$

Syntax Tree :-



DAG :-

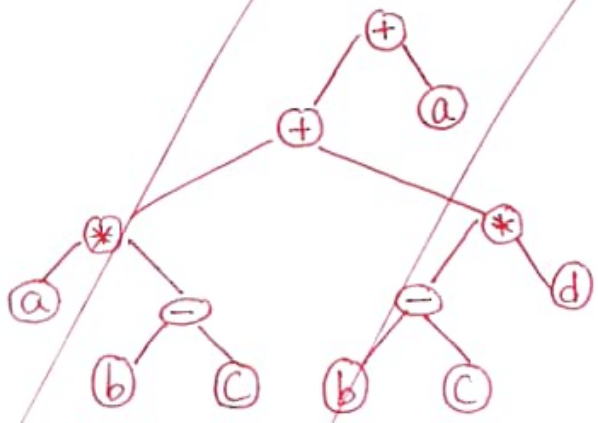


Given Expression :- $a + a*(b-c) + (b-c)*d$

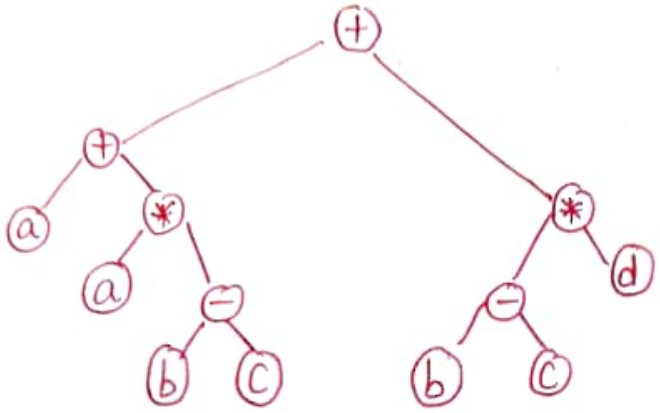
Three address Code :-

1) $t_1 = b-c$
2) $t_2 = b-c$
3) $t_3 = a*t_1$
4) $t_4 = d*t_2$
5) $t_5 = t_3 + a$
6) $t_6 = t_5 + t_3$

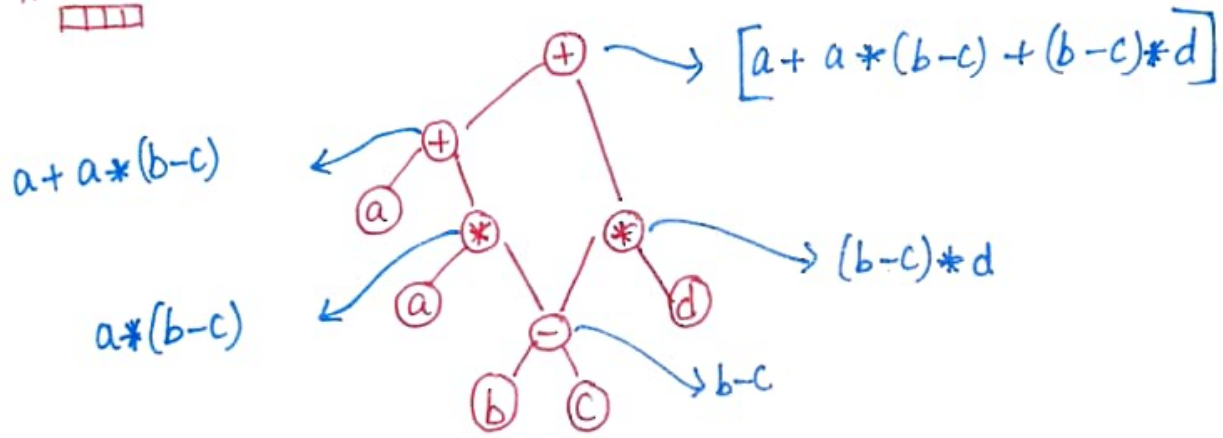
syntax tree :-



Syntax tree :-



DAG



Expression :-

a = b+c
b = a-d
c = b+c
d = a-d

Three address Code :-

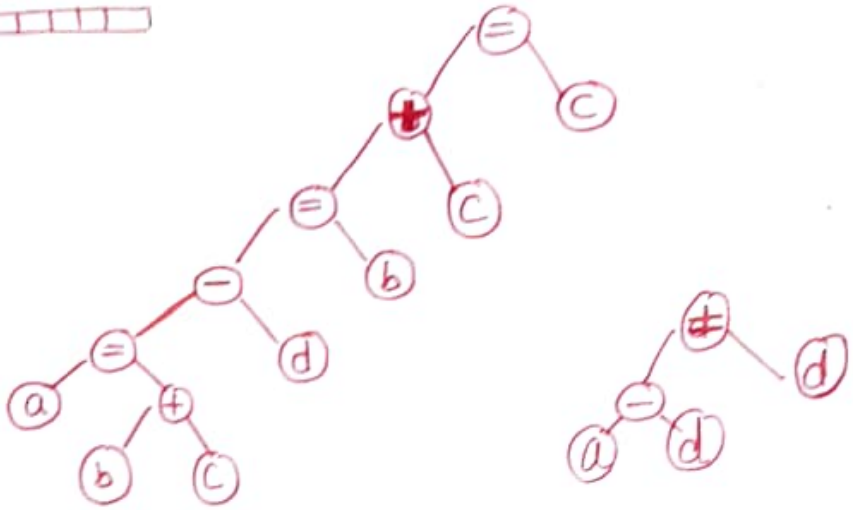
$t_1 = b + c$
$a = t_1$
$t_2 = a - d$
$b = t_2$
$t_3 = b + c$

$c = t_3$

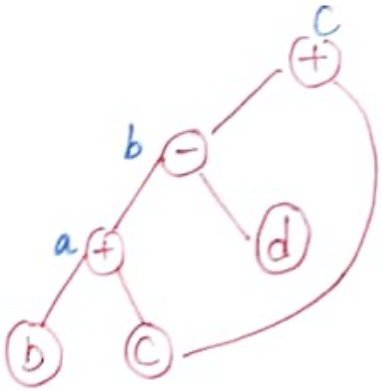
$t_4 = a - d$

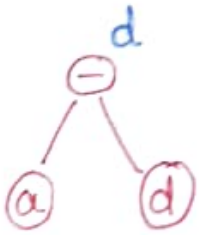
$d = t_4$

Syntax Tree



DAG :-





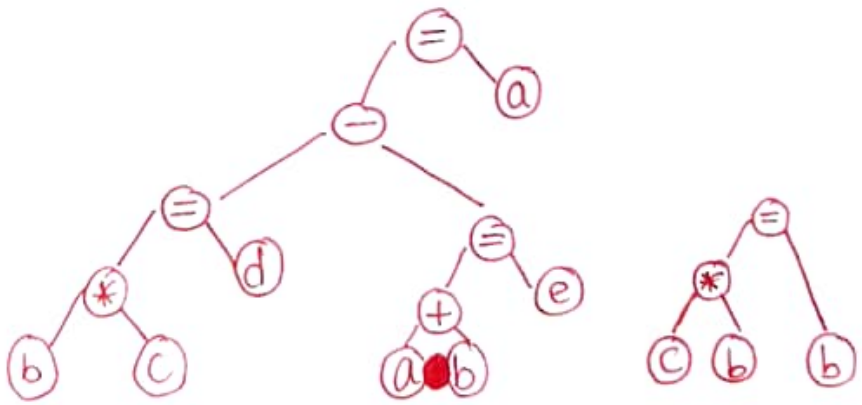
Expression :-

$d = b * c$
$e = a + b$
$b = b * c$
$a = e - d$

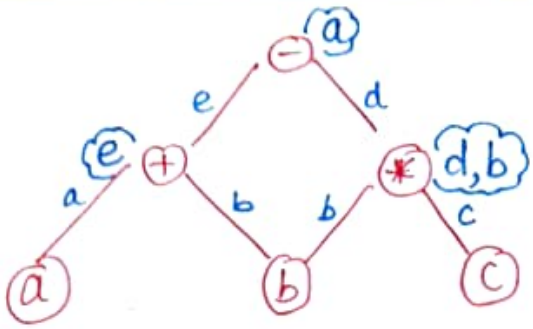
Three address code :-

1) $t_1 = b * c$
2) $d = t_1$
3) $t_2 = a + b$
4) $e = t_2$
5) $t_3 = b * c$
6) $b = t_3$
7) $t_4 = e - d$
8) $a = t_4$

syntax Tree :-



DAG



⇒ Try to keep the no. of times node minimum

⇒ reuse the node for $d = b*c$, $b = b*c$

Also maintain minimum edges.

---

given Expression :-

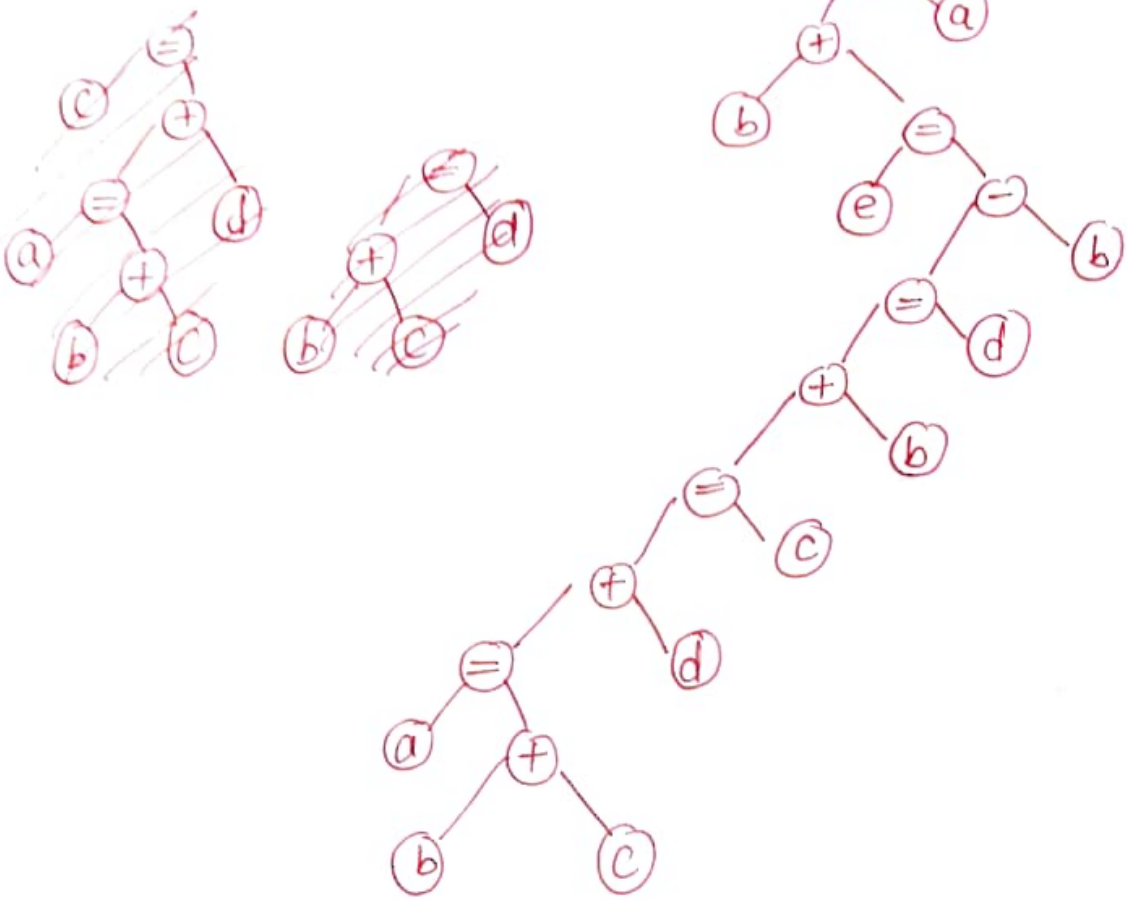
$a = b+c$
$c = a+d$
$d = b+c$
$e = d-b$
$a = e+b$

Design a DAG with minimum no. of nodes and edges.

Three address Code

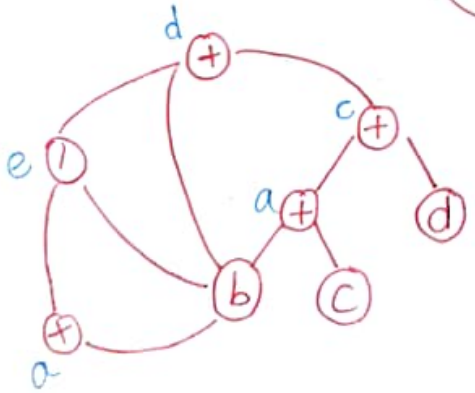
1) $t_1 = b+c$ ①
2) $a = t_1$ ②
3) $t_2 = b+c$ ⑤
4) $d = t_2$ ⑥
5) $t_3 = a+d$ ③
6) $c = t_3$ ④
7) $t_4 = d-b$ ⑦
8) $e = t_4$ ⑧
9) $t_5 = e+b$ ⑨
10) $a = t_5$ ⑩

Syntax tree :-



DAG :-

nodes = 8
edges = 10
→ not minimum



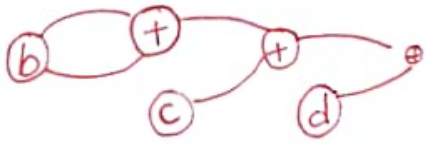
To obtain a DAG from the Expression with minimum nodes and edges :- last expression need to minimised with all the updated variables of expressions above it then the minimised last expression must be converted into DAG.

(i) $a = b+c$
(ii) $c = a+d$
(iii) $d = b+c$
(iv) $e = d-b$
(v) $a = e+b$.

<Ans>

$a = e+b$ ⇒ putting the value of $e = d-b$ from eq (iv)
$a = d-b+b$
$a = d$ ⇒ putting $d = b+c$ from eq (iii)
$a = b+c$ ⇒ putting the value of $c = a+d$ from eq (ii)
$a = b+a+d$ ⇒ putting the value of $a = b+c$ from eq (i)
$a = b+b+c+d$

↳ Converting this into DAG ⇒



nodes = 6
edges = 6

Code Optimization

⇒ Optiming the Intermediate Code is called Code optimization.

⇒ For e.g :- Reducing the no. of lines in the Three address Code.
Removing Extra temporary variables in TAC.

Code Optimization

- machine Independent
- Machine Dependent

Machine Independent Code Optimization :-

1) Loop optimization :- Reducing the no. of times the all the statements are executing with the help of loop, while keeping the logic and final output exactly same.
   - → Code motion or frequency Reduction
   - → Loop unrolling
   - → Loop janming
2) folding
3) Redundancy elimination
4) Strength Reduction
5) Algebric Simplification

Machine dependent Code optimization :-

- Register Allocation
- use of Addressing Modes
- peephole optimization
  - Redundant Load/Store
  - Flow of Control optimizations
  - Use of machine idioms

Loop optimization :-

1) To apply loop optimization, we must first detect loops.
2) For detecting Loops we use control Flow analysis using program Flow graph.
3) To find PFG, we need we to find Basic Blocks.
4) A basic block is a sequence of 3-address statements where Control enters at the beginning and leaves only at at the end without any jumps or halts.
5) If the Program flow graph formed of Basic blocks, Contains loop or cycles then the Loops are detected.

Basic Blocks :-

⇒ In order to find a basic blocks, we need to find leaders in the three-address Code.

⇒ A basic block will start from one leader to next leader but not including next leader.

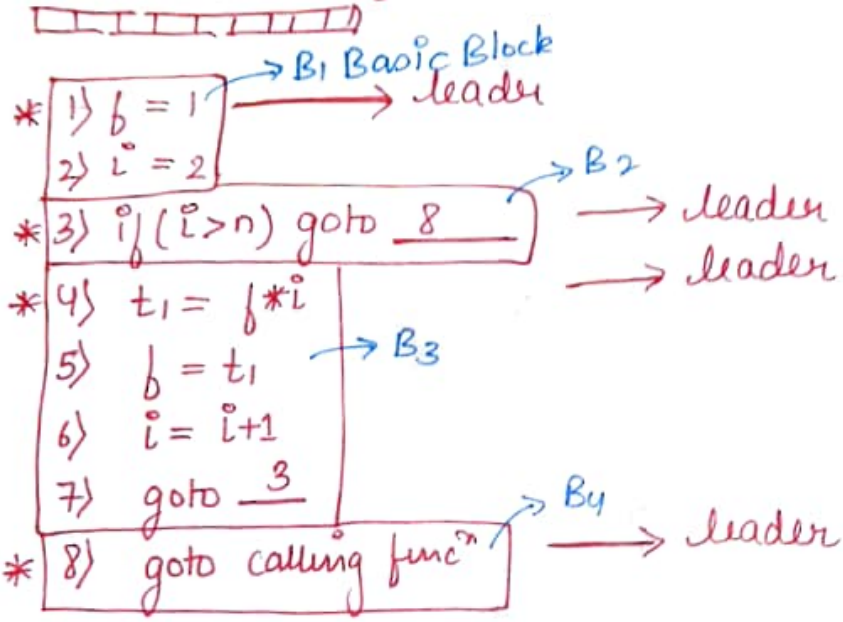
Finding Leaders in a basic block :-

- The first instruction in the three address code in the intermidiate Code is a leader.
- Any instruction that is is the target of Conditional and unconditional jump is a leader.
- An Instruction that immediately follows a Conditional or unconditional jump is a leader.

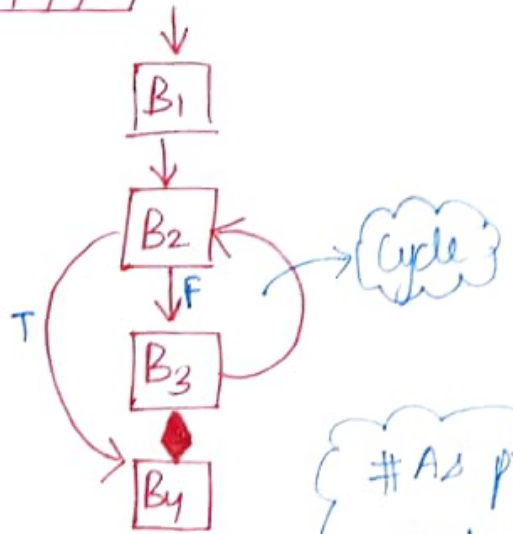
For eg :-

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

Three address code :-



Program flow graph :-

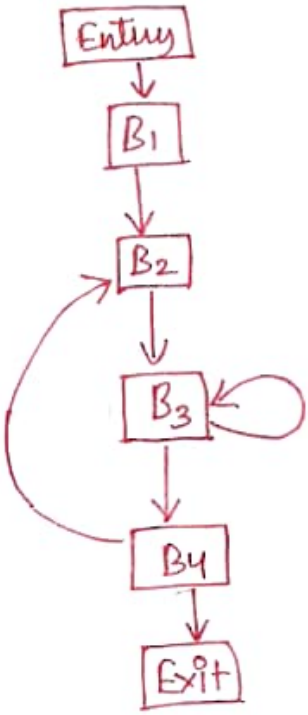


As program graph has cycle ⇒ loop exist in the TAC given

Example TAC :-

1) $i = 1$ → B_1
2) $j = 1$ → B_2
3) $t_1 = 5 * i$ → B_3
4) $t_2 = t_1 + j$
5) $t_3 = 4 * t_2$
6) $t_4 = t_3$
7) $a[t_4] = -1$
8) $j = j + 1$
9) if ($j <= 5$) goto 3
10) $i = i + 1$ → B_4
11) if ($i < 5$) goto 2

Program graph :-



Nodes = 6
edges = 5

Loop Optimization :- Once a loop is detected in the Code, then, optimizations techniques can be applied in the following ways :-

b) Frequency Reduction :- A statement or expression that can be moved outside the loop body without affecting the semantic/output of the Program.

Example :-

```
i=0
while (i<1000){
    A = (a/b) + i;
    i++;
}
```

→ optimization →

```
i=0
t=a/b
while(i<1000){
    A = t+i;
    i++;
}
```

This is also called Code motion :- moving some part of the code from inside the loop to outside to reduce the total number of instructions executed inside the loop.

The code inside the loop is high frequency code as it is executed more than the code outside. The code outside the loop is called low frequency code. Moving the code from high frequency region to low region is called frequency reduction.

(b) Loop unrolling :- Reducing the no. of times comparisions are made in the loop, without changing the logic or output.

Example :-

```
for(i=0; i<10; i++)
{
  Printf("hi");
}
```

} Printing 'hi' 10 times in the loop

↓ optimization

```
for(i=0; i<10; i=i+2){
    Printf("hi");
    Printf("hi");
}
```

} Printing 'hi' 10 times in the Loop.

(c) Loop Jamming :- Combine the bodies of two loops to again reduce the no. of instructions executed without changing the output of the code overall.

Example :-

```
for(i=0; i<5; i++){
    a = i+5;
}
for(i=0; i<5; i++){
    b = i+10;
}
```

↓ optimization

```
for(i=0; i<5; i++){
    a = i+5;
    b = i+10;
}
```

Peep-hole optimization :-

⇒ A technique which is applied on a small set of instructions. (peephole or window).

(a) Redundant LOAD/STORE elimination :-

For e.g: Two Code Expressions :-

```
a = b+c
d = a+e
```

Register language :-

```
LOAD  R0, b
ADD   R0, c
STORE a, R0
LOAD  R0, a
Add   R0, e
STORE d, R0
```

(STORE a, R0 / LOAD R0, a — Redundant)

optimization →

```
LOAD  R0, b
ADD   R0, c
ADD   R0, e
STORE d, R0
```

(b) Flow/Control optimization :-

1) Avoid Jumps on Jumps :-

L_1 : jump L_2

L_2 : jump L_3

L_3 : jump L_4

L_5 : jump L_4

L_4 : jump L_6

L_6 : $x = a+b*c$

→ Eliminate redundant jumps

L_1 : jump L_6

L_6 : $x = a+b*c$

rest Code is Eliminated

(a) Eliminate dead Code :-

```
int i=0;
if (i==1){
    printf("hi");
}
```

dead Code ← can be eliminated as it would never execute

(c) Use Machine idioms :-

Expression → i = i+1;

Indigineous/traditional

```
LOAD R0, i
ADD  R0, 1
Store i, R0
```

optm →

using machine

```
INC i
```

using predefined machine idioms can Reduce the Code's execution Complexity.

Constant Folding :- A machine Independent code optimization technique.

① Replacing an expression that can be Computed at Compile time by its value.

Examples (i) $C = 2*3.14*r \xrightarrow{opt^m} C = 6.28*r$

(ii) $x = 2+3+B+C \xrightarrow{opt^m} x = 5+B+C$

(II) Redundancy Elimination :- Useless / No repetion of variables.

Examples :-

(i) $a = b+c$, $e = d+b+c$ $\Longrightarrow$ $a = b+c$, $e = d+a$ → optimised

(III) Strength Reduction :- Replacing a expensive operator by cheaper operator.

Expensive		Cheaper
x^2	→	$x*x$
$2*x$	→	$x+x$
$x/2$	→	$x*0.5$
$B = A*2$	→	$B = A<<1$

(IV) Algebraic Simplication :- Replace a Complex Algebraic expression with a simpler Algebraic expression of same meaning.

For e.g :- (i) $x+0 = (x) = 0+x$

(ii) $x-0 = (0-x) = (x)$

(iii) $x*1 = 1*x = (x)$

(iv) $x/1 = (x)$