```
21.  a[t10] = x
22.  goto (5)
23.  t11 = 4 * i
24.  x = a[t11]
25.  t12 = 4 * i
26   t3 = 4 * n
27.  t14 = a[t13]
28.  a[t12] = t14
29.  t15 = 4 * n
30.  a[t15] = x
```

```
B1
i = m-1
j = n
t1 = 4*n
v = a[t1]

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

B3
j = j-1
t4 = 4*j
t5 = a[t4]
if t5 > v goto B3

B4
if i >= j goto B6

B5:
t6 = 4*i
x = a[t6]
t7 = 4*j
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2

B6
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

Code Optimization

Principle Sources of Optimization

- A transformation of a program is called local if it can be performed by looking only at the statements, in basic block, otherwise it is called global.
- Many transformation can be perfomed at both local & global level.
- Usully local transformations are perfomed first.

Functional Preserving Transformations

ⓐ Common sub expression elemination
ⓑ Copy propagation
ⓒ Dead code elimination
ⓓ Constant folding

Loop Optimization

ⓐ Code motion
ⓑ Induction variable elemination
ⓒ Reduction in strength.

1. Functional Preserving Transforms

~ These are the optimizing methods that optimizes code but preserves what the function comutes.

*1. Common Subexpression eldelemination (CSE)

~ We can eleminate subexpression which are redundant.

~ If E is an expression and En is assigned with E if and only if the later has not been altered in between.

~ Common Sub expressions are expressions whose values are computed already.

eg:-

Consider the expressions.

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

We observe, that both a & c are equal to b+c but we cannot write c = a since in b/w those two statements 'b' value gets changed

~ At the same time both b & d are equal to a-d and d can be written as d = b since neither a's nor d's value is altern when we traverse from b to d.

∴ The optimized block will be

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

eg:- Consider the B5 block of quick sort (Pg: 45)

B5		B5
t6 := 4*j		t6 := 4*i
x := a[t6]		x := a[t6]
t7 := 4*i	can be ⟹ optimized as	t8 := 4*j
t8 := 4*j		t9 := a[t8]
t9 := a[t8]		a[t6] := t9
a[t7] := t9		a[t8] := x
t10 := 4*j		goto B2
a[t10] := x		
goto B2		

*2. Copy Propogation

~ Assignmets of the form f := g is called copy statements, or copies for short.

eg:-

~ When statemls like B=A; C=B; D=C;
are written out without changin the values in betwem, then it can be optimized as:

B := A;	⟹	B := A ;	⟹	B := A
C := B;		C := A ;		C := A
D := C;		D := C;		D := A

~ The provide a potential platform to eliminate common sub expressions.

*3. Dead Code Elemination

~ A dead part of the code is, which never gets executed or the outcome of that part is never used.

eg:-

```
def fund (a, b):
    P = a+b
    return (a+b)
    print("sum")
```

never gets used

Never gets executed

eg:-

a = 1

$$\left.\begin{array}{l} \text{if } (a < 0) \\ \quad a = 0; \end{array}\right] \rightarrow \text{daad part} \quad \Rightarrow \quad a = 1 \rightarrow \text{optimized code.}$$

***4. Constant Folding**

~ If expression contains constat separated by operators which can be evaluated ad-hoc, then it is done so.

eg:- $a = 4 + 7 \xrightarrow[\text{optimized code}]{} a = 11$

~ This eliminate the overhead of performing that operation during run time/execution.

> Now according to use the are some othe optimizing techniques such as:

i) Renaming temporary variables.

ii) Inter change of statements

iii) Algebraic transformations

#2. Loop Optimization

~ The running time of a program may be improved if the number of instruction in an inner loop is decreased.

*1. Code Motion & Loop Invariant Computations

~ It is the approach which moves code outside the loop - if it won't have any difference if it executes inside or outsize a loop.

eg:- 1

```
n=10
for i in range(10):
    x = y+z; // redundant
    a[i] = 6*i
```

=>

```
n = 10
for i in range(10):
    a[i] = 6*i
x = y+z
```

eg:- 2

```
a,b,c = 10,20,30
for i in range(5):
    c = a+b
    d = a-b      } => loop invariant computation does not depend upon the loop
    e = a+b
    s *= i
```

=>

```
a,b,c = 10,20,30
c = a+b
d = a-b
e = a+b
for i in range(5):
    s *= i
```

*2. Induction Variable & Reduction in Strength.

~ Consider the loop.

```
B3 : j = j - 1
     t4 = 4 * j
     t5 = a[t4]
     if t5 > v goto B3
```

~ Every time j reduces by one t_4 decreass by 4.

~ Hence j is the induction variable & t_4 is the induced variable.

~ When there are two or more induction variables in a loop then it is possible to get rid of all but one.

~ Again in the above example since multiplication is more costly the subtraction it can be replaced as follows.

```
B3 : j = j - 1
     t4 = t4 - 4
     t5 = a[t4]
     if t5 > v got B3
```

[B] # Optimization of Basic Blocks

- Many of the structure preserving transformation is implented by constructing a DAG (Directed Acyclic Graph) for a block.

- There is a node 'n' associated with each ~~element~~ statement's within the block. The children of n are those nodes corresponding to statements that are the last definitions.

- These are prior to the statements s of the operands used by them. (s).

Basic Block = set of salements exequted in a sequencial manner.

- Every basic block contains one entry point & one exit point.

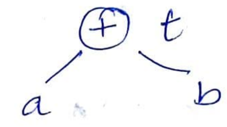
- Within a basic block there are no ~~conditional~~ control statements neither conditional nor ~~uncdt~~ unconditional.

* Properties of a DAG ~~Rules~~

1. Internal nodes in a DAG, represent operators.
2. Leaf node represents identifiers, constants
3. Internal node may also represent ~~identifier~~ result of expression.

eg:-

t = a+b



* Application of DAG

1. Determine the common sub expressions.
2. Determines which names are inside the block & are computed outside the block.
3. Helps in determining which statement of the block could have their values computed outside the block.
4. Simplying the list of quadraples by eleminating the common sub expressions.

* Rules for the Construction of a DAG

1. In a DAG leaf node represent identifiers

names, constants. Interior node represents operators.

2. While constructing DAG, there is a check made to find if there is an existing node with same children.

~ A new node is created only when such a node does not exists.

~ This helps to detect common sub expression and eliminate the same.

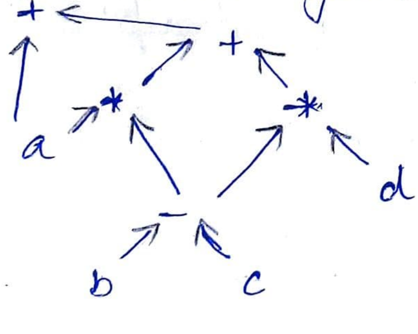
3. Assignment of the form $x = y$ must not be performed until unless it is a must.

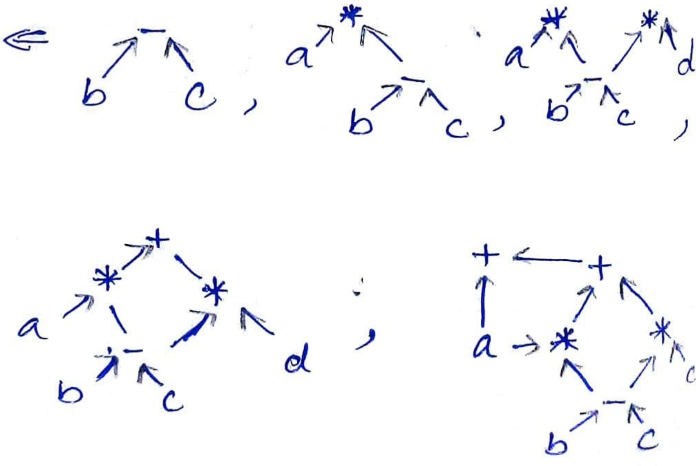
eg:-

1. Construct the DAG for the expression:

$a + a * (b - c) + (b - c) * d$

(paranthesis has highest priority)

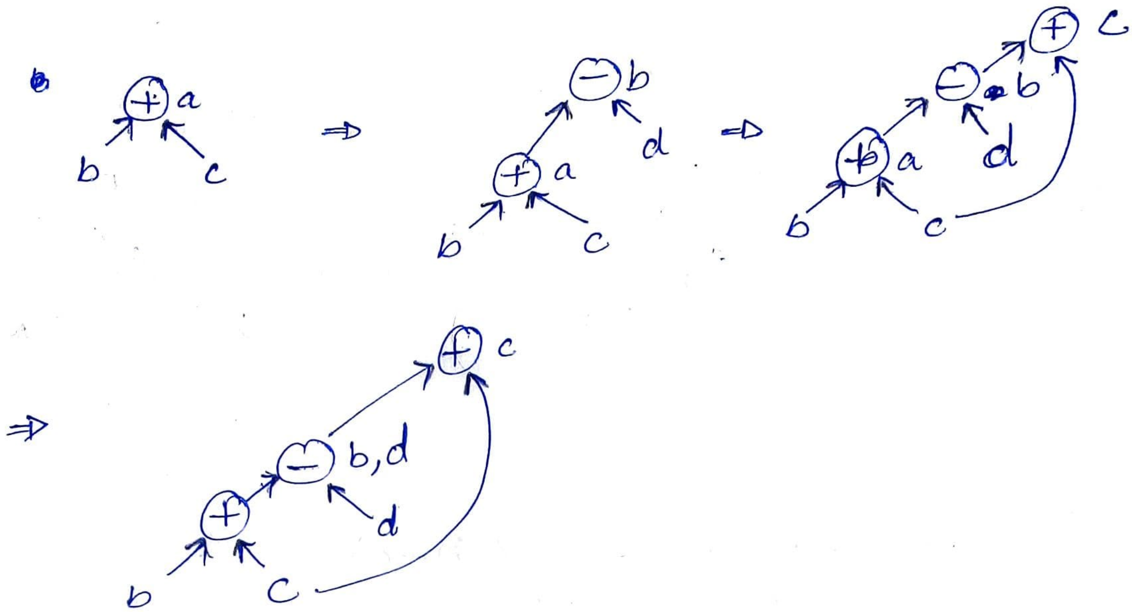




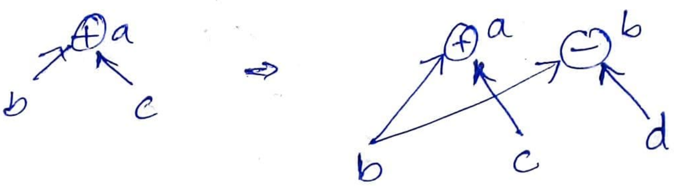
2. Construct DAG, for the block.

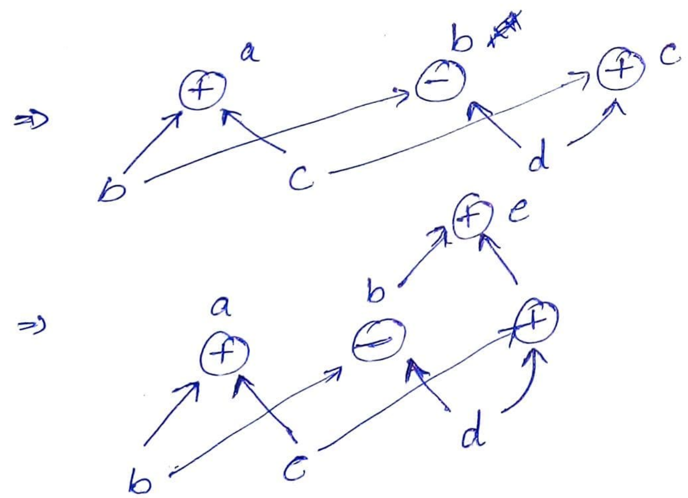
1. $a = b + c$ 3. $c = b + c$

2. $b = a - d$ 4. $d = a - d$



3. 1. $a = b + c$
2. $b = b - d$
3. $c = c + d$
4. $e = b + c$



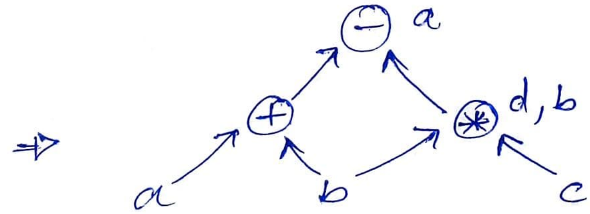


4. $d = b * c$

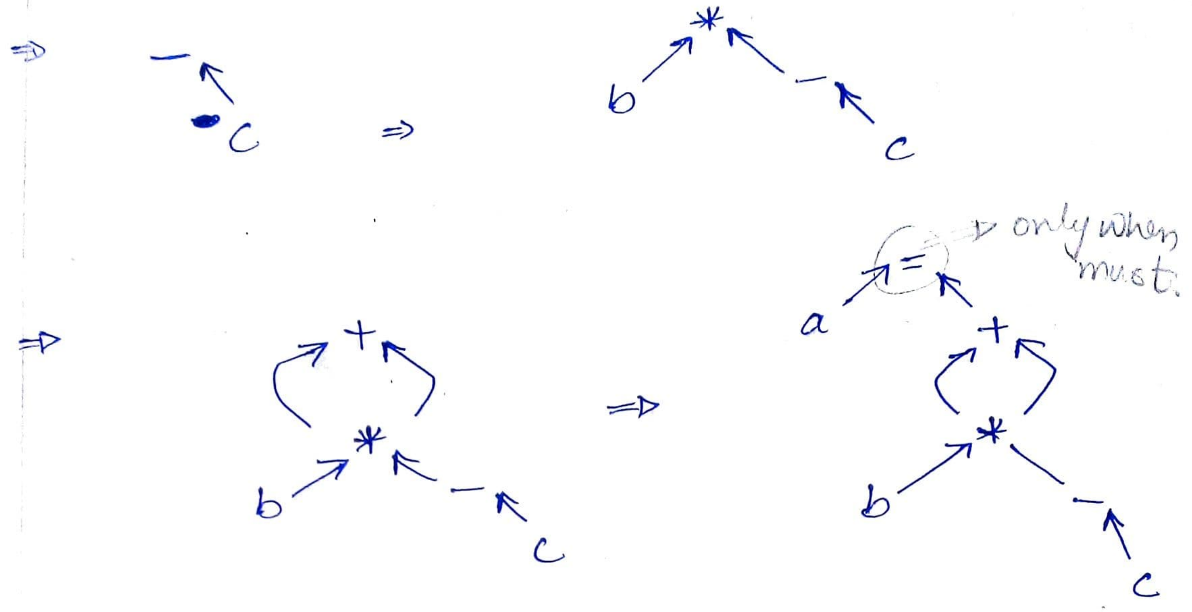
$e = a + b$

$b = b * c$

$a = e - d$



5. $a = b * -c + b * -c$



6. $a = (a * b + c) - (a * b - c)$