UNIT-V CONTROL FLOW & DATA FLOW ANALYSIS

Flow graph – It is a directed graph in which the flow control info is added to the basic block.

→ The nodes to the flow graph are represented by basic block.

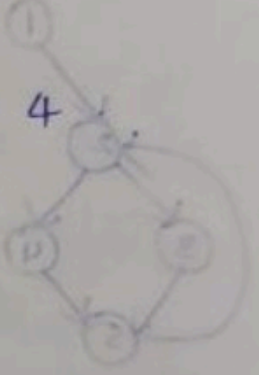
→ The block whose leader is the first stmt is called initial ~~stmt~~. block

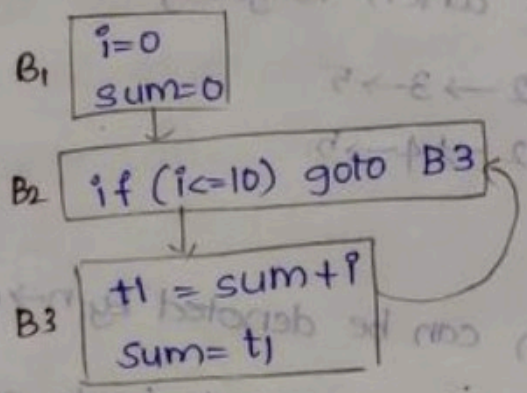
Ex:
```
i=0
sum=0
while (i<=10)
{ sum=sum+i;
}
```

⇒

3. Add code:

1) i=0
2) sum=0
3) if (i<=10) goto 4
4) t1 = sum+i
5) sum = t1

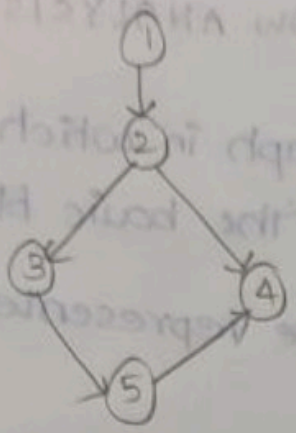




Loop: It is a collection of nodes in the flow graph.

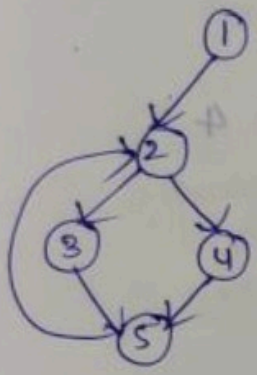
1) All such nodes are strongly connected that means there is path from node to any other node within that loop
2) The collection of nodes has unique entry that means there is only one path. One path from a node outside the loop to the node inside the loop.
3) The loop that contains no other loops is called

Ex:



Loops in a flow Graph:

1) Dominators



→ Node 1 is the initial node & it dominates every node as it is a initial node.

→ Node 2 dominates 3, 4, 5 as there exists only one path from initial node to node 2 which is going through the

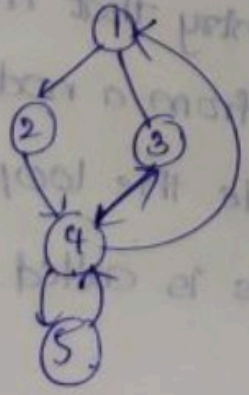
$1 \rightarrow 2 \rightarrow 3 \rightarrow 5$

$1 \rightarrow 2 \rightarrow 4 \rightarrow 5.$

2) Natural ~~Nodes~~ loops

→ Loop is a flow graph can be denoted by $n \rightarrow d$ such that d dominates n these edges are called back edges

→ If there is $p \rightarrow q \Rightarrow$ where p is the tail & q is head. head dominates tail.



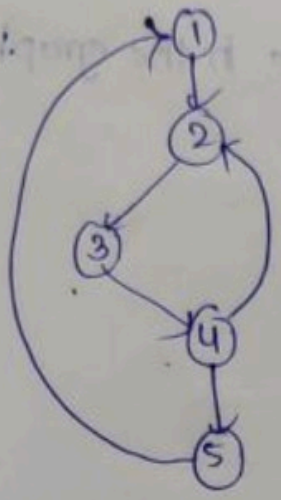
$4 \rightarrow 1$

1 dominates 4

3) Inner loop

→ It is a loop that contains no other loop.

ex:



The inner loop is 4→2

The edges are given by

2→3 3→4

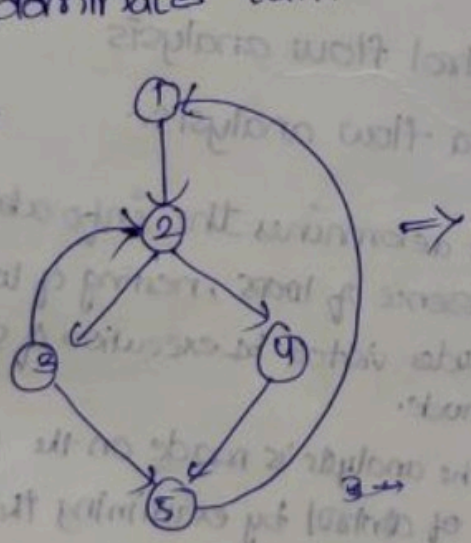
4) Reducable flow graph

→ It is a flow graph contains 2 edges. One is forward edge & the other is backward edge.

→ forward edges form an acyclic graph

→ Backward edges are such edges whose heads dominates tail.

ex:

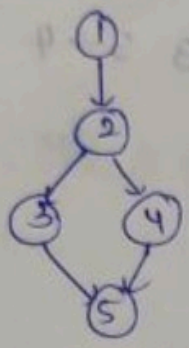


→ The above flow graph is reducable. We can reduce the graph by removing the backward edges 3→2 edge. Similarly removing the backward edge 5→1.

Non-reducable flow graph:

→ It is a flow graph in which there are no backward edges.

→ forward edges may produce cycle in the graph.



Global optimization:

→ It is a program in which each node is represented in the form of flow graph.

→ flow graph is a graphical representation in which each node represents the basic block & edges represents the flow of control from one block to another.

→ There are two types 1) Control flow analysis
2) Data flow analysis.

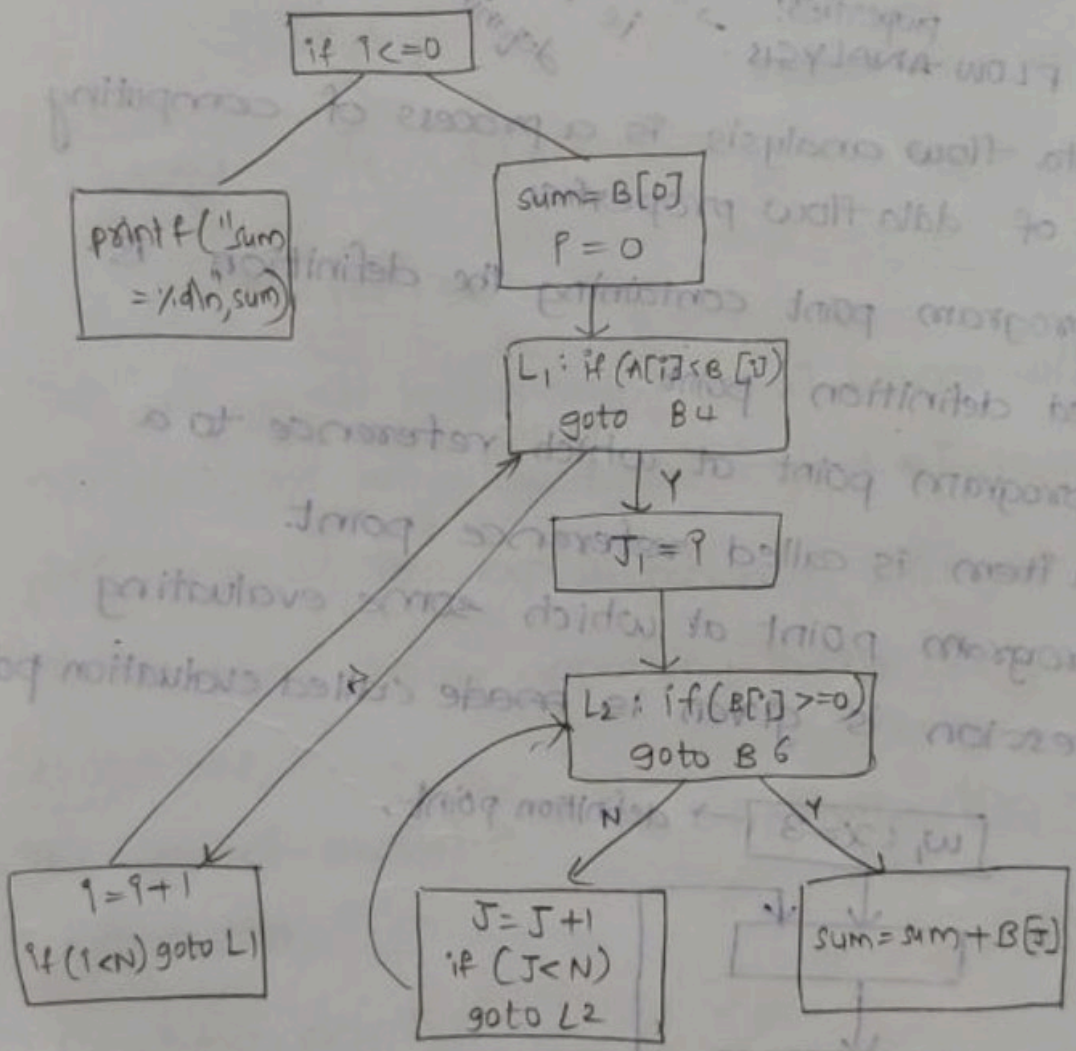
1) Control flow analysis:

→ It determines the info about presence of loops, nesting of loops & nodes visited by execution of specific node.

→ The analysis is made on the flow of control by examining the program flow graph.

Ex.
```
if (i<=10)
{
    sum = B[0];
    i=0;
L1: if (A[i]<B[i])
    { J1 = i;
    if (B[i]>=0)
        { sum=sum+ B[J];
        }
    J=J+1;
```

```
    if (J<N) goto L2;
  }
  I = I+1
  if (I<N) goto L1;
}
printf("sum = %d\n", sum);
}
```



2.Ex: L: a = K+2
c = d-b
d = a+b
if (d>I) goto E
f = b-d
K = d-2
b = a+f

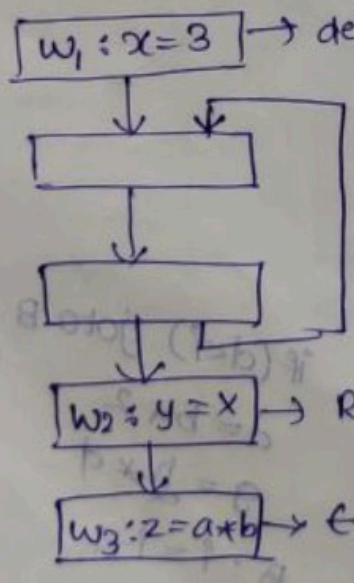
if (d<I) goto B
d = b*2
g = b*d
B: I = I+1
b = d-1
goto L
E: K = a-e
f = e+K
c = d+b

2) DATA FLOW ~~ANALYSIS~~ properties: → analysis is made on the data flow. ie it determines the info regarding the definition & use of the data.

→ The data-flow analysis is a process of computing values of data-flow properties

1. A program point containing the definition is called definition point
2. A program point at which reference to a data item is called reference point.
3. A program point at which some evaluating expression is given is ~~mode~~ called evaluation point

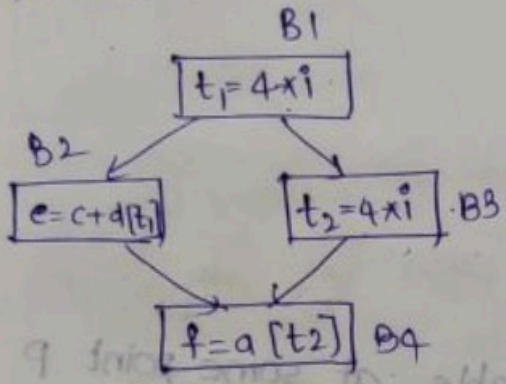
Ex:



PROPERTIES

1) Available Evaluable exp
2) Reaching definition.
3) Live variables
4) Busy exp

(1) Available Exp



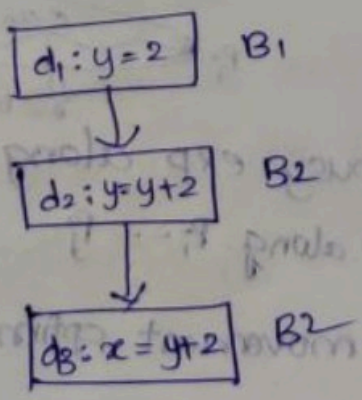
1. Exp is available at it's evaluation pt.
2. The exp can not be modified before their use.

That exp should be appearing in all blocks after the modifications are allowed. The exp $4 * i$ is available exp because this exp is not being changed by any of the block before appearing in B4 block

ADV:

→ The use of availabk is to eliminate the common sub expression.

(2) Reaching definition.



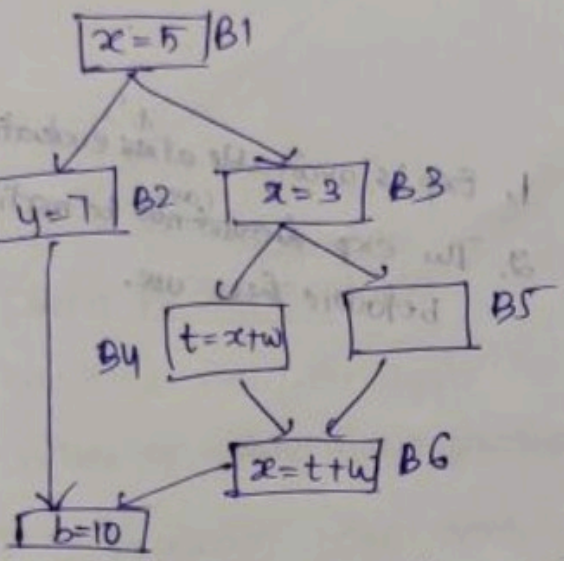
The definition d_1 is said to a reaching definition for block B2. But, the def^n d_1 is not a reaching def^n of block B3 becoz it is killed by def^n d_2 in B2

ADV

→ It is used in constant & variable propogation.

(3) Live variables:

→



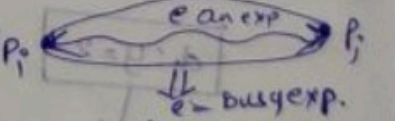
→ The variable x=5 is live variable at some point P if there is a path from P to exit along th which the value of x is used before it is redefined. Otherwise the variable is said to be dead at that point.

→ x is live at Block B1, B3, B4, B6 but killed at B6.

ADV

→ It is used for dead code elimination.

(4)

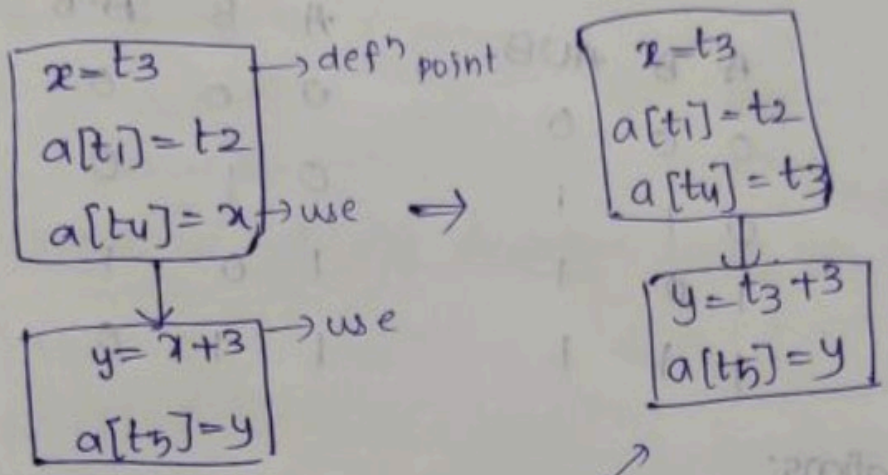


~~Your~~ An exp is said to be a busy exp along some path Pi to Pj iff e is evaluated along Pi --- Pj

→ Busy exp are useful in code movement optimization.

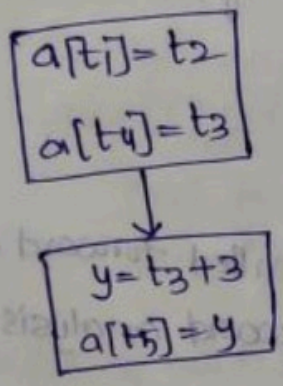
REDUNDANT COMMON SUB-EXPRESSION

*) Copy propogation. $a := b$ → copy statement.



→ we will replace x by t_3

→ eliminating the copy statement finally we get



*) Iterative dataflow Analysis.

→ we can perform a data-flow analysis by computing data-flow equations using

1) Available exp
2) Reaching Def^n
3) Live variable analysis
4) Logical AND (or) Logical OR
5) Difference.

4)

logical AND

A	B	A∩B
0	0	0
0	1	0
1	0	0
1	1	1

logical OR

A	B	A∪B
0	0	0
0	1	1
1	0	1
1	1	1

Difference.

A	B	A-B
0	0	0
0	1	0
1	0	1
1	1	0

Data flow equations:

→ Equations representing the expressions that are appearing in the program. These equations used in computing live variables, available expressions and the reaching definition.

→ Data flow equations are 2 types

1. Forward
2. Backward

→ Available expression & reaching def^n is called forward analysis

→ live variable analysis is called backward analysis

→ Out[s] = gen[s] U [in[s] - kill[s]]

→ s represents the statement for which the data flow eqn can be written. gen represents the information generated within the statement. In represents the gathered information before entering loop. kill represents the information that is killed within the controlled flow.

Live variable analysis:

→ Data flow equations are

use[S] → represents set of variables used by S

def[S] → " " " " defined by S.

→ every stmt uses some set of variables (read from them) & defines some set of variables (writes to them)

Ex: $x = y + z$

def = x
defined
use value = y, z

$x = x + 1$

use = x
def = x.

Algorithm for data flow equations using live variable analysis

in[i] = ∅
out[i] = ∅

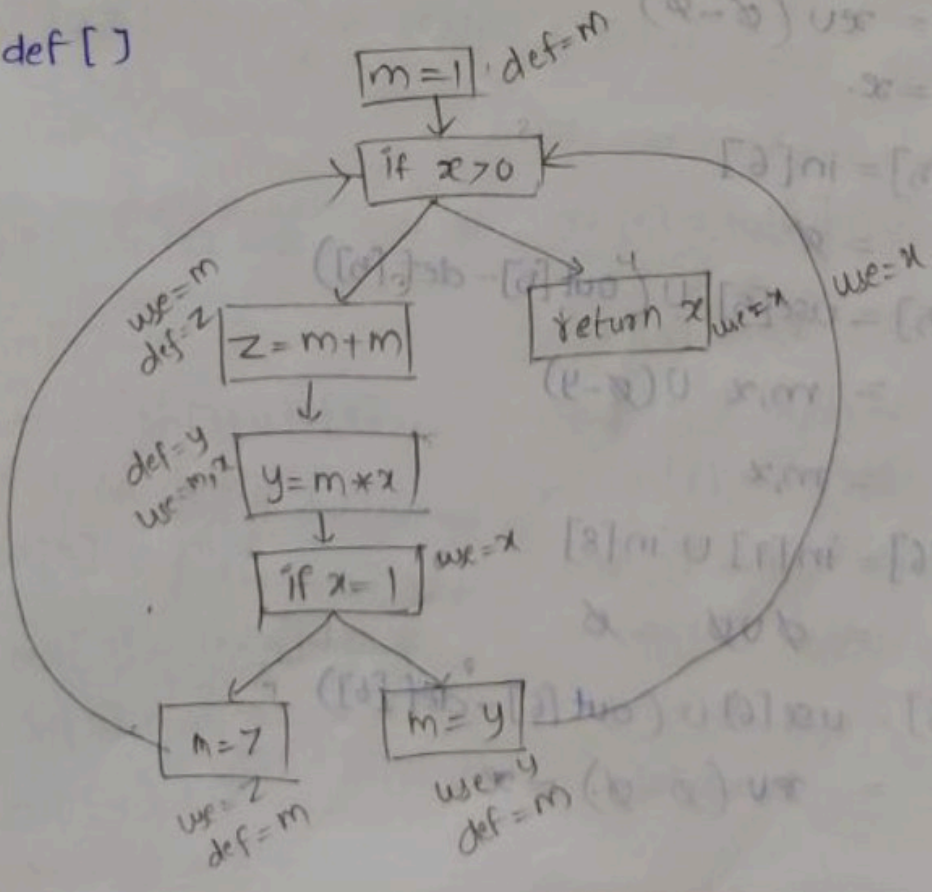
Repeat until no change for all i.

$$out[i] = \bigcup_{i \in succ[i]} in[i]$$

$$in[i] = use[i] \cup (out[i] - def[i])$$

End for
End Repeat.

Ex: in[i] & out[i] for following prog using use[] & def[]



→ first we define use[] & def[] for each block. find for above flow graph.

→ Initially we assume in[i] & out[i] = ∅.

	in[i]	out[i]
1	∅	∅
2	∅	∅
3	∅	∅
4	∅	∅
5	∅	∅
6	∅	∅
7	∅	∅
8	∅	∅

out[2] = in[3] ∪ in[4]
= ∅ ∪ ∅ = ∅.

in[2] = use[2] ∪ (out[2] − def[2]
= x ∪ (∅ − ∅)
= x.

out[3] = in[5] = ∅

in[3] = m ∪ (∅ − z)
in[3] = m.

out[4] = ∅

in[4] = use[4] ∪ (out[4] − def[4])
= x ∪ (∅ − ∅)
= x.

out[5] = in[6]
= ∅

in[5] = use[5] ∪ (out[5] − def[5])
= m, x ∪ (∅ − y)
= m, x

out[6] = in[7] ∪ in[8]
= ∅ ∪ ∅ = ∅

in[6] = use[6] ∪ (out[6] − def[6])
= x ∪ (∅ − ∅) = x.

out[7] = in[2]
= x.

in[7] = use[7] ∪ (out[7] − def[7])
= z ∪ (x − m)
= x, z

out[8] = in[2]
= x

in
out[8] = use[8] ∪ (out[8] − def[8])
= y ∪ (x − m) = x, y.

After 1st pass we get.

	in[i]	out[i]
1	∅	∅
2	{x}	∅
3	{m}	∅
4	{x}	∅
5	{m, x}	∅
6	{x}	∅
7	{x, z}	x
8	{x, y}	x.

out[1] = in[2] = x

in[1] = x

out[2] = in[3] ∪ in[4]
= m, x.

in[2] = m, x

out[3] = in[5] = m, x

in[3] = m, x

out[4] = ∅

in[4] = x.

out[5] = in[6]
= x

in[5] = m, x

out[6] = x, y, z

in[6] = x, y, z

out[7] = m, x

in[7] = x, z

out[8] = m, x

in[8] = x, y

After 2nd pass we get

	in[i]	out[i]
1	x	x
2	m, x	m, x
3	m, x	m, x
4		
5	m, x	x
6	x, y, z	x, y, z
7	x, z	m, x
8	x, y	m, x

→ Since in[4] & out[4] are same as pass 1, so we will not consider.

out[i] = m, x
in[i] = x
out[2] = m, x
in[2] = m, x

~~out[3]~~ - since in[2] & out[2] are same as in pass 1 we will not consider

out[3] = m, x
in[3] = m, x
} we will not consider.

out[5] = x, y, z
in[5] = x, m, z

out[6] = x, y, z
in[6] = x, y, z
} since in[6], in[7], in[8] are not cosidered.

out[7] = m, x
in[7] = x, z

out[8] = m, x
in[8] = x, y

	out[i]	in[i]
(1)	{m,x}	{x}
(5)	{x,y,z}	{x,z,m}

out [1] = m,x
in [1] = x

out [5] = x,y,z
in [5] = x,z,m

} we will not consider.

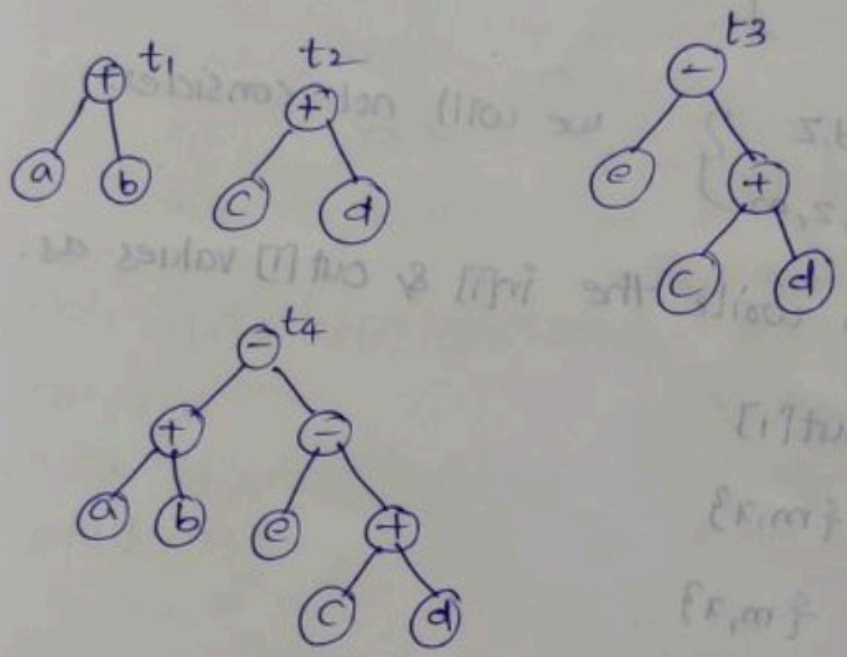
finally we can write the in[i] & out[i] values as.

	in[i]	out[i]
1	{x}	{m,x}
2	{m,x}	{m,x}
3	{m,x}	{m,x}
4	{ϕ}	{x}
5	{m,y,z}	{x,y,z}
6	{x,y,z}	{x,y,z}
7	{x,z}	{m,x}
8	{x,y}	{m,x}

Compute the following 3 address code

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

Consider the DAG as an example. Explain the process of code generation from DAG



Three address code	Target code	Register description
	MOV a, R0	R0 contains a
$t_1 = a + b$	ADD b, R0	R0 contains t_1
	MOV c, R1	R1 contains c
$t_2 = c + d$	ADD d, R1	R1 contains t_2
	MOV e, R2	R2 contains e
$t_3 = e - t_2$	sub R2, R1	R2 contains t_3
$t_4 = t_1 - t_3$	sub R0, R2	R2 contains t_4

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

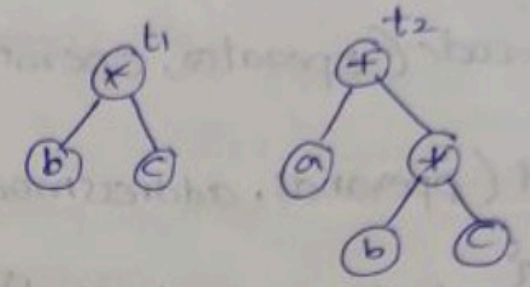
$t_1 = b*c$

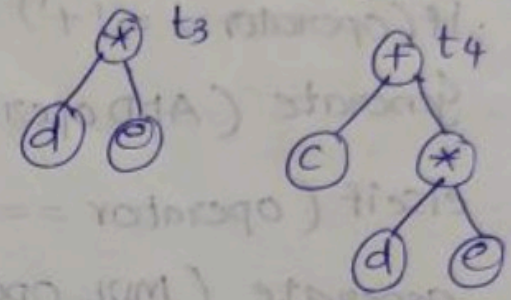
$t_2 = a+t_1$

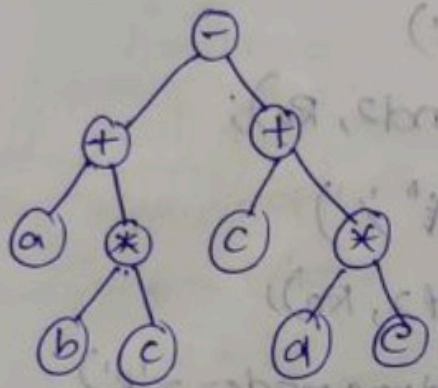
$t_3 = d*e$

$t_4 = c+t_3$

$t_5 = t_2 - t_4$







Three add code	Target code	Register description
$t_1 = b*c$	MOV b, R0	R0 contains b
	MUL c, R0	" " t_1
$t_2 = a+t_1$	MOV a, R1	R1 contains a
	ADD R0 R1	" " t_2
$t_3 = d*e$	MOV d, R2	R2 contains d
	MUL e, R2	" " t_3
$t_4 = c+t_3$	MOV C, R3	R3 contains c
	ADD R2, R3	" " t_4
$t_5 = t_2 - t_4$	SUB R1, R3	R3 contains t_5

Algorithm for code generation:

```
Gen-code(operator, operand1, operand2)
{
  if (operand1.addressmode == 'R')
  {
    If(operator == '+')
    Generate (ADD operand 2, R0);
    elseif (operator == '*')
    generate (MUL operand2, R0);
    elseif (operator == '/')
    generate (DIV operand 2, R0);
  }
  elseif (operand 2. address mode == 'R')
  {
    if (operator == '+')
    generate (ADD operand1, R0);
    elseif (operator == '*')
    generate (MUL operand, R0)
    elseif (operator == '/')
    generate (DIV operand, R0)
  }
else
  { Generate (MOV operand2, R0)
    if (operator == '+')
    generate (ADD operand2, R0);
  dse if (operator == '-')
    generate (SUB operand2, R0);
```

elseif (operator == '*')
generate (MUL operand2, R0);
}

Generate code for following C program using code generation algorithm.

$t_1 = f(a)$
$x = ++t_1$
$t_1 = t_1 + 1$
$x = t_1$

$x = ++f(a)$

Param a
Call f, 1 (1 only 1 parameter)
return t_1
$t_1 = t_1 + 1$
$x = t_1$

CODE GENERATION:

```
MOV a, R0
GOTO 100 /*At location 100 in f(a)*/
MOV AX, R1
ADD #1, R1
MOV R1, X
```

Q) $x = f(a) / g(b,c)$

```
MOV a, R0
GOTO 100 /* At location 100 in f(a)*/
MOV Ax, R1
ADD
```

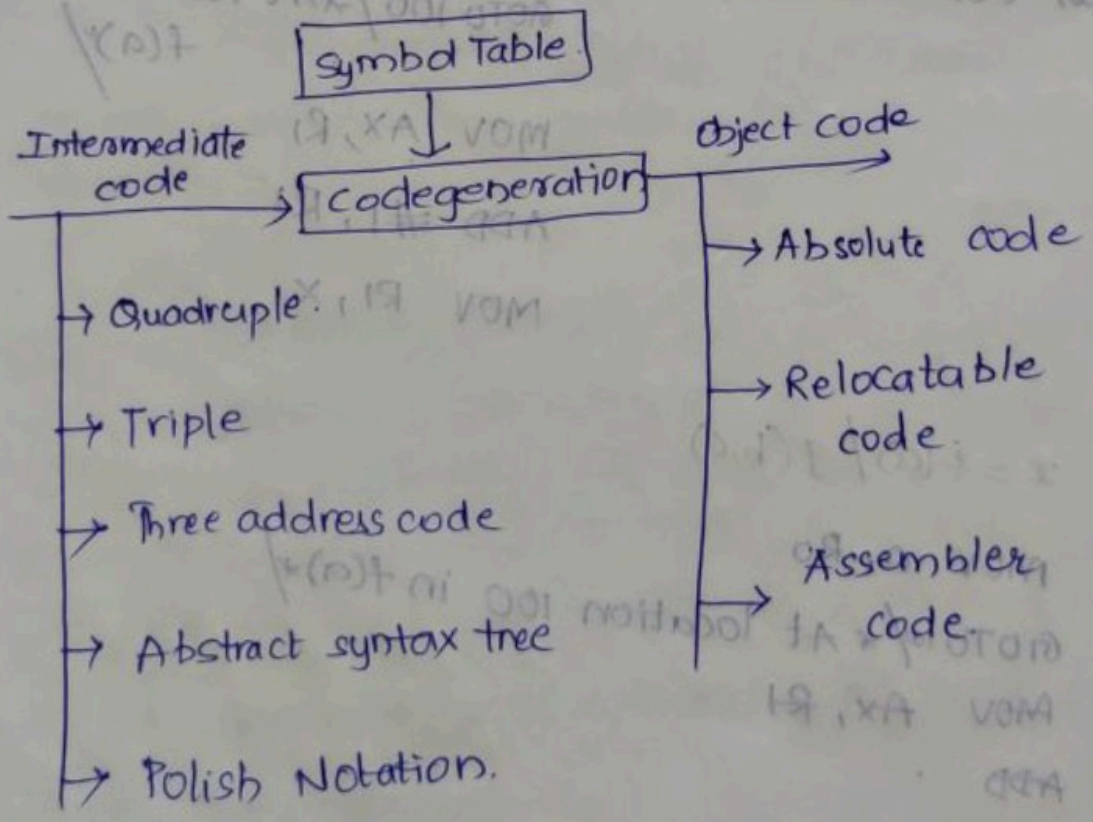
Q) Generate code for $x = f(f(a))$

Param a
Call f,1
return t1
param t1
call f,1
return t2
x = t2

code Gen^n:

```
MOV a, R0     /* Parameter a in R0 */
GoTo 100      /* At location 100 in f(a) */
MOV AX, R1    /* return address of
                 Ax is stored in R1 */
MOV R1, R2    /* take parameter
                 R1 in Register R2 */
GoTo 200      /* At loc 200 in f(f(a)) */
MOV AX, R3    /* return address of
                 Ax is stored in R3 */
MOV R3, X     /* finally x is in R3 */
```

*** Object code forms



→ Object code forms are 3 types

1. Absolute code
2. Relocatable code
3. Assembler code.

1.

→ It is a machine code that contains reference to actual address within program address space.

→ The generated code can be placed directly in the memory & execution starts immediately.

2.

→ producing a relocatable machine language program as o/p allows sub programs to be compiled seperately.

→ A set of relocatable object modules can be linked together & loaded for execution with the help of linker loader

→ High flexibility

3.

→ Producing an assembly language program as o/p makes the process of code generation easier & slower

→ We can generate symbolic instructions & use the macro facilities of the assembler to help in gen? of code.

Register Allocation & Assignment:

→ During register allocation select appropriate set of variables that will reside in registers

Register Assignment:

→ During register assignment pick up the specific register in which corresponding ~~reg~~ variables will reside.

a) $x = a[I] + 1$

```
MOV I, R0
MOV A[R0], R1
ADD R1, #1
MOV R1, x
```

Q) $a[I] = a[I] + b[J]$

```
MOV J, R0
MOV b[R0], R1
MOV I, R2
MOV a[R2], R3
ADD R1, R3
MOV R3, a[R2]
```

Q) $a[I] = b[c[I]]$

```
MOV I, R0
MOV c[R0], R1
MOV b[R1], R2
MOV R2, a[R0]
```

→ Register Allocation & assignment are 4 types

1. Global register allocation
2. Usage count
3. Register assignment for outer loop
4. Graph coloring for register assignment.

1. Global register allocation:

→ It has a strategy of storing the most frequently used variables in fixed registers throughout the loop

→ Another strategy is to des assign some fixed no. of global ~~variables~~ registers to hold the most active values in each inner loop.

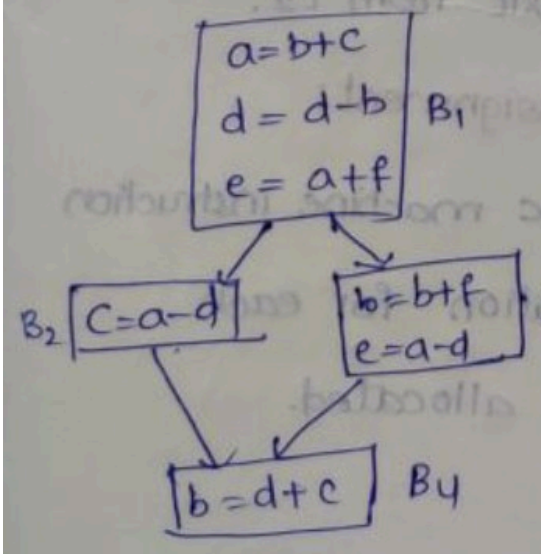
→ The registers not already allocated may be used to hold values local to one block.

→ In certain language like C. can do the register allocation by using register declaration.

2. Usage count:

→ It is the count for the use of some variable in some register used in any basic block. The usage count use the Idea of above how many units of count can be saved by selecting a specific variable for global register allocation.

$$\Sigma(use(X,B) + 2 * live(X,B))$$



→ The usage count for Block B_1 for variable a is

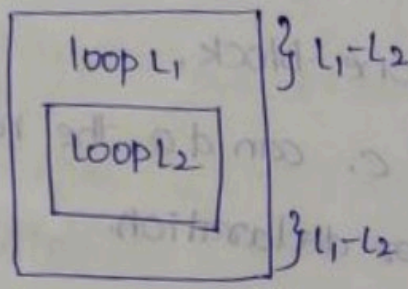
use (a, B) = 0

2 * live (a, B)

2 * 1 = 2

Σ (0+2) = 2.

3. Register assignment for outer loop:



There are 2 loops. loop L_1 & L_2. L_1 is the outer loop & L_2 is the inner loop.

→ If a is allocated in loop L_2 then it should not be allocated in L_1-L_2

→ If a is allocated in L_1 & it is not allocated in L_2 then store in entrance to L_2 & load while leaving L_2.

→ If a is allocated in L_2 & not in L_1 then load a entrance of L_2 & store on exit from L_2.

4. Graph colouring for register assignment:

→ In the first pass, the specific machine instruction is selected for register allocation for each variable a symbolic register is allocated.

→ Graph colouring technique is applied for this register inference graph.

→ The k colour can be assumed to be no. of assignable registers.

→ In graph coloring ~~thec~~ technique no two adjacent nodes can have same colour. Hence in register inference graph using such graph colouring principle each node is assigned to the symbolic registers so that no symbolic registers can interplace with each other with assigned physical register.