chapter - 9

Code Optimization

Need for Code Optimization -

- The code optimization is required to produce an efficient target code.
- There are two important issues that need to be considered while applying the techniques for code optimization & those are.

1) The semantic equivalence of the source program must not be changed.
2) The improvement over the program efficiency must be achieved without changing the algorithm of the program.

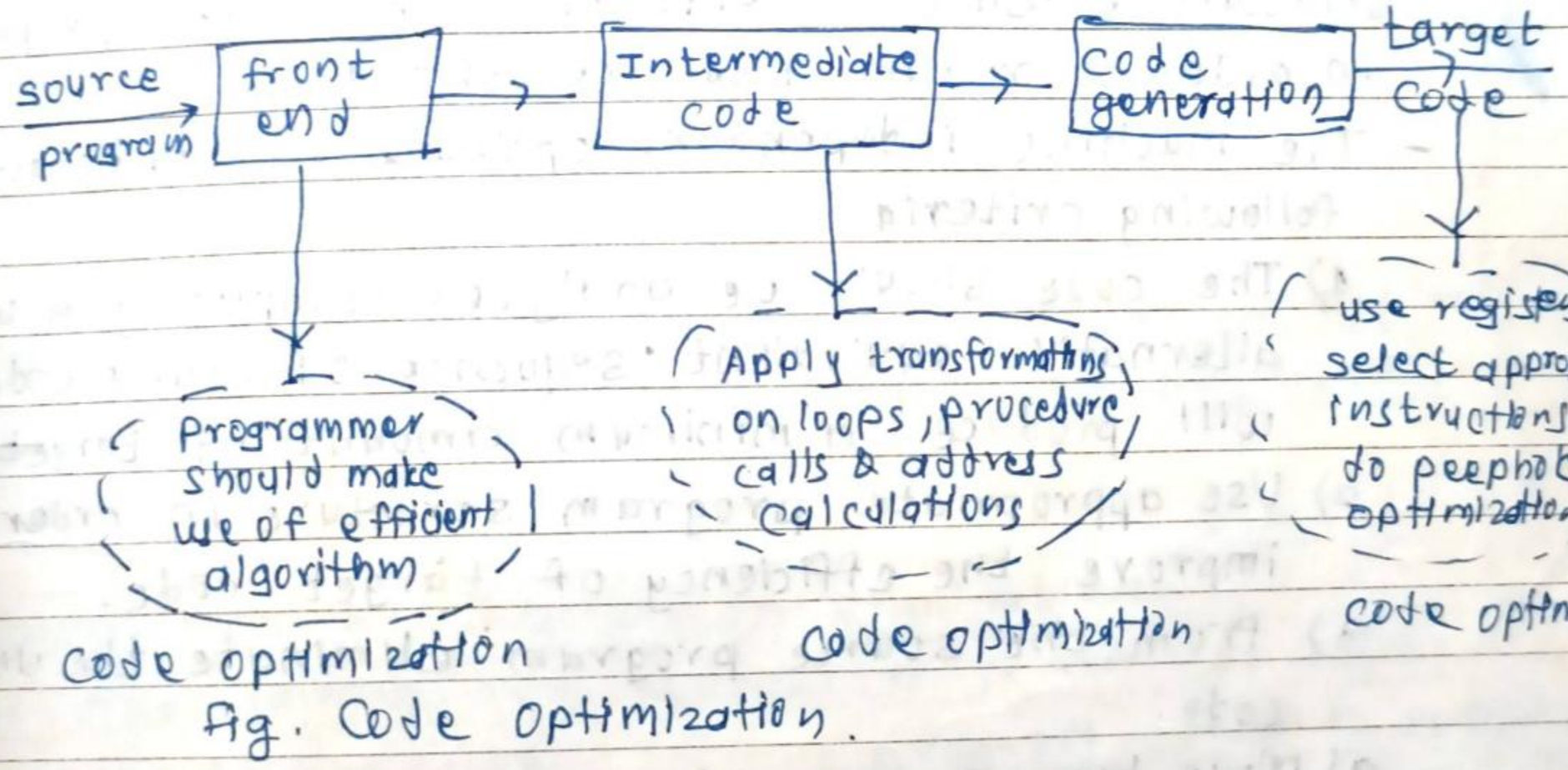
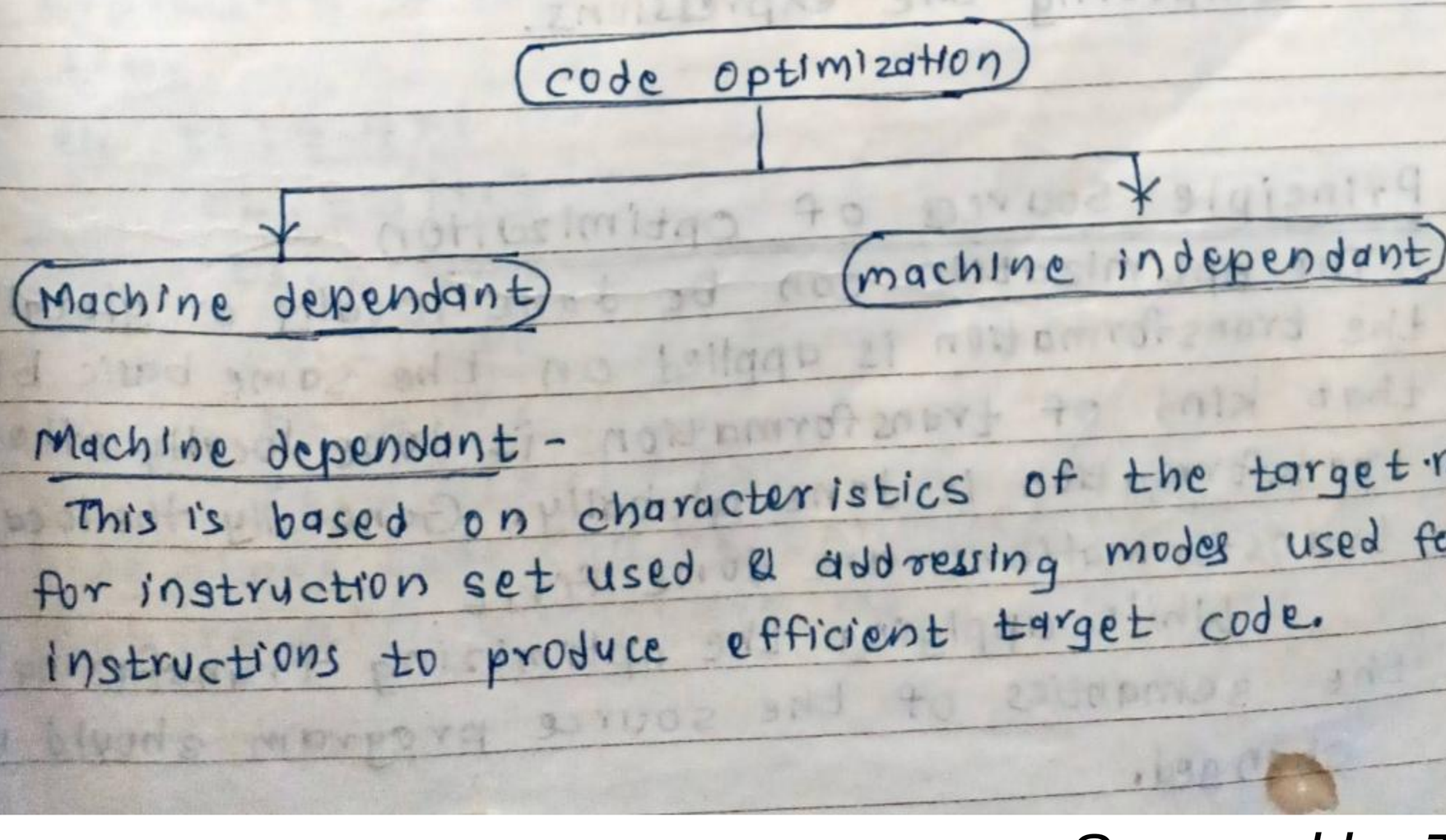


fig. Code Optimization.

Classification of Optimization -



Machine dependant -

This is based on characteristics of the target machine for instruction set used & addressing modes used for the instructions to produce efficient target code.

Machine dependant optimization can be achieved using following criteria.

1) allocation of sufficient number of resources to impr the execution efficiency of the program.
2) Using intermediate instructions wherever necessary.
3) The use of intermix instructions. The intermixing of instruction along with the data increases the spee of execution.

Machine independant optimization

This optimization is based on the characteristics of the programming languages for appropriate programming structure & usage of efficient arithmetic properties in order to reduce the execution time.

- The machine independant optimization can be achieved following criteria

1) The code should be analyzed completely & use alternative equivalent sequence of source code tha will produce a minimum amount of target cod
2) Use appropriate program structure in order to improve the efficiency of target code.
3) From the source program eliminate the unreach code.
4) Move two or more identical computations at one pl & make use of the result instead of each time computing the expressions.

* Principle Sources of optimization -

The optimization can be done locally or globally. I the transformation is applied on the same basic block. that kind of transformation is done locally otherwise transformation is done globally. Generally the local transformations are done first.

While applying the optimizing transformations the semantics of the source program should not b changed.

1) Compile Time Evaluation -

- Compile time evaluation means shifting of computations from run time to compilation. time.
- There are two methods used to obtain the compile time evaluation.

1) folding -

In the folding technique the computation of constant is done at compile time instead of execution time.

ex. length = (22/7) * d

Folding is implied by performing the computation of 22/7 at compile time instead of execution time.

2) constant propogation -

In this technique the value of variable is replaced & computation of expression is done at compile time

ex. pi = 3.14

r = 5

Area = pi * r * r

here, at compilation time the value of pi is replaced by 3.14 & r by 5 then computation of 3.14*5*5 is done during compilation.

2) Common sub Expression Elimination -

The common sub expression is an expression appearing repeatedly in the program which is computed previously. Then if the operands of this sub expression is used instead of recomputing it each time.

ex. t1 := 4*i

t2 := a[t1]

t3 := 4*j

t4 := 4*j

t5 := n;

t6 := b[t4] + t5

The above code can be optimized using the common subexpression elimination as

t1 : = 4 * i
t2 : = a [t1]
t3 : = 4 * j
t5 : = n;
t6 : = b [t3] + t5

The common sub expression t4 : = 4 * i is elimina
its computation is already in t1 & value of i
been changed from definition to use.

3) Variable Propogation -

Variable propogation means use of one variable in
of another.

ex. x = pi
. . . .
area = x * r * r

The optimization wing variable propogation can be done
follows.

area = pi * r * r;

here, the variable x is eliminated.

4) Code Movement -

There are two basic goals of code movement.

1) to reduce the size of the code i.e. to obtain
space complexity.
2) To reduce the freguency of execution of code.
i.e. to obtain the time complexity.

ex. for (i = 0; i <= 10; i++)
{
x = y * 5;
. . . . .
k = (y * 5) + 50;
}

This can be optimized as

```
temp = y * 5;
for(i=0; i<=10; i++)
{
    x = temp
    - - temp
    k = z+50;
}
```

The code for y*5 will be generated only once. And simply the result of that computation is used wherever necessary.

5) Strength Reduction —

- The strength of certain operators is higher than others.
- for instance strength of * is higher than +.
- In strength reduction technique the higher strength operators can be replaced by lower strength operators.

```
ex. for(i=1; i<=50; i++)
    {
        - - - .
        count = i*7;
    }
```

This code can be replaced by using strength reduction as follows.

```
    temp = 7
for(i=1; i<=50; i++)
{
    count = temp;
    temp = temp+7;
}
```

6) Dead Code ellmination -

A variable is said to live in a program if the value contained into it is used subsequently. The variable is said to be dead at a point in a program if the value contained into it is never been used.

ex. i = j ;
. . .
x = i + 10 ;
. . . . .

The optimization can be performed by eliminating assignment statement i=j ; This assignment statement is called dead assignment.

ex. i = 0 ;
if(i = 1)
{
a = x + 5 ;
}

here if statement is a dead code as this condition will never get satisfied hence, if statement can be elliminated & optimization can be done.

7) Loop optimization -

- The code optimization can be significantly done in the part of program.
- Specially inner ~~place~~ loop is a place where program spends large amount of time.
- Hence, if number of instructions are less in inner loop then the running time of the program will be decreased to a large extent.
- Hence, loop otimization is a technique in which code optimization performed on inner loops.
- The loop optimization is carried out by following methods
  1) Code motion
  2) Induction variable & strength reduction
  3) loop invariant method
  4) loop unrolling
  5) loop fusion.

1) Code motion -

- Code motion is a technique which moves the code outside the loop.
- If there lies some expression in the loop whose results remains unchanged even after executing the loop for several times, then such an expression should be placed just before the loop.
- Here, before the loop means at the entry of the loop

ex.

```
while( i<=Max-1)
{
   sum = sum + a[i];
}
```

The above code can be optimized by removing the computation of Max-1 outside the loop.
The optimized code can be.

```
n = Max-1;
while(i<=n)
{
   sum = sum + a[i];
}
```

2) Induction variables & reduction in strength

A variable x is called an induction variable of loop L if the value of variable gets changed every time. It is either decremented or incremented by some constant.

ex.

```
B1 | i := i+1
   | t1 := 4*j
   | t2 = a[t1]
   | if t2 < 10 goto B1
```

In above code the values of i & t1 are in lockstep state. i.e. when value of i gets incremented by 1 then t1 gets incremented by 4. Hence i & t are induction variable.

Reduction in strength -
The strength of certain operators is higher th
others.
ex. strength of * is higher than +. In strength
reduction technique the higher strength operat
can be replaced by lower strength operators.
ex.

```
for(i=1;i<=50;i++)
{
    .....
    count = i * 7;
    .....
}
```

here, we get values of count as 7,14,21 & so on
This code can be replaced by using strength
reduction as follows.

```
    temp = 7
for(i=1;i<=50;i++)
{
    .....
    count = temp
    temp = temp + 7;
}
```

The replacement of multiplication by addition will speed up the object code. Hence, the strength of operation is reduced without changing the meaning of above code.

3) Loop invariant method -

In this optimization technique the computation inside the loop is avoided & thereby the comput overhead on compiler is avoided.

ex. for i=0 to 10 do begin
k = i+a/b;
- - -
- - - .
end;

can be written as,

t = a/b;
for i=0 to 10 do begin
k = i+t;
- - -
end;

4) Loop unrolling -
In this method the number of jumps & tests can be reduced by writing the code two times.
ex.

```
int i=1;
while (i<=100)
{
  a[i] = b[i];
  i++;
}
```

can be written as

```
int i=1;
while (i<=100)
{
  a[i] = b[i];
  i++;
  a[i] = b[i];
  i++;
}
```

5) Loop fusion -
In loop fusion method several loops are merged to one loop

```
For i=1 to n do
for j=1 to m do
a[i,j] = 10
```

can be written as

```
for i=1 to n*m do
a[i] = 10
```

* Optimization of Basic Blocks -

A basic block is a sequence of statements in wh flow of control enters at the beginning & leaves end without halt or possibility of branching.

Two types of optimizations that can be done basic blocks.

1) structure preserving transformations.
2) use of algebric identities.

1) Structure preserving transformations -

- structure preserving transformations can be appl by applying some principle technique such as common sub expression elimination, variable & con propogation, code movement, dead code elimination
- The structure preserving transformation is a l based transformation. That means a DAG is cons for the basic block then the above said transform can be applied.
- The common subexpression can be easily detect by observing the DAG for corresponding basic b

ex. $m := n * p$
$n := m + q$
$p := n * p$
$q := m + q$

We assume that value of $n=1$, $p=2$ & $q=3$
∴ Then the expression becomes

$m := n * p = 1 * 2 = 2$
$n := m + q = 2 + 3 = 5$
$p := n * p = 5 * 2 = 10$
$q = m + q = 2 + 3 = 5$

Thus, the above seqn of instructions contain tw common sub expressions such as $n*p$ & $m+q$. But for the common subexpression $n*p$ the of n gets changed when it ~~ap~~ reappears. hen $n*p$ is not a common sub expression.

But the expression $m+q$ gives the same resu in respective appearance & values of m & q are

are consistent each time. Therefor m+q i's a sub expression. Ultimately n & q are the same. The DAG is constructed & the common expressions can be identified.

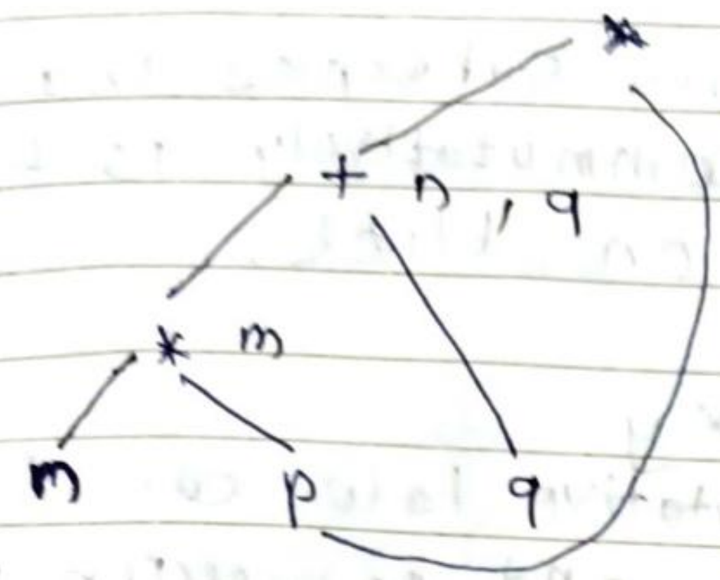


fig. DAG for identifying common subexpressions.

For, u the common expressions means the expressions that are guranted to compute same value. The DAG construction method for optimization is effective for dead code elimination.

2) Use of Algebric Identities -

Algebric identities are used in peephole optimization techniques. Also some simple transformations can be applied in order to optimize the code.

ex. a+0 = a,

a * 1 = a,

a/1 = a

Thus, corresponding identities can be applied on corresponding algebric expressions.

ex.

- The algebric transformation can be obtained using strength ~~reduction~~ technique operator makes the code

ex. Instead of using 2*a we can use a+a

Instead of using a/2 we can use a*0.5

Thus, use of lower strength operator instead of higher strength operators makes the code efficient.

B-divD 3,5,7,12,18,31,41,52,54,56,62,63,65 -

(12) 2,7,9,11,17,24,25,26,29,37,38,68,72 - 13

The constant folding technique can be applied t
achieve the algebric transformations.
ex: Instead of using a = 2 * 5.4 we can use a1)
This saves the effort of compiler in do
Computations.

- The use of common subexpressions elimination,
associativity & commutatively is to apply alg
transformations on block.

ex. x := y * 2

t = 2 * r * y

here, the commutative law can be applied
y * 2 = 2 * y & from 2nd expression we can re
2 * y by x hence optimized block becomes.

x := y * 2

t = x * r

Loops in flow graphs -

1) Dominators -

- In a flow graph, a node d dominates n if every path to node n from initial node goes through d only.
- This can be denoted as 'd dom n'.
- Every initial node dominates all the remaining nodes in the flow graph.
- Similarly every node dominates itself.

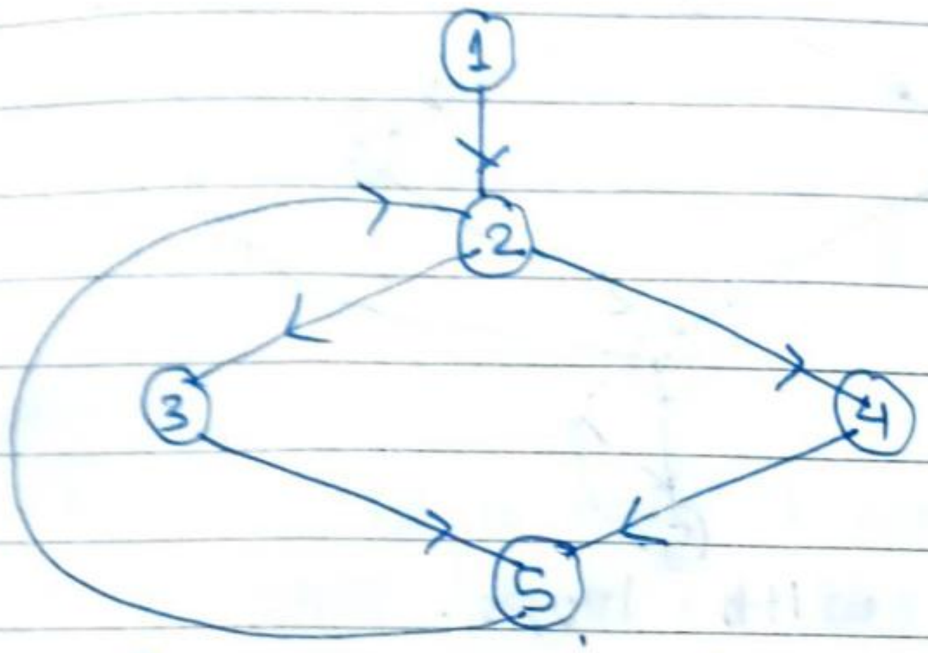


fig. flow graph.

In the above flow graph

- Node 1 is intial 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 2 (it is 1-2-3) Similarly, for node 4 the only path exists is 1-2-4 & for node 5 the only path exists 1-2-4-5 or 1-2-3-5 which is going through 2.
- Node 3 dominates itself. Similarly 4 dominates itself.
- The reason is that for going to remaining node 5 we have another path 1-4-5 which is not through 3. Similarly to going to node 5 there is another path 1-3-5 which is not through 4.
- Node 5 dominates no node.

2) Natural loops -

- Loop in a flow graph can be denoted by $n \to d$ such that d dom n.
- These edges are called back edges & for a loop can be more than one back edges.
- If there is $p \to q$ then q is a head & p is a ta And head dominates tail.

ex.

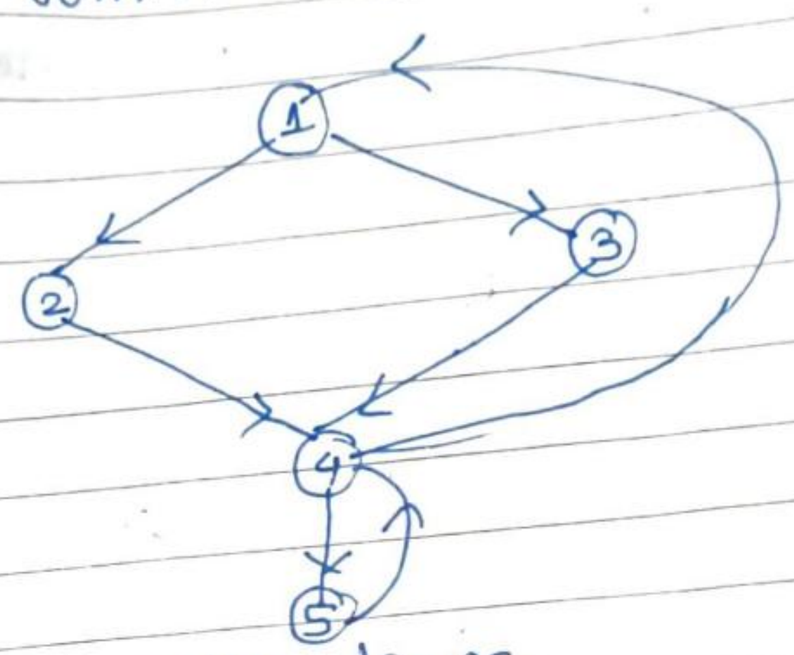
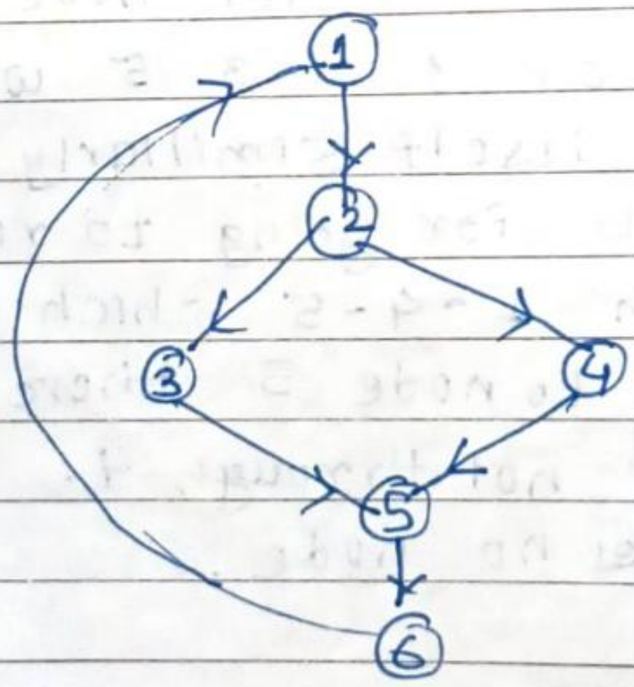


fig. flow graph with loops.

- The loops in the above graph can be denoted by 4 i.e. 1 dom 4. Simillary $5 \to 4$ i.e. dom & dom 5

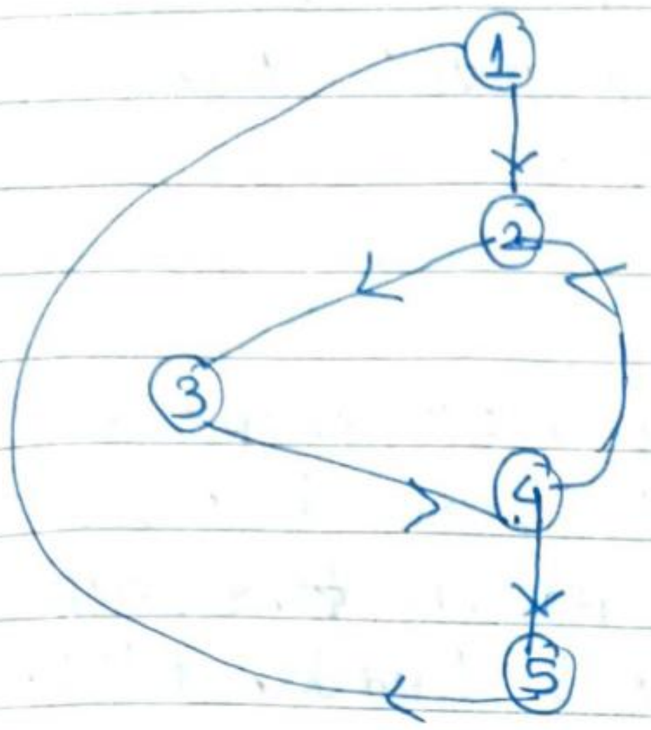
The natural loop can be defined by a edge $n \to d$ su that, there exists a collection of all the nodes th can reach to n without going through d & at th same time d also can be added to this collection.



$6 \to 1$ is a natural loop because we can reach all the remaining nodes from 6.

3) Inner loops -

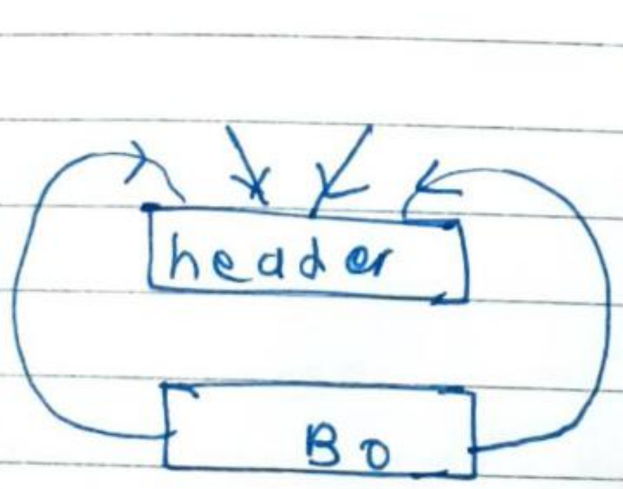
The inner loop is a loop that contains no other loop.

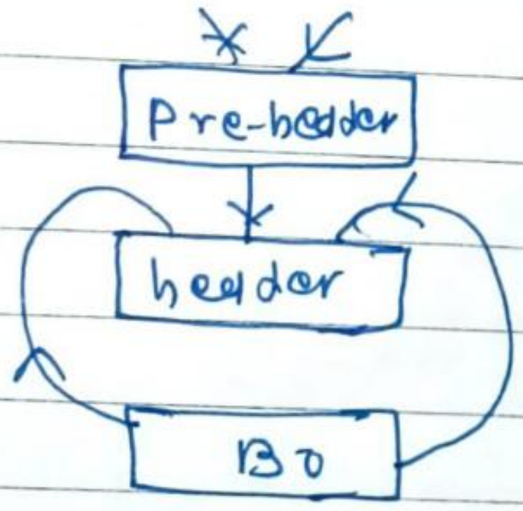


here, the inner loop is 4 → 2 that means edge given by 2.

4) Pre-header -

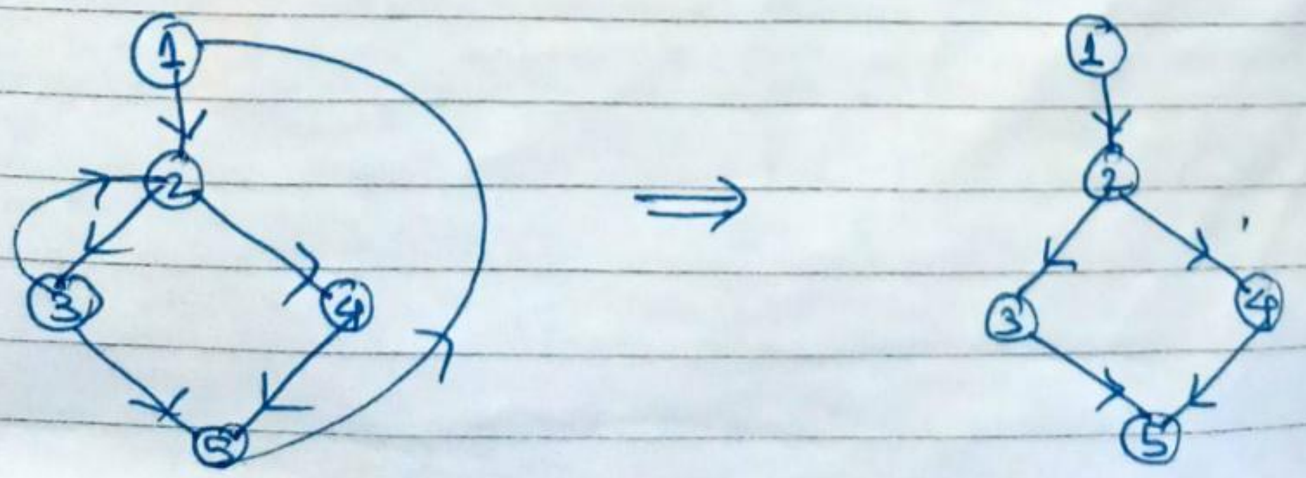
- The pre-header is a new block created such that successor of this block is the header block.
- All the computations that can be made before the header block can be made before the pre-header block





5) Reducible flow graphs -

- The reducible graph is a flow graph in which there are two types of edges forward edges & backward edges.
- These edges have following properties.
  i) The forward edge from an acyclic graph.
  ii) The back edges are such edges who head dominates their tail.



- The above flow graph is reducible. We can reduce graph by removing the back edge from 3 to 2 edge
- Similarly by removing the backedge from 5 to 1 we reduce the above flow graph. And the resultant graph is a cyclic graph.

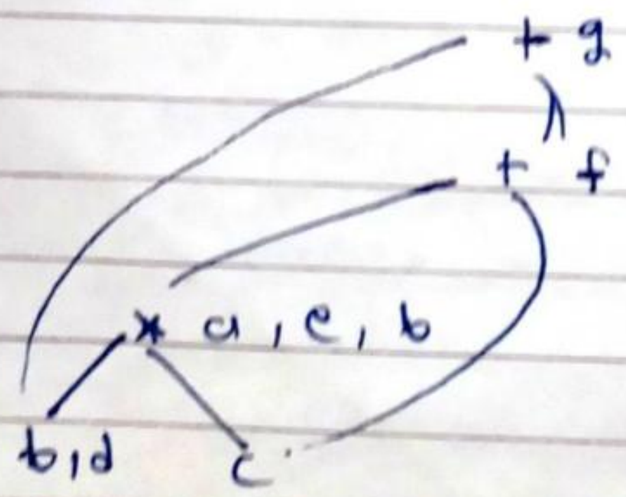
Q.2 Consider the following flow graph & find
i) Dominators for each basic block.
ii) Detect all the loops in ~~the~~graph for each find the back edge & header block information.

* Local optimization -
- The local optimization is done at restricted scope
- In other words local optimization is kind of opti that can done on some sequence of statemen
- Typically local optimization can be done within Specific basic blocks.
- DAG is constructed for the basic block & the optimization can be done.

DAG based Local optimization -
A DAG can be constructed for a block & certain transformmations such as common subexpression elimination can be applied for performing the lo optimization.

ex. Construct the DAG for the following blo

a := b * c
d := b
e := d * c
b := e
f := a + c
g := f + d



The optimized code can be traversing the DAG
The local optimization on the above block can be don
1. A common expression e = d * c which is actually b is eliminated.
2. A dead code b = e is eliminated.

The optimized code & basic block is.

```
a := b * c
d := b
f := a + c
g = f + d
```

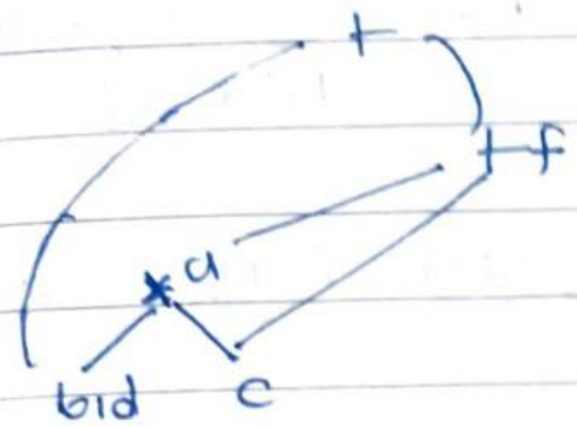
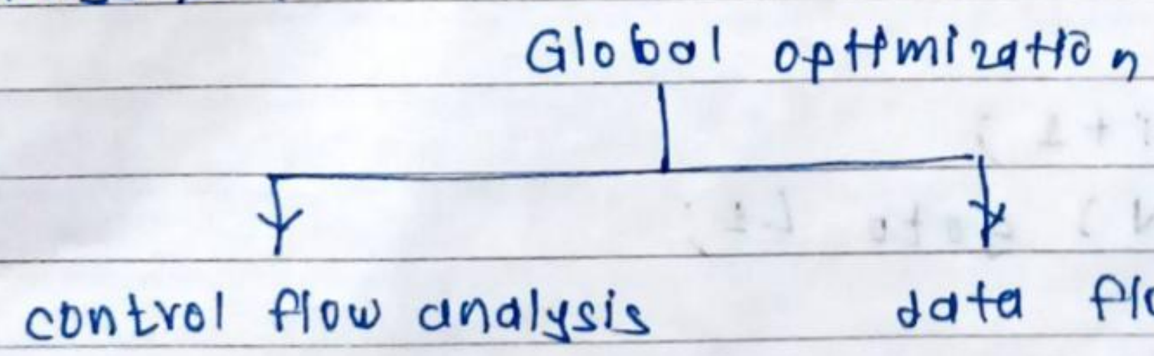


fig. optimized code & DAG

* Global Optimization -
- The local optimization has very restricted scope where as global optimization is applied over a broad scope such as procedure or function body.
- For a global optimization a program is represented in the form of program flow graph.
- The program flow graph representation in which each node represents the basic block & edges represent the flow of control from one block to another.
- There are two types of analysis performed for global optimizations control flow analysis & data flow analysis.



Control flow analysis -
- The control flow ~~analy~~ analysis determines the information regarding arrangement of graph node, presence of loops, nesting of loops & nodes visited before execution of specific node.
- Using this analysis optimization can be performed.
- Thus, in control flow analysis the analysis is made the flow of control by carefully examining the program flow graph.

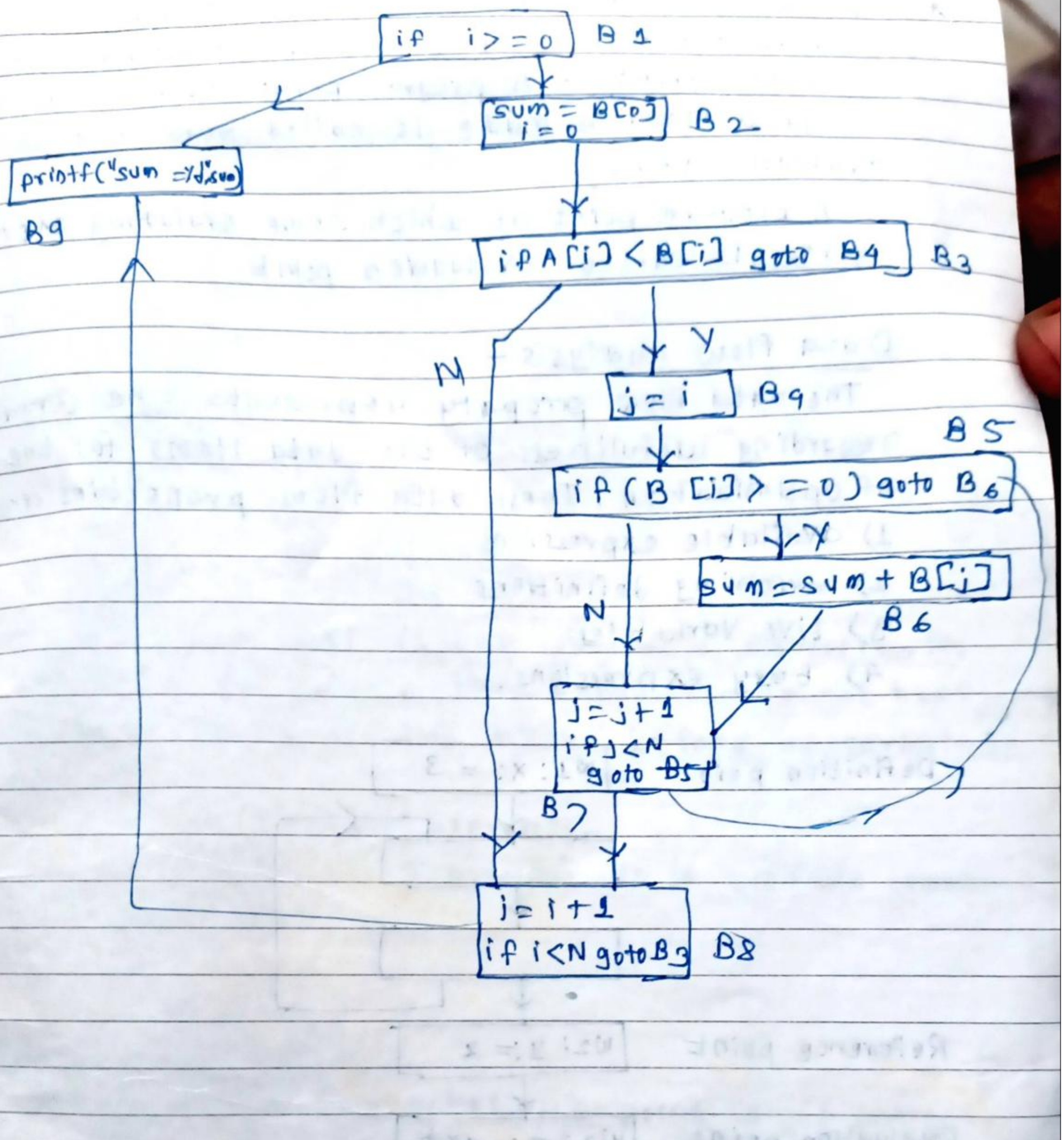
Data flow analysis -

Data flow analysis is made on the data flow.
i.e. The Data flow analysis determines the infor
regarding the definition & use of the data in t
- Using this kind of analysis optimization can be d
- The data flow analysis is basically a proces in wh
the values are computed using data flow proper

ex. Consider following program fragment.

```
if (i >=0)
{
   sum = B[0];
   i=0;
L1 : if (A[i] < B[i])
   {
      j=i;
L2 :
      if (B[i] >=0)
      {
       sum = sum + B[j];
      }
      j = j+1;
     if (j<N) goto L2;
   }
     i=i+1;
  }
    i = i+1;
if(i< N) goto L1;
}
printf("sum = %d \n", sum);
```

Draw control flow graph for this code. Show the basi
blocks clearly in your control flow graph.



* Computing Global Data flow Information -

Definition point - A program point containing the defin

Reference point - A program point at which reference data item is made is called reference point.

Evaluation point

A program point at which some evaluating expression g given is called evaluation point.

Data flow analysis -

The data flow property represents the certain infor regarding wefullness of the data items for the purpos of optimization. These data flow properties are.

1) available expressions
2) reaching definitions
3) live variables
4) busy expressions.

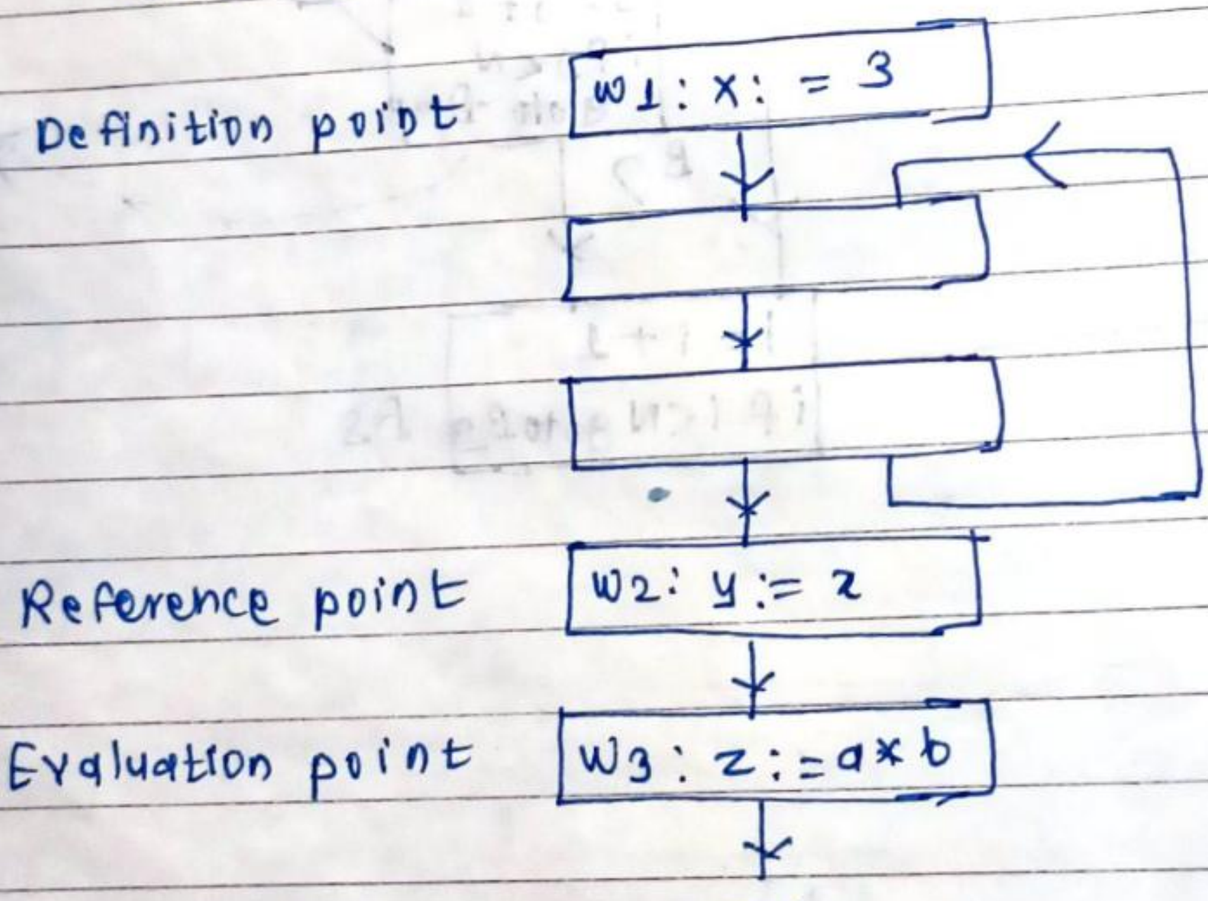


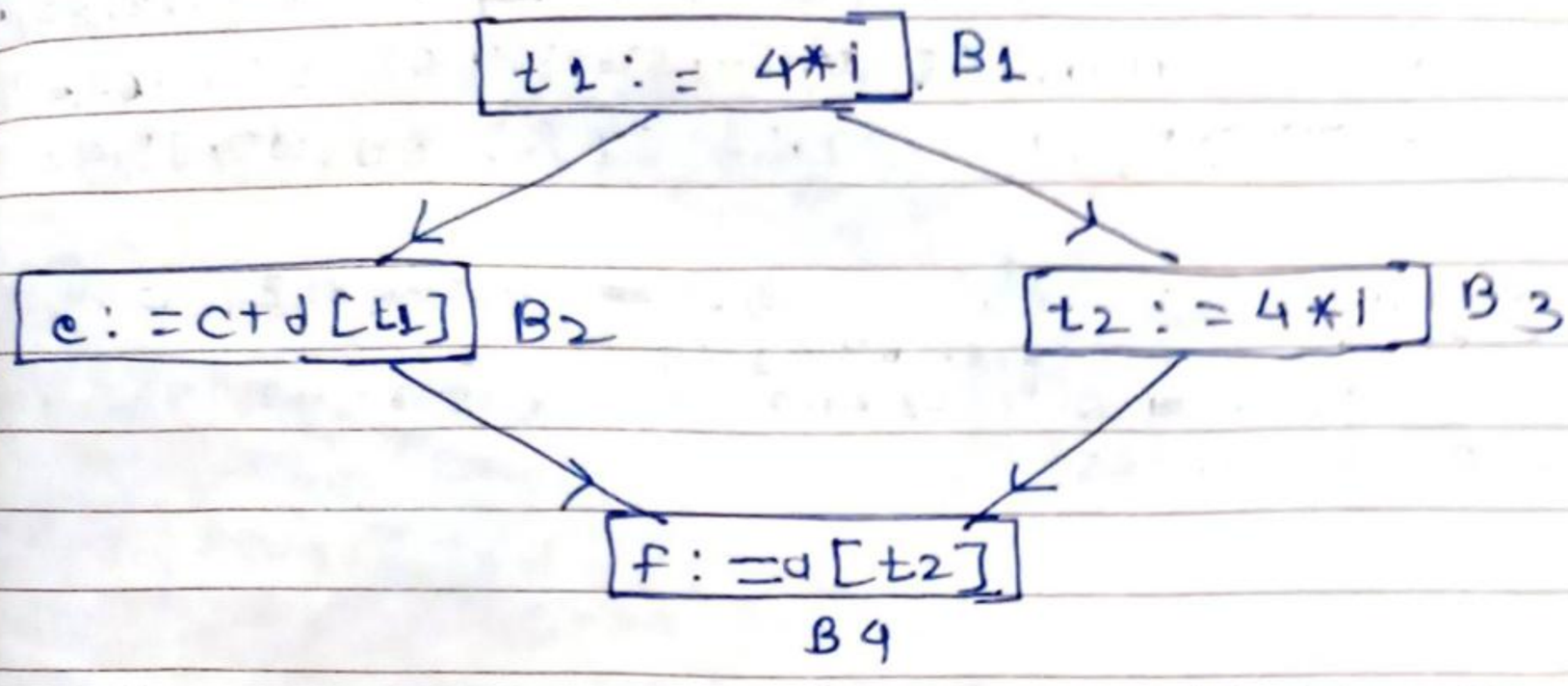
fig. program points.

1) Available expression -

An expression x+y is available at program poin iff along all paths are reaching to w.

1. The expression x+y is said to be available at i evaluation point.

2. The expression x+y is said to be available if no definition of any operand of expression follows its last evaluation along the path. In other word, if neither of two operands get modified before their use.



The expression 4*i is the available expression for B2, B3 & B4 because this expression is not been changed by any of the block before appearing in B

Advantage of available expression

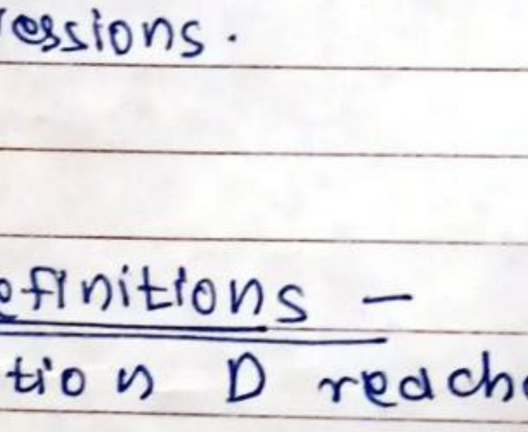
- The use of available expression is to eliminate common sub expressions.

2) Reaching definitions –

A definition D reaches at point p if there is path from D to p along which D is not killed.

A definition D of variable x is killed when there is a redifinition of x.

ex.



The definition d1 is said to a reaching definition for block B2. But the definition d1 is not a reaching definition in block B3, because it is killed by d2 in block B2