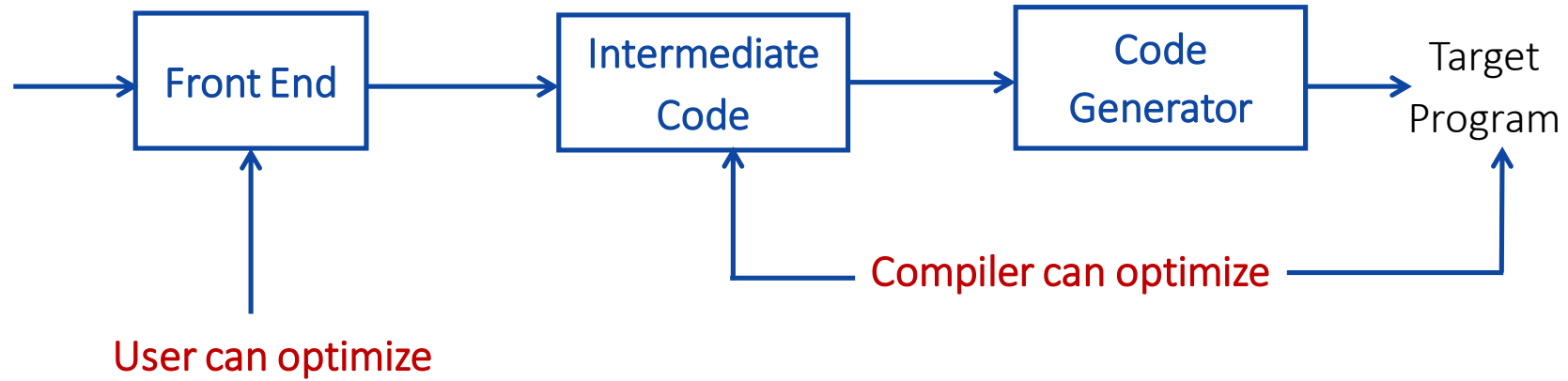


Module 5 – Code Optimization

Code Optimization



Compile time evaluation

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

Folding

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

Example : $\text{length} = (22/7) * d$

- Here folding is implied by performing the computation of $22/7$ at compile time.

Cont.,

Constant/ Variable propagation

- In this technique the value of variable is replaced and computation of an expression is done at compilation time.

Example : $\pi = 3.14$; $r = 5$;

$\text{Area} = \pi * r * r$;

- Here at the compilation time the value of π is replaced by 3.14 and r by 5 then computation of $3.14 * 5 * 5$ is done during compilation.

Principal Sources of Optimization

Principal Sources of Optimization

Semantics-Preserving Transformations

1. Common sub expressions elimination
2. Dead code elimination
3. Code Motion
4. Copy Propagation
5. Induction variable elimination & Reduction in Strength
6. Constant folding

1. Common sub expressions elimination

- The common sub expression is an **expression appearing repeatedly in the program** which is computed previously.
- If the operands of this sub expression do not get changed at all then result of such sub expression is used instead of re-computing it each time.
- **Example:**

t1 := 4 * i
t2 := a[t1]
t3 := 4 * j
t4 := 4 * i
t5 := n
t6 := b[t4] + t5


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

t6 = 4*i x = a[t6] t7 = 4*i t8 = 4*j t9 = a[t8] a[t7] = t9 t10 = 4*j a[t10] = x goto B ₂	→	t6 = 4*i x = a[t6] t8 = 4*j t9 = a[t8] a[t6] = t9 a[t8] = x goto B ₂	B ₅
-----------------------------------------------------------------------------------------------------------------------------	---	---------------------------------------------------------------------------------------------------	----------------

2. Dead code elimination

- The variable is said to be **dead at a point in a program** if the **value contained into it is never been used.**
- The code containing such a variable supposed to be a dead code.
- **Example:**

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



Dead Code

- If statement is a dead code as this condition will never get satisfied hence, statement can be eliminated and optimization can be done.

3. Code Motion

- Optimization can be obtained by **moving some amount of code outside the loop** and placing it just before entering in the loop.
- This method is also called **loop invariant computation**.
- **Example:**

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



```
a=200;
while(a>0)
{
    b=x+y;
    a--;
    if(a%b==0)
        printf("%d",a);
}
```



```
a=200;
b=x+y;
while(a>0)
{
    if(a%b==0)
        printf("%d",a);
}
```

4. Copy Propagation

- Copy propagation means **use of one variable instead of another.**
- Example:

~~x = pi;~~

area = x * r * r;



area = pi * r * r;

x=a

y=x*b

z=x*c



x=a

y=a*b


z=a*c

5. Induction variable elimination & Reduction in Strength

- Priority of certain operators is higher than others.
- For instance strength of $*$ is higher than $+$.
- In this technique the higher strength operators can be replaced by lower strength operators.
 - Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.
 - Multiplication is cheaper than exponentiation. So we can replace exponentiation with multiplication within the loop.

- **Example:**

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



- Here, we get the count values as 7, 14, 21.... and so on.

Cont... [Another Example]

```
i = 1;  
while (i < 10)  
{  
    t = i * 4;  
    i = i + 1;  
}
```

\Rightarrow

```
t = 0;  
while (t < 40)  
{  
    t = t + 4;  
}
```

6. Constant Folding

- If the value of an expression is constant then use constant instead of the expression
- Ex: $\text{pi} = 22/7 \rightarrow \text{pi} = 3.14..$

Global Data Flow Analysis

Global Data Flow Analysis

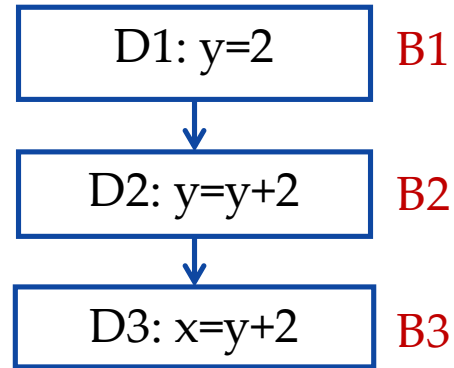
- To do code optimization and code generation
 - Compiler needs to collect information about the whole program
 - Distribute it to each block in the whole program
- Data flow equations are the equations representing the expressions that are appearing in the flow graph.
- Data flow information can be collected by setting up and solving systems of equations that relate information at various points in a program.
- The data flow equation written in a form of equation such that,
$$\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$$
 - Data flow equation can be read as “the information at the end of a statement is either generated within a statement, or enters at the beginning and is not killed as control flows through the statement”.

Path

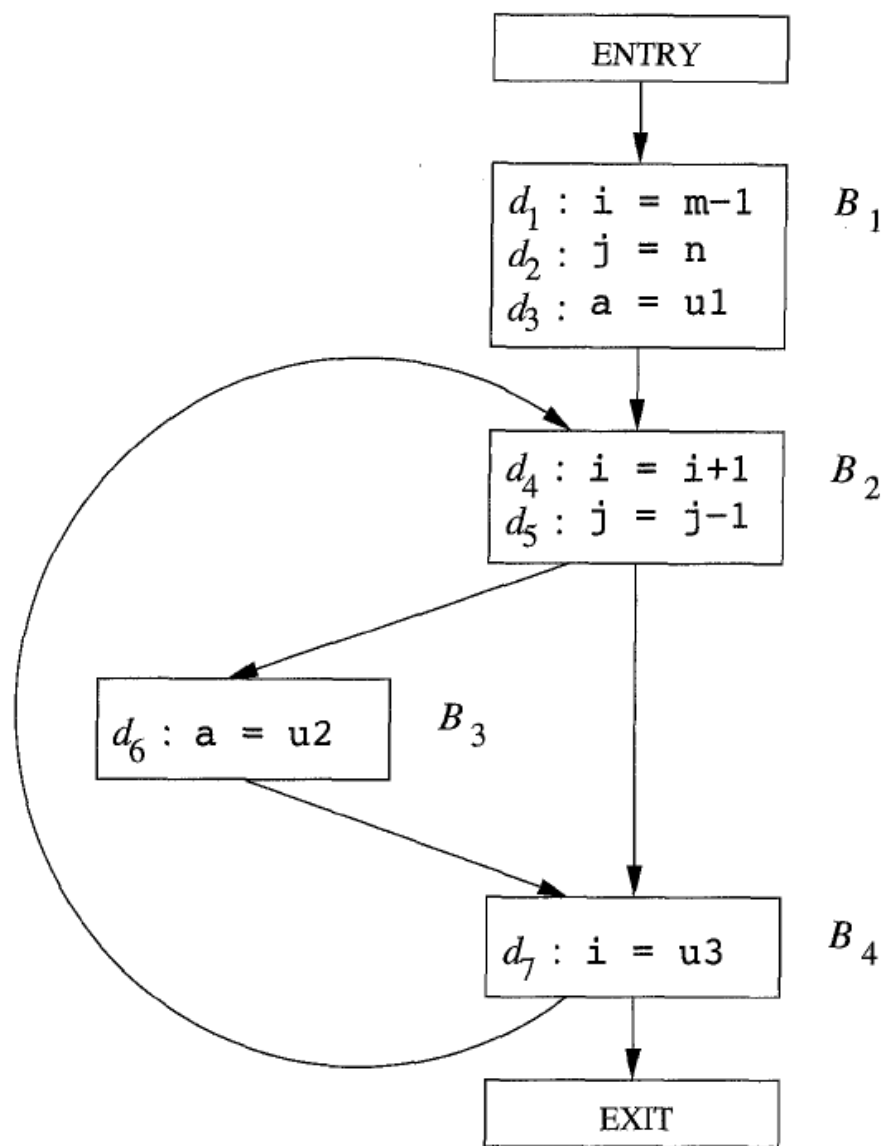
- A path from p_1 to p_n is sequence of points p_1, p_2, \dots, p_n such that for each i between 1 and $n-1$, either,
 - i. P_i – point immediately preceding the statement
 P_{i+1} – point immediately following that statement in the same block
 - ii. P_i – End of some block
 P_{i+1} – beginning of a successor block

Reaching Definition

- A definition D reaches at the point P if there is a path from D to P along which D is not killed.
- A definition D of variable x is killed when there is a redefinition of x .



- The definition $D1$ is reaching definition for block $B2$, but the definition $D1$ is not reaching definition for block $B3$, because it is killed by definition $D2$ in block $B2$.



$$gen_{B_1} = \{ d_1, d_2, d_3 \}$$

$$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$$

$$gen_{B_2} = \{ d_4, d_5 \}$$

$$kill_{B_2} = \{ d_1, d_2, d_7 \}$$

$$gen_{B_3} = \{ d_6 \}$$

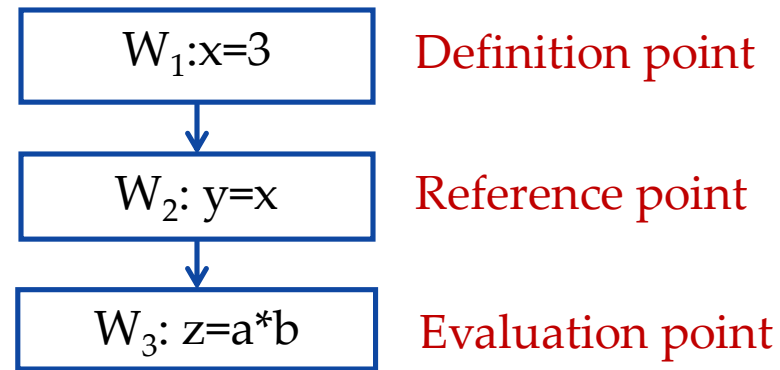
$$kill_{B_3} = \{ d_3 \}$$

$$gen_{B_4} = \{ d_7 \}$$

$$kill_{B_4} = \{ d_1, d_4 \}$$

Data Flow Properties

- A program point containing the definition is called **Definition point**.
- A program point at which a reference to a data item is made is called **Reference point**.
- A program point at which some evaluating expression is given is called **Evaluation point**.

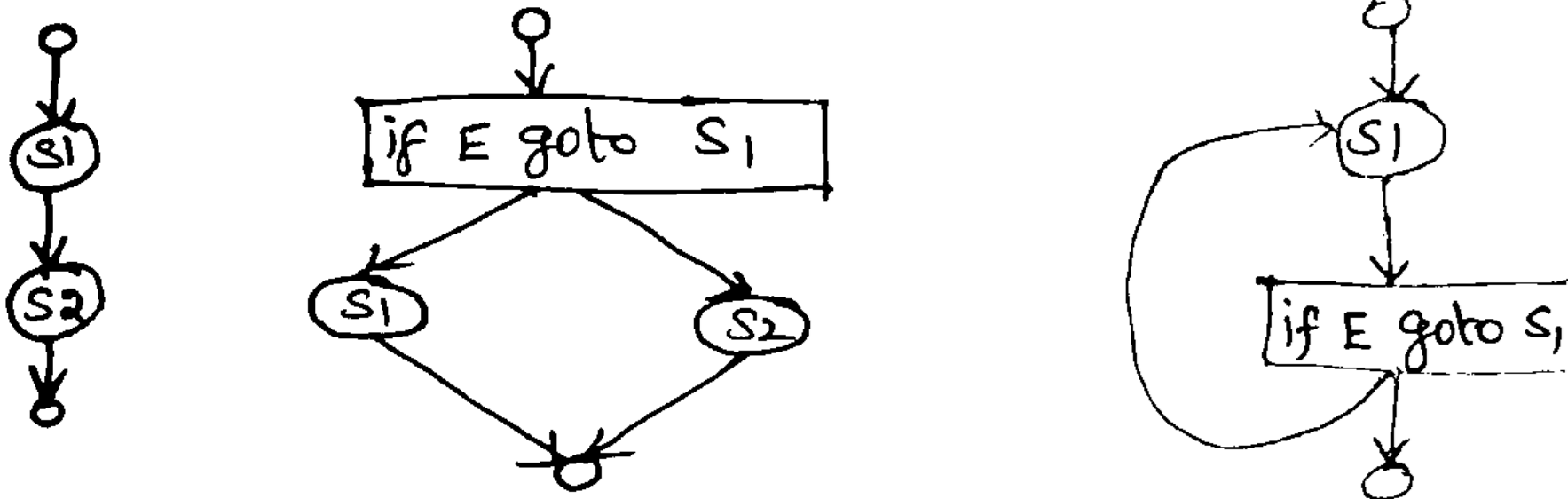


Dataflow analysis of structured programs

$S \rightarrow \text{id}:E \mid S; S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{do } S \text{ while } E$

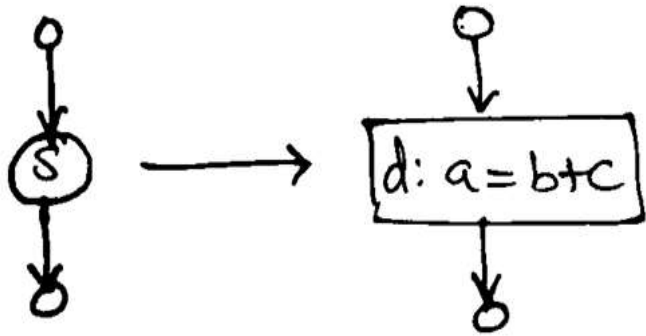
$E \rightarrow \text{id}+\text{id} \mid \text{id}$

Example:



Data flow equations for reaching definitions – [statement(s)]

Single Statement

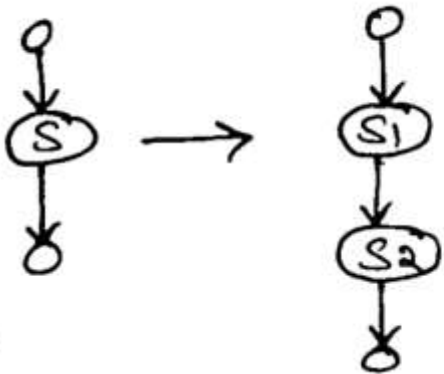


$gen[S] = \{d\}$ ← Newly generated definition, d for a

$kill[S] = D_a - \{d\}$ ← All previous definition for a will be killed except the newly generated definition, d

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

Sequence of Statements



$$gen[S] = gen[S_2] \cup (gen[S_1] - kill[S_2])$$

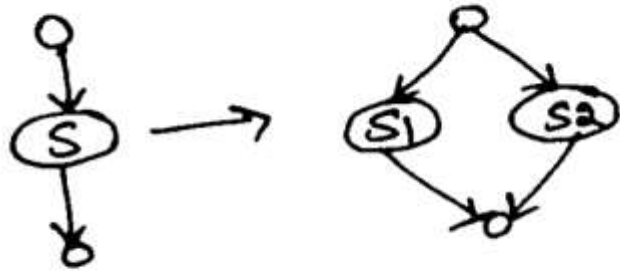
$$kill[S] = kill[S_2] \cup (kill[S_1] - gen[S_2])$$

$$in[S_1] = in[S]$$

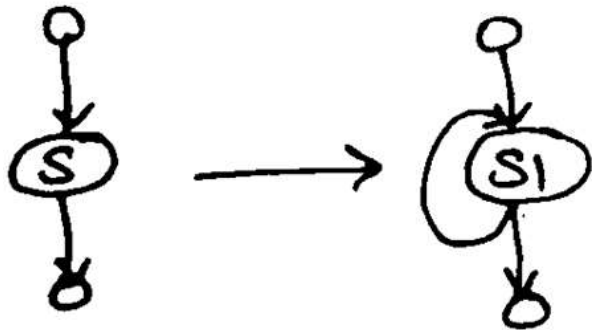
$$in[S_2] = out[S_1]$$

$$out[S] = out[S_2]$$

Data flow equations for reaching definitions – [if..else & loop]



$$\begin{aligned} \text{gen}[S] &= \text{gen}[S_1] \cup \text{gen}[S_2] \\ \text{kill}[S] &= \text{kill}[S_1] \cap \text{kill}[S_2] \\ \text{in}[S_1] &= \text{in}[S] \\ \text{in}[S_2] &= \text{in}[S] \\ \text{out}[S] &= \text{out}[S_1] \cup \text{out}[S_2] \end{aligned}$$



$$\begin{aligned} \text{gen}[S] &= \text{gen}[S_1] \\ \text{kill}[S] &= \text{kill}[S_1] \\ \text{in}[S_1] &= \text{in}[S] \cup \text{gen}[S_1] \\ \text{out}[S] &= \text{out}[S_1] \end{aligned}$$

Basic Block & Flow Graph

Basic Blocks

- A basic block is a **sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end** without halt or possibility of branching except at the end.
- The following sequence of three-address statements forms a basic block:

$t1 := a * a$

$t2 := a * b$

$t3 := 2 * t2$

$t4 := t1 + t3$

$t5 := b * b$

$t6 := t4 + t5$

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements.

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1. We first determine the set of **leaders**, for that we use the following rules:
 - a. The first statement is a leader.
 - b. Any statement that is the target of a conditional or unconditional goto is a leader.
 - c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example1: Partition into basic blocks

begin

prod := 0;

i := 1;

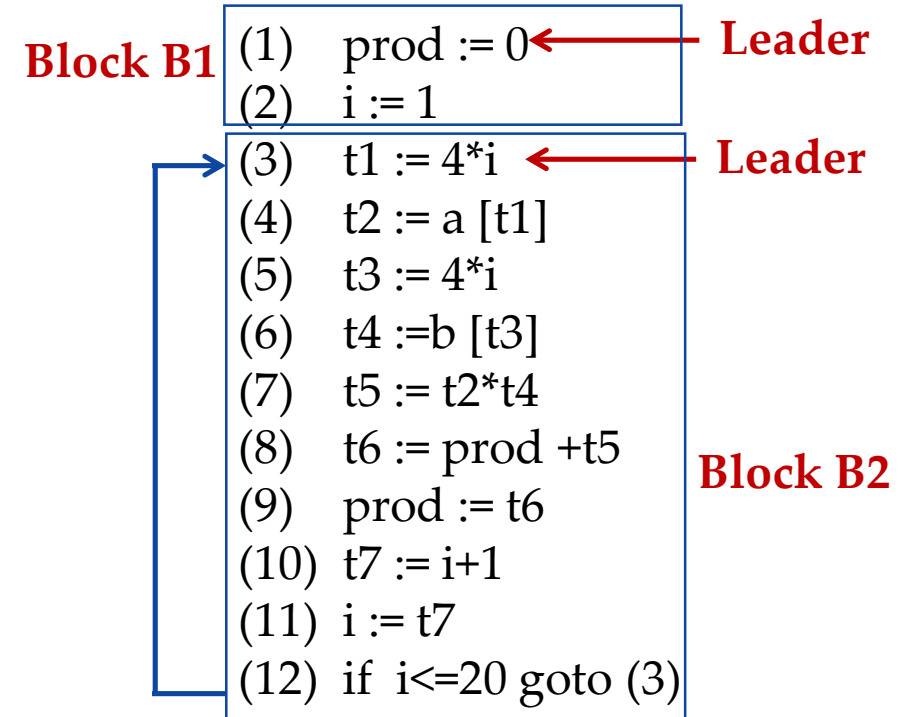
do

prod := prod + a[t1] * b[t2];

i := i+1;

while i <= 20

end



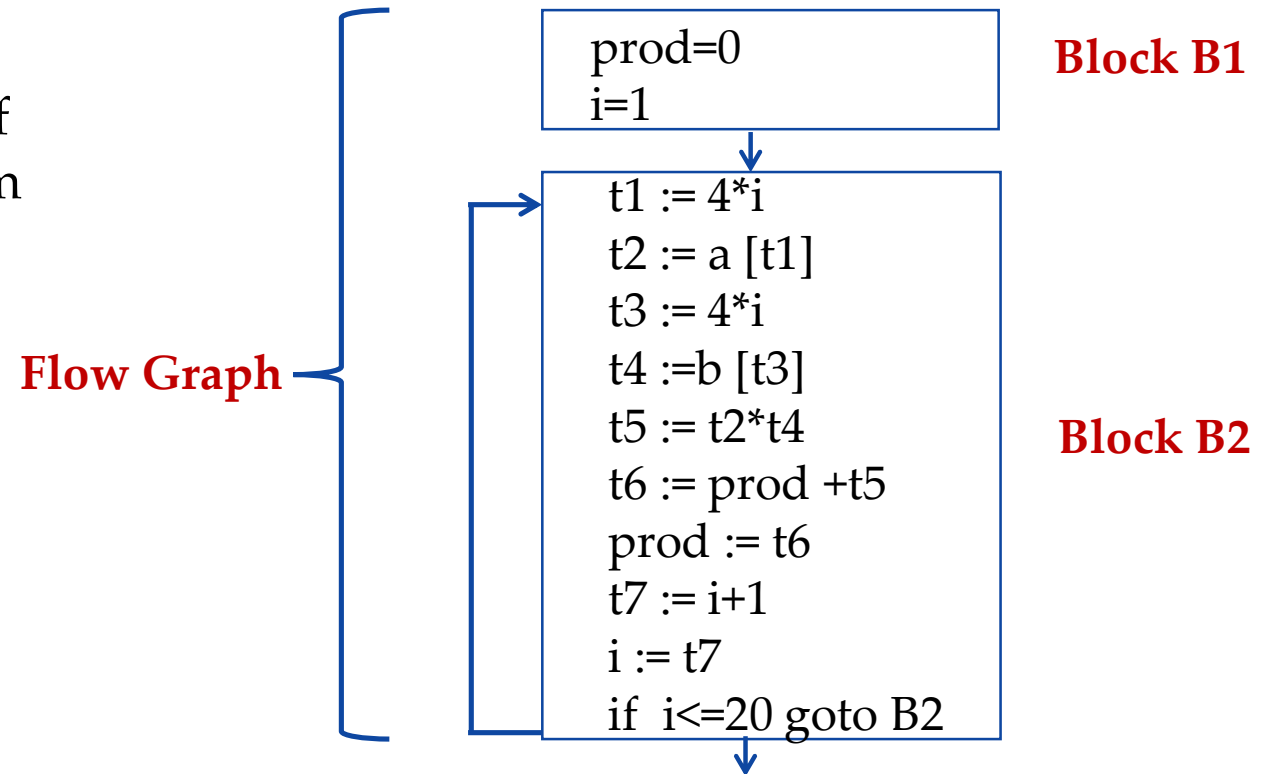
**Three Address
Code**

Flow Graph

- We can add flow-of-control information to the set of basic blocks making up a program by constructing a direct graph called a **flow graph**.
- Nodes in the flow graph represent computations, and the edges represent the flow of control.
- Example of flow graph for following three address code:

Construction of Flow Graph

1. Draw directed edge from block1 to block2 if
 - a. Conditional or unconditional jump from block1 to block2
 - b. Block2 immediately follows block1



Transformation on Basic Blocks

Transformation on Basic Blocks

- A number of transformations can be applied to a basic block without changing the set of expressions computed by the block.
- Many of these transformations are useful for improving the quality of the code.
- Types of transformations are:
 - I. Structure preserving transformation
 - II. Algebraic transformation

I. Structure Preserving Transformations

- Structure-preserving transformations on basic blocks are:
 1. Common sub-expression elimination
 2. Dead-code elimination
 3. Renaming of temporary variables
 4. Interchange of two independent adjacent statements

1. Common sub-expression elimination

- Consider the basic block,

a:= b+c

b:= a-d

c:= b+c

d:= a-d

- The second and fourth statements compute the same expression, hence this basic block may be transformed into the equivalent block:

a:= b+c

b:= a-d

c:= b+c

d:= b

2. Dead-code elimination

- Suppose x is dead, that is, never subsequently used, at the point where the statement $\mathbf{x} := \mathbf{y} + \mathbf{z}$ appears in a basic block.
- Above statement may be safely removed without changing the value of the basic block.

3. Renaming of temporary variables

- Suppose we have a statement
 $t := b + c$, where t is a temporary variable.
- If we change this statement to
 $u := b + c$, where u is a new temporary variable,
- Change all uses of this instance of t to u , then the value of the basic block is not changed.
- In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary.
- We call such a basic block a *normal-form* block.

4. Interchange of two independent adjacent statements

- Suppose we have a block with the two adjacent statements,

$t1 := b + c$

$t2 := x + y$

- Then we can interchange the two statements without affecting the value of the block if and only if neither x nor y is $t1$ and neither b nor c is $t2$.
- A normal-form basic block permits all statement interchanges that are possible.

II. Algebraic Transformation

- Countless algebraic transformation can be used to change the set of expressions computed by the basic block into **an algebraically equivalent set**.
- The useful ones are those that **simplify expressions or replace expensive operations by cheaper one**.
- Example:
 - $x := x + 0$ or $x := x * 1$ can be eliminated from a basic block.
 - The exponential statement $x := y * * 2$ can be replaced by $x := y * y$.

Peephole Optimization

Peephole optimization

- Peephole optimization is a simple and effective technique for locally improving target code.
- Replaces short sequences of target instructions (PEEPHOLE) by a shorter or faster sequence.
 - It is applied to improve the performance of the target program
- Peephole is a small, moving window on the target program.

Techniques of peephole optimization

1. Redundant Loads & Stores
2. Unreachable code elimination
3. Flow of Control Optimization
4. Algebraic simplification
5. Use of Machine idioms

Redundant Loads & Stores

- Especially the **redundant loads and stores** can be eliminated in following type of transformations.

- Example:

MOV R_0, x

MOV x, R_0

- We can **eliminate the second instruction** since x is in already R_0 unless it is referred by some other statements [preceded by any label].

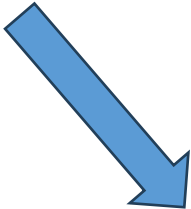
2. Unreachable code elimination

```
x=0;  
if (x==1)  
{ a=b;  
}
```

Intermediate code.

```
x=0  
if x=1 goto L1  
goto L2  
L1: a=b;  
L2:
```

OPTIMIZED CODE after removing
unnecessary jump statements



```
x=0  
if x≠1 goto L2  
a=b; X  
L2:
```


UNREACHABLE
CODE can be removed

3. Flow of Control Optimization

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.
- We can replace the jump sequence.

Goto L1
..... 
L1: goto L2

- It may be possible to eliminate the statement **L1: goto L2** provided it is preceded by an unconditional jump. Similarly, the sequence can be replaced by:

If a<b goto L1
..... 
L1: goto L2

4. Algebraic simplification

- Peephole optimization is an effective technique for algebraic simplification.
- The statements such as $x = x + 0$ or $x := x * 1$ can be eliminated by peephole optimization.
- Reduction in strength
 - Certain machine instructions are cheaper than the other.
 - In order to improve performance of the intermediate code we can replace these instructions by equivalent cheaper instruction.
 - For example, $x * x$ is cheaper than x^2 , $x + x$ is cheaper than $2 * x$, $x \ll 1$ is cheaper than $x * 2$, $x \gg 1$ is cheaper than $x/2$
 - Similarly addition and subtraction are cheaper than multiplication and division. So we can add effectively equivalent addition and subtraction for multiplication and division.

5. Use of Machine idioms

- The target instructions have equivalent machine instructions for performing some operations.
 - We can replace these target instructions by equivalent machine instructions in order to improve the efficiency.
- Example: Some machines have **auto-increment** or **auto-decrement addressing modes**.
- These modes can be used in code for statement like $i=i+1 \rightarrow \text{INCR } i$
 $i=i-1 \rightarrow \text{DECR } i$

DAG Representation of Basic Block

Algorithm: DAG Construction

We assume the three address statement could of following types:

Case (i) $x:=y \text{ op } z$

Case (ii) $x:=\text{op } y$

Case (iii) $x:=y$

With the help of following steps the DAG can be constructed.

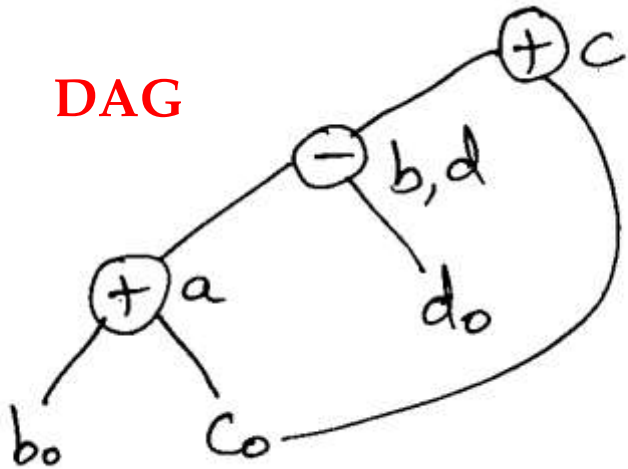
- **Step 1:** If y is undefined then create $\text{node}(y)$. Similarly if z is undefined create a node (z)
- **Step 2:**
 - Case(i)** create a node(op) whose left child is $\text{node}(y)$ and $\text{node}(z)$ will be the right child. Also check for any common sub expressions.
 - Case (ii)** determine whether is a node labeled op , such node will have a child $\text{node}(y)$.
 - Case (iii)** node n will be $\text{node}(y)$.
- **Step 3:** Delete x from list of identifiers for $\text{node}(x)$. Append x to the list of attached identifiers for node n found in 2.

Examples

EX

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

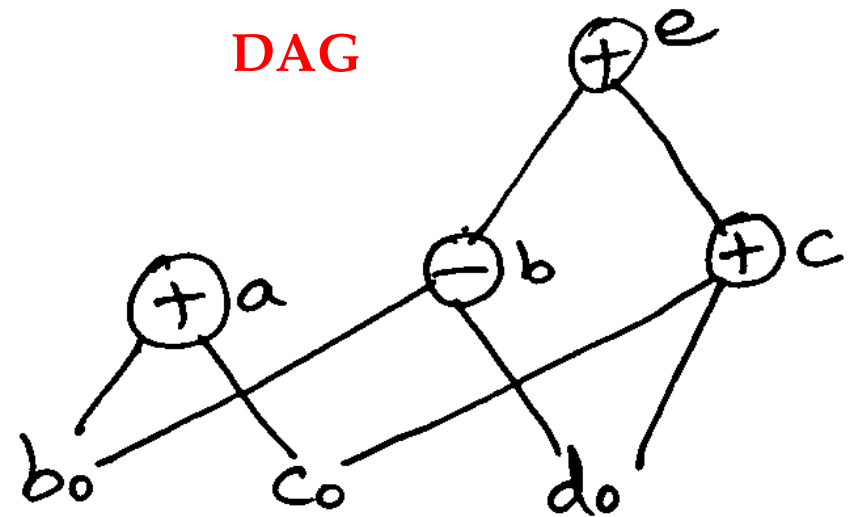
DAG



EX 2

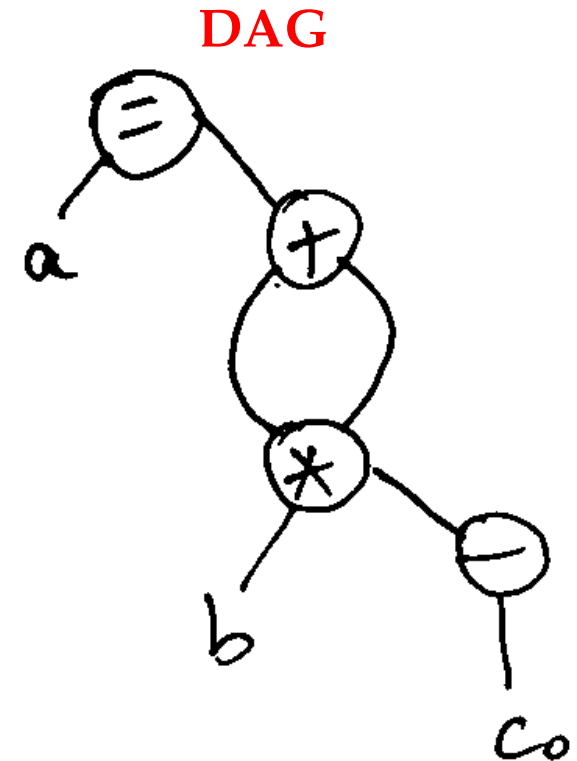
$a = b + c$
 $b = b - d$
 $c = c + d$
 $e = b + c$

DAG



Cont...

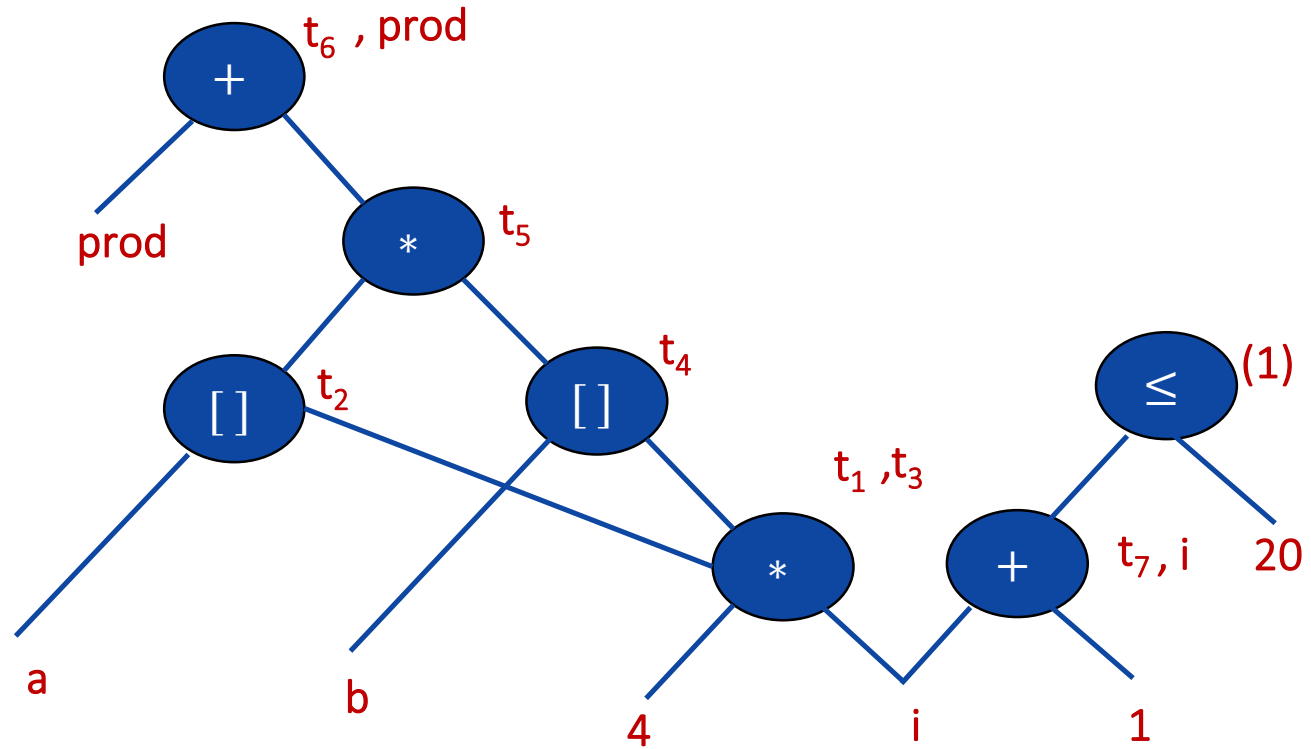
EX:3
 $a = b * -c + b * -c$



DAG Representation of Basic Block

Example:

- (1) $t_1 := 4 * i$
- (2) $t_2 := a[t_1]$
- (3) $t_3 := 4 * i$
- (4) $t_4 := b[t_3]$
- (5) $t_5 := t_2 * t_4$
- (6) $t_6 := \text{prod} + t_5$
- (7) $\text{prod} := t_6$
- (8) $t_7 := i + 1$
- (9) $i := t_7$
- (10) if $i \leq 20$ goto (1)



Generation of Code from DAGs

Generation of Code from DAGs

- Methods generating code from DAGs are:
 1. Rearranging Order
 2. Heuristic ordering
 3. Labeling algorithm

Rearranging Order

- The order of three address code affects the cost of the object code being generated.
- By changing the order in which computations are done we can obtain the object code with minimum cost.
- Example:

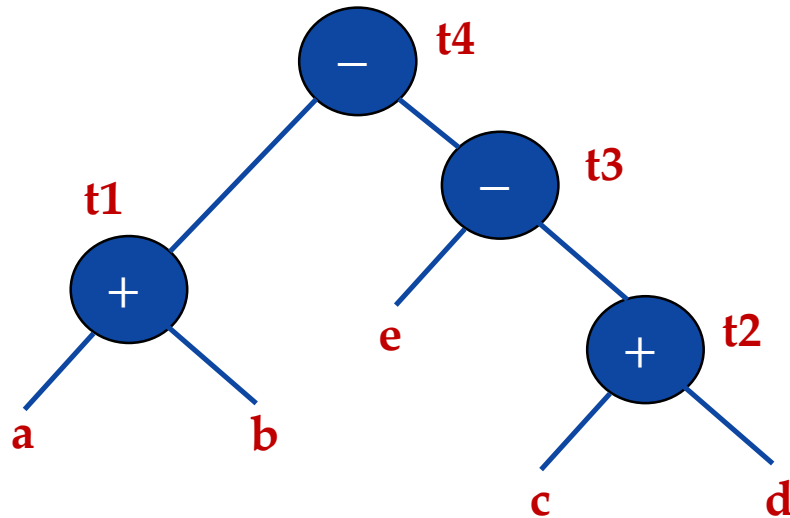
t1:=a+b

t2:=c+d

t3:=e-t2

t4:=t1-t3

Three Address
Code



Example: Rearranging Order

t1:=a+b

t2:=c+d

t3:=e-t2

t4:=t1-t3

Three Address Code

Re-arrange



t2:=c+d

t3:=e-t2

t1:=a+b

t4:=t1-t3

Three Address Code

MOV a, R0

ADD b, R0

MOV c, R1

ADD d, R1

MOV R0, t1

MOV e, R0

SUB R1, R0

MOV t1, R1

SUB R0, R1

MOV R1, t4

Assembly Code

MOV c, R0

ADD d, R0

MOV e, R1

SUB R0, R1

MOV a, R0

ADD b, R0

SUB R1, R0

MOV R0, t4

Assembly Code

Algorithm: Heuristic Ordering

Obtain all the interior nodes. Consider these interior nodes as unlisted nodes.

while(unlisted interior nodes remain)

{

 pick up an unlisted node n, whose parents have been listed

 list n;

 while(the leftmost child m of n has no unlisted parent AND is not leaf)

 {

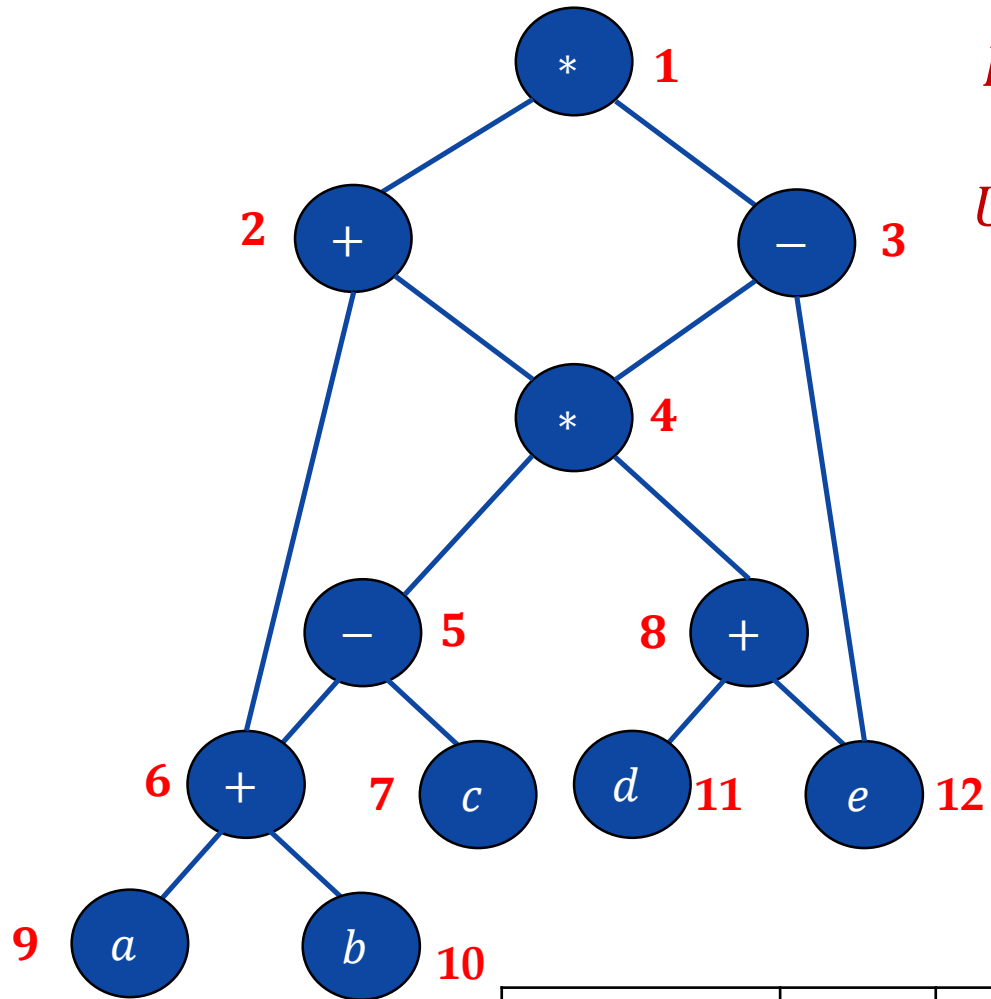
 List m;

 n=m;

 }

}

Example: Heuristic Ordering



Interior nodes = 1 2 3 4 5 6 8

Unlisted nodes = ~~1~~ ~~2~~ 3 4 5 6 8

Pick up an unlisted node,
whose parents have been listed

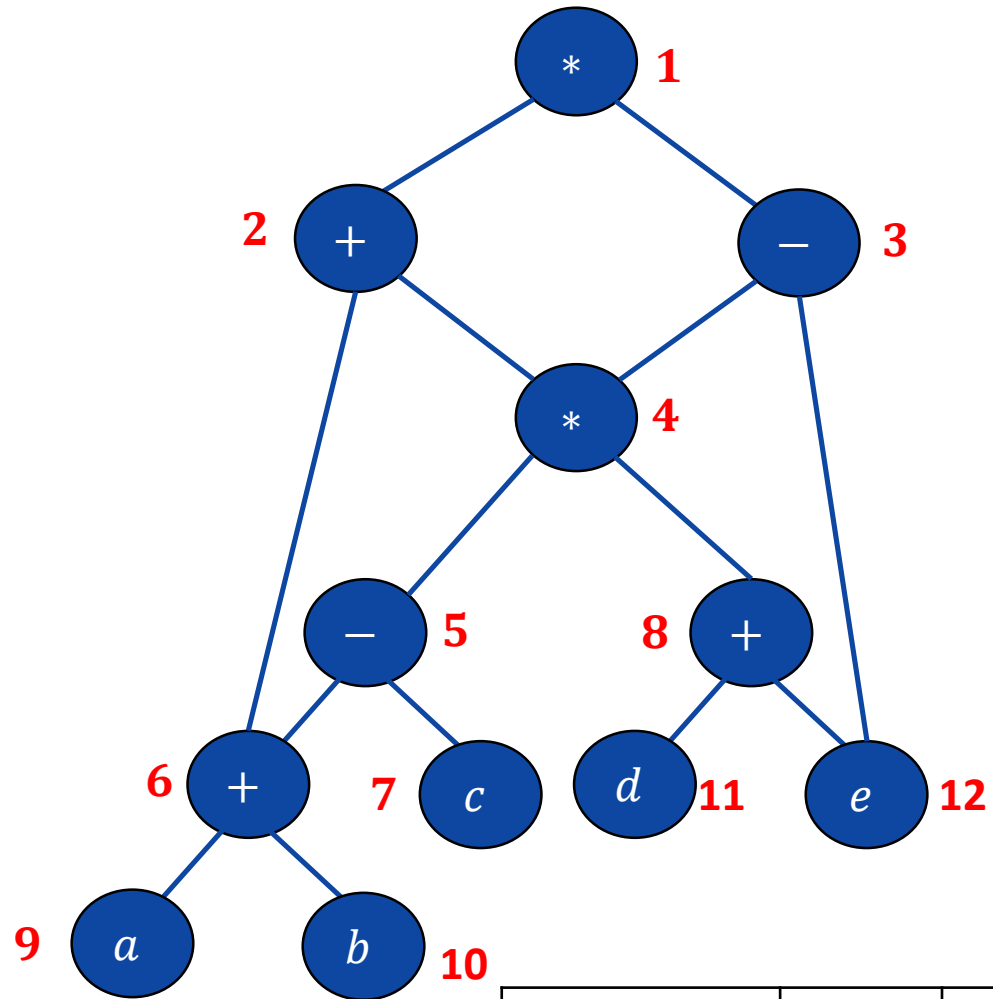
1

Left child of 1 = **2**
Parent 1 is listed so list 2

Left child of 2 = **6**
Parent 5 is not listed so can't list 6

Listed Node							
-------------	--	--	--	--	--	--	--

Example: Heuristic Ordering



Interior nodes = 1 2 3 4 5 6 8

Unlisted nodes = ~~1~~ ~~2~~ ~~3~~ ~~4~~ 5 6 8

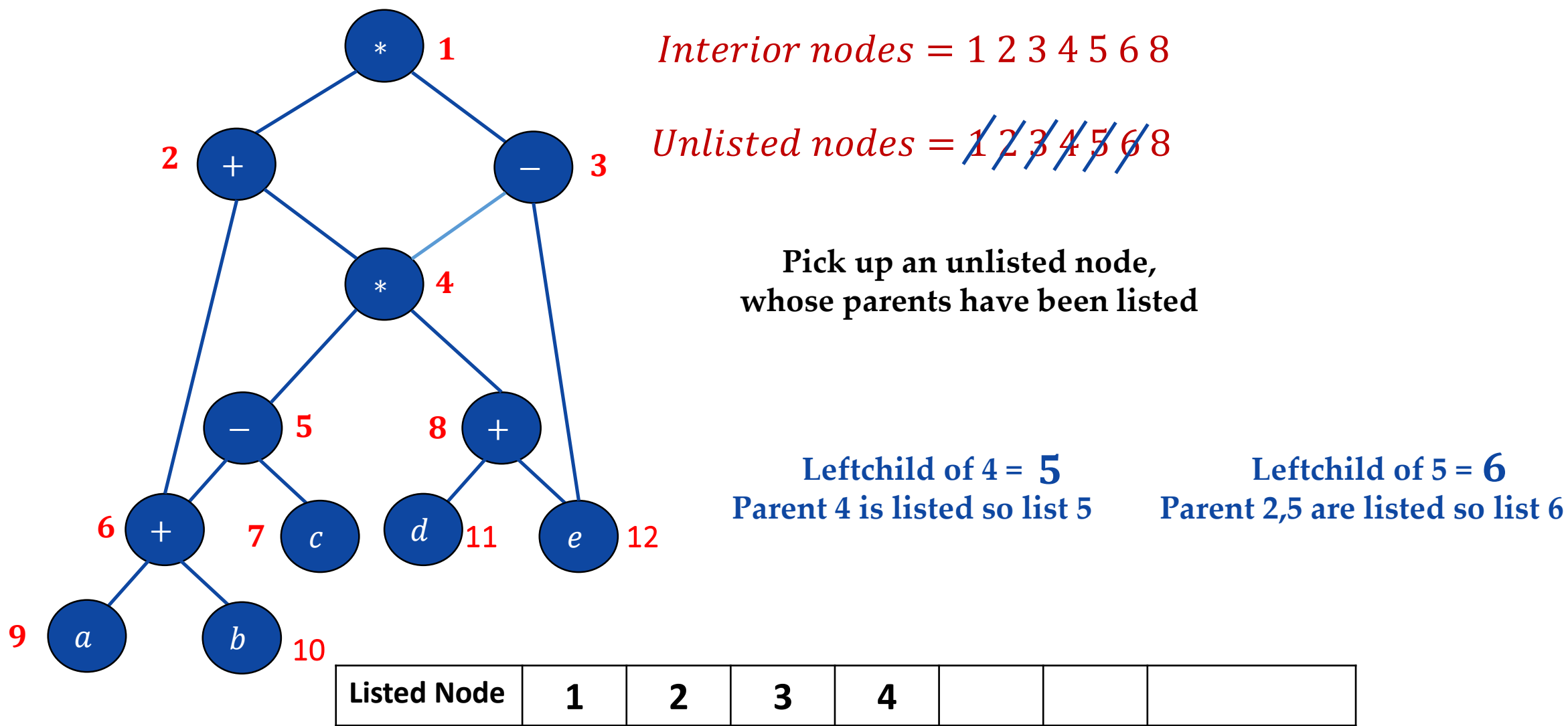
Pick up an unlisted node,
whose parents have been listed

Rightchild of 1 = **3**
Parent 1 is listed so list 3

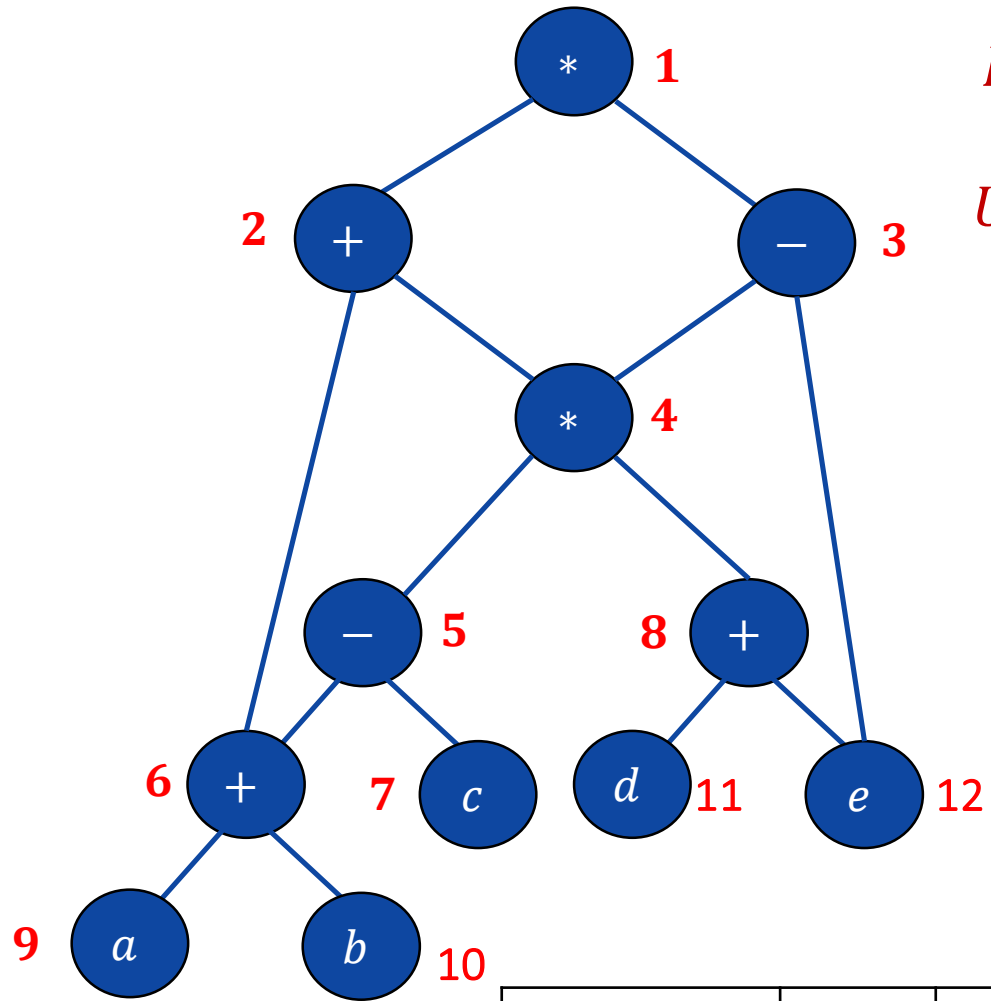
Leftchild of 3 = **4**
Parent 2,3 are listed so list 4

Listed Node	1	2					
-------------	---	---	--	--	--	--	--

Example: Heuristic Ordering



Example: Heuristic Ordering



Interior nodes = 1 2 3 4 5 6 8

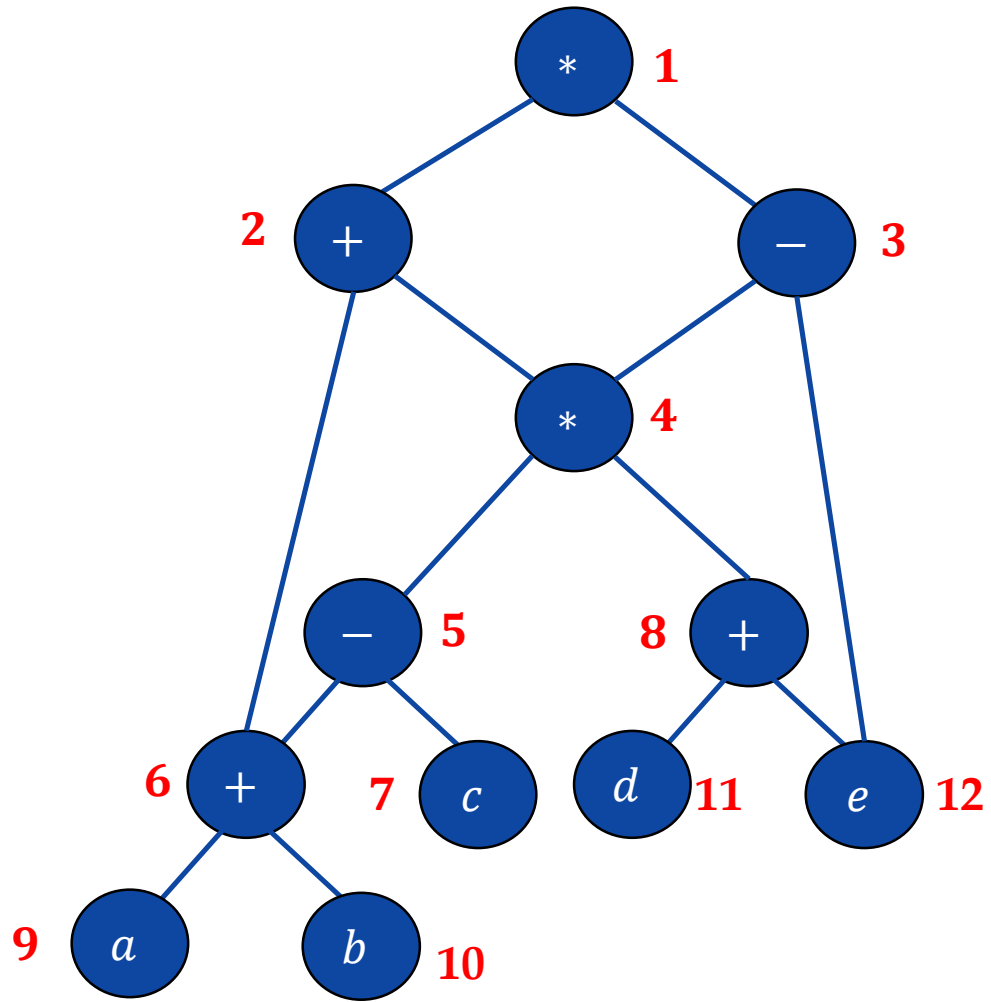
Unlisted nodes = ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~8~~

Pick up an unlisted node,
whose parents have been listed

Rightchild of 4 = **8**
Parent 4 is listed so list 8

Listed Node	1	2	3	4	5	6	
-------------	---	---	---	---	---	---	--

Example: Heuristic Ordering



Listed Node	1	2	3	4	5	6	8
-------------	---	---	---	---	---	---	---

Reverse Order for three address code = 8 6 5 4 3 2 1

t8=d+e
t6=a+b
t5=t6-c
t4=t5*t8
t3=t4-e
t2=t6+t4
t1=t2*t3

Optimal Three
Address
code

Labeling Algorithm

- The labeling algorithm is used to find out how many registers will be required by a program to complete its execution.
- Using labeling algorithm, the labeling can be done to tree by visiting nodes in bottom up order.
- For computing the label at node n with the label $L1$ to left child and label $L2$ to right child as,

$$\text{Label}(n) = \max(L1, L2) \quad \text{if } L1 \text{ not equal to } L2$$

$$\text{Label}(n) = L1 + 1 \quad \text{if } L1 = L2$$

- We start in bottom-up fashion and label left leaf as 1 and right leaf as 0.

Example: Labeling Algorithm

t1:=a+b

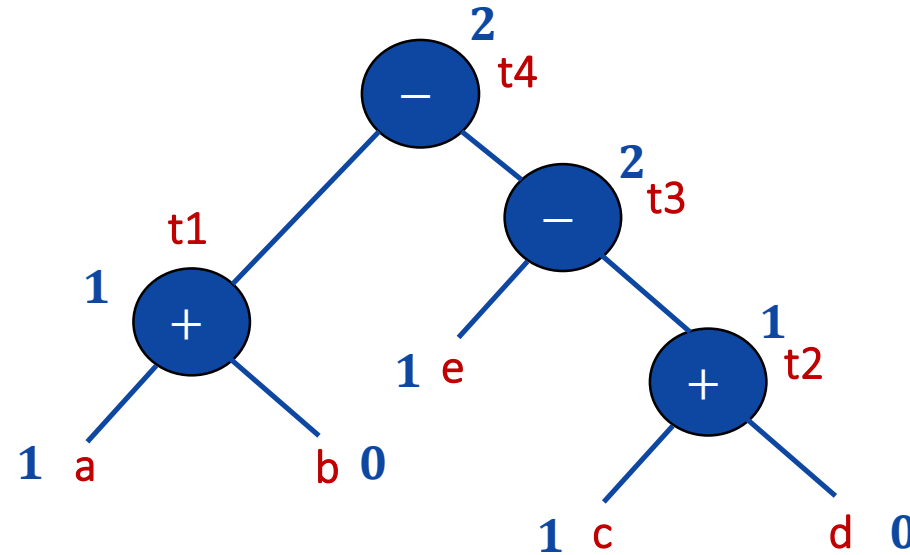
t2:=c+d

t3:=e-t2

t4:=t1-t3

Three Address Code

$$Label(n) = \begin{cases} Max(l1, l2) & \text{if } l1 \neq l2 \\ l1 + 1 & \text{if } l1 = l2 \end{cases}$$



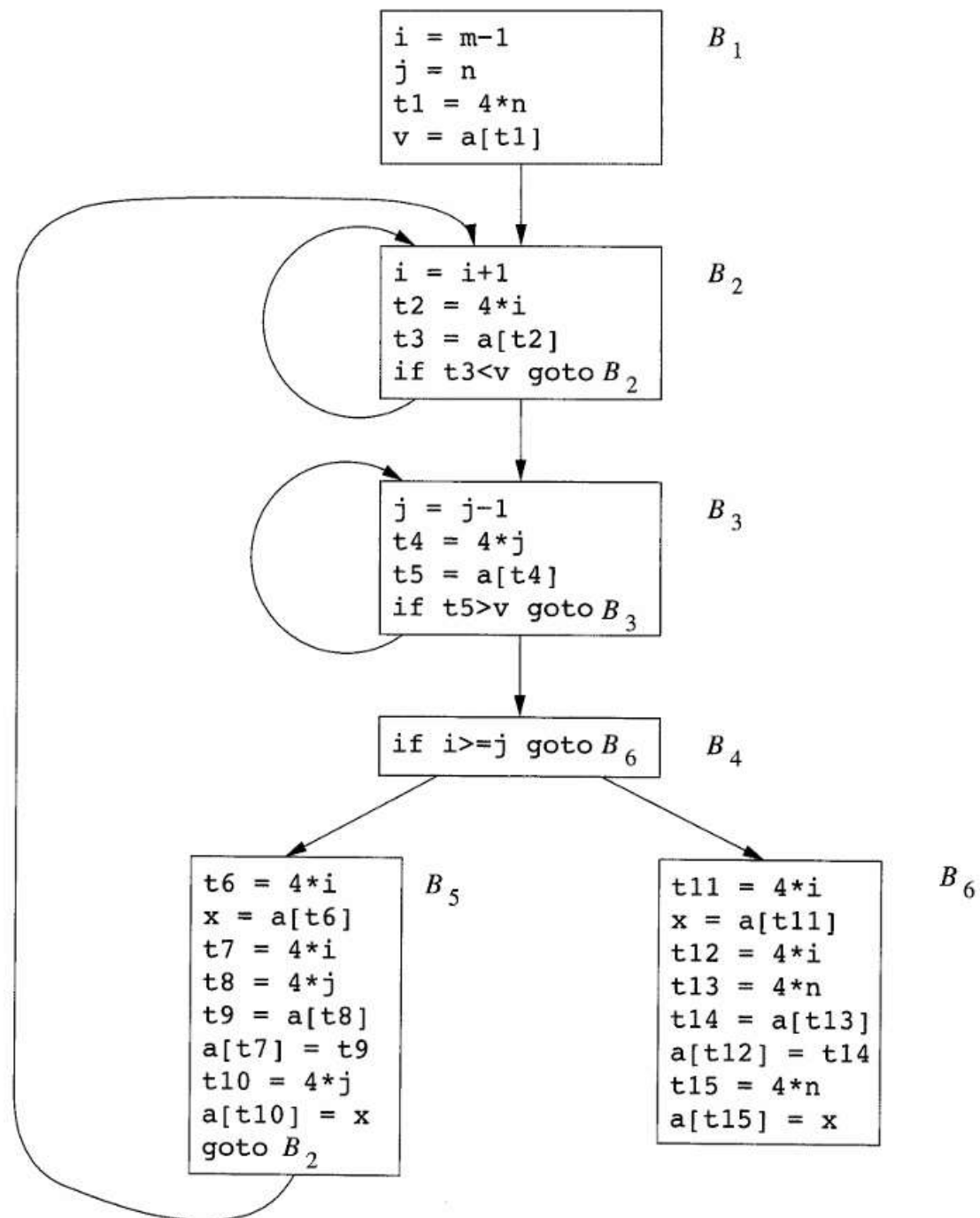
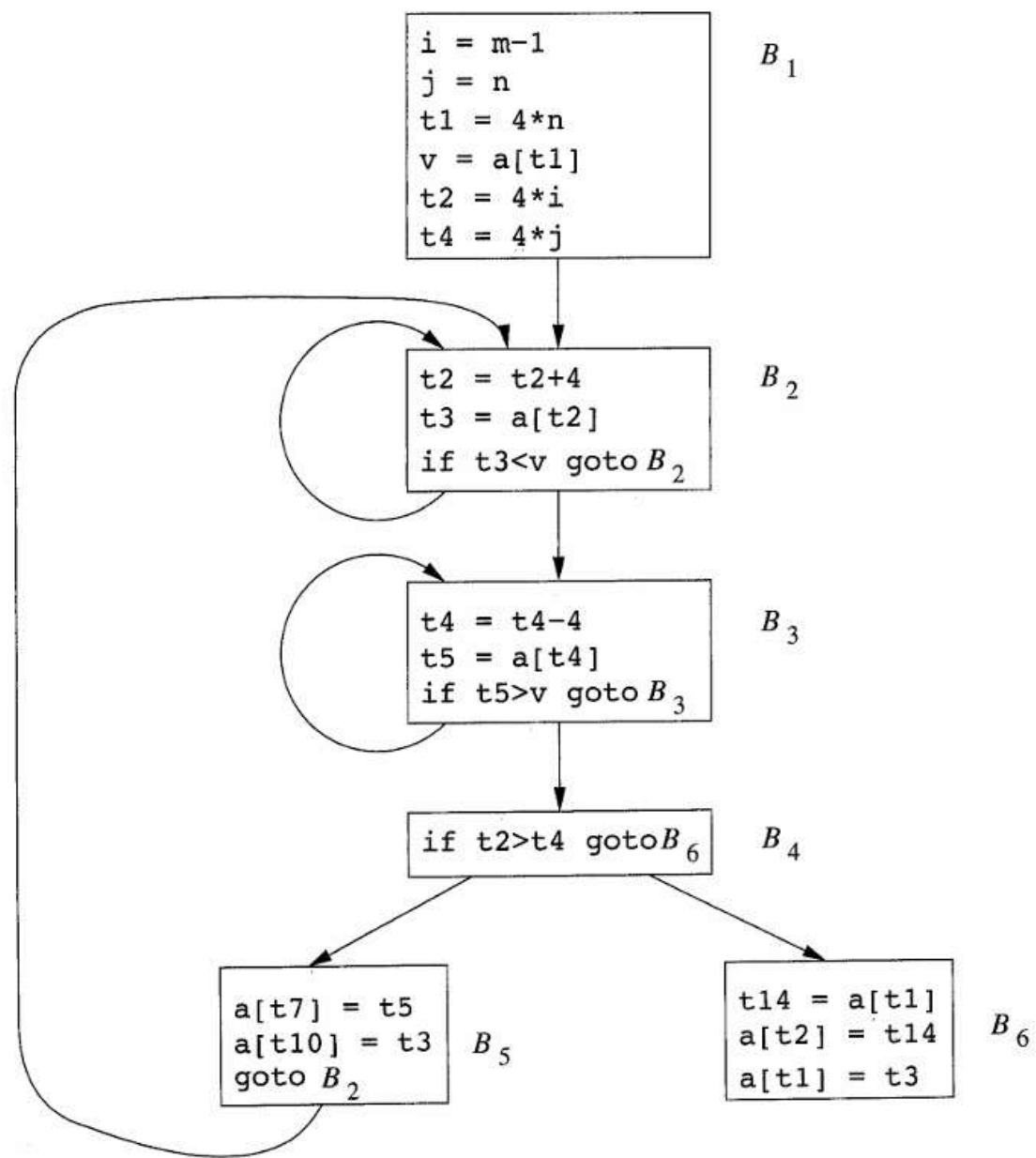
postorder traversal = a b t1 e c d t2 t3 t4

So, this three address code will require maximum 2 registers to complete its execution.

Three Address Code

```
(1)    i = m-1
(2)    j = n
(3)    t1 = 4*n
(4)    v = a[t1]
(5)    i = i+1
(6)    t2 = 4*i
(7)    t3 = a[t2]
(8)    if t3<v goto (5)
(9)    j = j-1
(10)   t4 = 4*j
(11)   t5 = a[t4]
(12)   if t5>v goto (9)
(13)   if i>=j goto (23)
(14)   t6 = 4*i
(15)   x = a[t6]
```

```
(16)   t7 = 4*i
(17)   t8 = 4*j
(18)   t9 = a[t8]
(19)   a[t7] = t9
(20)   t10 = 4*j
(21)   a[t10] = x
(22)   goto (5)
(23)   t11 = 4*i
(24)   x = a[t11]
(25)   t12 = 4*i
(26)   t13 = 4*n
(27)   t14 = a[t13]
(28)   a[t12] = t14
(29)   t15 = 4*n
(30)   a[t15] = x
```

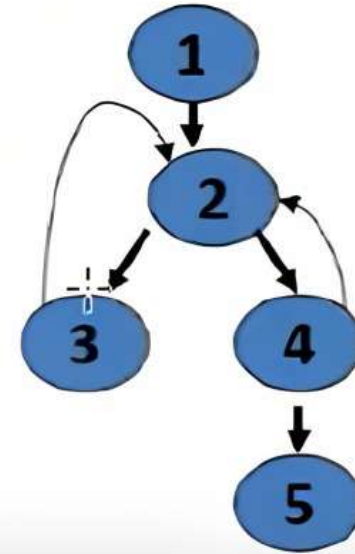


Loops in Flow Graphs

Control Flow Graph (CFG)

- **Control Flow Graph**
 - A graph which may contain **loops**, known as strongly connected components
- **Loop**
 - A directed graph whose nodes can reach all other nodes along some path
 - **Goto** statements can create any loops whereas **break** statements creates add-on exits

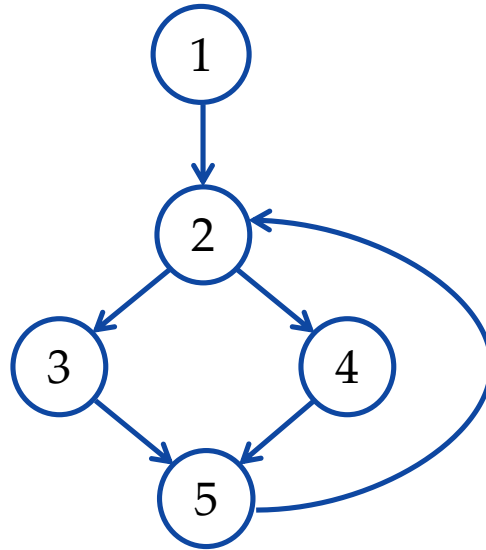
Loops



2 proper loops, one unstructured loop
Loop1: 2, 3; Loop2: 2, 4; Loop3: 2, 3, 4

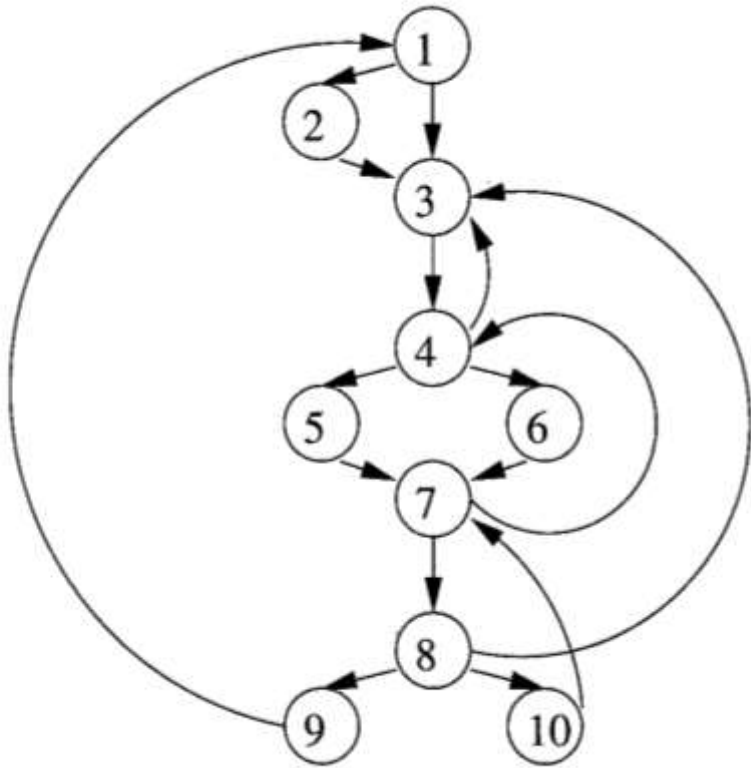
Dominators

- In a flow graph, a **node 'd'** is said to **dominate node '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.
- Every node dominates itself.



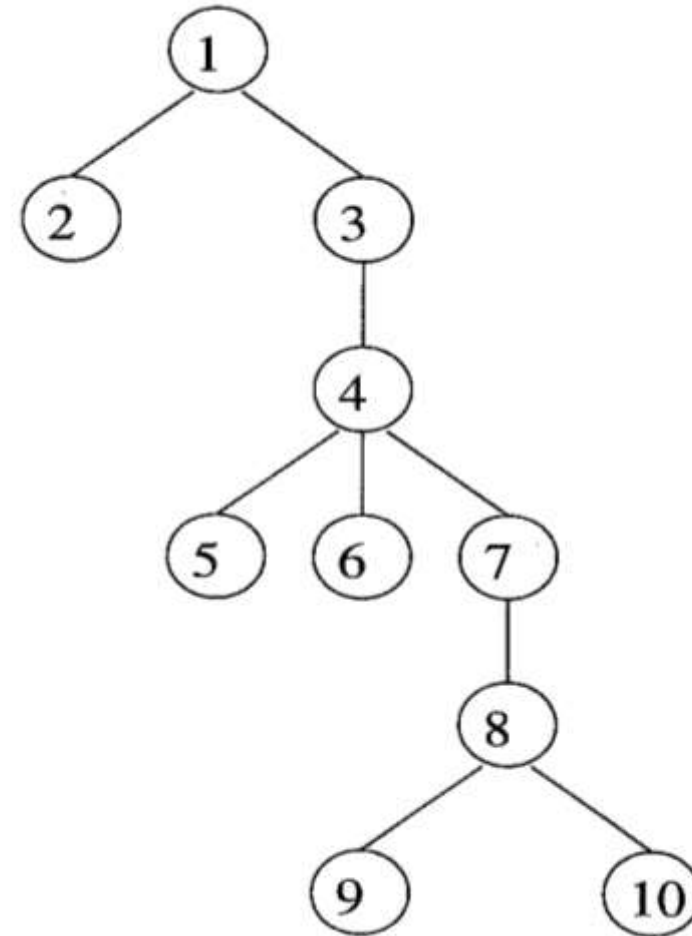
- Node 1 is initial node and it dominates every node as it is initial node.
- Node 2 dominates 3, 4 and 5.

Cont.,



A flow graph

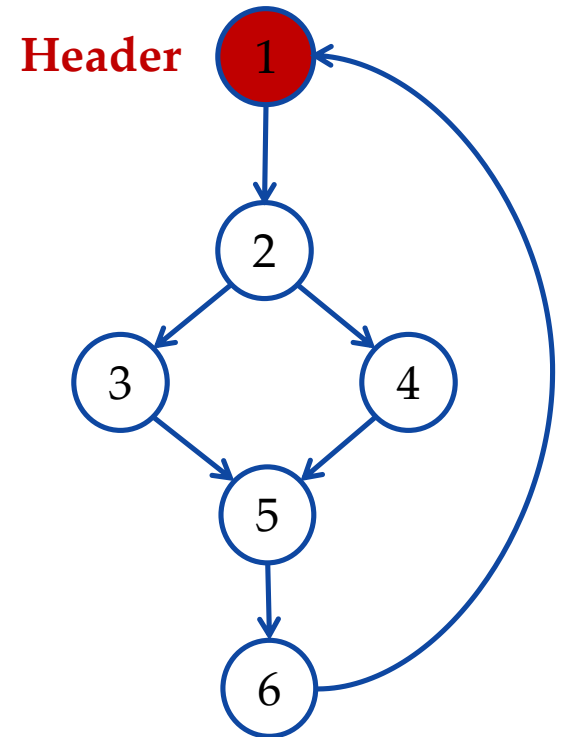
Dominator tree



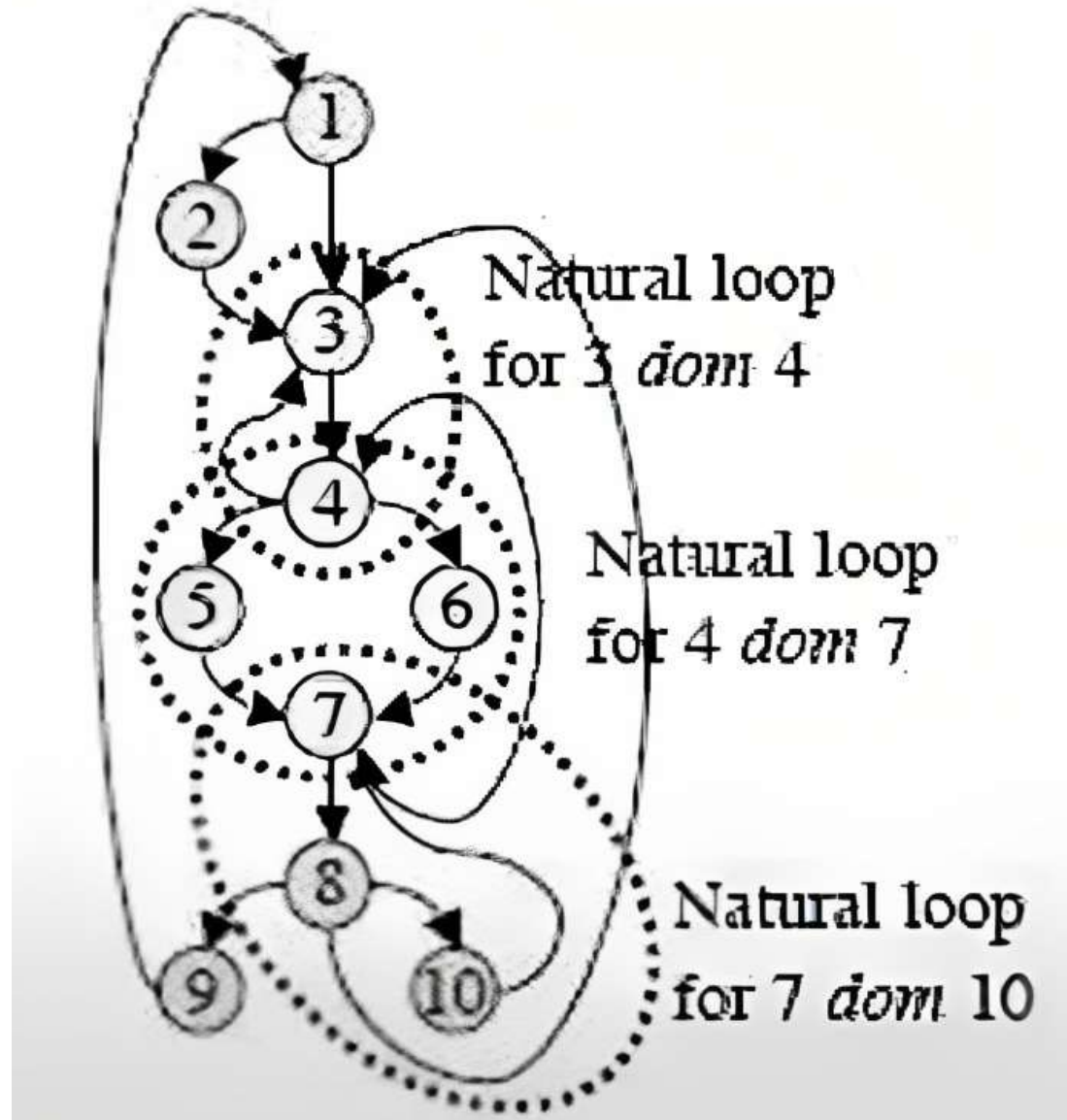
Natural Loops

- There are two essential properties of natural loop:
 1. A loop must have single entry point, called the **header**. This point dominates all nodes in the loop.
 2. There must be at least one way to iterate loop i.e. at least one path back to the loop header.
- Back edge is an edge $a \rightarrow b$ whose head, b dominates its tail, a
- Given a back edge $n \rightarrow d$, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d .
- Node d is the header of the loop.

$6 \rightarrow 1$ is natural loop.

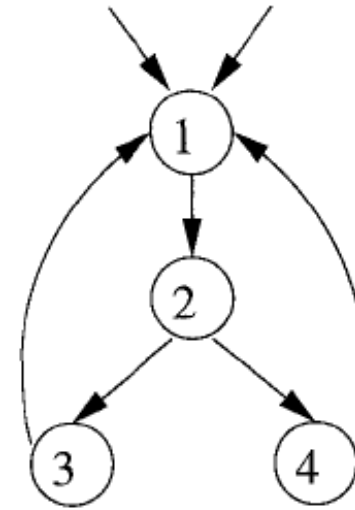
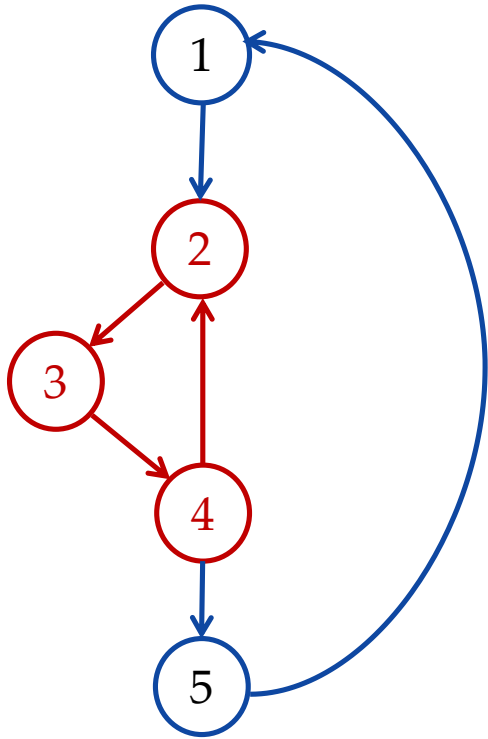


Example



Inner Loops

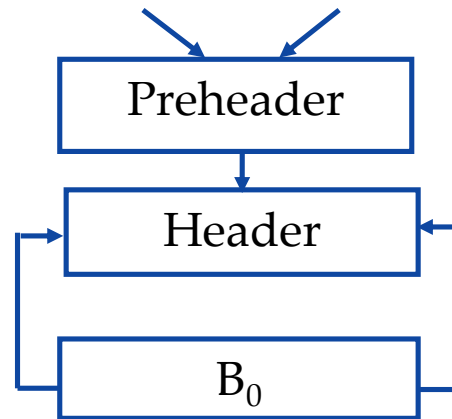
- The inner loop is a loop that contains no other loop.
- Here the inner loop is $4 \rightarrow 2$ that mean edge given by 2-3-4.



Two loops with the same header

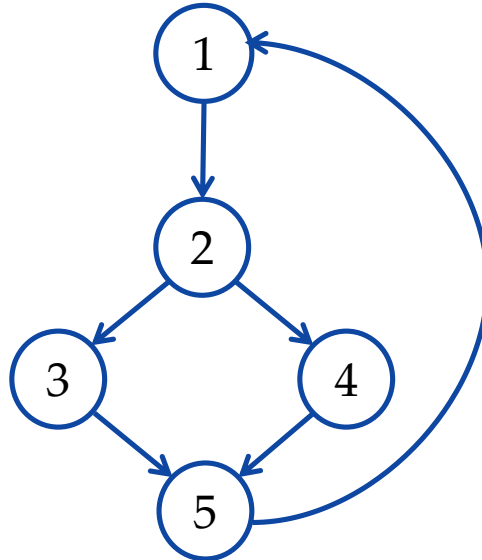
Preheader

- Several loop transformation require us to move statements “before the header”. [Ex: code motion, strength reduction, ...]
- Therefore, begin treatment of a loop L by creating a new block, called the preheader.
- The preheader has only the header as successor.



Reducible Flow Graph

- The reducible graph is a flow graph, if and only if we can partition the edges into two disjoint groups, in which often called the forward edges and backward edges.
- These edges have following properties,
 1. The **forward edge forms an acyclic graph** in which every node has to be reached from the initial node.
 2. The **back edges** are such edges whose head dominates their tail.



Non-reducible Flow Graph

- A non reducible flow graph is a flow graph in which:
 1. There are no back edges.
 2. Forward edges may produce cycle in the graph.

