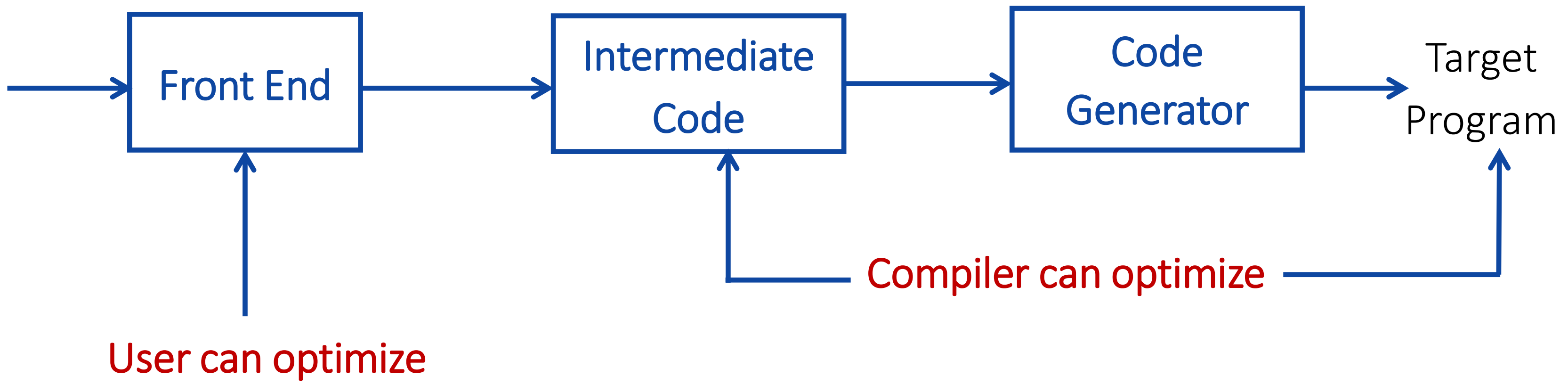
Module 5 – Code Optimization

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 : 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;

  Area = pi * r * r;

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

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.

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

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];
}
```

$\longrightarrow$

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

$\Rightarrow$

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

Copy Propagation

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

~~x = pi;~~

area = x * r * r;

$$\Big\downarrow$$

area = pi * r * r;

$$\begin{array}{lcl} x=a & & x=a \\ y=x*b & \Longrightarrow & y=a*b \\ z=x*c & & z=a*c \end{array}$$

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.
- Example:

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

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

Peephole optimization

- Peephole optimization is a simple and effective technique for locally improving target code.
- Replaces short sequences of target instructions by a shorter or faster sequence.
- This technique is applied to improve the performance of the target program by examining the short sequence of target instructions (called the peephole) and replacing these instructions by shorter or faster sequence whenever possible.
- Peephole is a small, moving window on the target program.

Redundant Loads & Stores

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

```
MOV R0,x
MOV x,R0
```

- We can eliminate the second instruction since x is in already R0.

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
```

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^2 is cheaper than $x * x$.
- Similarly addition and subtraction are cheaper than multiplication and division. So we can add effectively equivalent addition and subtraction for multiplication and division.

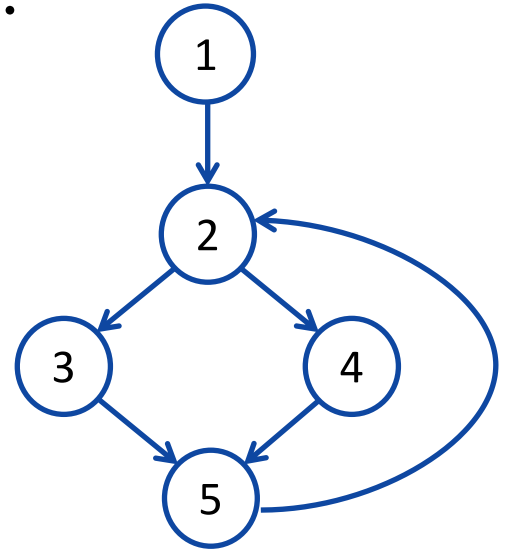
Machine idioms

- The target instructions have equivalent machine instructions for performing some operations.
- Hence 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.

Loops in Flow Graphs

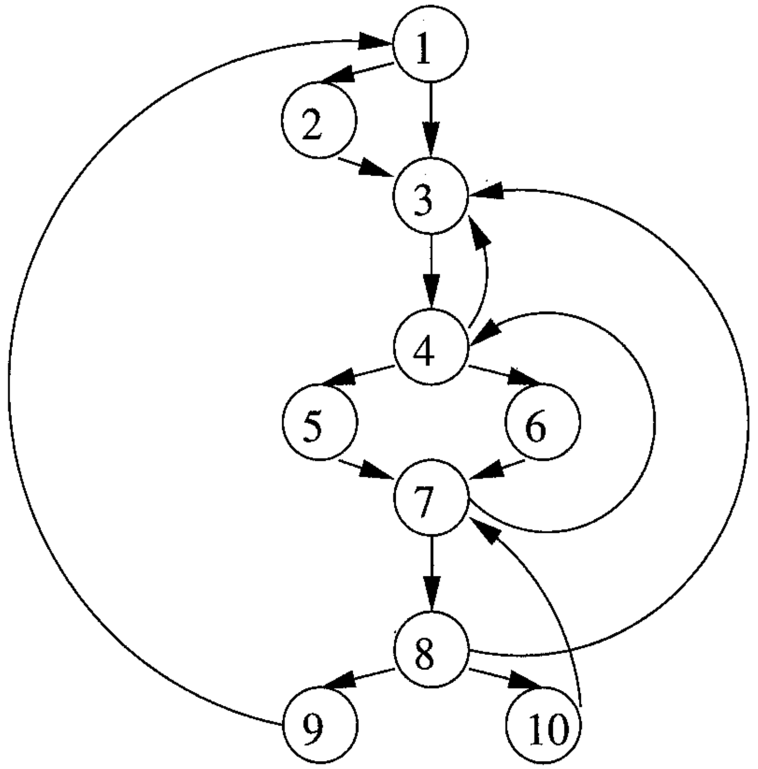
Dominators

- In a flow graph, a node ‘d’ is said to be dominates 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.



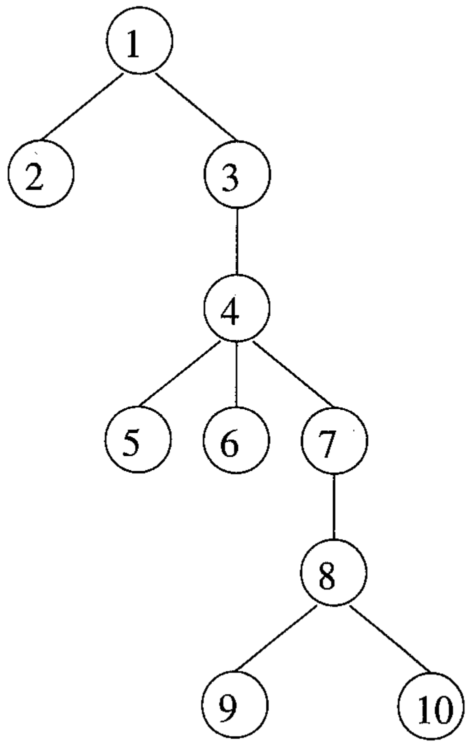
- Node 1 is initial node and it dominates every node as it is initial node.
- Node 2 dominates 3, 4 and 5.

Cont.,



A flow graph

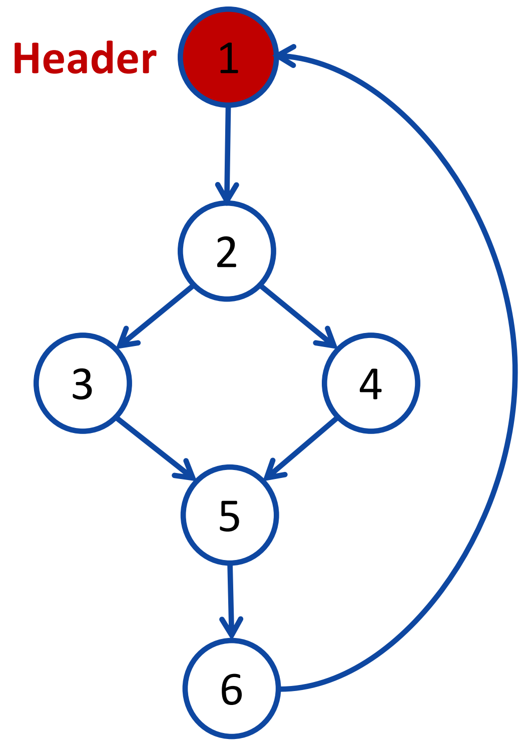
Dominator tree



Natural Loops

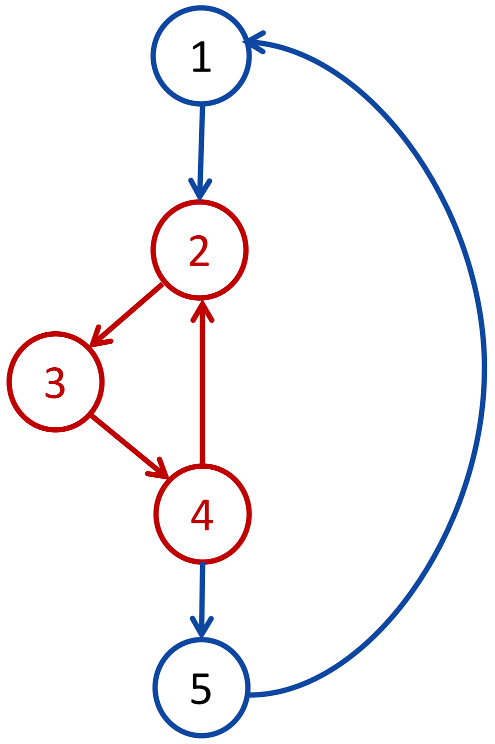
- There are two essential properties of natural loop:
    1. A loop must have single entry point, called the header. This point dominates all nodes in the loop.
    2. There must be a back edge that enters the loop header. There must be at least one way to iterate loop.
    3. Given a back edge n + d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d.
    4. Node d is the header of the loop.

6→1 is natural loop.



Inner Loops

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



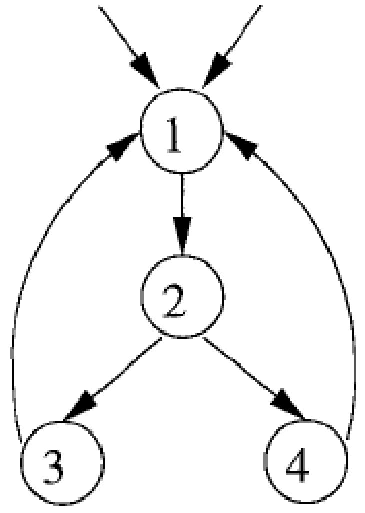
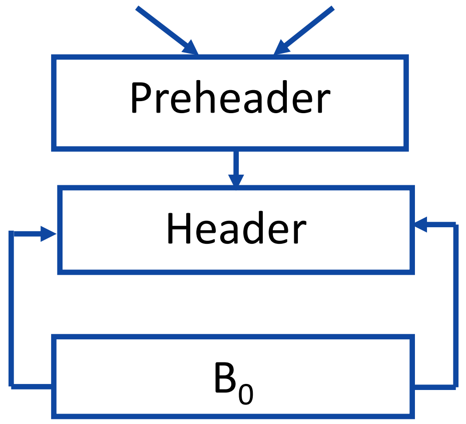


Figure 9.46: Two loops with the same header

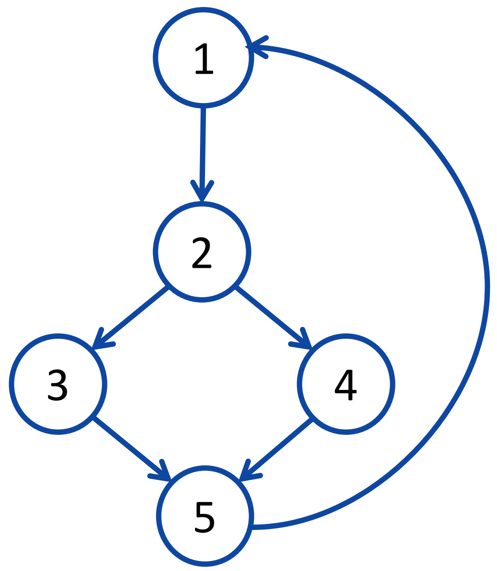
Preheader

- Several transformation require us to move statements "before the header".
- Therefore begin treatment of a loop *L* by creating a new block, called the preheader.
- The preheader has only the header as successor.



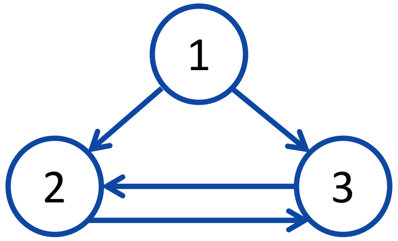
Reducible Flow Graph

- 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.



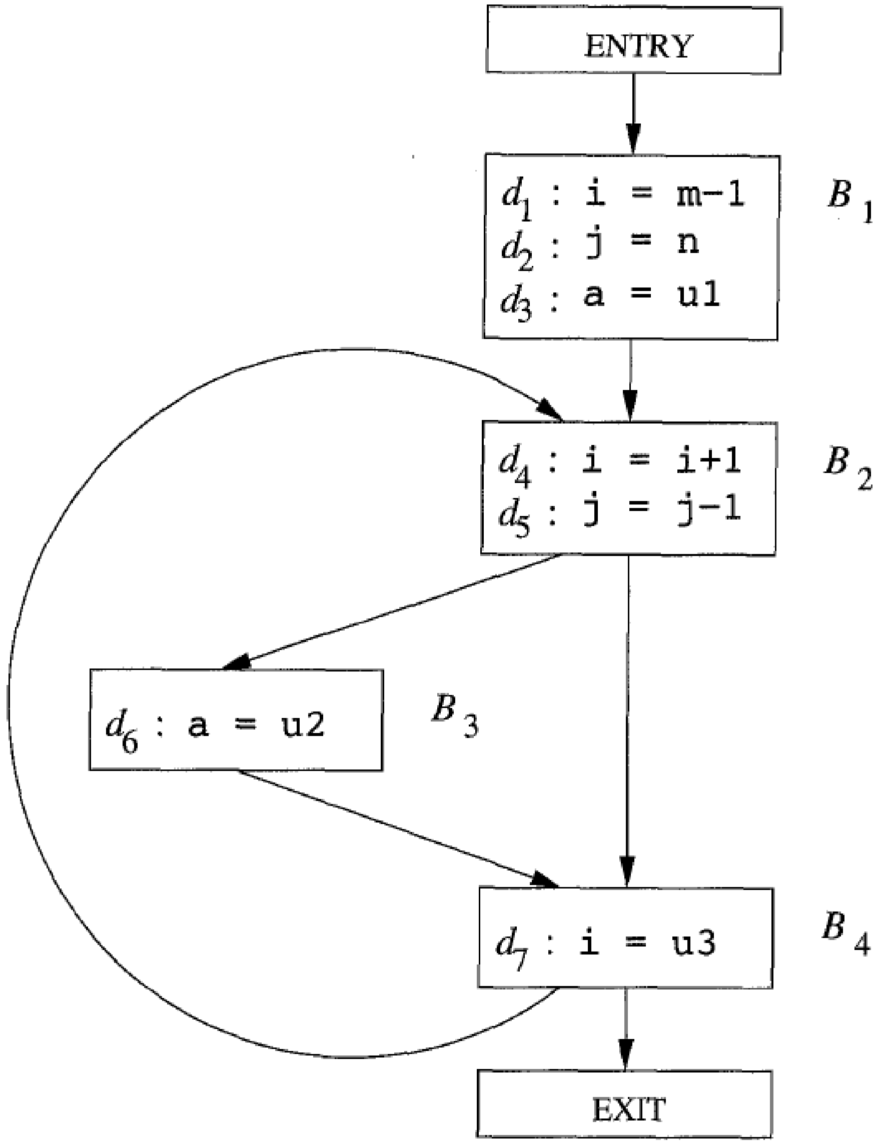
Global Data Flow Analysis

Global Data Flow Analysis

- 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,

$$\boldsymbol{out}\ [\boldsymbol{S}] = \boldsymbol{gen}[\boldsymbol{S}]\ \boldsymbol{U}\ (\boldsymbol{in}[\boldsymbol{S}] - \boldsymbol{kill}[\boldsymbol{S}])$$

- 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”.



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

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

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

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

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

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

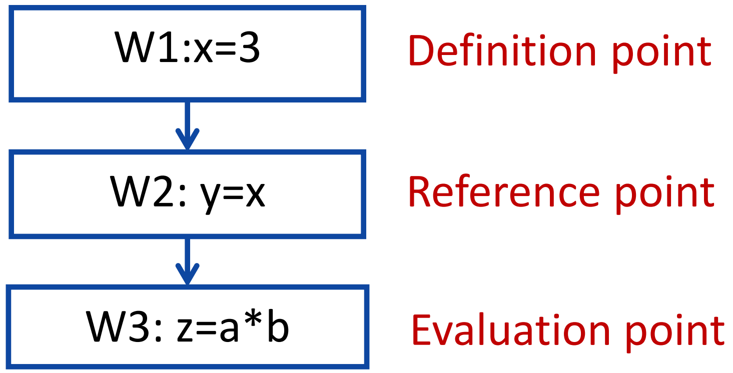
$gen_{B_4} = \{ d_7 \}$

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

Dataflow Properties

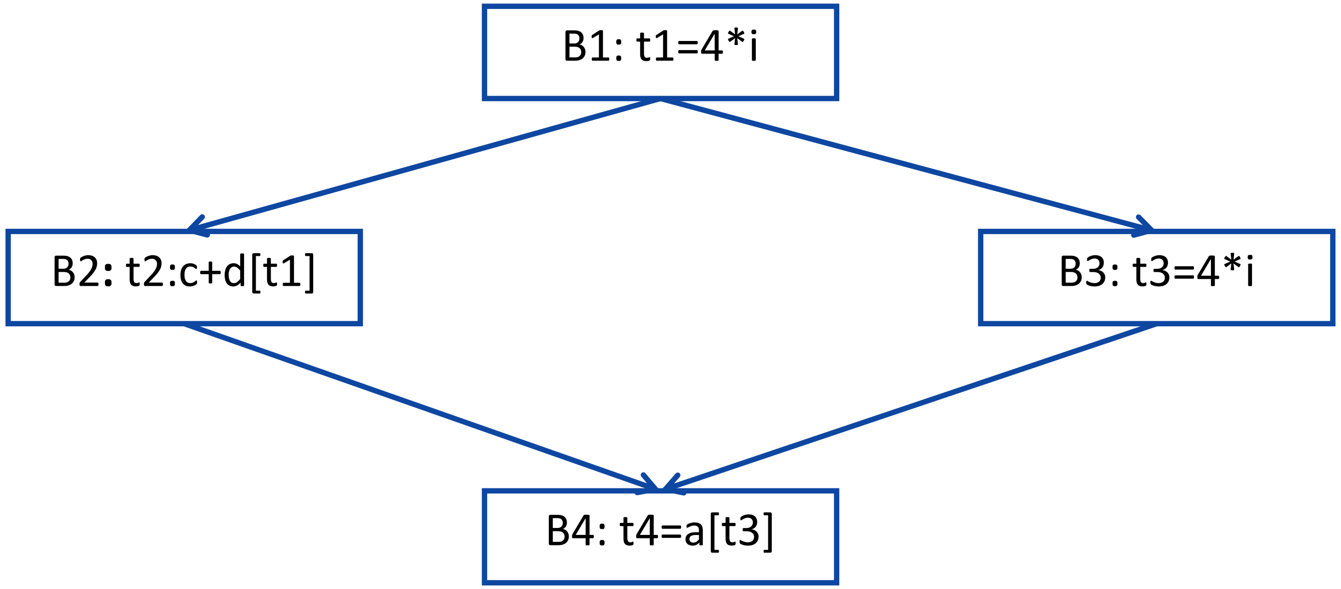
Data Flow Properties

- 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.



Available Expression

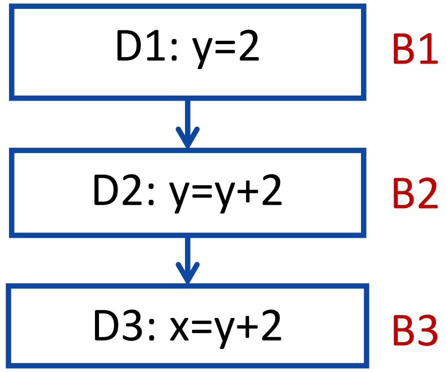
- An expression $x + y$ is available at a program point w if and only if along all paths are reaching to w.
  1. The expression $x + y$ is said to be available at its evaluation point.
  2. Neither of the two operands get modified before their use.



- Expression $4 * i$ is the available expression for B_2, B_3 and B_4 because this expression has not been changed by any of the block before appearing in $B_{4.}$

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 $B2$.

Live variable

- A live variable x is live at point p if there is a path from p to the exit, along which the value of x is used before it is redefined.
- Otherwise the variable is said to be dead at the point.
- Example:

```
b = 3
c = 5
a = f(b * c)
```

- The set of live variables are {b, c} because both are used in the multiplication on line 3.

Busy Expression

- An expression e is said to be busy expression along some path $p_i..p_j$ if and only if an evaluation of e exists along some path $p_i \ldots p_j$ and no definition of any operand exist before its evaluation along the path.

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.

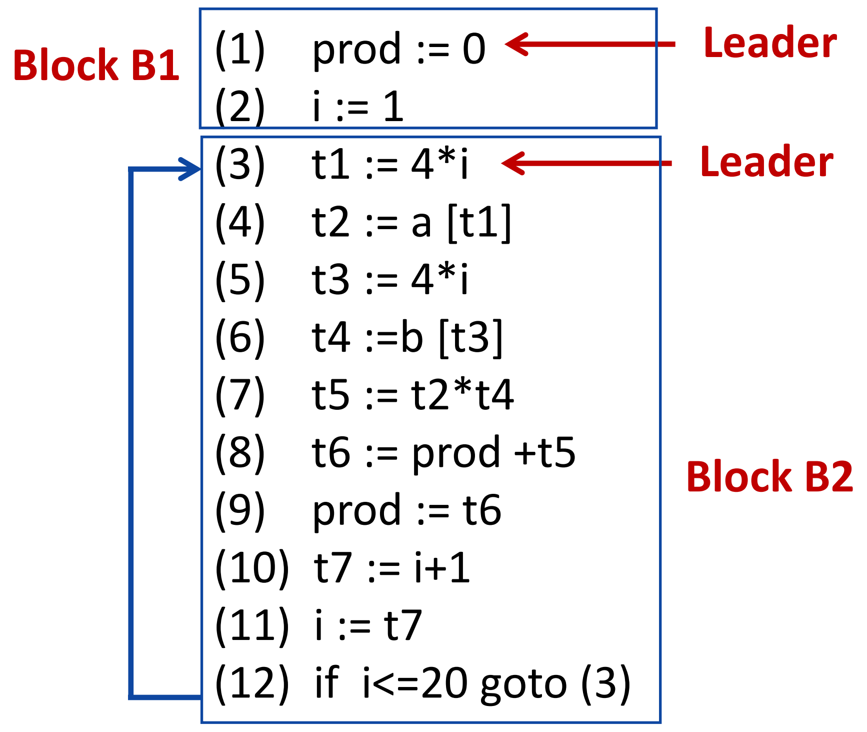
Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1. We first determine the set of **leaders**, for that we use the following rules:
   - I. The first statement is a leader.
   - II. Any statement that is the target of a conditional or unconditional goto is a leader.
   - III. 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.

Example: Partition into basic blocks

```
begin
        prod := 0;
                i := 1;
          do
                prod := prod + a[t1] * b[t2];
                i := i+1;
          while  i<= 20
end
```

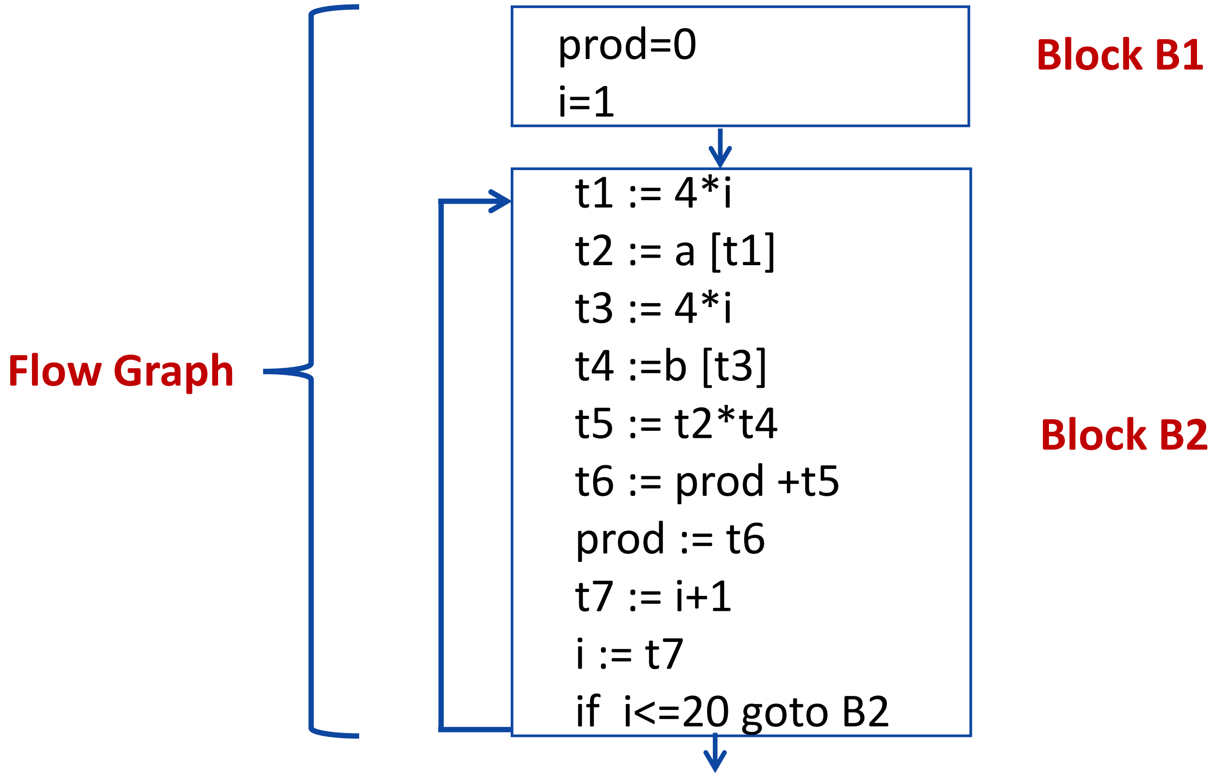


```
(1)    prod := 0
(2)    i := 1
(3)    t1 := 4*i
(4)    t2 := a [t1]
(5)    t3 := 4*i
(6)    t4 :=b [t3]
(7)    t5 := t2*t4
(8)    t6 := prod +t5
(9)    prod := t6
(10)  t7 := i+1
(11)  i := t7
(12)  if  i<=20 goto (3)
```

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:



```
prod=0
i=1
```

```
t1 := 4*i
t2 := a [t1]
t3 := 4*i
t4 :=b [t3]
t5 := t2*t4
t6 := prod +t5
prod := t6
t7 := i+1
i := t7
if  i<=20 goto B2
```

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:
    1. Structure preserving transformation
    2. Algebraic transformation

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

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
```

Dead-code elimination

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

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.

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.

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.

DAG Representation of Basic Block

Algorithm: DAG Construction

We assume the three address statement could of following types:

Case (i) x:=y op z

Case (ii) x:=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 node(y). Similarly if z is undefined create a node (z)
- Step 2:

  Case(i) create a node(op) whose left child is node(y) and 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 node(y).

  Case (iii) node n will be node(y).
- Step 3: Delete x from list of identifiers for node(x). Append x to the list of attached identifiers for node n found in 2.

Problems

$a=b+c$

$b=a-d$

$c=b+c$

$d=a-d$

DAG Representation of Basic Block

Example:

(1) t1 := 4*i

(2) t2 := a [t1]

(3) t3 := 4*i

(4) t4 :=b [t3]

(5) t5 := t2*t4

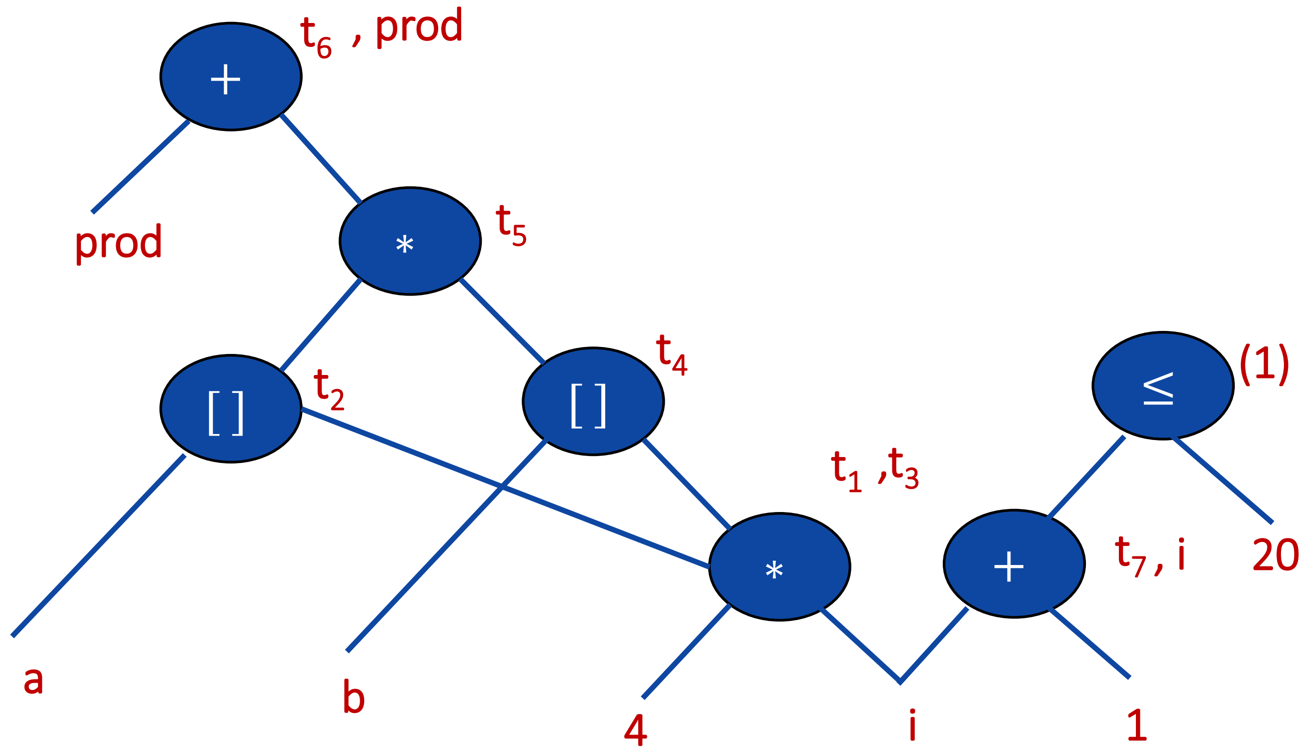
(6) t6 := prod +t5

(7) prod := t6

(8) t7 := i+1

(9) i := t7

(10) if i<=20 goto (1)



Practice Problem

- Problem 1:

  a=b + c

  b=b - d

  c=c + d

  e=b + c

- Problem 2:

  a=b*-c + b* -c

Applications of DAG

- The DAGs are used in following:
  1. Determining the common sub-expressions.
  2. Determining which names are used inside the block and computed outside the block.
  3. Determining which statements of the block could have their computed value outside the block.
  4. Simplifying the list of quadruples by eliminating the common sub-expressions and not performing the assignment of the form x:=y unless and until it is a must.

Generation of Code from DAGs

Generation of Code from DAGs

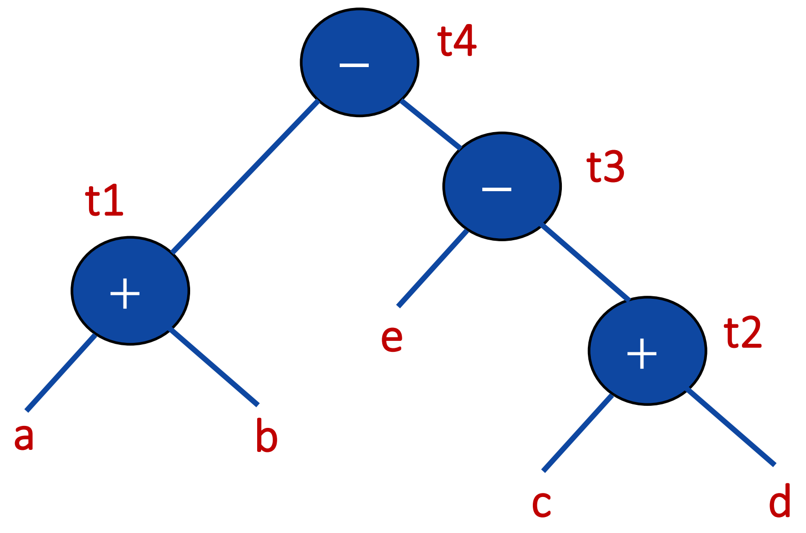
- Methods generating code from DAGs are:
    1. Rearranging Order
    2. Heuristic ordering
    3. Labeling algorithm

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

$$\xrightarrow{\text{Re-arrange}}$$

```
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

```
t2:=c+d
t3:=e-t2
t1:=a+b
t4:=t1-t3
```

Three Address Code

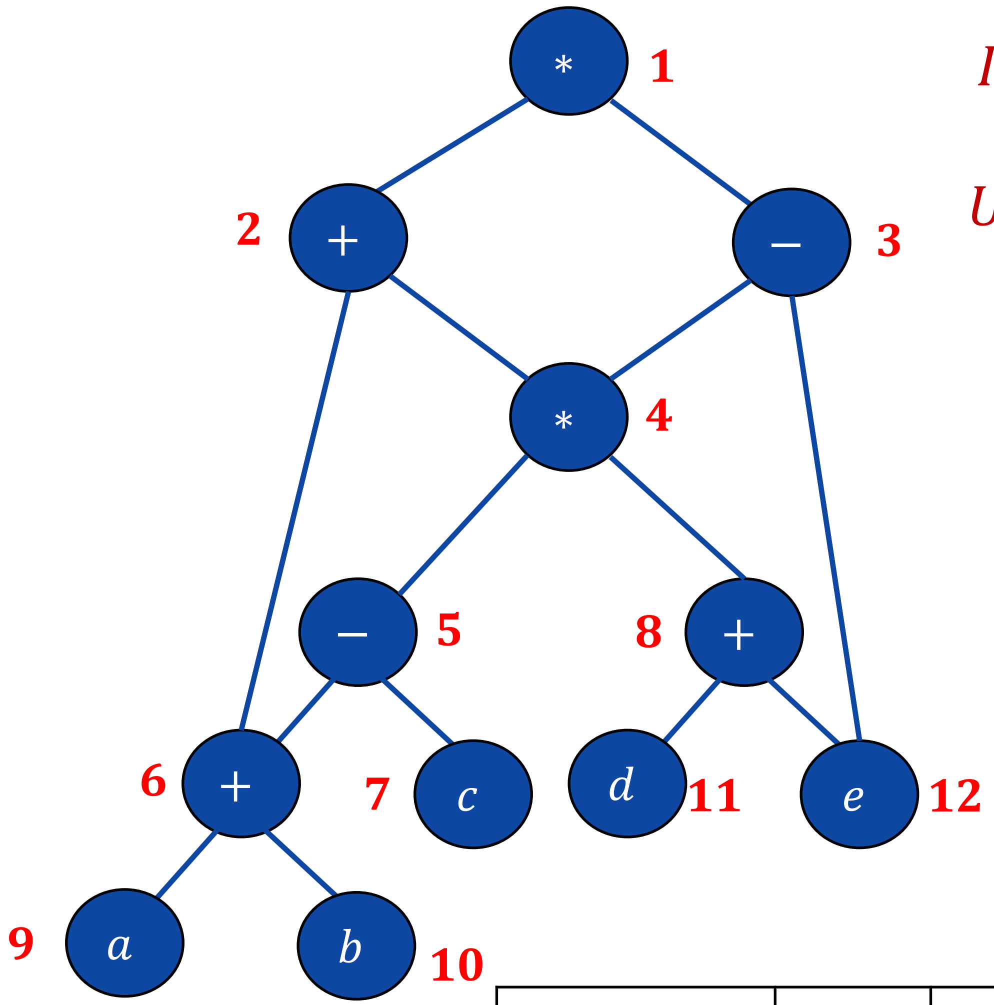
```
MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4
```

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

Leftchild of 1 = 2

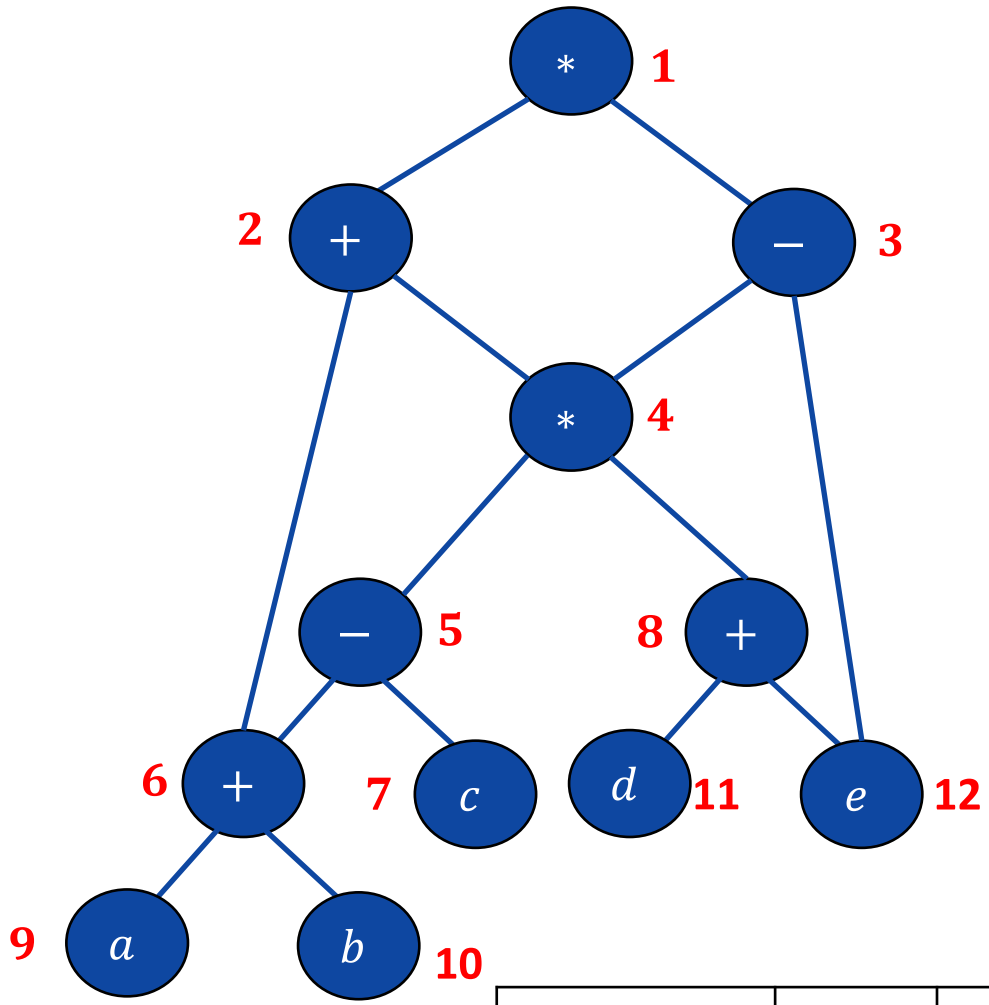
Leftchild of 2 = 6

Parent 1 is listed so list 2

Parent 5 is not listed so can't list 6

Listed Node							

Example: Heuristic Ordering



$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$

$Unlisted\ nodes = \cancel{1}\ \cancel{2}\ \cancel{3}\ \cancel{4}\ 5\ 6\ 8$

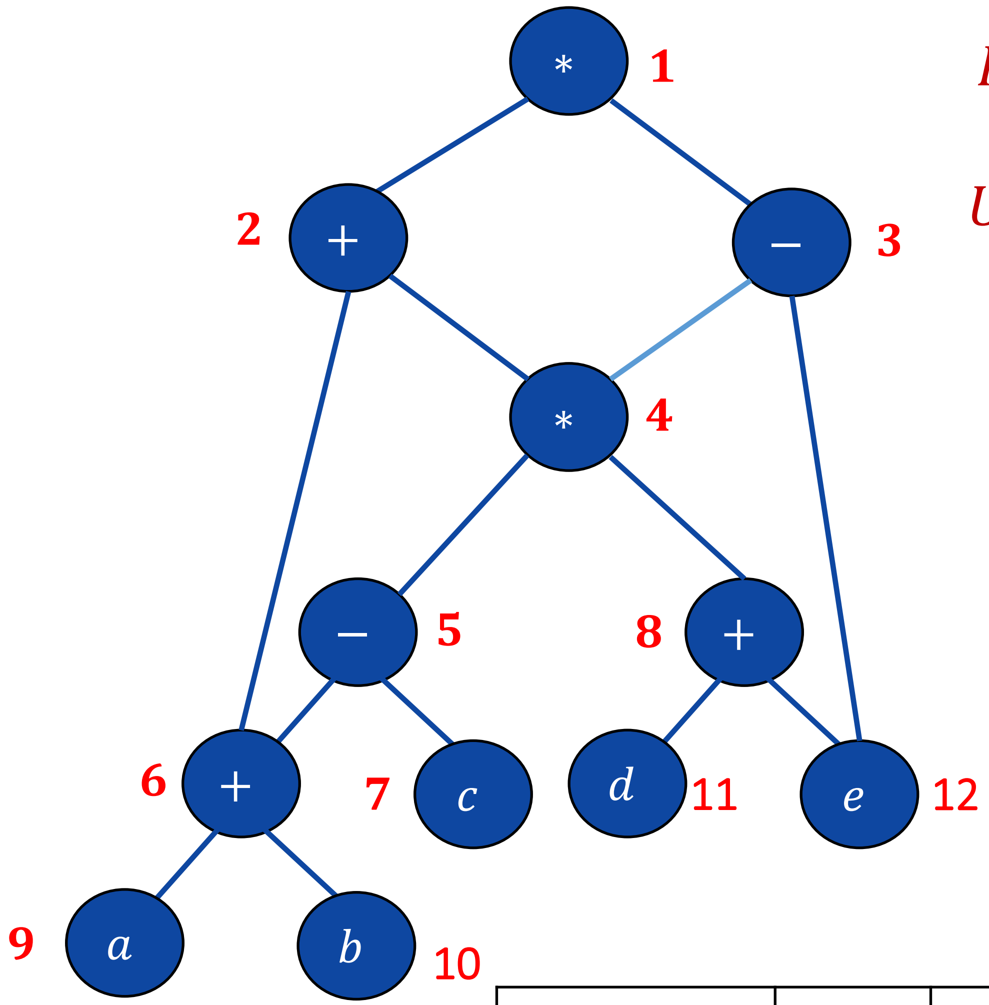
Pick up an unlisted node,
whose parents have been listed

Right child of 3 = 4 / Left child of 1 = 2

Parent 2, 3 are listed so list 4 / Parent 1 is listed so list 3

Listed Node	1	2					

Example: Heuristic Ordering



$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$

$Unlisted\ nodes = \cancel{1}\ \cancel{2}\ \cancel{3}\ \cancel{4}\ \cancel{5}\ \cancel{6}\ 8$

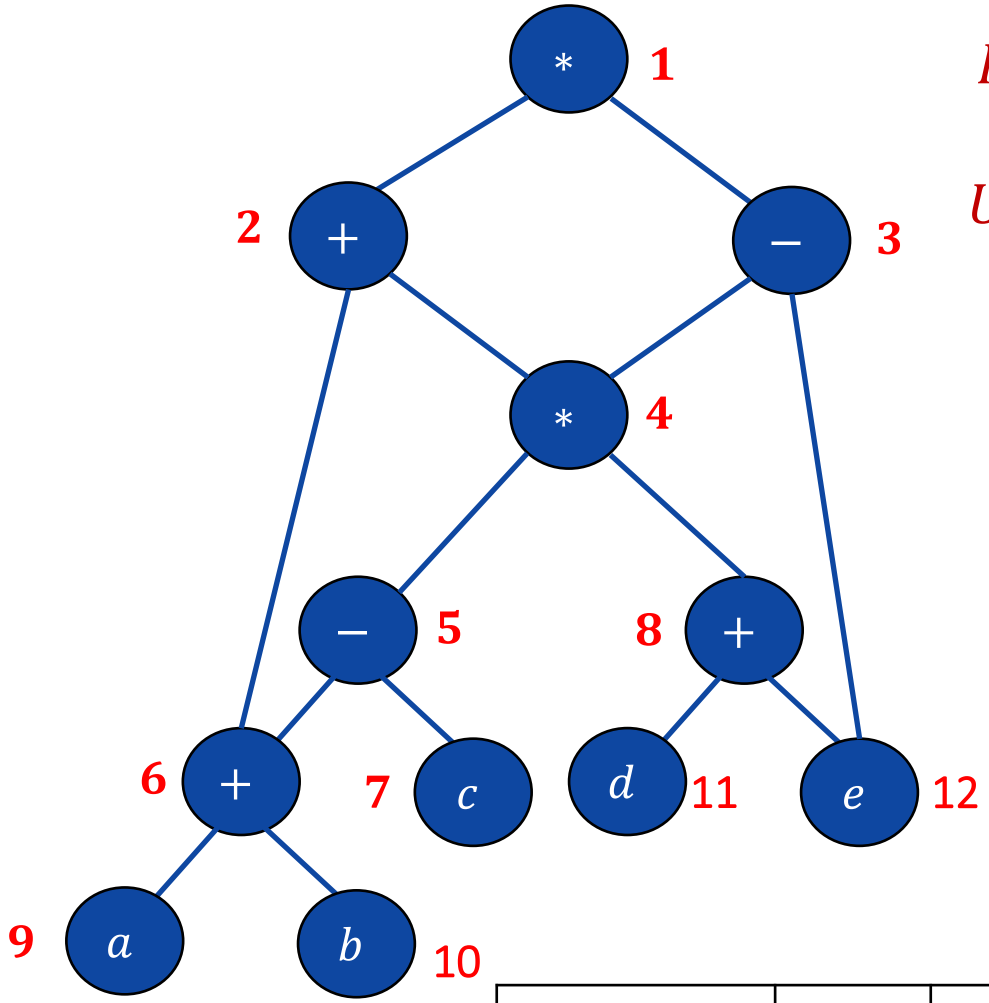
**Pick up an unlisted node,
whose parents have been listed**

Leftchild of 4 = 5
Leftchild of 5 = 6

Parent 4 is listed so list 5
Parent 2,5 are listed so list 6

Listed Node	**1**	**2**	**3**	**4**			

Example: Heuristic Ordering



$Interior\ nodes = 1\ 2\ 3\ 4\ 5\ 6\ 8$

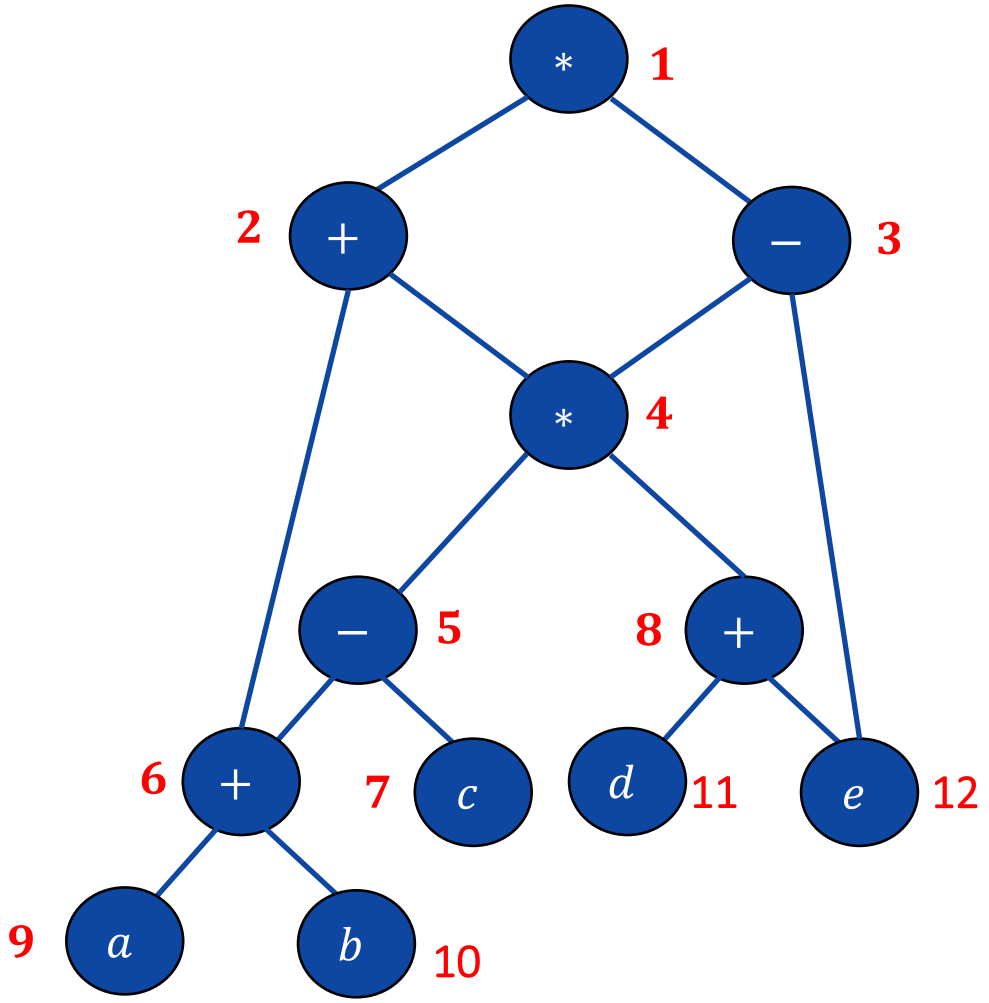
$Unlisted\ nodes = \cancel{1}\ \cancel{2}\ \cancel{3}\ \cancel{4}\ \cancel{5}\ \cancel{6}\ \cancel{8}$

Pick up an unlisted node,
whose parents have been listed

Rightchild of 4 = $\mathbf{8}$
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

$$\left.\begin{array}{l} t8=d+e \\ t6=a+b \\ t5=t6-c \\ t4=t5*t8 \\ t3=t4-e \\ t2=t6+t4 \\ t1=t2*t3 \end{array}\right\}$$

Optimal Three Address code

Labeling Algorithm

- The labeling algorithm generates the optimal code for given expression in which minimum registers are required.
- 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,

$$Label\ (n)\ =\ max(L1, L2)\ if\ L1\ not\ equal\ to\ L2$$
$$Label(n)\ =\ L1 + 1\ if\ L1 = L2$$

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

Example: Labeling Algorithm

t1:=a+b

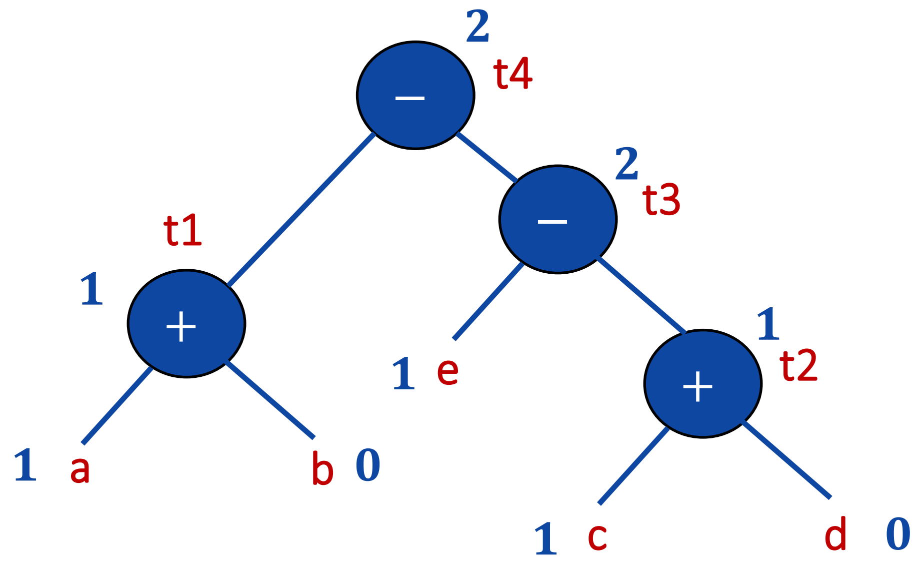
t2:=c+d

t3:=e-t2

t4:=t1-t3

Three Address Code

$$Label(n) = \begin{cases} Max(l1, l2) & if\ l1 \neq l2 \\ L1 + 1 & if\ l1 = l2 \end{cases}$$



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