# SSA-based Optimization (Objectives)

- Given an CFG with $\phi$-nodes, the student will be able to perform global common subexpression elimination (redundancy elimination) using a dominator-based approach.
- Given a CFG in SSA form, the student will be able to perform global constant propagation.
- Given a CFG in SSA form, the student will be able to perform strength reduction by finding loop, calculating loop invariants, finding induction variables and then applying the strength reduction transformation.
- Given a CFG in SSA form, the student will be able to perform dead-code elimination.
- Given a CFG in SSA form, the student will be able to perform global value numbering.

# Dominator-based Global Common Subexpression Elimination

- A limited form of global CSE
  - used before dependence based optimization and other SSA-based optimizations
  - no code motion
  - redundancy found only along paths in the dominator tree
- In SSA all syntactically equivalent expression are semantically equivalent.
- Method:
  - keep a block structured table of available expression
    - StartBlock – add a scope in the expression table for this block.
    - EndBlock – remove the scope for the current block
  - perform CSE on the way back up the dominator tree while constructing SSA.

# Algorithm

```
OPTRENAME(b) {
  for each T0 = φ(T1,...,Tn) ∈ Φ(b)
    push NewName() on NameStack(T0)
  StartBlock(b)
  for each I ∈ b in execution order {
    for each T ∈ Operand(I)
      replace T by Top(NameStack(T))
    if I.expr() ∈ AVAIL { // insert if ∉AVAIL
      T = I.lval()
      push GetTarget(AVAIL,i) on NameStack(T)
      DEAD ∪= {I}
    }
```

# Algorithm

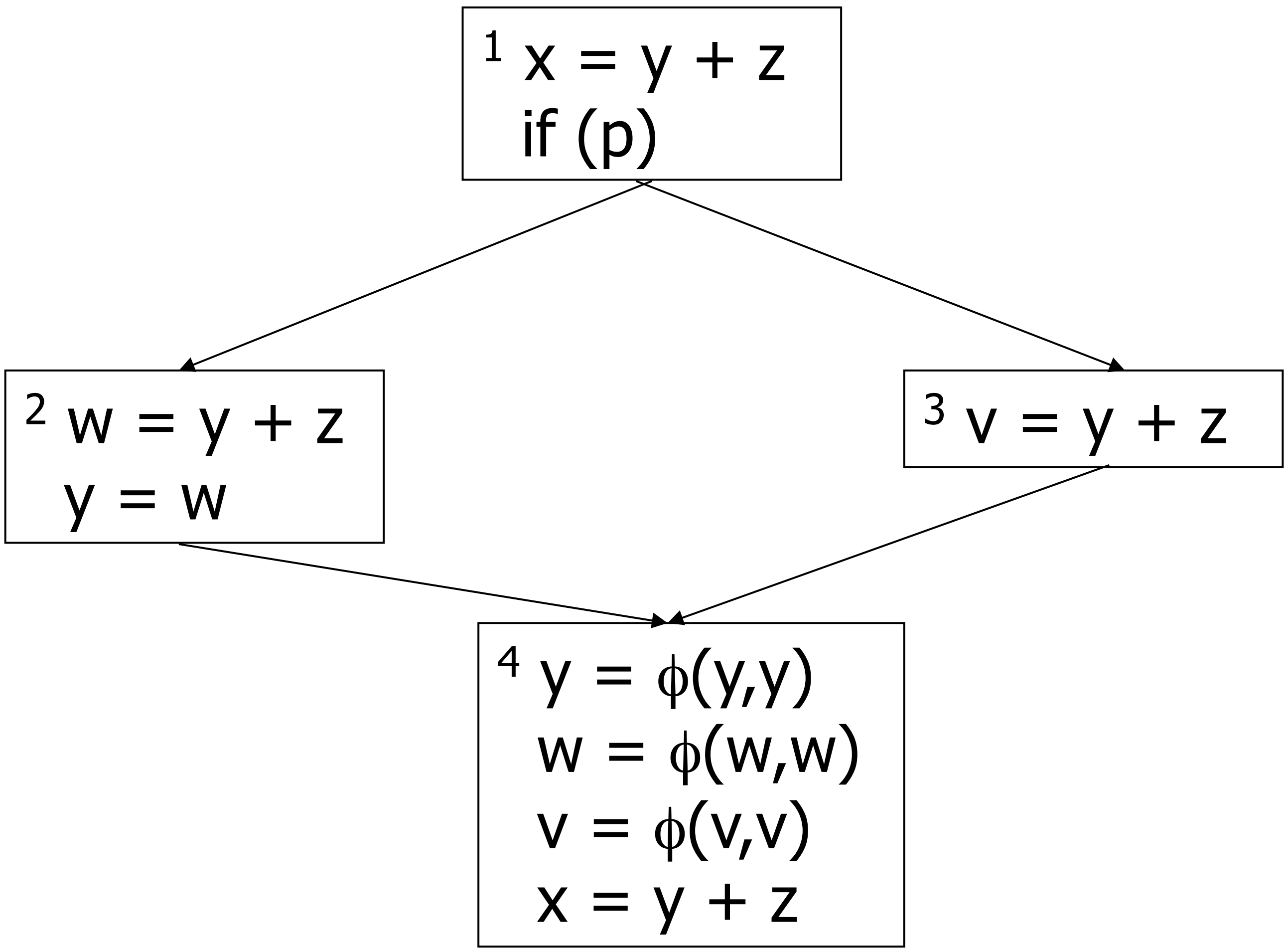
```
    else
      push NewName() on Top(NameStack(I.lval()))
  }
for each s ∈ succ(b) {
  j = WhichPredecessor(s,b)
  for each T0 = φ(T1,…,Tn) ∈ Φ(s)
    replace Tj with Top(NameStack(Tj))
}
for each c ∈ children(b)
  OPTRENAME(c)
```

# Algorithm

```
for each I ∈ b in reverse order {
  X = Pop(NameStack(I.lval()))
  if I ∈ DEAD
    remove I
  else
    replace I.lval() with X
}
```

for each $T_0 = \phi(T_1,\ldots,T_n) \in \Phi(b)$
  replace $T_0$ by Pop(NameStack($T_0$))
EndBlock(b)
}

# Example



# Constant Propagation

- Propagate constants globally on a sparse representation
  - cheaper than previous algorithm
- Incorporate the effects of branch folding
  - if a block cannot be reached, it will be ignored
- Meet operations occur at $\phi$-nodes

# Algorithm

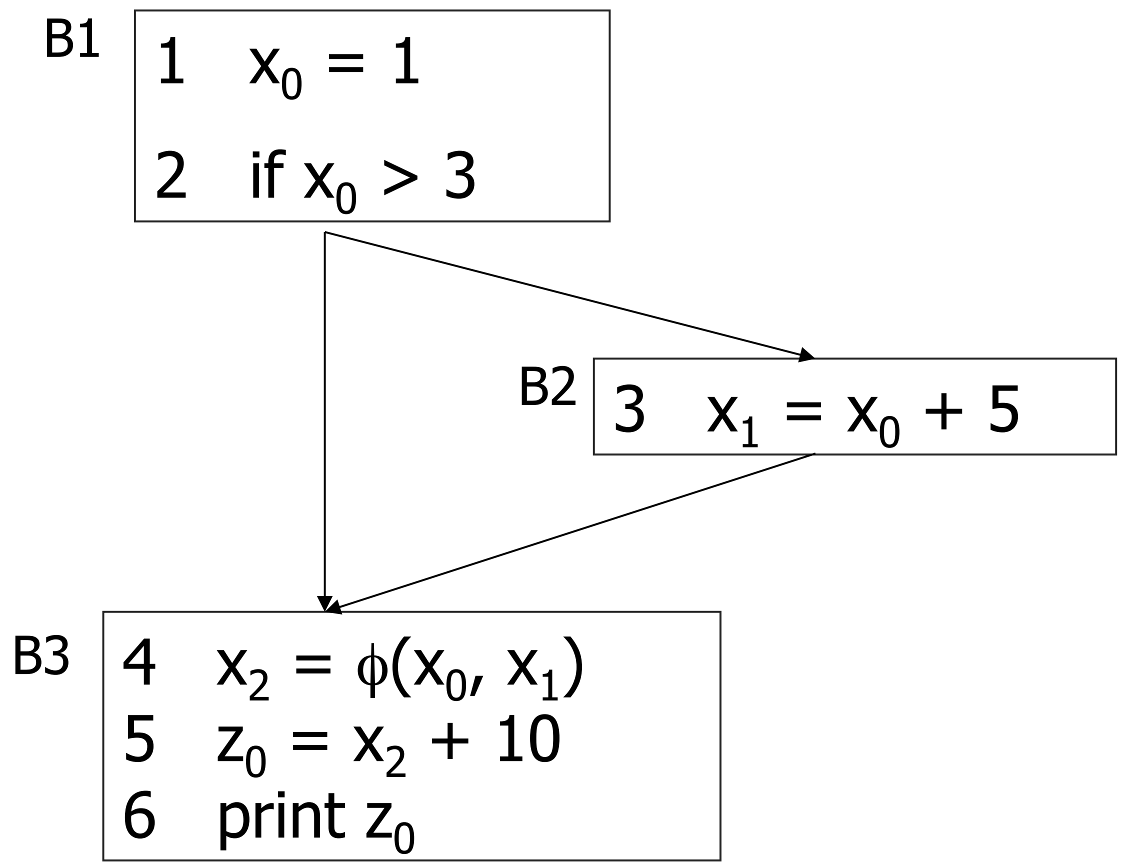
```
Procedure ConstProp {
    mark all edges in CFG not executable
    initialize all nodes in SSA Graph to ⊤
    Work = ∅;
    Visited = ∅;
    Blocks = {ENTRY}

  while Work ≠ ∅ ∧ Blocks ≠ ∅ {
    while Work ≠ ∅ {
      take I from Work
      EvalInstruction(I)
    }
    while Blocks ≠ ∅ {
      take b from Blocks
      for each I ∈ Φ(b) {
        EvalInstruction(I)
      if b ∉ Visited {
        Visited ∪= {b}
        for each I ∈ b
          EvalInstruction(I)
      }
    }
  }
}
```

# Algorithm

```
EvalInstruction(I) {
    if I is an arithmetic instruction or ϕ-node {
       evaluate I
       if result lowered
         for each j ∈ Uses(I.lval()) {
            propagate result           // may to non-executable block
            if j.Block() ∈ Visited  // only for executable block
              Work ∪= {j}
         }
      }
    else if I is a branch
       for each possible destination of I, S
         if edge from I.block() to S is not executable {
           mark it as executable
             Blocks ∪= {S}
         }
}
```

# Example



# Strength Reduction

- Replace multiplication of a regularly varying variable by a constant in a loop with an addition.
- Example

```
i = 1
loop {
  j = 2*i
  i += 1
}
```

- Gets converted to

```
j = 0;
i = 1
loop {
  j += 2
  i += 1
}
```

- Useful for enabling opportunities for auto-increment mode
- cheaper instructions

# Method

1. Find loops in CFG
2. Find the variables in a loop that are loop invariant.
3. Find loop induction variables (vary regularly)
4. Reshape expressions into canonical form
5. perform strength reduction

# Step 1: Finding Loops

- Def$^{n}$: A loop is a set of basic blocks, $L$, such that if $b_0, b_1 \in L$ then there is a path from $b_0$ to $b_1$ and from $b_1$ to $b_0$. A block $b \in L$ is an entry block if $b$ has a predecessor that is not in $L$. A block $b \in L$ is an exit block if $b$ has a successor not in $L$.
  - We will look at natural loops where the entry block dominates all other blocks in the loop (single entry).
- Computing loops involves finding a block that has an incoming back edge (head dominates the tail).

# Loop Tree

- Organize the loops in a function hierarchically.
  - A loop L1 is a child of loop L2 in the loop tree iff L1 $\subseteq$ L2
- The tree structure is recorded by (X is a loop or block)
  - LoopParent(X) – an attribute indicating which node in the tree of which this node is a child. It also indicates the loop in which a loop or block is contained. LoopParent(X) may be a special root node indicating that the loop is contained in no other loop.
  - LoopContains(X) – the set of children of a node in the loop tree. The blocks or loops contained in a loop.
  - LoopEntry(X) – the entry node of the loop.

# Computing the Loop Tree

```
LoopTree() {
   compute post-order numbering for the CFG
   for each b ∈ G {
     LoopParent(B) = NIL
     LoopEntry(B) = B
     LoopContains(B) = B;
   }
   for each b ∈G in postorder
     FindLoop(b)
   Make all nodes w/o parents have a Root node as parent
}
```

# Computing the Loop Tree

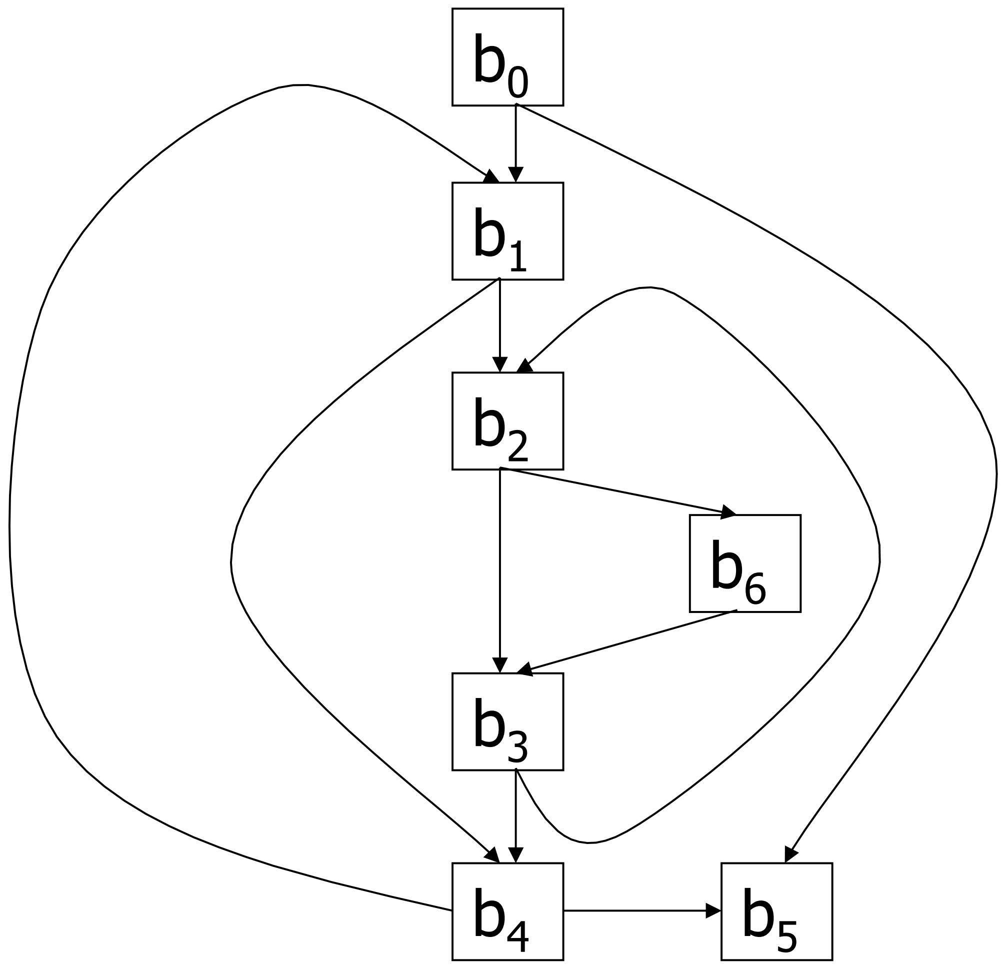
```
FindLoop(b) {
   Loop = ∅; Found = false
   for each p ∈ pred(b)
      if b ≫ p {
         Found = true;  // b is the entry node of Loop
         if p ∉ Loop ∧ p ≠ b {
            Loop ∪= {p}
         }
      }
   if Found
     FindBody(Loop,b)
}
```

# Computing the Loop Tree

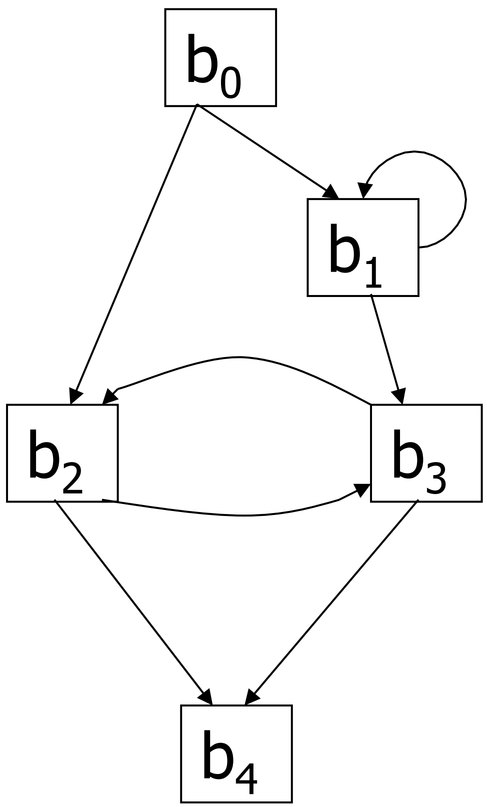
```
FindBody(Generators, H) {
    Loop = ∅; Queue = ∅
    for each b ∈ Generators {
      L = LoopAncestor(b)
      if L ∉ Loop then {
        Loop ∪= {L};Queue ∪= {L}
      }
    }
    while (Queue ≠ ∅) {
      b = Queue.Dequeue()
      Pred= pred(LoopEntry(b))
      for each p ∈ Pred
        if p ≠ H {
          L = LoopAncestor(p)
          if L ∉ Loop {
           Queue.Enqueue(L)
           Loop ∪= {L}
          }
        }
    }
    Loop ∪= {H}
    X = new Loop Tree node
    LoopContains(X) = Loop
    LoopEntry(X) = H
    LoopParent(X) = NIL
    for each b ∈ Loop
      LoopParent(b) = X
}

LoopAncestor(b) {
    while LoopParent(b) ≠ ∅
        b = LoopParent(b)
    return b
}
```

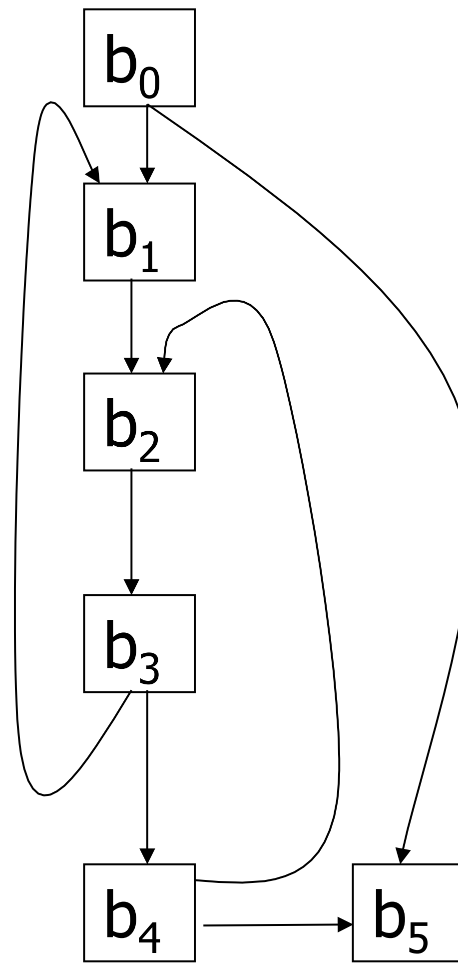
# Example



# Example




Irreducible loop



# Tarjan's Algorithm

- Compute maximal SCR (strongly connected region) on a directed graph
- Robert Tarjan, "Depth-first Search and Linear Graph Algorithms", SIAM J. Computing, 1:2, pp. 146-160, June 1972.
  - Uses a depth-first spanning tree, left-to-right preorder number in *Number*
  - Tracks the lowest numbered *v* to which each vertex has a path in *Lowlink*
  - Determines a number for SCR to which *v* belongs

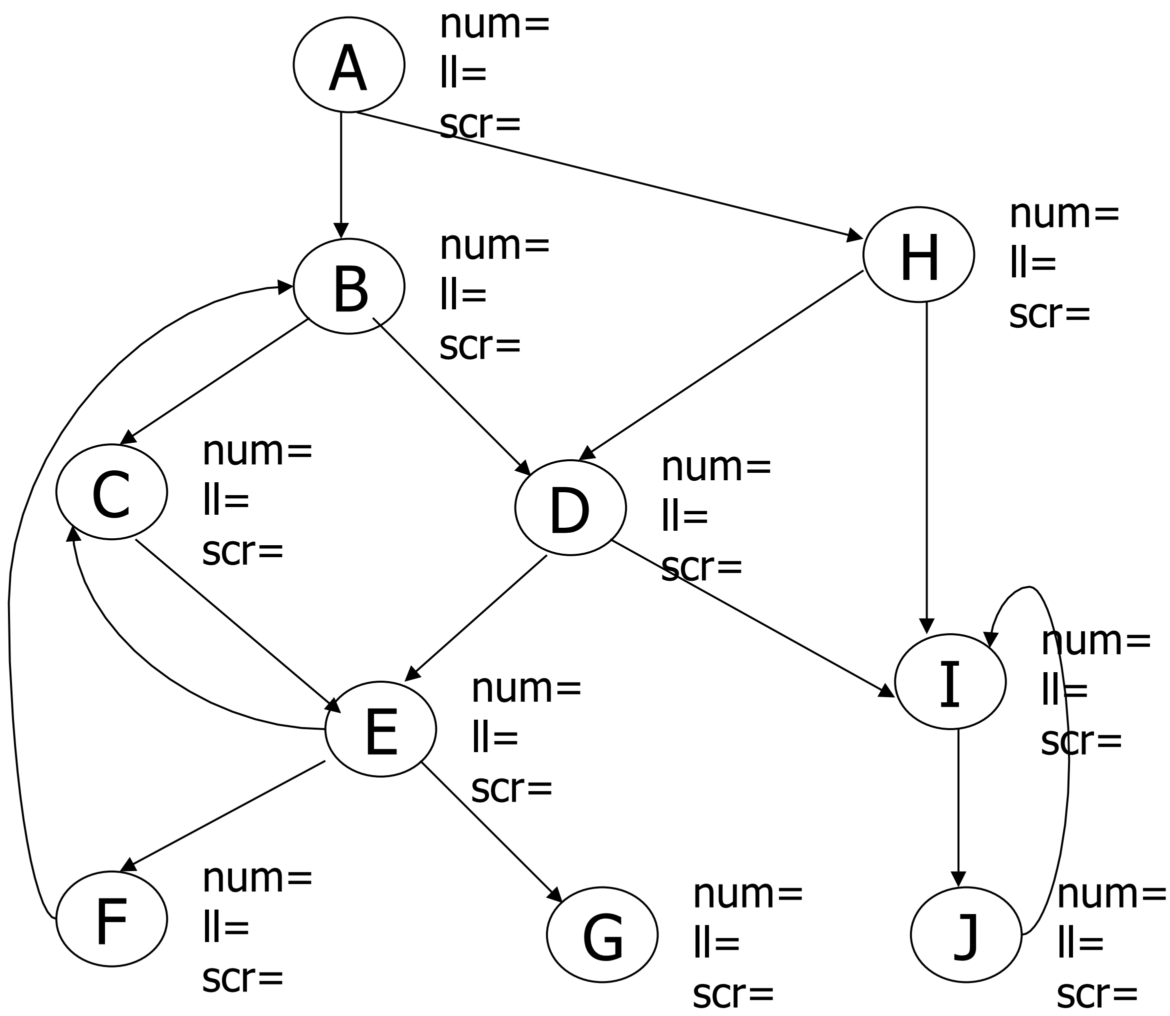
# Tarjan's Maximal SCR Algorithm

```
Number(*) = 0;
SCRnum = 0;
InStack(*) = false;
Stack = empty;
for (v ∈ V) {
   if (Number(v) == 0)
      Tarjan(v);
}
```

# Tarjan's Maximal SCR Algorithm (cont.)

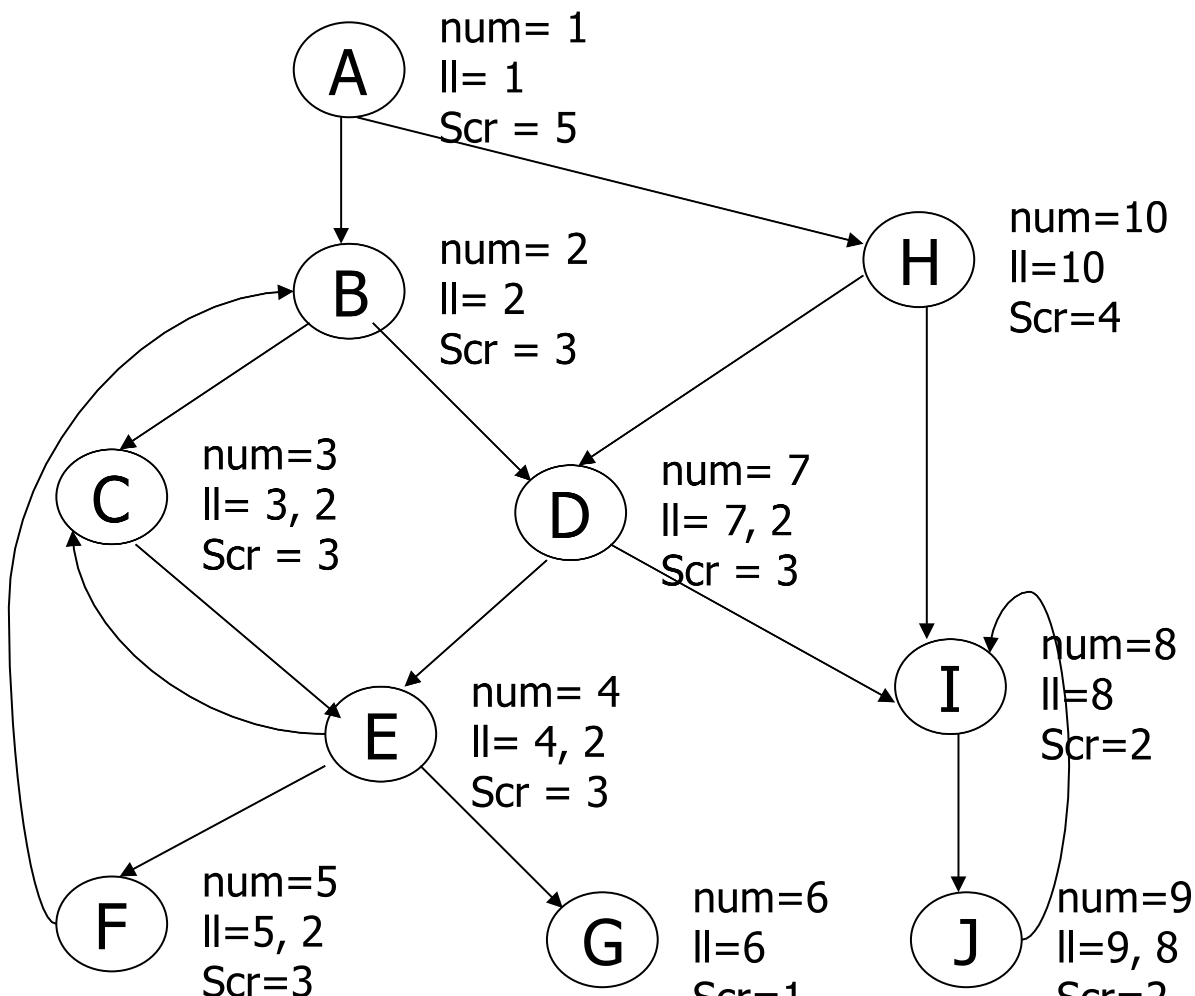
```
Tarjan(v)
{
   Number(v)=Lowlink(v)=++i;
   Instack(v) = true;
   push(v);
   for (w ∈ succ(v)) {
      if (Number(w) == 0) {
          Tarjan(w);
           Lowlink(v)= min(LowLink(v), Lowlink(w));
      } else if (InStack(w)) {
           Lowlink(v)= min(LowLink(v), Lowlink(w));
   }
   if (LowLink(v) == Number(v)) {
       SCRnum++;
       do {
             w=pop();   InStack(w) = false;  SCR(w)=SCRnum;
       } while (w !=v)
   }
}
```

# Tarjan's Maximal SCR Algorithm - Example




STACK

# Tarjan's Maximal SCR Algorithm - Example




STACK

# Step 2: Loop Invariants

- Def$^n$: A variable is loop invariant if it is either not computed in a loop or its operands are invariant.
- Compute variant(T), the innermost loop in which T is not invariant.
  - if T = $\phi$(..), T is defined to be variant in the innermost loop containing it.
  - for pure functions like add, variant in the innermost loop that one of the operands is variant
  - for a LOAD, it varies in the innermost loop in which a store operation might modify the same location.
- Walk the dominator tree in preorder

# Finding Loop Invariants

```
CalculateLoopInvariants()
{
   CalculateDominatorTree();
   CalculateLoopTree();
   for each b ∈ N in preorder on dominator tree
        CalculateLoopInvariants(b);
}
```

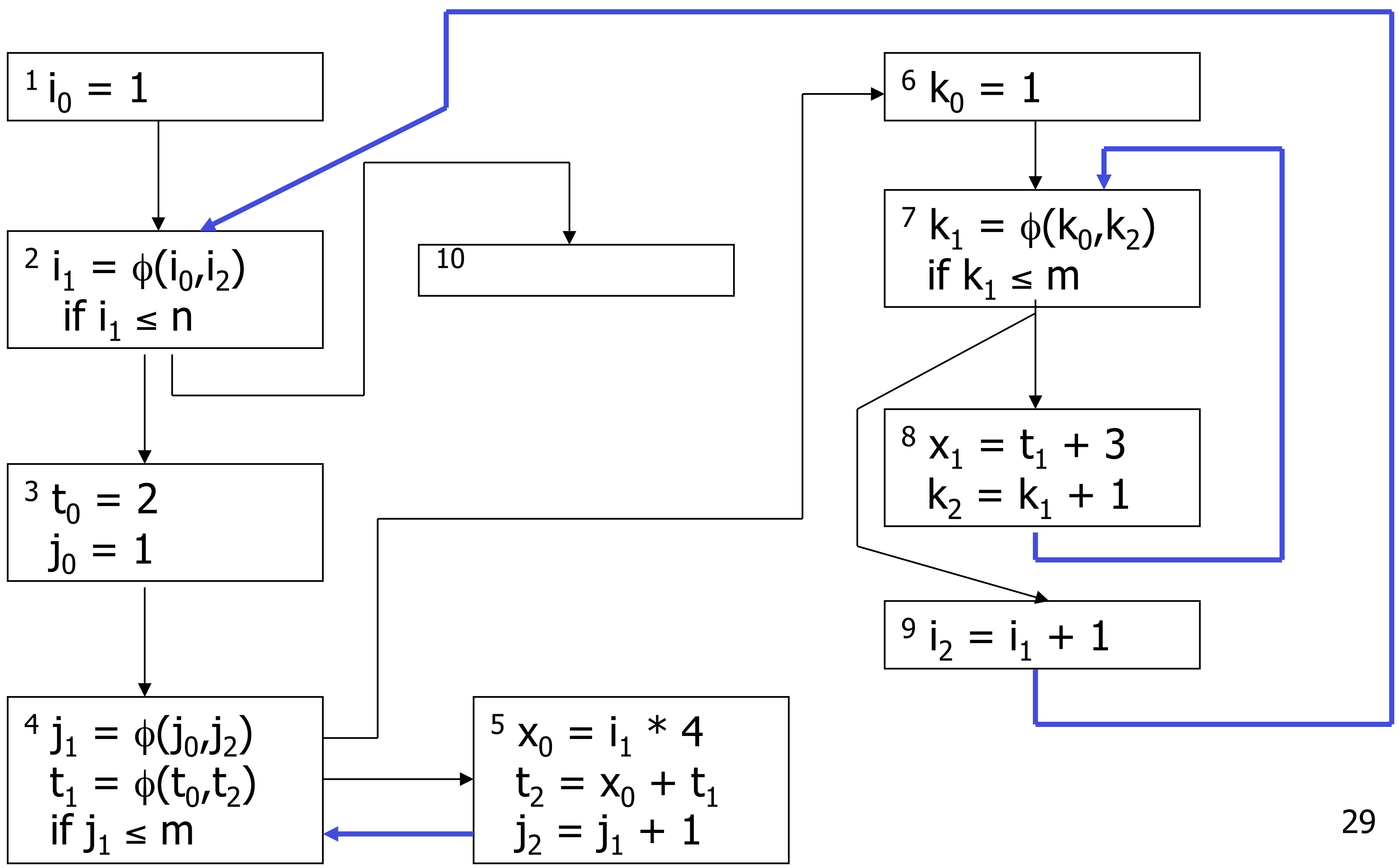
# Finding Loop Invariants

```
CalcLoopInvariants(b) {
   for each T0 = φ(T1,...,Tn) ∈ Φ(b)
     variant(T0) = LoopParent(b)
   for each I ∈b in execution order {
     Varying = Root;
     for each T ∈ Operands(I) {
       TVarying = LoopNearestAncestor(variant(T),b)
       if LoopNearestAncestsor(Varying,TVarying) == Varying
         Varying = TVarying
     }
     variant(I.lval()) = Varying
   }
}
```

# Finding Loop Invariants

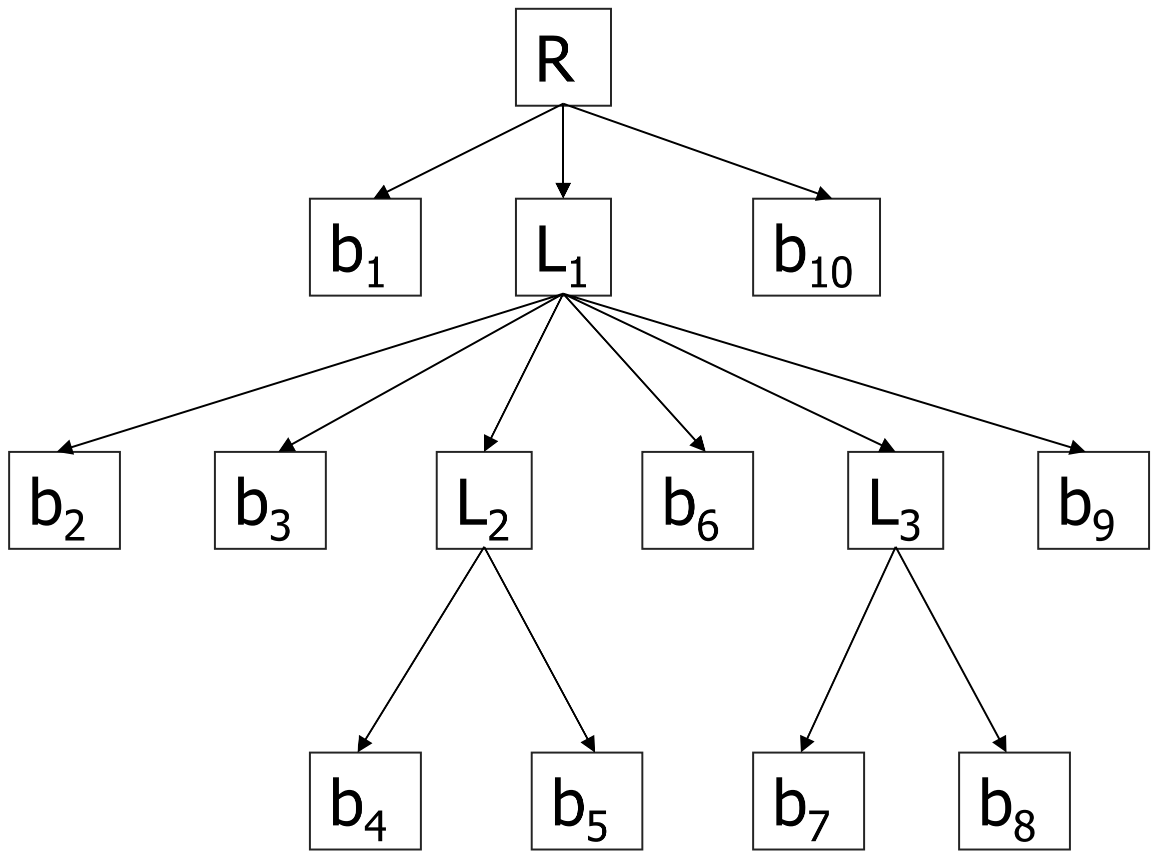
```
LoopNearestAncestor(L1,L2) {
   if is_ancestor(L2,L1)
     return L2
   L = L1
   while !is_ancestor(L,L2)
     L = LoopParent(L)
   return L
}
```

# Example

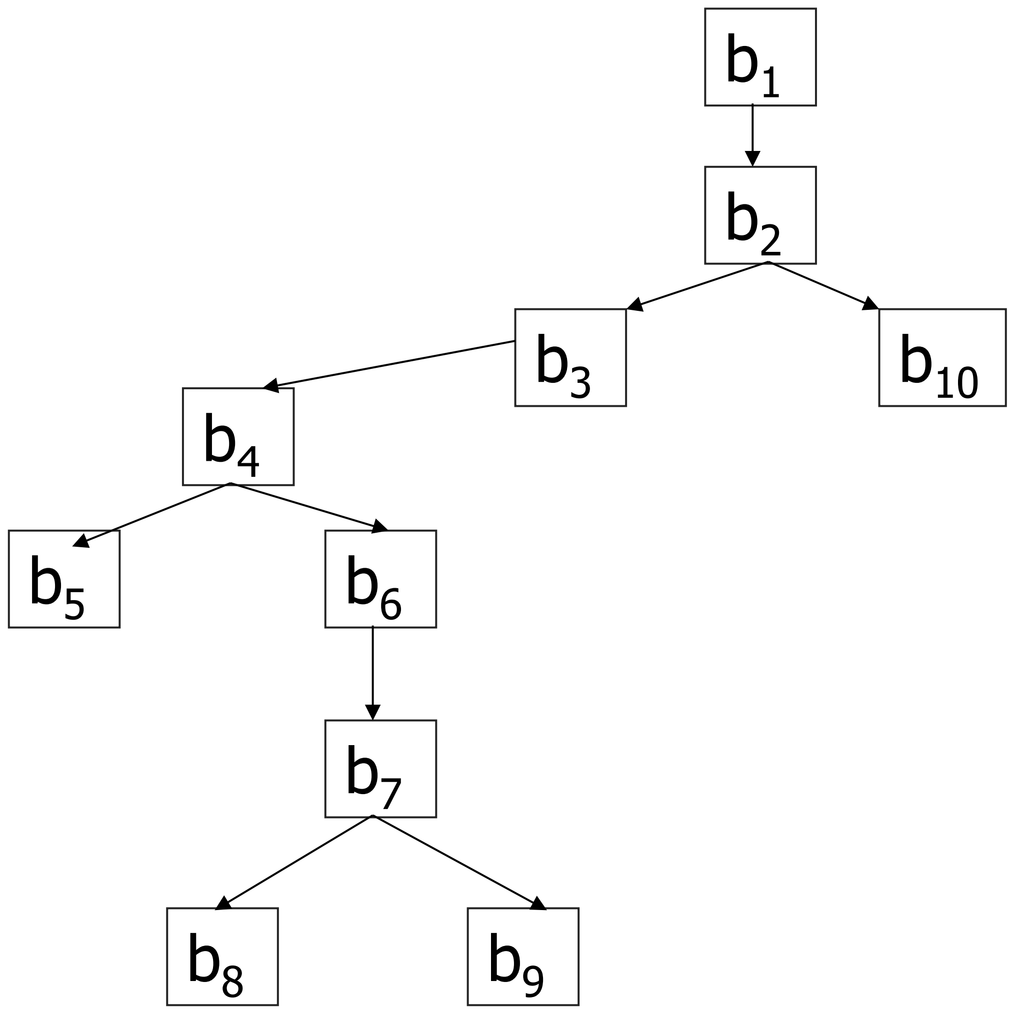


1 $i_0 = 1$
2 $i_1 = \phi(i_0,i_2)$
if $i_1 \le n$
3 $t_0 = 2$
$j_0 = 1$
4 $j_1 = \phi(j_0,j_2)$
$t_1 = \phi(t_0,t_2)$
if $j_1 \le m$
5 $x_0 = i_1 * 4$
$t_2 = x_0 + t_1$
$j_2 = j_1 + 1$
10
6 $k_0 = 1$
7 $k_1 = \phi(k_0,k_2)$
if $k_1 \le m$
8 $x_1 = t_1 + 3$
$k_2 = k_1 + 1$
9 $i_2 = i_1 + 1$

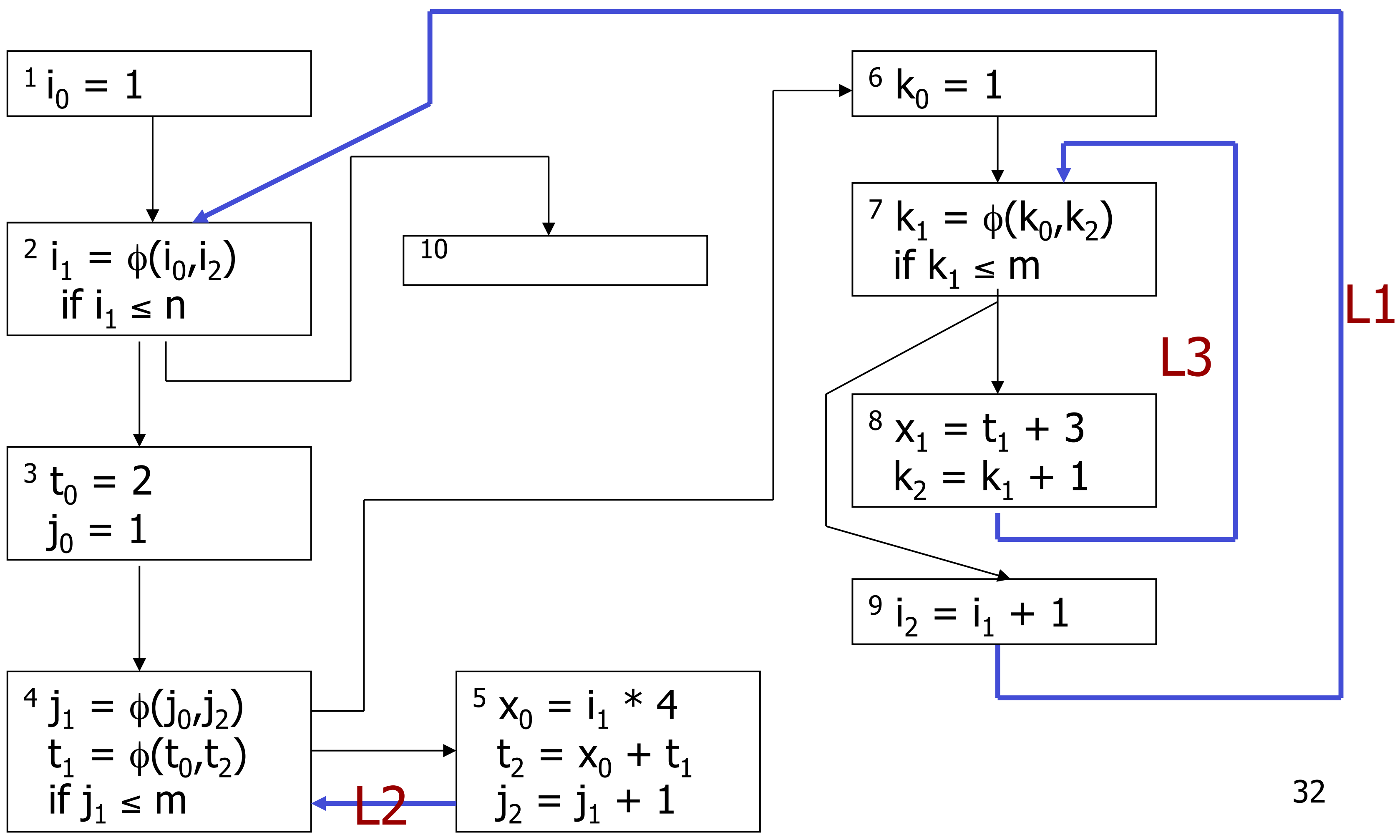
# Example: Loop Tree



# Example: Dominator Tree



# Example



1 $i_0 = 1$
2 $i_1 = \phi(i_0,i_2)$
if $i_1 \le n$
10
3 $t_0 = 2$
$j_0 = 1$
4 $j_1 = \phi(j_0,j_2)$
$t_1 = \phi(t_0,t_2)$
if $j_1 \le m$
5 $x_0 = i_1 * 4$
$t_2 = x_0 + t_1$
$j_2 = j_1 + 1$
6 $k_0 = 1$
7 $k_1 = \phi(k_0,k_2)$
if $k_1 \le m$
8 $x_1 = t_1 + 3$
$k_2 = k_1 + 1$
9 $i_2 = i_1 + 1$
L1
L2
L3

# Step 3: Finding Induction Variables

- Def$^n$: A temporary $T$ is a candidate temporary for loop $L$ iff $T$ is computed in $L$ and the computation has one of the following forms:
  a) $T = T_i \pm T_j$ where one operand is a candidate in $L$ and the other is loop invariant
  b) $T = \pm T_k$ where $T_k$ is a candidate in $L$ or is loop invariant in $L$
  c) $T = \phi(T_1,\ldots,T_n)$ where each of the operands is either a candidate in $L$ or a loop invariant in $L$

# Algorithm: Finding Induction Variables

```
CalcCandidates(L) {
   Candidates = ∅
   Work = ∅
   for each b ∈ L {
     for each I ∈ Φ(b) ∪ b of the form T = ...
       if Typeof(T) is integer
         if T has candidate syntax {
           Candidates ∪= {T}
           Work ∪= {T}
         }
   } // for each b
```

# Algorithm

```
while Work ≠ ∅ {
 take T from Work
 CandidatePrune(T)
 if T ∉ Candidates
  for each I ∈ Uses(T) where I ∈ L {
   U = I.lval()
   if (U ∈ Candidates ∧ U ∉ Work)
    Work ∪= {U}
  }
 }
}
```

# Algorithm

```
CandidatePrune(T) {
  I = T.instruction()
  case on form of I {
    T = φ(T1,...,Tn):
            for i = 1, n
             if Ti∉Candidates ∧ !invariant(Ti,L) {
               Candidates -= {T}
               return
             }
```

# Algorithm

$T = T_i \pm T_j$: if $T_i \in Candidates \wedge invariant(T_j, L)$

```
return
else
  if Tj ∈ Candidates ∧ invariant(Ti,L)
    return
  else {
    Candidates -= {T}
    return
  }
```

# Algorithm

```
T = ±Tk:  if Tk ∉ Candidates ∧ !invariant(Tk,L) {
             Candidates -= {T}
             return
          }
   }
}
```

$T = \pm T_k$: if $T_k \notin$ Candidates $\wedge$ !invariant($T_k$,L) {

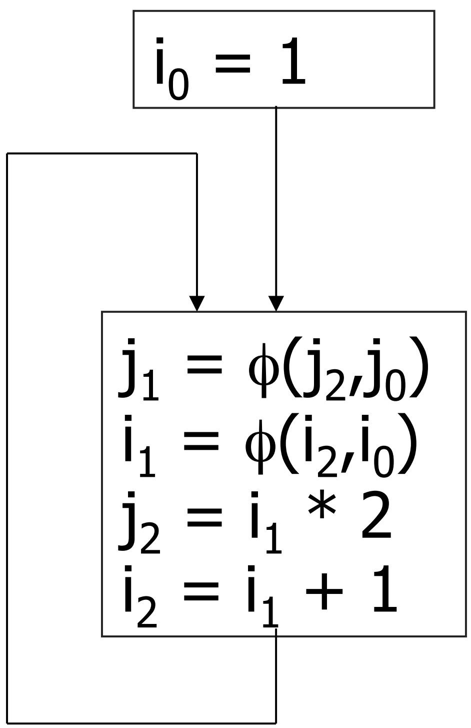
# Example

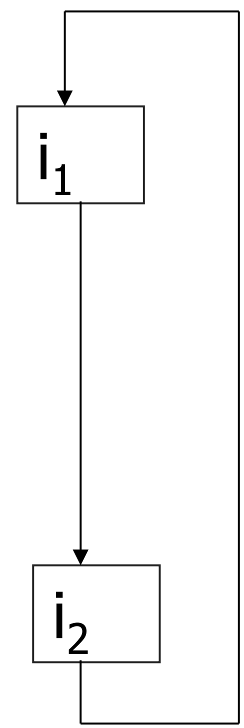
- Detect induction variable candidates in previous example

# Induction Sets

- Consider a graph where candidates are nodes and an edge is between two nodes, T and U, if T is used to compute U. And induction temporary is a temporary in a SCC in this graph. An induction set is the set of temporaries in the SCC.

# Example



# Algorithm

```
CalcInduction(L) {
   CalcCandidates(L)
   Construct candidate graph, G
   compute SCC(G)
   Anchors = {T | T is a target of a φ-node in
                        LoopEntry(L)}
   for each s ∈ SCC(G)
     if |s| > 1 ∧ Anchors ∩ s ≠∅
       add s to InductionSets
}
```

# Example

- Compute the induction variables in the previous example.

# Step 4: Reshape Expression

- Use commutative, associative, and distributive properties to reshape expressions contained in n loops as

$$E = E' + (LC_1 + (LC_2 + \ldots + LC_n))$$
$$E' = E'' + FD_1 * I_1 + FD_2 * I_2 + \ldots + FD_m * I_m$$

  where $LC_i$ is invariant in $L_i$, $I_i$ is the induction variable of $L_i$ and $FD_i$ is a loop invariant expression.
- $LC_i$ can be moved outside of $L_i$
- Can cause an increase in cost (invariants into loops)

# Strength Reduction

Consider an expression of the form $E = FD_i * I_i + LC_i$

Let $IS_i$ be the induction set of $I_i$

Create temporaries $E_0, \ldots, E_q$, one for each element of $IS_i$ plus any initial values coming in from outside the loop.

for all $T_j = T_k \pm c$ in the loop such that $T_j, T_k \in IS_i$
insert $E_j = E_k \pm FD_i * c$ after this point

for all $T_j = \pm T_k$ in the loop such that $T_j, T_k \in IS_i$
insert $E_j = \pm E_k$ after this point

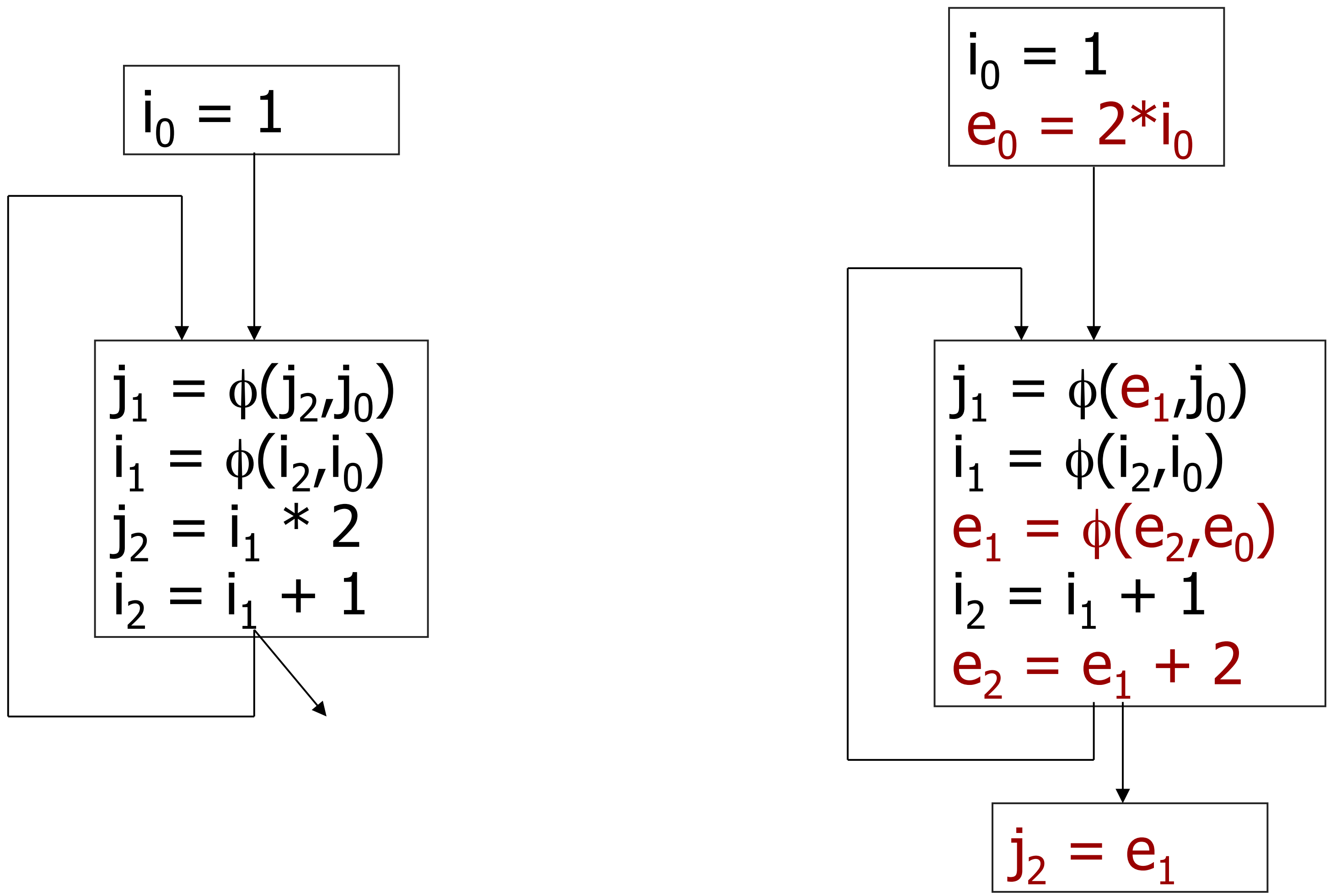
replace uses of E with the corresponding $E_i$ whose definition reaches the use

replace $E = FD_i * I_i + LC_i$ with the assignment $E = E_i$. If the block containing this assignment is executed on every path through the loop to a loop exit, it can be moved after the loop following each loop exit.

# Handling $\phi$-nodes

- Given $T_0 = \phi(T_1, \ldots T_n)$, $T_0 \in IS_i$, create a new $\phi$-node $E' = \phi(\ldots)$
- for each predecessor block $P_j$
  - if the temporary $T_j$ is in the induction set of $T_0$, put the temporary holding E at the end of $P_j$ in the $j^{th}$ position of the $\phi$-node for E' ($P_j$ must be in the loop because $T_j$ is in the induction set).
  - if $T_j$ is not in the induction set for $T_0$, insert the computation $E_j = FD_i{*}T_j + FC_i$ at then end of $P_j$ and place $E_j$ into the corresponding entry in the $\phi$-node for E' ($P_j$ is not in the loop).
  - change E' to be the exposed use of a temporary for E.

# Example



i0 = 1
j1 = φ(j2,j0)
i1 = φ(i2,i0)
j2 = i1 * 2
i2 = i1 + 1
i0 = 1
e0 = 2*i0
j1 = φ(e1,j0)
i1 = φ(i2,i0)
e1 = φ(e2,e0)
i2 = i1 + 1
e2 = e1 + 2
j2 = e1

# Dead-code Elimination

- Use the SSA graph (sparse) to detect dead code.
  - SSA graph are the set of nodes representing temporaries and connected by def-use links.
- Method
  - remove instructions that do not directly or indirectly use data that is observable outside the procedure.
  - allow for eliminating branches that are never taken (can eliminate loops this way)
    - uses control dependence

# Control Dependence

- Use the idea of postdominators
- Def$^n$: A block X postdominates a block B iff every path from B to Exit contains X.
- Def$^n$: pdom(B) represents the immediate postdominator of B and is the parent of B in the postdominator tree.
- Compute postdominators using the dominator relation on the reverse control flow graph.

# Review: Computing Dominators

$D(v_0) = \{v_0\}$

for each $n \in V - \{v_0\}$

  $D(n) = V$

do {

  for each $n \in V - \{v_0\}$

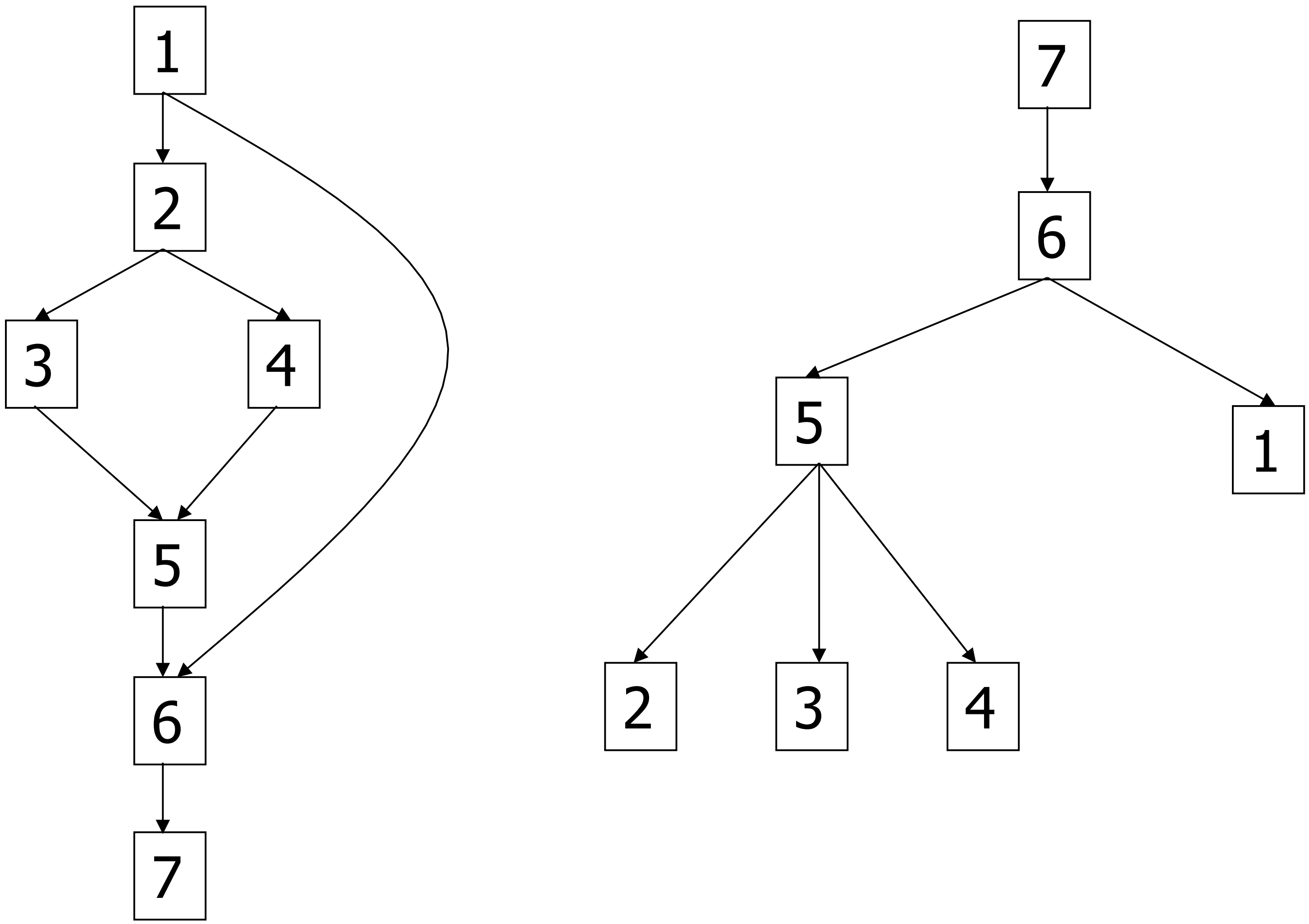
   $D(n) = \{n\} \cup \bigcap_{p \in preds(n)} D(p)$

} until no $D(n)$ changes

$n \underline{\gg} m \Leftrightarrow \forall p \in pred(m)\ n \underline{\gg} p$

- ENTRY dominates all nodes
- Since $\underline{\gg}$ is a partial order, we can construct an ordering of all the nodes that each node dominates to construct a dominator tree.
- The immediate dominator of n, denoted idom(n), is its parent in the dominator tree.

# Example: Postdominator Tree



# Control Dependence

- Consider two blocks, B and X. When does B control the execution of X?
  1. If B has only one successor block, it does not control the execution of anything. B must have multiple successors.
  2. B must have some path leaving it that leads to the Exit block and avoids X. X cannot postdominate B
  3. B must have some path leaving it that leads to X.
  4. B should be the latest block that has these properties.

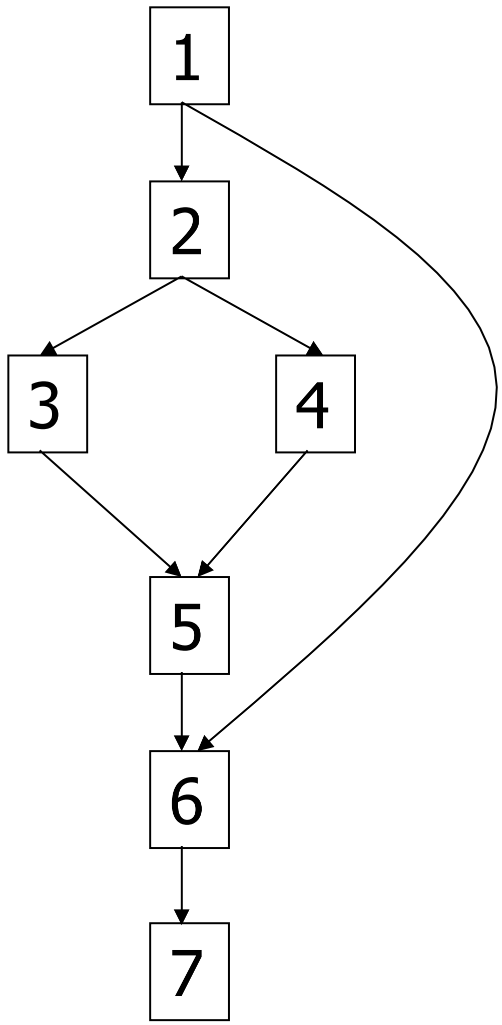
# Control Dependence

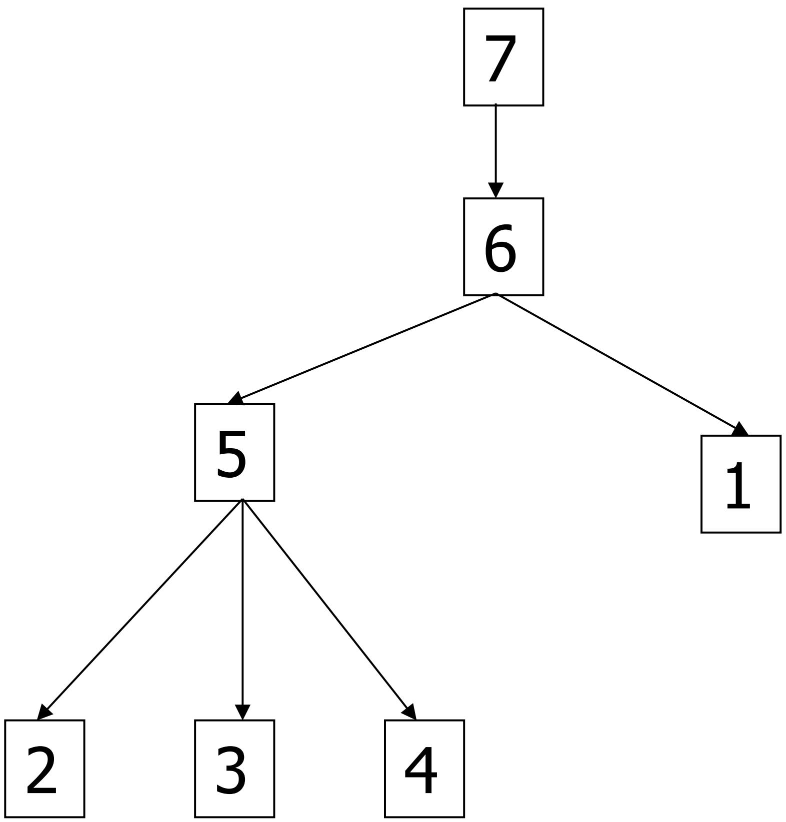
- A block X is control dependent on a block B iff there is a non-empty path from B to X such that X postdominates each block on the path except B. And, X = B or X does not postdominate B.
- A block X is control dependent on an edge (B,S) iff there is a non-empty path from B to X starting with edge (B,S) such that X postdominates each block on the path except B. And, X = B or X does not postdominate B.
- Compute control dependence by find the dominance frontier of every node in the reverse control-flow graph.

# Computing Control Dependence

```
foreach n ∈ PDT in postorder{
   DF(n) = ∅
   for each c ∈ child(n)
     for each m ∈ DF(c)
       if !(n stricly postdominates m)
         DF(n) ∪= {m}
   for each m ∈ pred(n)
     if !(n strictly postdominates m)
       DF(n) ∪= {m}
}
```

# Example: Calculate Control Dependence



# Algorithm

```
EliminateDeadCode()
    WorkList = ∅
    Necessary = ∅
    for each B ∈ N do
      for each I ∈ B do
        if (I stores into external data) ∨
            (I is an i/o instruction) ∨ (I is a call) {
          Necessary ∪= {I}
          WorkList ∪= {I}
        }
```

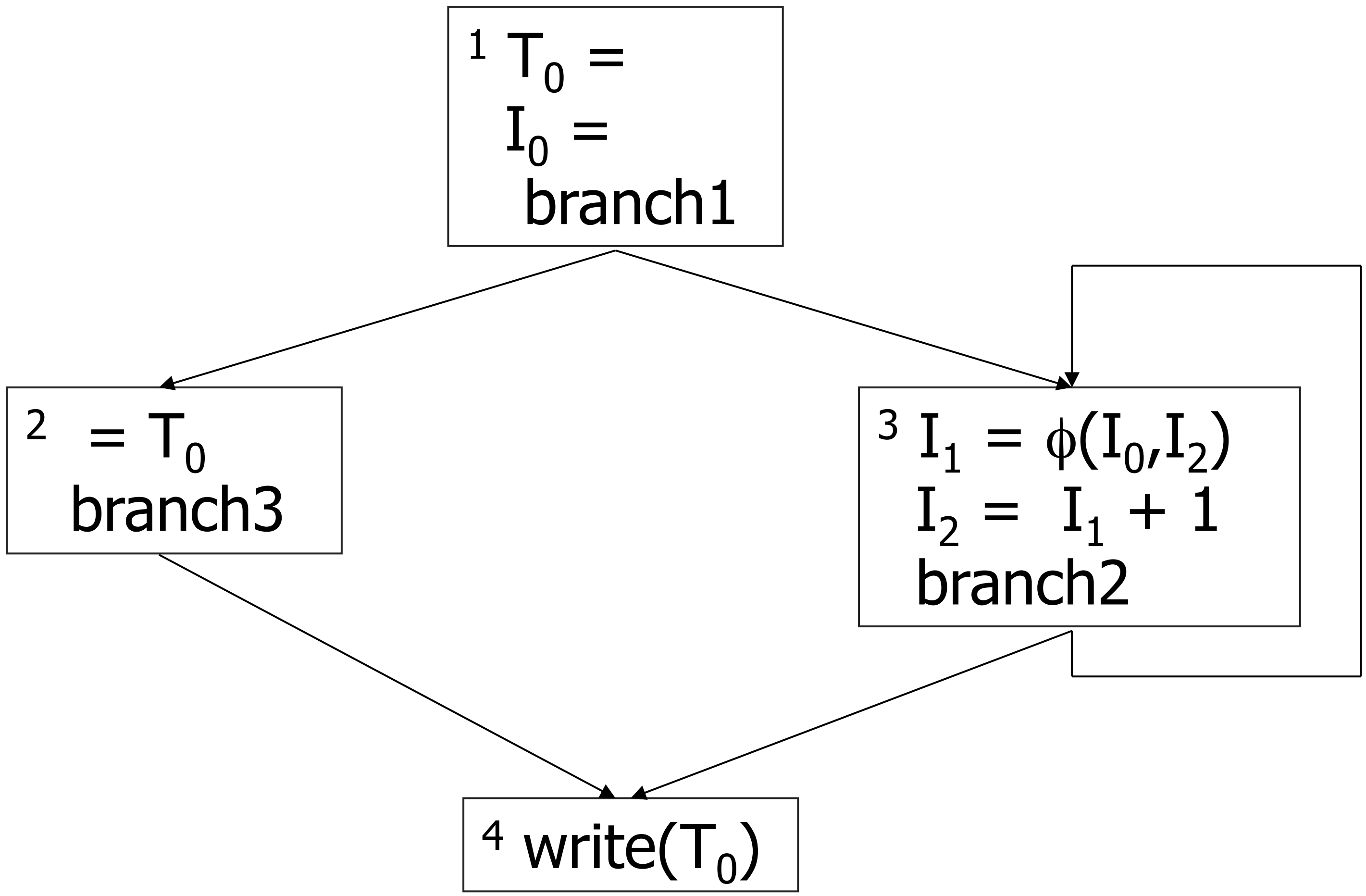
# Algorithm

```
while WorkList ≠ ∅ {
  take I from WorkList
  b = I.ContainingBlock()
  for each C on which B is control dependent {
    J = branch in C
    if J ∉ Necessary {
      Necessary ∪= {J}
      WorkList ∪= {J}
    }
  }
```

# Algorithm

```
    for each T ∈ Operand(I) {
      J = Definition(I)
      if J ∉ Necessary {
        Necessary ∪= {J}
        WorkList ∪= {J}
      }
    }
  }
  for each B ∈ N
    for each I ∈ B
      if I ∉ Necessary
        remove I
      else if I is a branch ∧ I ∉ Necessary
         change branch to immediate postdominator of block
}
```

# Example



1 $T_0$ =
$I_0$ =
branch1
2 = $T_0$
branch3
3 $I_1 = \phi(I_0, I_2)$
$I_2 = I_1 + 1$
branch2
4 write($T_0$)

# Global Value Numbering

- Apply value numbering to a global context for better redundancy elimination.
- Associate a field for each temporary to hold its value number
- If two temporaries have the same value number then they are equivalent.
- If there are no loops a reverse postorder walk of the CFG is sufficient (all operands defined before used)
- $\phi$-nodes can only be equivalent in the same basic block
  - need control-flow information to compare $\phi$-nodes from different blocks

# Global Value Numbering

- Work on SSA graph
- What can we do about SCCs in the SSA graph?
  - The value number of some operands will not be known when trying to process an instruction.
  - This will happen at $\phi$-nodes
  - Solution: assume the best case (an unknown value number does not affect the result) and iterate
  - Process nodes in an SCC in reverse postorder (as other nodes)

# Processing $\phi$-nodes

- There are 3 possibilities
  1. If a corresponding entry for the $\phi$-node/block is already in the value table, then assign the target of this $\phi$-node the same value_representative value.
  2. Consider the operands that do not have a value_representative value of NULL. If at least two of them have different values, assign the target a new value # and enter it into the value table
  3. Consider the operands that do not have a value_representative value of NULL. If all of them have the same value, then give the target the same value number and enter it into the table.

# Efficiency Improvements

- When processing a SCC, use a temporary value table called a scratch table. Once the values in the scratch table have stabilized, move the results to the value table.

# Algorithm

```
procedure CalcGlobalValue {
   compute the SCC of the SSA Graph: C_1,...,C_s ordered by SSA
      edges so that defs precede uses
   ValTab = ∅; ScratchTab = ∅;
   for each T ∈ Temporaries
     ValRep(T) = NULL;
   for i = 1, s
     if |C_i| > 1 {
       call CalcGlobalValueSCC(C_i)
       for each T ∈ C_i in reverse postorder {
         I = Definition(T); U = ValRep(T);
         apply algebraic simplification to I
         if ⟨opcode(I),ValRep(Operands(I))⟩ ∉ ValTab
           ValTab ∪= {⟨opcode(I),ValRep(Operand(i),U⟩}
     }
```

# Algorithm

```
// let I be the single instruction in C_i
  else if I is a φ-node
    CalcφValue(I,ValTab)
  else {
      apply algebraic simplification to I
      T = Target(I)
      if ⟨opcode(I),ValRep(Operands(I)⟩ ∉ ValTab {
         ValRep(T) = T;
         ValTab ∪= {⟨opcode(I),ValRep(Operands(I),ValRep(T)⟩}
      }
      else
         ValRep(T) = value from ValTab
   }
}
```

# Algorithm

```
procedure CalcGlobalValueSCC(C) {
    change = false;
    repeat
     for each T ∈ C in reverse postorder {
      I = Definition(T)
      if I is a ϕ-node
       CalcϕValue(I,ScratchTab)
      else {
         process algebraic simplification but don't change instructions
        if ⟨opcode(I),ValRep(Operands(I))⟩ ∈ ScratchTab
          NewValue = value in ScratchTab
        else {
          NewValue = T
          ScratchTab ∪= {⟨opcode(I),ValRep(Operands(I),T⟩}
        }
       }
      if NewValue ≠ ValRep(T) {
          change = true; ValRep(T) = NewValue;
       }
     }
    until not(change)
}
```

# Algorithm

```
procedure CalcφValue(I,Table) {
    Let I be T0 = φ(T1,...,Tn)
    if <φ,ValRep(Operands(I))> ∉ Table {
      if ∃Ti,Tj | ValRep(Ti) ≠ NULL ∧ ValRep(Tj) ≠ NULL ∧
                       ValRep(Ti) ≠ ValRep(Tj)
        NewValue = T0
      else
        NewValue = ValRep(Ti) where ValRep(Ti) ≠ NULL
      Table ∪= {⟨opcode(I),ValRep(Operands(I)),NewValue⟩}
    }
    else
       NewValue = ValRep from Table
}
```

# Example

1 $I_0 = 1$
$J_0 = 1$

2 $I_1 = \phi(I_0, I_2)$
$J_1 = \phi(J_0, J_2)$
$U = I_1 - J_1$
$I_2 = I_1 + 1$
$J_2 = J_1 + 1$

# Now What?

- Give all temporaries with the same value # the same partition, and convert to normal form
- Apply common subexpression elimination
  - dominator-based
  - traditional AVAIL-based
  - partial redundancy elimination