CS142B Language Processor Construction

# Building Static Single Assignment Form

Yeoul Na

UCI

April 16, 2019

# Static Single Assignment Form

- A property of an intermediate representation
- Each variable is assigned exactly once
- Control-flow and data-flow are explicit
  - Simplifies and improves a variety of compiler optimizations

# A Simple Example

y = 1

y = 2

x = y

# A Simple Example

$y = 1$

$y = 2$

$x = y$

$y_1 = 1$

$y_2 = 2$

$x_1 = y_2$

# A Simple Example

$y = 1$

$y = 2$

$x = y$

$y_1 = 1$

$y_2 = 2$

$x_1 = y_2$

# Statically Unknown Use

```
if(x > 4){
   y = x - 2;
} else{
    y = x + 9;
}
w = x + y;
```

```
if(x1 > 4){
   y1 = x1 - 2;
} else{
    y2 = x1 + 9;
}
w1 = x1 + y?;
```

# *Φ Function*

```
if(x > 4){
   y = x - 2;
} else{
    y = x + 9;
}
w = x + y;
```

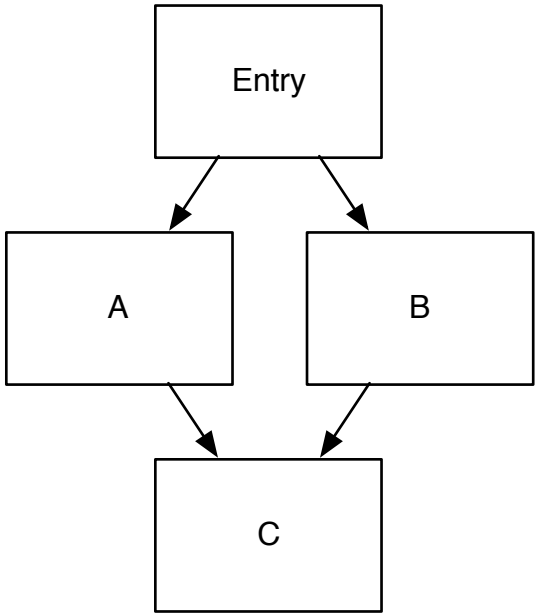
```
if(x1 > 4){
   y1 = x1 - 2;
} else{
    y2 = x1 + 9;
}
y3 = Φ(y1,y2);
w1 = x1 + y3;
```

# Where to put these Φ functions?

- Insert Φ for every definition at every join node
- How to find the minimal join nodes
  - Dominance Frontiers

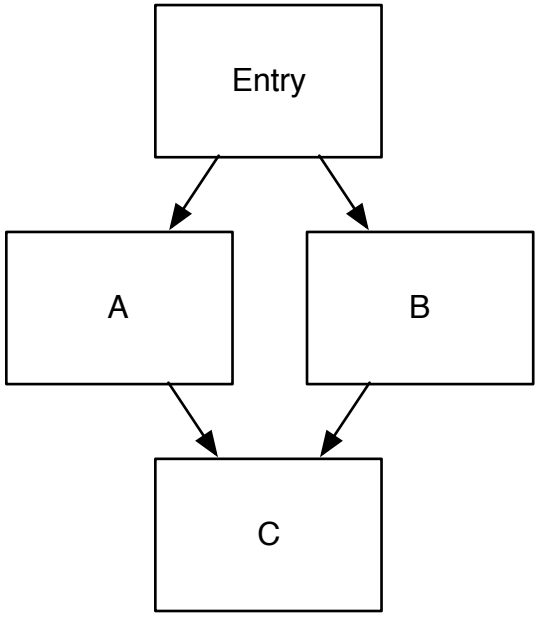
# Dominance Property

- Dominators
  - *X* dominates *Y* if all paths from *Entry* to *Y* must pass *X*
  - *X* strictly dominates *Y*, if *X* dominates *Y* and *X* != *Y*
- In SSA form, definitions must dominate uses!



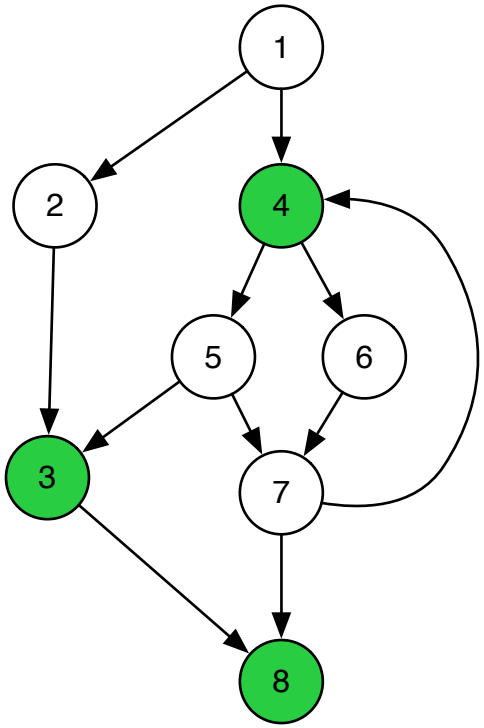
# Dominance Frontier

- *Y* is a dominance frontier of a node *X* iff
  - *X* dominates a predecessor of *Y*, and
  - *X* does not strictly dominate *Y*



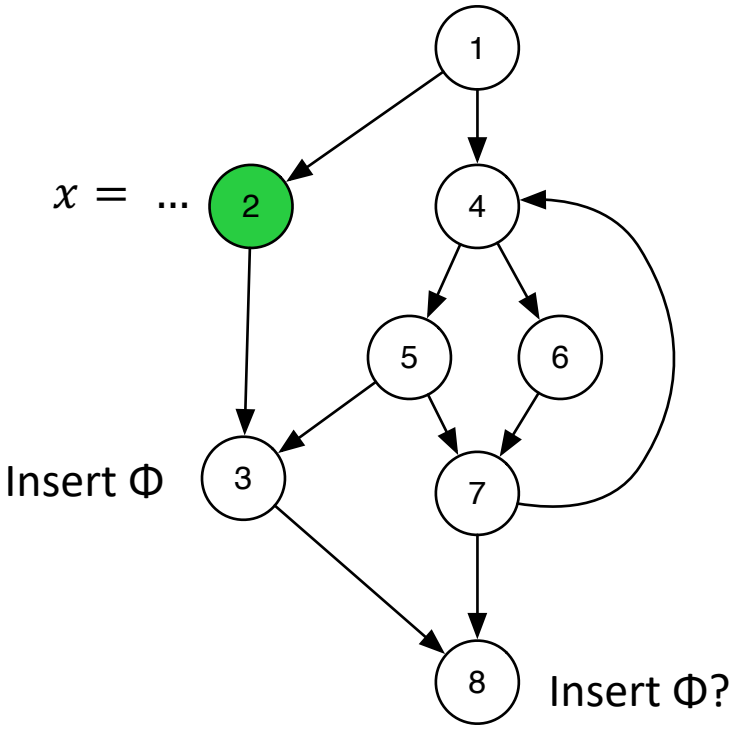
# Dominance Frontier Example

- $DF(d) = \{n \mid \exists p \in pred(n),\ d \text{ dom } p \text{ and } d \text{ ! sdom } n\}$
- $Dom(4) = \{4, 5, 6, 7\}$
  - $iSucc(4) = \{\sout{5}, \sout{6}\},\ iSucc(5) = \{3, \sout{7}\},\ iSucc(6) = \{\sout{7}\},\ iSucc(7) = \{4, 8\}$
- $DF(4) = \{3, 4, 8\}$



# Φ Function Exercise

- $Dom(2) = \{2\}, iSucc(2) = \{3\} = DF(2)$



# Algorithm for Inserting Φ Functions

```
for each variable v
    WorkList ← ∅
    EverOnWorkList ← ∅
    AlreadyHasPhiFunc ← ∅
    for each node n containing an assignment to v
        WorkList ← WorkList ∪ {n}
    EverOnWorkList ← WorkList
    while WorkList ≠ ∅
        Remove some node n from WorkList
        for each d ∈ DF(n)
            if d ∉ AlreadyHasPhiFunc
                Insert a ϕ-function for v at d
                AlreadyHasPhiFunc ← AlreadyHasPhiFunc ∪ {d}
                if d ∉ EverOnWorkList
                    WorkList ← WorkList ∪ {d}
                    EverOnWorkList ← EverOnWorkList ∪ {d}
```

# Algorithm for Inserting Φ Functions

- For each variable,
  - Find all basic blocks that define the variable
  - For each of these basic blocks,
    - Find its dominance frontiers
    - Insert Φ for each of the dominance frontier

# Build SSA Step by Step

- Build CFG
  - Create basic blocks
  - Link basic blocks with successors, predecessors
  - Fill the basic blocks with instructions
- Insert Φ nodes
- Rename variables to be in SSA form

# Create Basic Blocks

1. Determine a set of leaders, the first instruction of basic blocks
    1. The first instruction is a leader
    2. Instruction L is a leader if it's a target of a branch instruction
    3. Instruction L is a leader if it immediately follows a branch instruction
2. A basic block consists of a leader and all the following instructions until the next leader

# Example Basic Block Creation

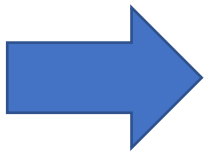
```
0: iconst_0
1: istore_1
2: iload_1
3: bipush        20
5: iadd
6: istore_2
7: iload_2
8: istore_3
9: iload_1
10: ifne          16
13: bipush        53
15: istore_3
16: iload_3
17: invokestatic  #4
20: return
```

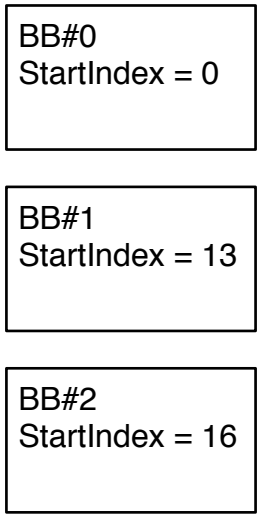
# Example Basic Block Creation

```
Leader  0: iconst_0
        1: istore_1
        2: iload_1
        3: bipush        20
        5: iadd
        6: istore_2
        7: iload_2
        8: istore_3
        9: iload_1
        10: ifne          16
        13: bipush        53
        15: istore_3
        16: iload_3
        17: invokestatic  #4
        20: return
```

# Example Basic Block Creation

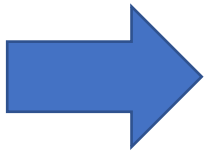
```
Leader  0: iconst_0
        1: istore_1
        2: iload_1
        3: bipush        20
        5: iadd
        6: istore_2
        7: iload_2
        8: istore_3
        9: iload_1
        10: ifne          16
Leader  13: bipush        53
        15: istore_3
Leader  16: iload_3
        17: invokestatic  #4
        20: return
```





# Example Basic Block Creation

```
Leader  0: iconst_0
        1: istore_1
        2: iload_1
        3: bipush        20
        5: iadd
        6: istore_2
        7: iload_2
        8: istore_3
        9: iload_1
        10: ifne          16
Leader  13: bipush        53
        15: istore_3
Leader  16: iload_3
        17: invokestatic  #4
        20: return
```



BB#0
StartIndex = 0

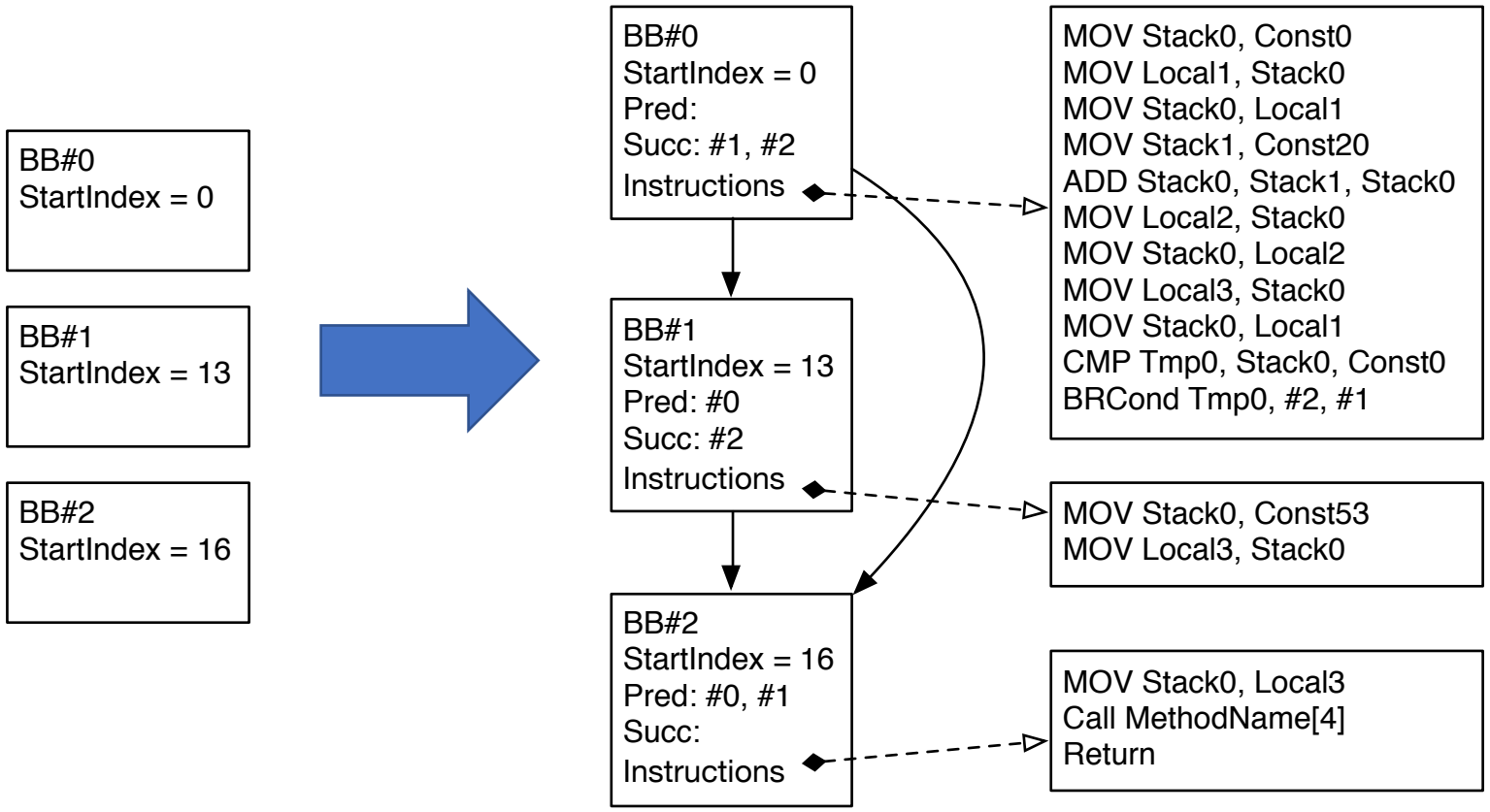
BB#1
StartIndex = 13

BB#2
StartIndex = 16

# Link and Fill Basic Blocks

- Each basic block contains a list of successors and a list of predecessors
- At the same time, fill basic blocks with instructions

# Link and Fill Basic Blocks



# Intermediate Representation

- Two-Address Code
  - E.g., add %0, %1
- Three-Address Code
  - E.g., add %0, %1, %2
  - Quadruples: <Opcode, Op1, Op2, Op3>

- Deal with Stack : make def/use of data on the operand stack explicit
  - Stack Numbering

# Intermediate Representation

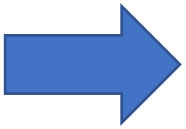
- Value: empty parent class
- AbstractVariable: inherits from Value
- Instruction (Opcode) inherits from Value
  - The following lists inherit from Instruction
  - MoveInstruction (Opcode, Op1, Op2)
  - BinaryInstruction (Opcode, Op1, Op2, Op3)
  - CmpInstruction (Opcode, Cond, Op1, Op2)
  - CondBranchInstruction (Opcode, Cond, TargetBB1, TargetBB2)
  - UncondBranchInstruction (Opcode, TargetBB)
  - CallInstruction (Opcode, TargetMethod)
  - ReturnInstruction (Opcode, Op1)
  - PhiInstruction (Opcode, {BB1, Op1}, {BB2, Op2}, …}

# Stack Numbering

```
LiveInStackDepth : 0

0: iconst_0                 StackDepth++ : 1
1: istore_1                 StackDepth— : 0
2: iload_1                  StackDepth++ : 1
3: bipush        20         StackDepth++ : 2
5: iadd                     StackDepth— : 1
6: istore_2                 StackDepth— : 0
7: iload_2                  StackDepth++ : 1
8: istore_3                 StackDepth— : 0
9: iload_1                  StackDepth++: 1
10: ifne          16        StackDepth— : 0
```

```
LiveInStackDepth : 0

13: bipush        53        StackDepth++ : 1
15: istore_3                StackDepth— : 0
```

```
LiveInStackDepth : 0

16: iload_3                 StackDepth++ : 1
17: invokestatic  #4        StackDepth— : 0
20: return
```



```
MOV Stack0, Const0
MOV Local1, Stack0
MOV Stack0, Local1
MOV Stack1, Const20
ADD Stack0, Stack1, Stack0
MOV Local2, Stack0
MOV Stack0, Local2
MOV Local3, Stack0
MOV Stack0, Local1
CMP Tmp0, Stack0, Const0
BRCond Tmp0, #2, #1
```

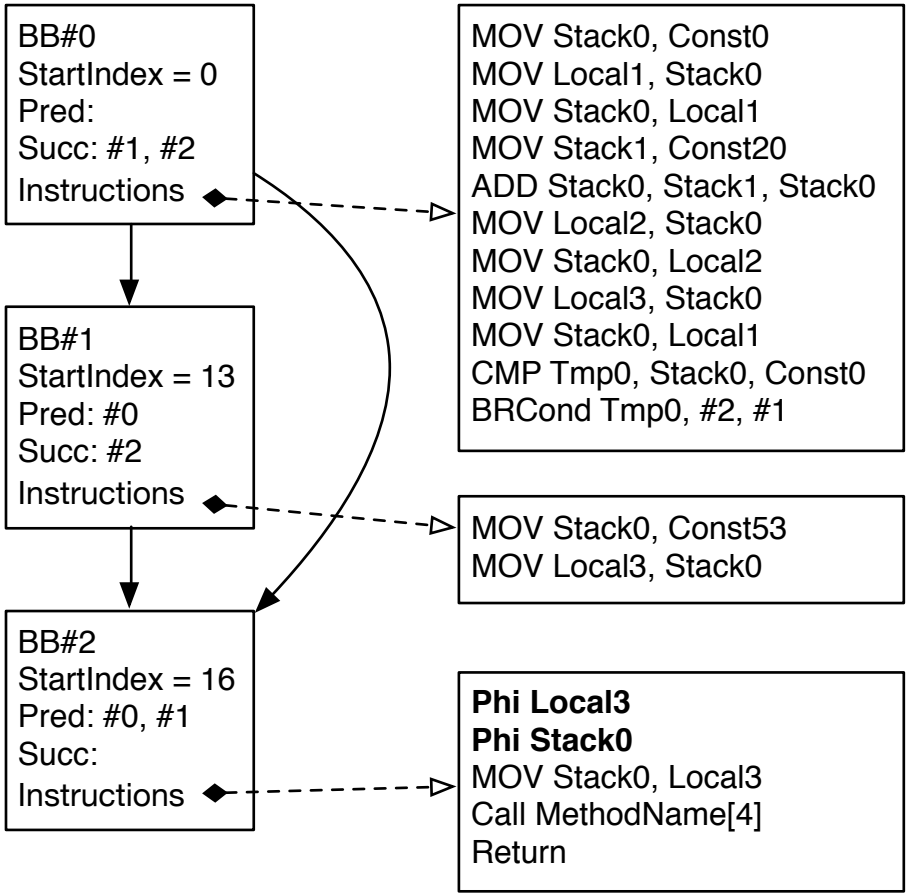
```
MOV Stack0, Const53
MOV Local3, Stack0
```

```
MOV Stack0, Local3
Call MethodName[4]
Return
```

# Algorithm for Inserting Φ Functions

- For each variable,
  - Find all basic blocks that define the variable
  - For each of these basic blocks,
    - Find its dominance frontiers
    - Insert Φ for each of the dominance frontier

# After Inserting Φ Functions



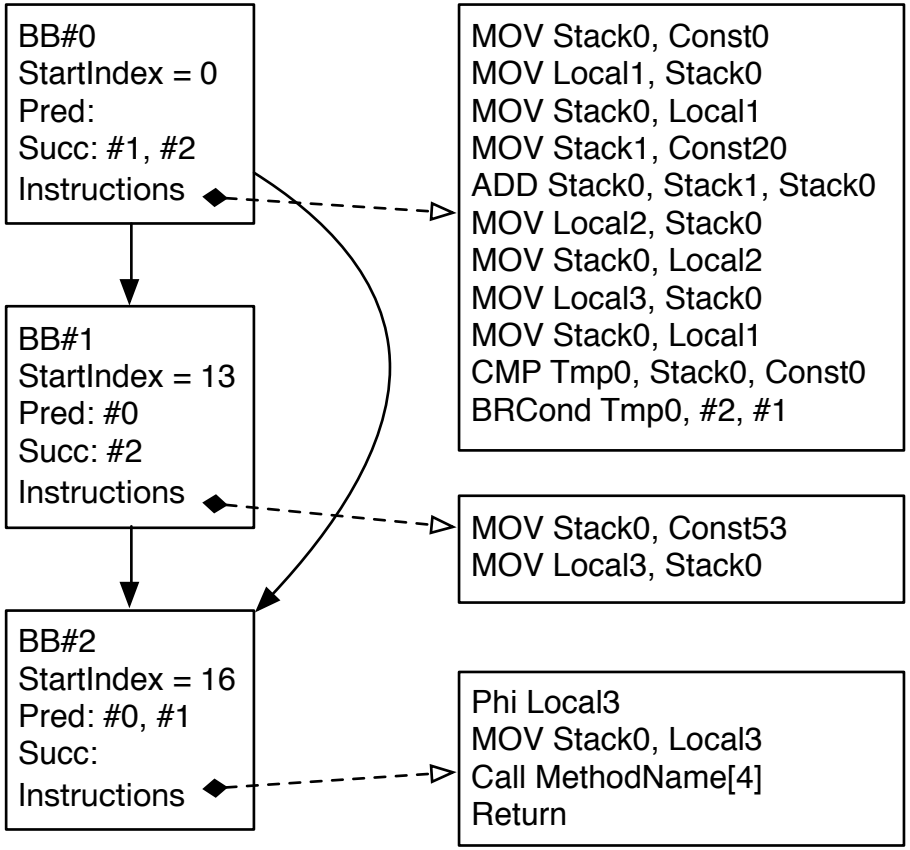

**Phi Stack0** is Redundant!

# Optimization

- Dead Definition
  - No need to insert Phi, if the definition *LIVE-OUT* for the defining block or *LIVE-IN* for the join block
  - But this requires liveness analysis
- Simple optimization for the operation stack
  - Record the stack depth (live-in stack) at the beginning of each block
  - Do not insert Phi if the live-in stack depth of the dominance frontier is smaller than or equal to the stack variable index

# After Phi Optimization

```
0: iconst_0
1: istore_1
2: iload_1
3: bipush        20
5: iadd
6: istore_2
7: iload_2
8: istore_3
9: iload_1
10: ifne          16
13: bipush        53
15: istore_3
16: iload_3
17: invokestatic  #4
20: return
```



# Variable Renaming (SSA Form)

- For a variable definition, create a new name for it
- For a variable use, use on the top of the name stack
  - i.e., the recent definition in the dominator

$y = 1$

$y = 2$

$x = y$

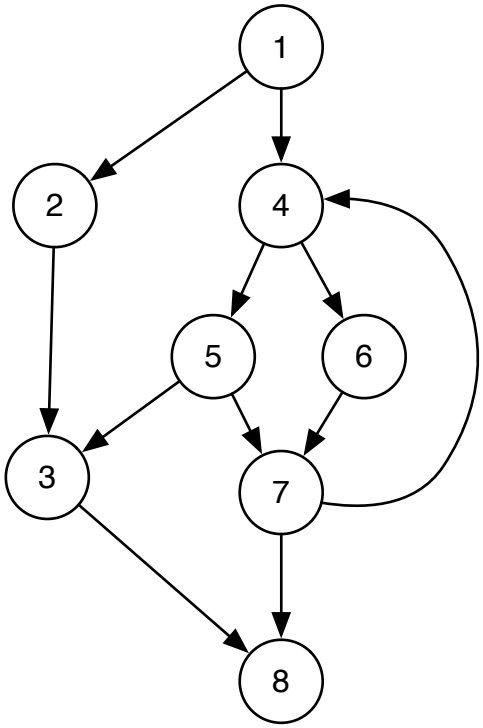
$y_1 = 1$

$y_2 = 2$

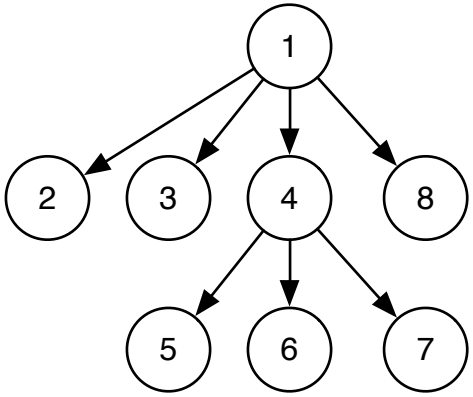
$x_1 = y_2$

# Dominance Tree

CFG



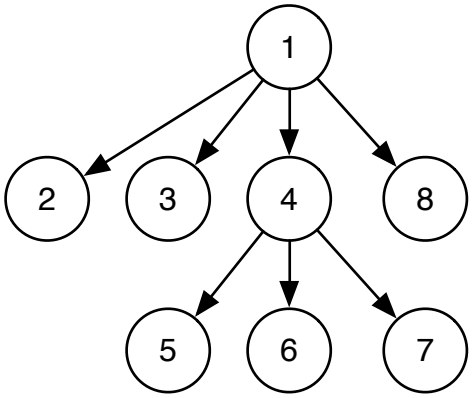

Dominance Tree



# Variable Renaming (SSA Form)

- Data Structure
  - Stacks[v] : Holds the most recent definition of variable v, initially empty
- Use Dominance Tree to maintain Variable name stacks

```
procedure GenName(variable v)
  vn = new Value(v)
  Push vn onto Stacks[v]
  return vn
```



# Variable Renaming Algorithm

Replace with most recent definition of v On the stack

DFS in DT

Unwind the stack

```
call Rename(entry)

procedure Rename(block b)
  if b previously visited return
  for each φ-function p in b
    v = LHS(p)
    vn = GenName(v) and replace v with vn
  for each statement s in b (in order)
    for each variable v ∈ RHS(s)
      replace v by Top(Stacks[v])
    for each variable v ∈ LHS(s)
      vn = GenName(v) and replace v with vn
  for each s ∈ succ(b) (in CFG)
    j ← position in s's φ-function corresponding to block b
    for each φ-function p in s
      replace the jth operand of RHS(p) by Top(Stacks[v])
  for each s ∈ child(b) (in DT)
    Rename(s)
  for each φ-function or statement t in b
    for each vi ∈ LHS(t)
      Pop(Stacks[v])
```

# After Variable Renaming

After Renaming Variables (in two different representations)

```
MOV Stack0_1, Const0
MOV Local1_1, Stack0_1
MOV Stack0_2, Local1_1
MOV Stack1_1, Const20
ADD Stack0_3, Stack1_1, Stack0_2
MOV Local2_1, Stack0_3
MOV Stack0_4, Local2_1
MOV Local3_1, Stack0_4
MOV Stack0_5, Local1_2
CMP Tmp0, Stack0_5, Const0
BRCond Tmp0, BB#2, BB#1
```

```
MOV Stack0_6, Const53
MOV Local3_2, Stack0_6
```

```
Phi Local 3_3, (Local3_1, BB#1), (Local3_2, BB#2)
MOV Stack0_7, Local3_3
Call MethodName[4]
Return
```

Quadruples

```
#1 : MOV Const0
#2 : MOV (#1)
#3 : MOV (#2)
#4 : MOV Const20
#5 : ADD (#4), (#3)
#6 : MOV (#5)
#7 : MOV (#6)
#8 : MOV (#7)
#9 : MOV (#6)
#10 : CMP Tmp0, (#9), Const0
BRCond (#10), BB#2, BB#1
```

```
#12 : MOV Const53
#13 : MOV (#12)
```

```
#14 : Phi ((#8), BB#1), ((#13), BB#2)
#15 : MOV (#14)
Call MethodName[4]
Return
```

Triples

# Implementation Example

```
class Value {
  bool isInstruction;
  …
};

class AbstractVariable : public Value {
   bool isStack;
   int Index;
   …
};
class Instruction : public Value {
  Opc;
  bool hasLHS;
};
class BinaryInstruction : public Instruction {
  Value *Op1; Value *Op2; Value *Op3;
};
```

```
class PhiInstruction : public Instruction {
  Value *Dst;
  pair<BasicBlock*,Value*> InEdges[]
  …
};
…
```

# Summary

- SSA form - explicit use-def relationships
- Build CFG
  - Create Basic Blocks with the identified leaders
  - Link Basic Blocks and fill them with instructions
    - Rename operand stack variables
    - Instructions are not in SSA from yet!
- Insert Phi Functions based on the dominance frontier algorithm
- Rename variables to have SSA property

# Interpreting with SSA IR for fun