SSA Form: Construction

Lecture 4: CPEN 400P

Karthik Pattabiraman, UBC

Outline

Def-Use Chains

Why do we need SSA form ?

Dominator Trees and Dominance Frontiers

Inserting Phi-Nodes

Renaming Variables

Converting Out of SSA form

Information Chains

A tuple that connects 2 data-flow events is a *chain*

event ≅ definition or use

- Chains express data-flow relationships directly
- Chains provide a graphical representation
- Chains jump across unrelated code, simplifying search

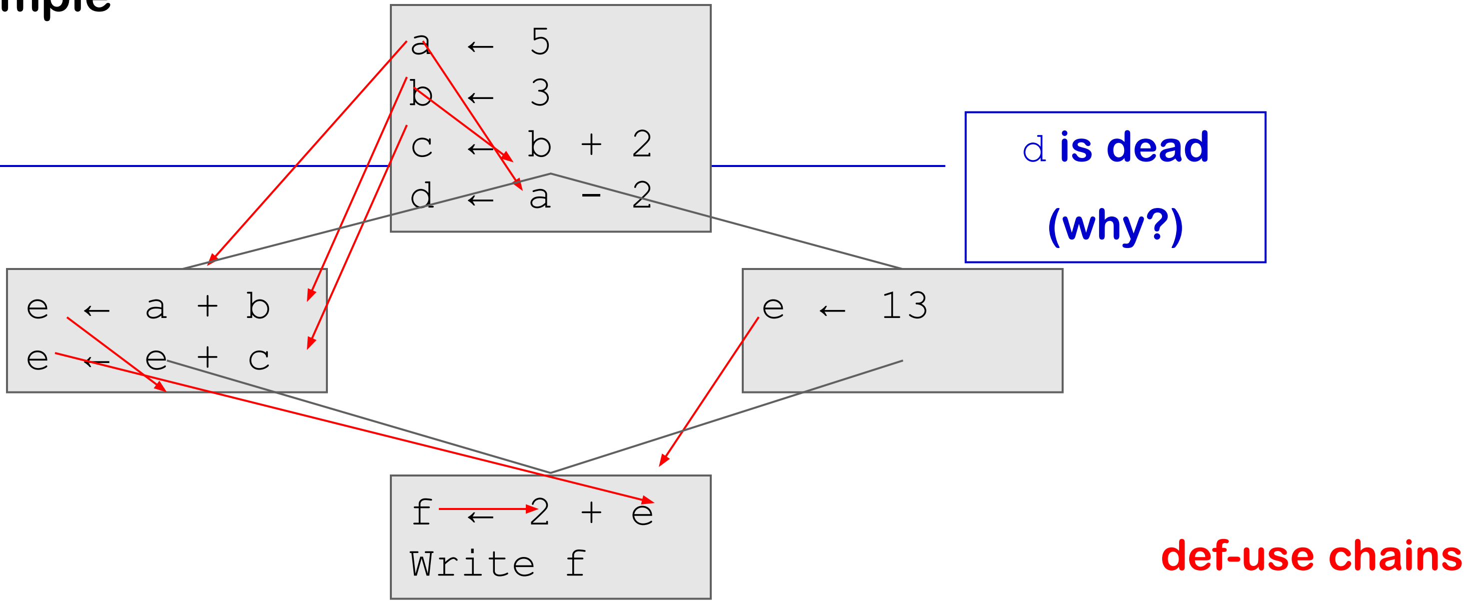
We can build chains efficiently

Def-Use chains are the most common

Four interesting types of chain

Information Chains

Example



Notation

Assume that, ∀ operation *i* and each variable *v*,

- ***DEFS(v,i)* is the set of operations that may have defined *v* most recently before *i*, along some path in the CFG**
- ***USES(v,i)* is the set of operations that may use the value of *v* computed at *i*, along some path in the CFG**

$$x \in DEFS(A,y) \Leftrightarrow y \in USES(A,x)$$

To construct DEF-USE chains, we solve *reaching definitions* (*how?*)

Domain is |definitions|, same as number of operations

Reaching Definitions

The equations

$$REACHES(n) = \emptyset,\ \forall\ n \in N$$

$$REACHES(n) = \cup_{p \in preds(n)} \left(DEDEF(p) \cup \left(REACHES(p) \cap \overline{DEFKILL(p)}\right)\right)$$

- *REACHES(n)* is the set of definitions that reach block *n*
- *DEDEF(N)* is the set of definitions in *n* that reach the end of *n*
- *DEFKILL(n)* is the set of defs obscured by a new def in *n*

Outline

Def-Use Chains

Why do we need SSA form ?

Dominator Trees and Dominance Frontiers

Inserting Phi-Nodes

Renaming Variables

Converting Out of SSA form

Issues with simple def-use Chains

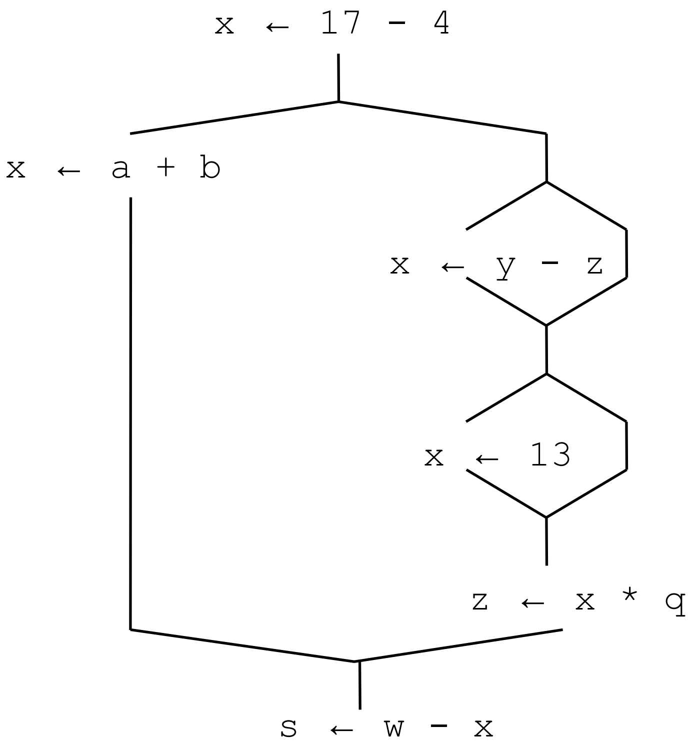
Don’t encode control-flow information and data-flow in a unique way

Can’t trace each value to it’s definition as definition may be non-unique

Update of values is dependent on number of “meet points” in standard data-flow analysis - can take a long time to converge

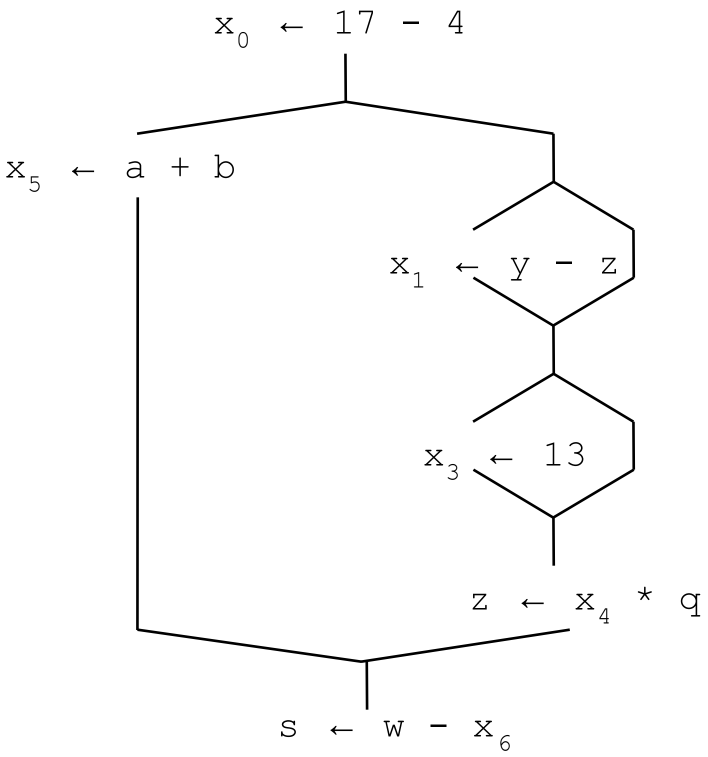
Need a way to encode both data-flow and control information efficiently to allow multiple data-flow analysis problems to be run on them

Example (without SSA form)



There are four birth points for `x`

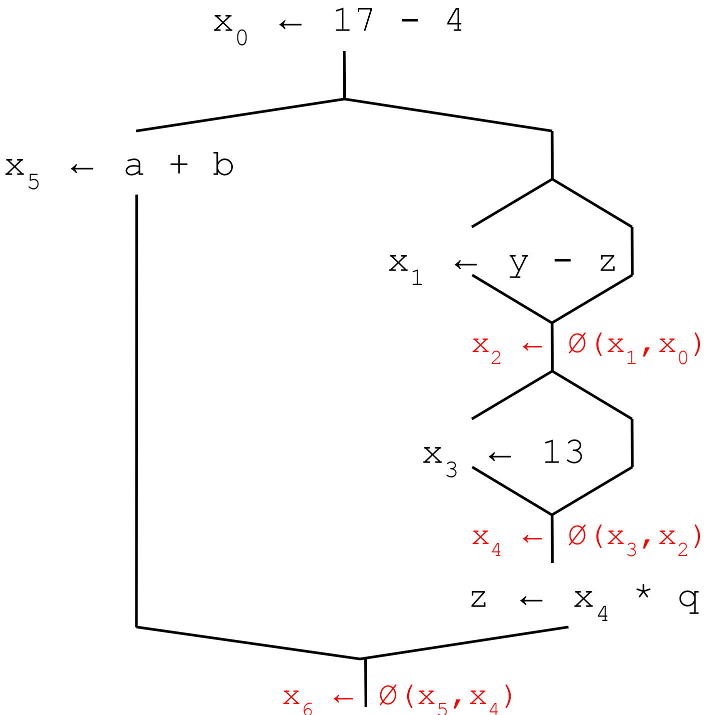
Example (without SSA form, but with renaming)



There are four birth points for x

Example (with SSA form)

Making Birth Points Explicit



Building Static Single Assignment Form

SSA-form

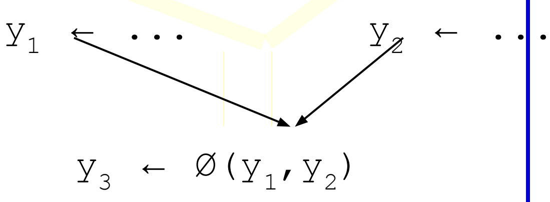
- Each name is defined exactly once
- Each use refers to exactly one name

What's hard

- Straight-line code is trivial
- Splits in the CFG are trivial
- Joins in the CFG are hard

A ϕ-function is a special kind of copy that selects one of its parameters.

The choice of parameter is governed by the CFG edge along which control reached the current block.



Few machines implement a ϕ-function in hardware.

Simple Algorithm For Building SSA form ?

Simple algorithm

1. Insert a ϕ at each join point for each name

2. Solve *reaching definitions*

3. Rename each name to get single definition & single use

This produces

- **Correct SSA form**
- **More ϕ's than any other known algorithm for SSA construction (too many to be practical)**

The rest is optimization (!)

{

SSA Construction Algorithm - 1 *(Detailed sketch)*

1. Insert ϕ-functions

   a.) calculate dominance frontiers

   b.) find global names

   for each name, build a list of blocks that define it

   c.) insert ϕ-functions

   $\forall$ global name n

   $\forall$ block b in which n is assigned

   $\forall$ block d in b's dominance frontier

   insert a ϕ-function for n in d

   add d to n's list of defining blocks

Moderately complex

Compute list of blocks where each name is assigned & use as a worklist

This adds to the worklist !

Creates the iterated dominance frontier

Use a checklist to avoid putting blocks on the worklist twice; keep another checklist to avoid inserting the same ϕ-function twice.

SSA Construction Algorithm - 2 *(Detailed sketch)*

2. Rename variables in a pre-order walk over dominator tree
   (use an array of stacks, one stack per global name)

   1 counter per name for subscripts

   Staring with the root block, *b*

   a.) generate unique names for each ϕ-function
       and push them on the appropriate stacks

   b.) rewrite each operation in the block
       i. Rewrite uses of global names with the current version
          (from the stack)
       ii. Rewrite definition by inventing & pushing new name

   c.) fill in ϕ-function parameters of successor blocks

   d.) recurse on *b*'s children in the dominator tree

   Reset the state

   e.) <on exit from block *b*> pop names generated in *b* from stacks

   Need the end-of-block name for this path

Outline

Def-Use Chains

Why do we need SSA form ?

Dominator Trees and Dominance Frontiers

Inserting Phi-Nodes

Renaming Variables

Converting Out of SSA form

Dominators (Recap)

Definition:

x <u>dominates</u> y if and only if every path from the entry of the control-flow graph to the node for y includes x

- **By definition, x <u>dominates</u> x**
- **The first entry node of a procedure dominates every block in it**
- **We associate a DOM set with each node**
- **$|DOM(x)| \geq 1$**

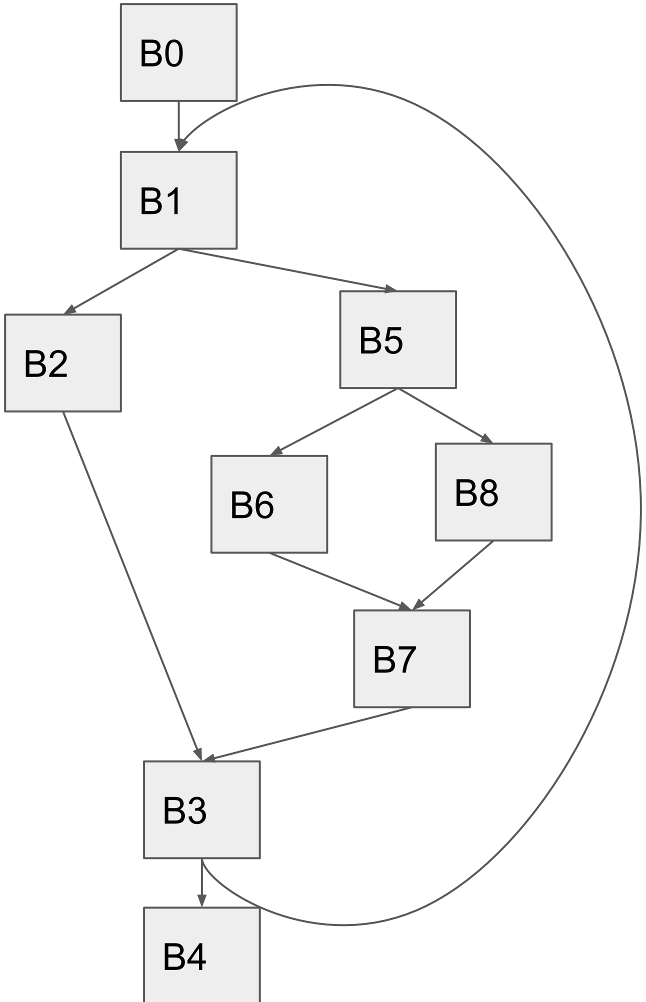
Immediate Dominators and Dominator Tree

Immediate dominator:

- **For any node *x*, there must be a *y* in DOM(*x*) closest to *x* *(different from x)***
- **We call this *y* the <u>immediate</u> <u>dominator</u> of *x***
- **Note that**
- **As a matter of notation, we write this as IDOM(*x*)**

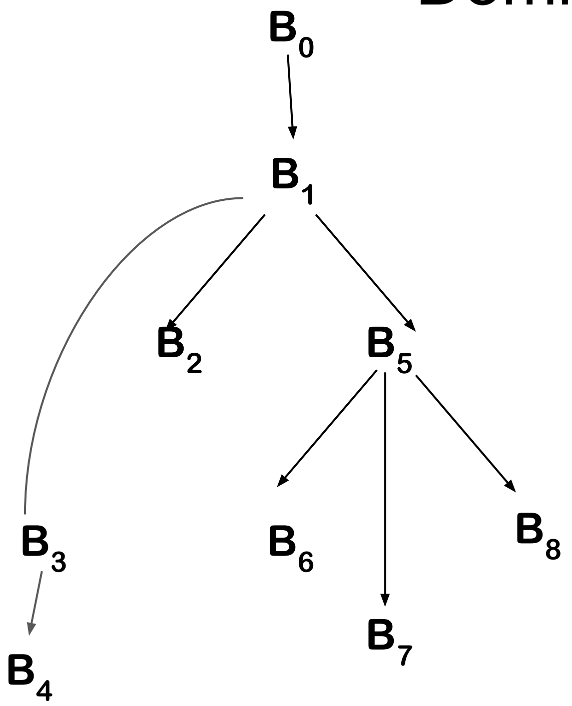
Dominator Tree: Contains every node of the flow graph and encodes IDOM set:

- If m is IDOM(n), then there is an edge in the dominator tree from m to n
- Given a node in the dominator tree, it’s parent is IDOM(n)



Basic Block	Dominator Set	IDOM
B0	{ 0 }	-
B1	{ 0, 1 }	0
B2	{ 0, 1, 2 }	1
B3	{ 0, 1, 3 }	1
B4	{ 0, 1, 3, 4 }	3
B5	{ 0, 1, 5 }	1
B6	{ 0, 1, 5, 6 }	5
B7	{ 0, 1, 5, 7 }	5
B8	{ 0, 1, 5, 8 }	5

Dominator Tree



Basic Block	Dominator Set	IDOM
B0	{ 0 }	-
B1	{ 0, 1 }	0
B2	{ 0, 1, 2 }	1
B3	{ 0, 1, 3 }	1
B4	{ 0, 1, 3, 4 }	3
B5	{ 0, 1, 5 }	1
B6	{ 0, 1, 5, 6 }	5
B7	{ 0, 1, 5, 7 }	5
B8	{ 0, 1, 5, 8 }	5

Dominance Frontiers: Intuition

Where does an assignment in block *n* induce ϕ–functions in SSA form?

- ***n DOM m*** **⇒ no need for a ϕ–function in *m***
  - **Definition in *n* blocks any previous definition from reaching *m***
- **If *m* has multiple predecessors, and *n* (strictly) dominates some of them, but not all of them, *m* needs a ϕ–function for each definition in *n***

This is also known as the dominance frontier of m - these are the locations at which phi nodes need to be inserted (with some minor optimizations)

Dominance Frontier: Formal Definition

More ***formally***, *m* **is in the dominance frontier of** *n* **if and only if**

1. $\exists\ p \in preds(m)$ such that $n \in DOM(p)$, and
2. *n* does not ***strictly dominate*** *m* $(n \notin DOM(m) - \{\ m\ \})$

- **This notion of dominance frontier is precisely what we need to insert ϕ–functions:**

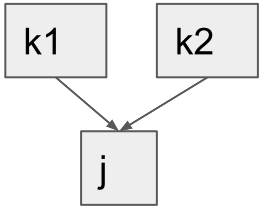
  a def in block n induces a ϕ–function in each block in DF(n).

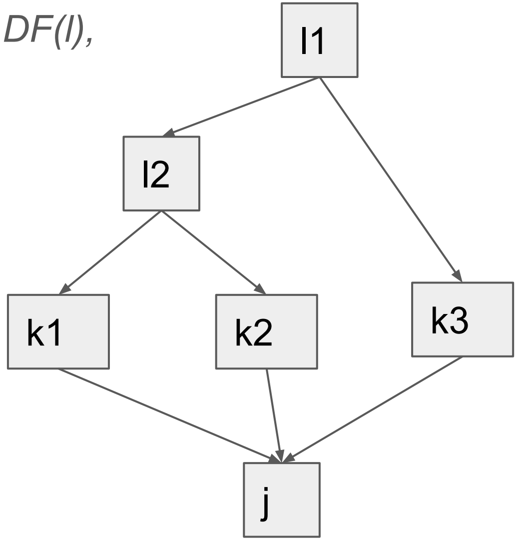
Why do you need "strict" dominance in the above definition ?

- ***Single loop basic blocks (as n doesn't strictly dominate itself)***

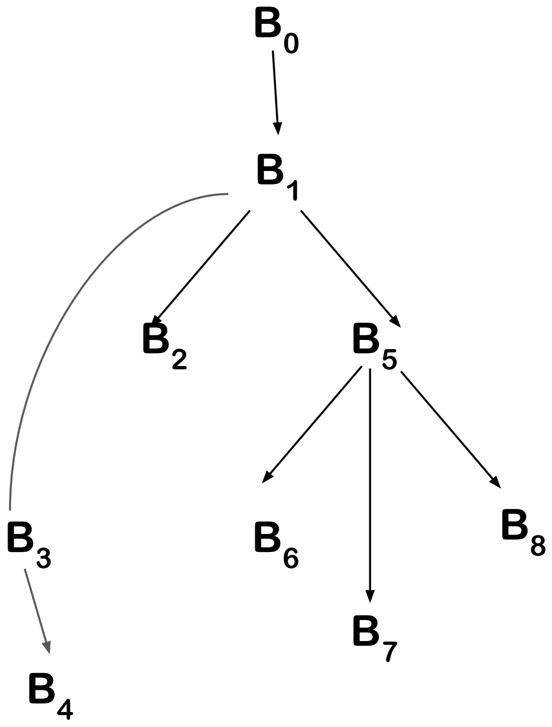
Algorithm for computing DFs: Observations

1. Only nodes in the join points of the CFG can be in the dominance frontier
2. For a joint point *j*, each predecessor *k* of *j* must have *j* in *DF(k)* (Why ?)
3. If *j* is in *DF(k)* for some predecessor *k,* then
   - For each node *l* in *DOM(k), j* must also be in *DF(l),*
   - Unless l is in DOM(j) (Why ?)





Algorithm for Computing DFs: Intuition



Computing Dominance Frontiers

- Only join points are in *DF(n)* for some *n*
- Leads to a simple, intuitive algorithm for computing dominance frontiers

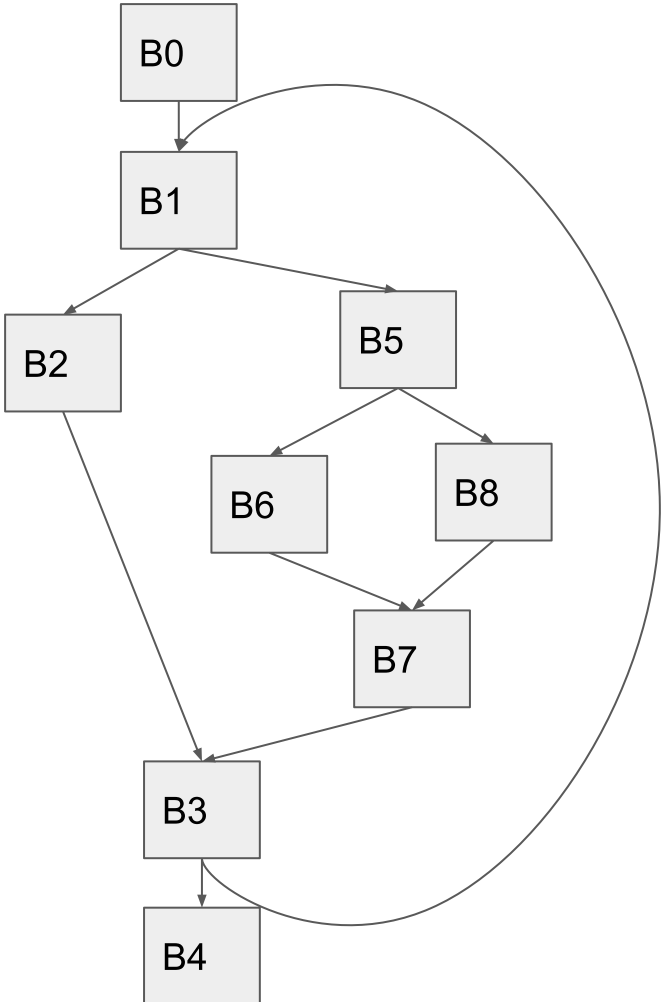
For each join point *x* (*i.e.*, |*preds(x)*| > 1)

For each *CFG* predecessor *p* of *x*

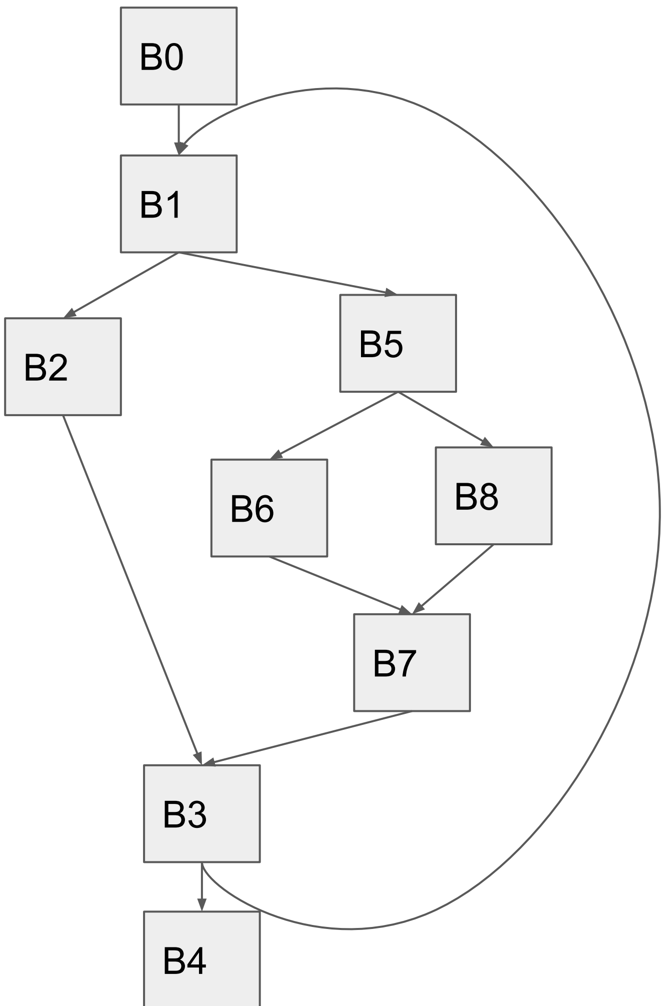
Run from p to *IDOM(x)* *in the dominator tree*, & add *x* to *DF(n)* for each *n* from *p* up to but not *IDOM(x)*

Algorithm for computing DFs: Pseudo-code

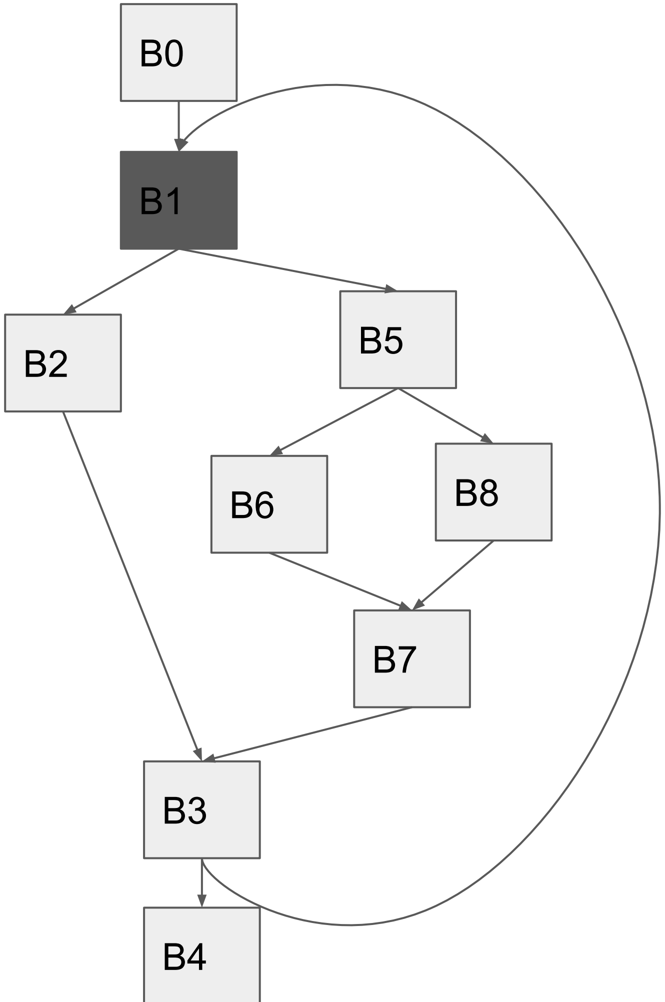
```
For all nodes, n, in the CFG
    DF(n) ← ϕ
For all nodes, n, in the CFG
    If n has multiple predecessors then
        For each predecessor p of n
            runner ← p
            While ( runner =/= IDOM(n) )
                DF(runner) ← DF(runner) ∪ { n }      // Add the node to the DF of the runner
                runner ← IDOM(runner)                 // Go to predecessor node in the DOM Tree
```



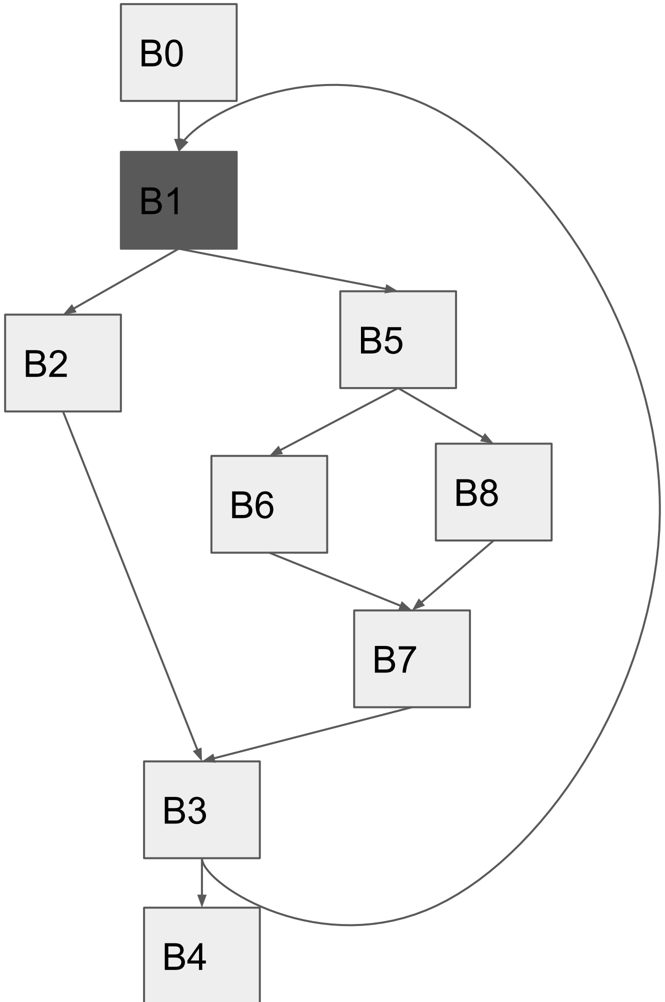
Basic Block	IDOM	DF
B0	-	
B1	0	
B2	1	
B3	1	
B4	3	
B5	1	
B6	5	
B7	5	
B8	5	



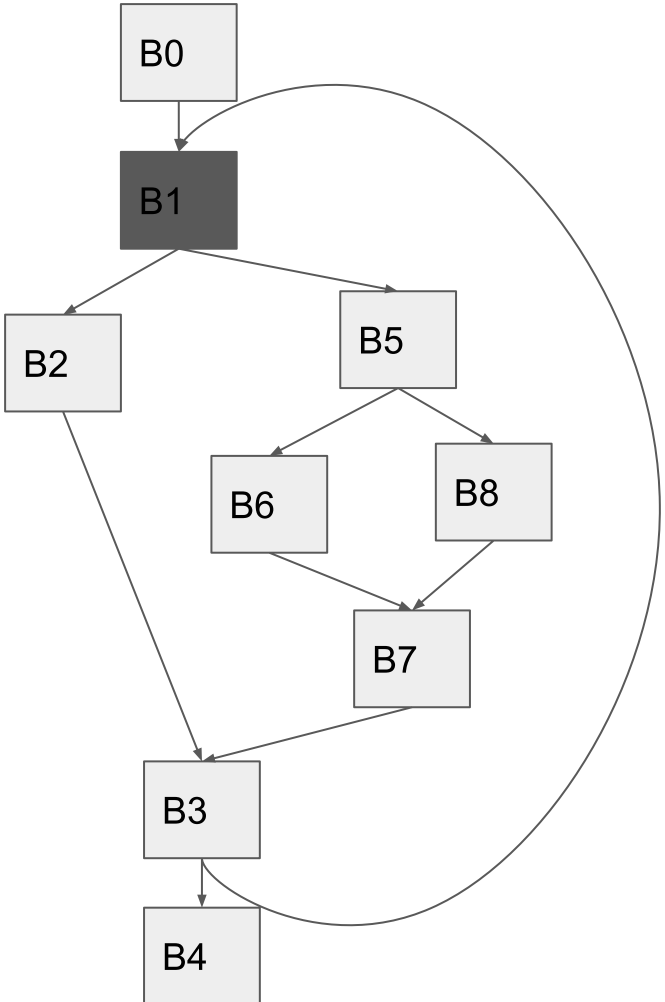
Basic Block	IDOM	DF
B0	-	
B1	0	
B2	1	
B3	1	
B4	3	
B5	1	
B6	5	
B7	5	
B8	5	



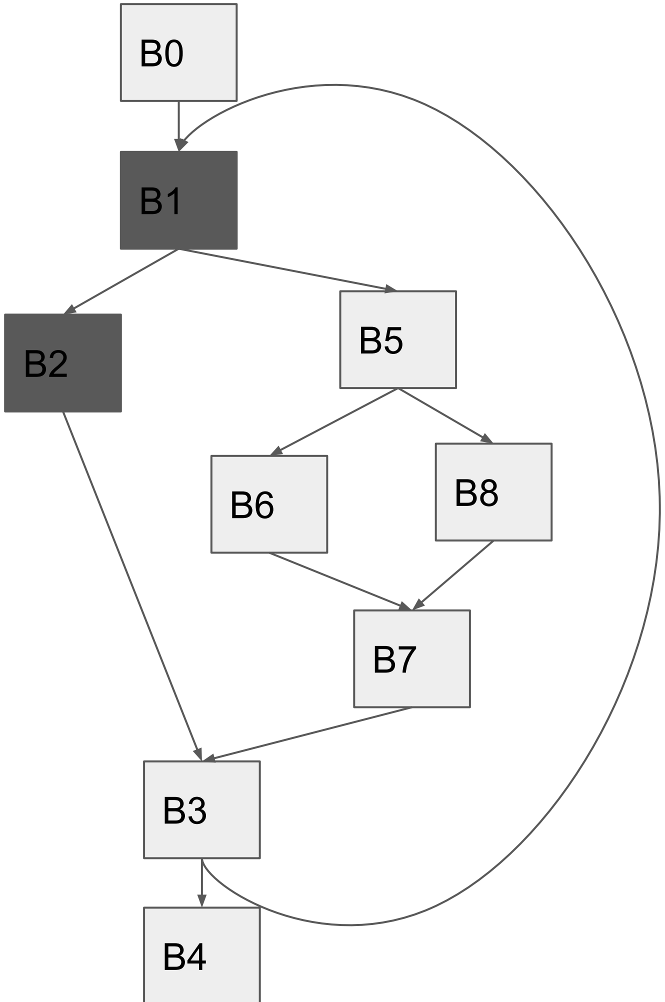
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ }
B2	1	{ }
B3	1	{ }
B4	3	{ }
B5	1	{ }
B6	5	{ }
B7	5	{ }
B8	5	{ }



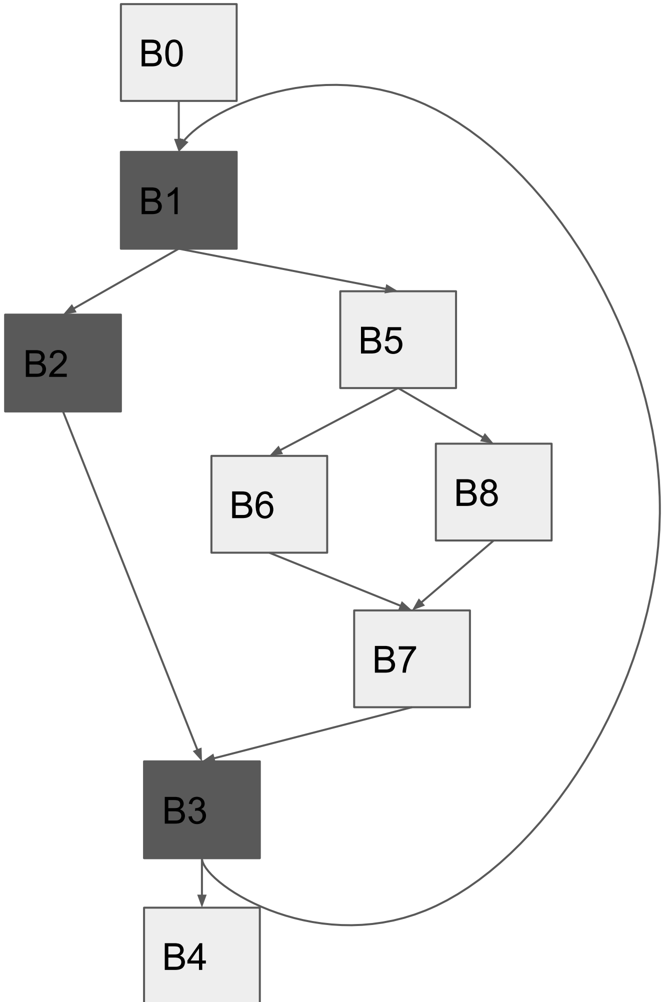
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ }
B2	1	{ }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ }
B6	5	{ }
B7	5	{ }
B8	5	{ }



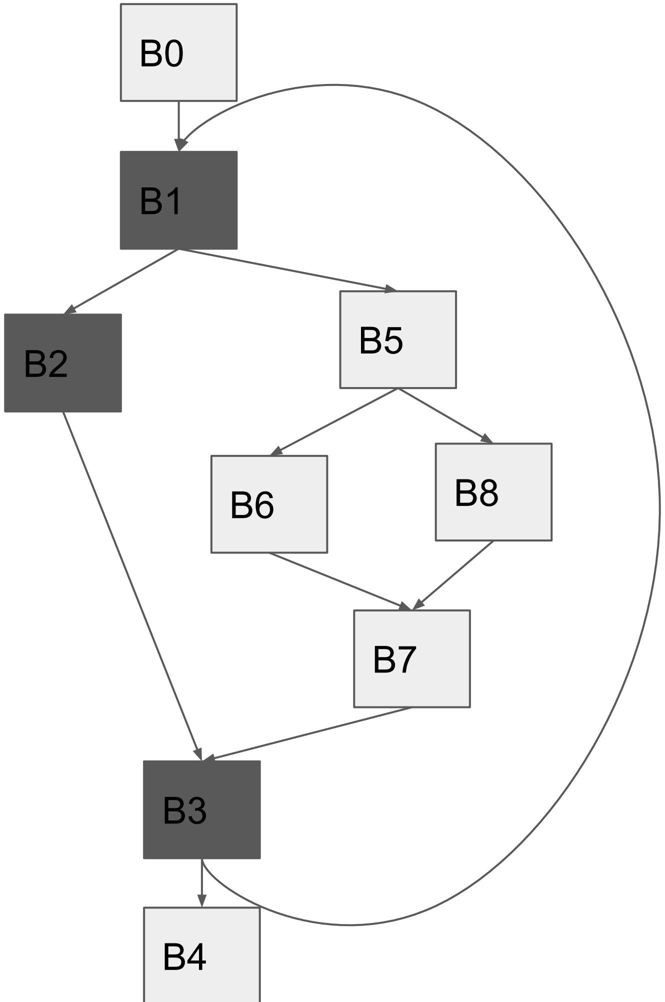
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ }
B6	5	{ }
B7	5	{ }
B8	5	{ }



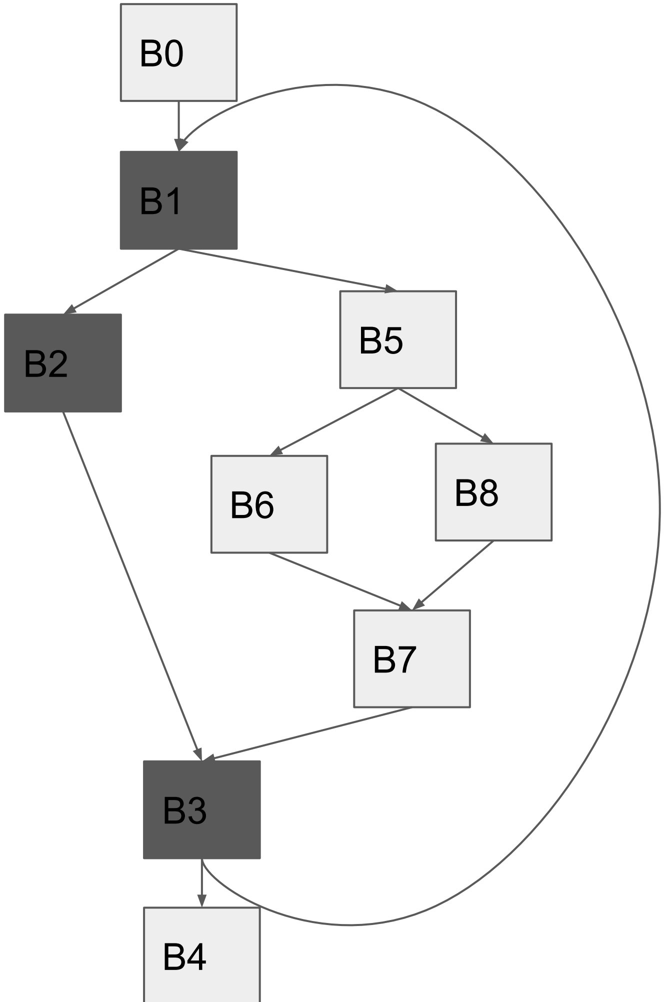
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ }
B6	5	{ }
B7	5	{ }
B8	5	{ }



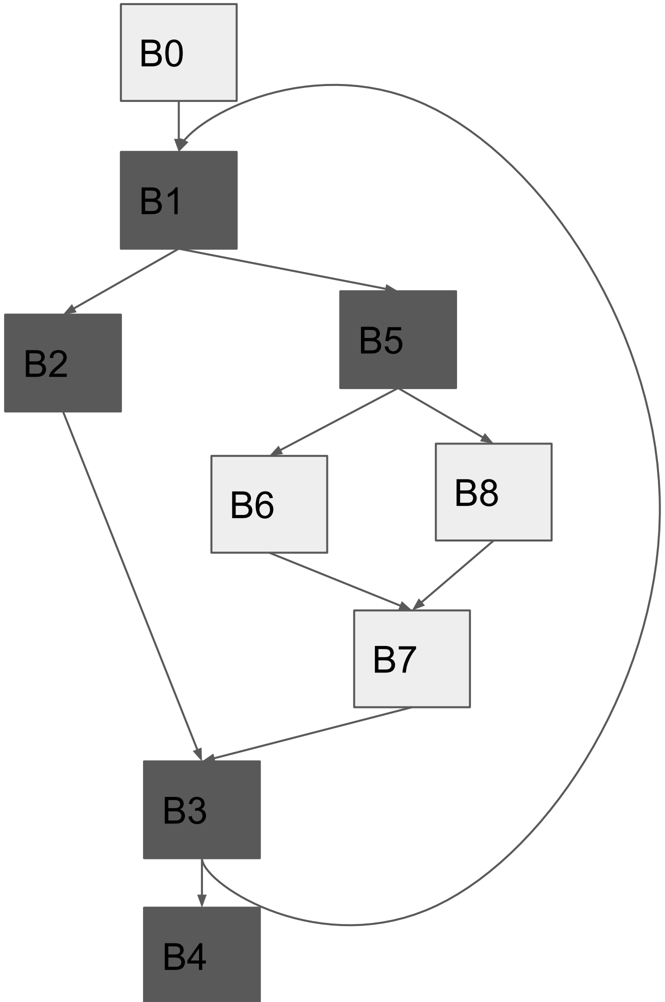
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ 3 }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ }
B6	5	{ }
B7	5	{ }
B8	5	{ }



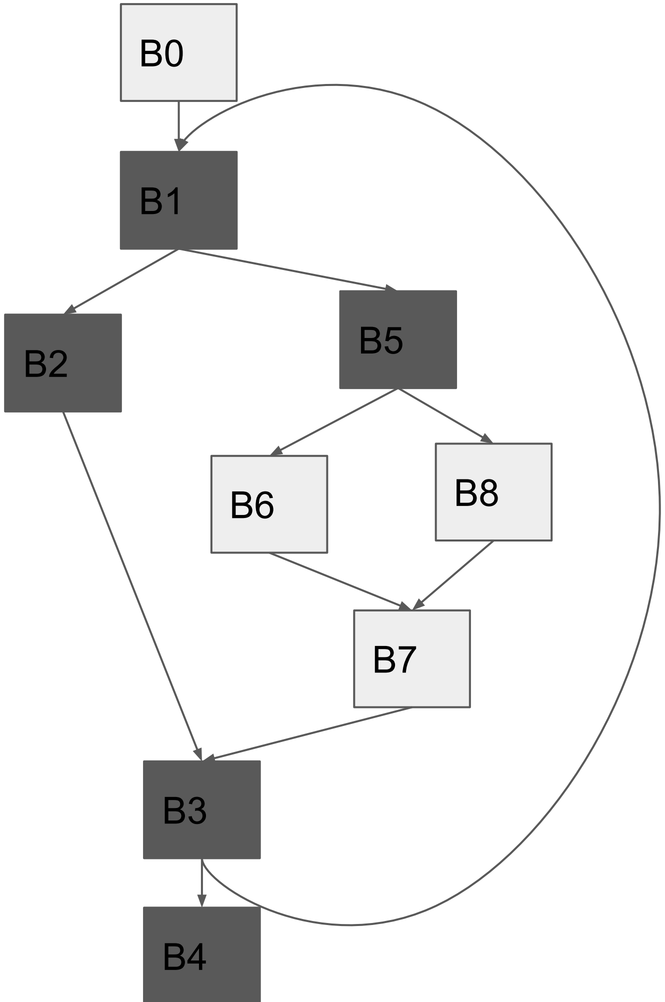
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ 3 }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ }
B6	5	{ }
B7	5	{ 3 }
B8	5	{ }



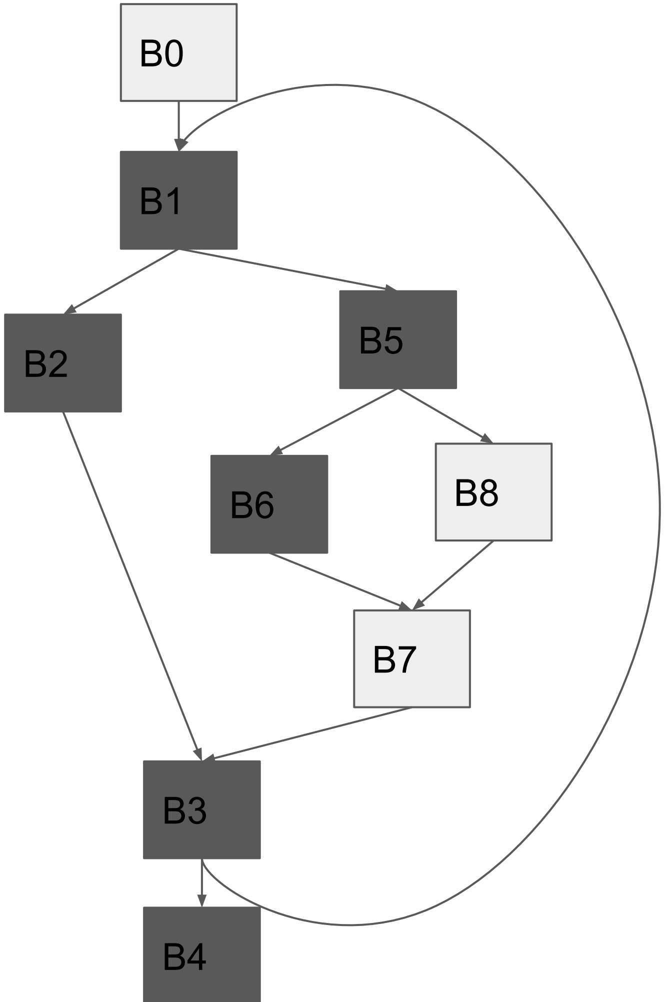
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ 3 }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ 3 }
B6	5	{ }
B7	5	{ 3 }
B8	5	{ }



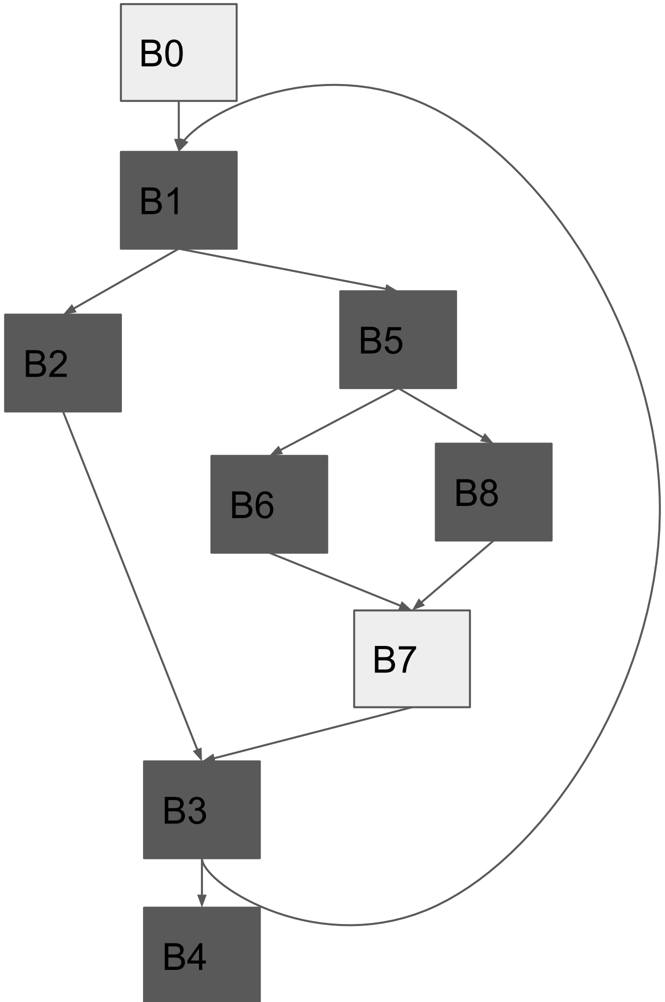
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ 3 }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ 3 }
B6	5	{ }
B7	5	{ 3 }
B8	5	{ }



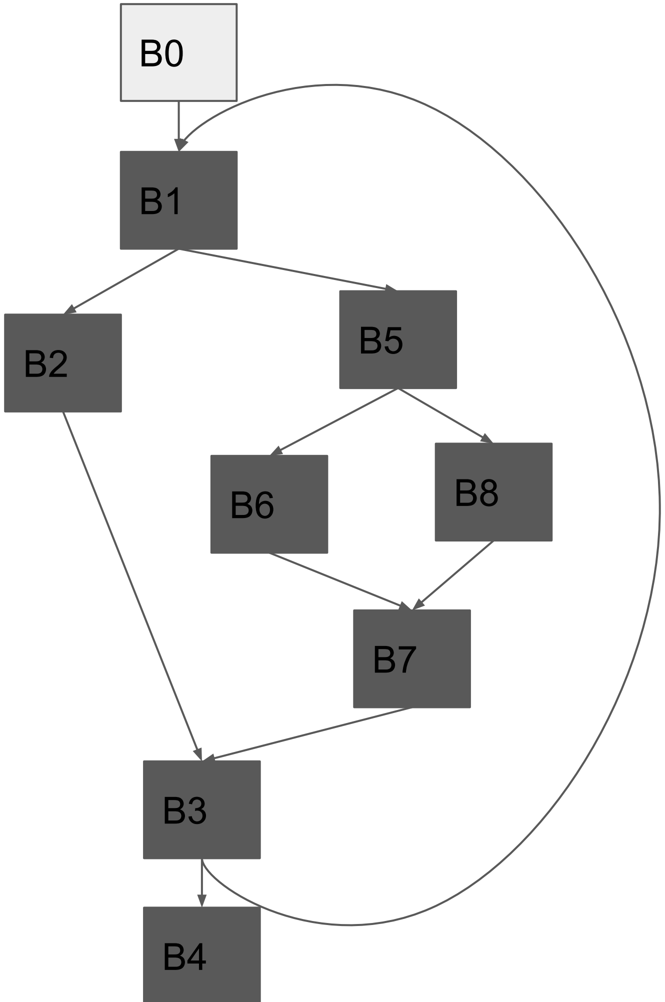
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ 3 }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ 3 }
B6	5	{ }
B7	5	{ 3 }
B8	5	{ }



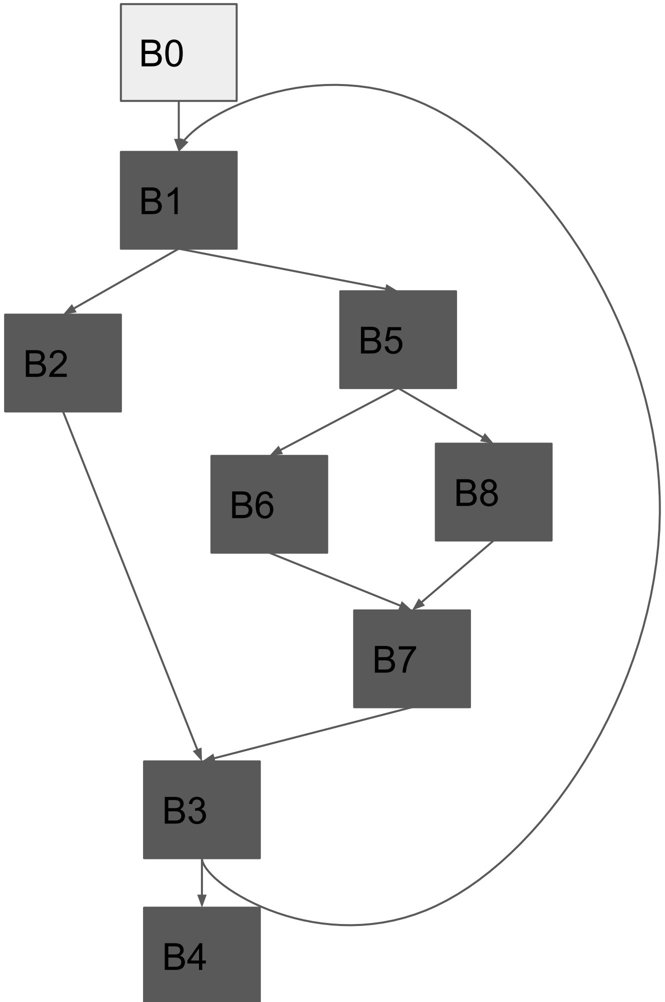
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ 3 }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ 3 }
B6	5	{ }
B7	5	{ 3 }
B8	5	{ }



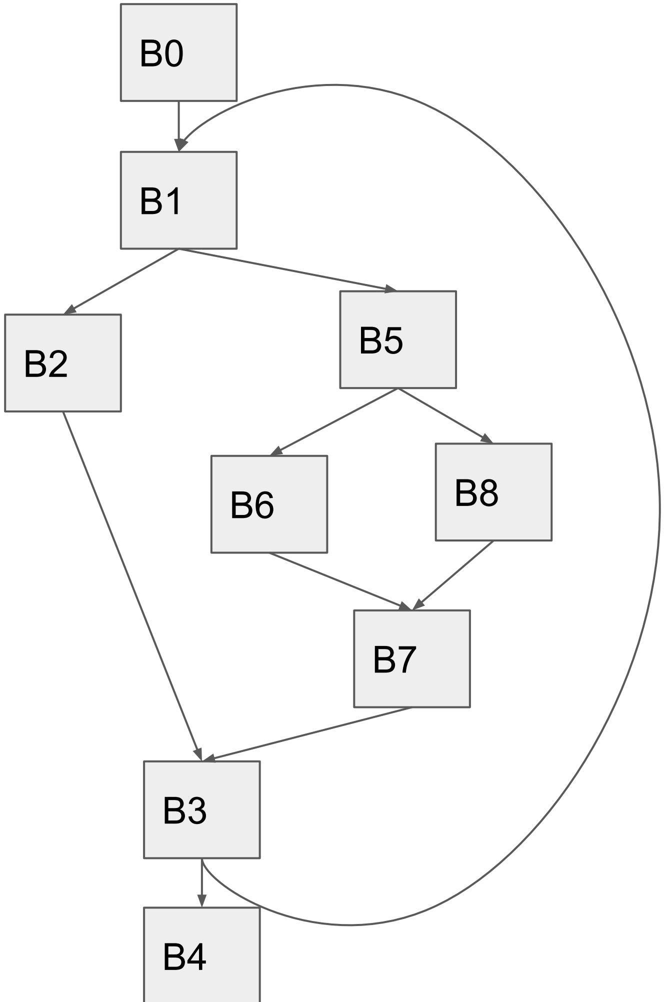
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ 3 }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ 3 }
B6	5	{ }
B7	5	{ 3 }
B8	5	{ }



Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ 3 }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ 3 }
B6	5	{ 7 }
B7	5	{ 3 }
B8	5	{ }



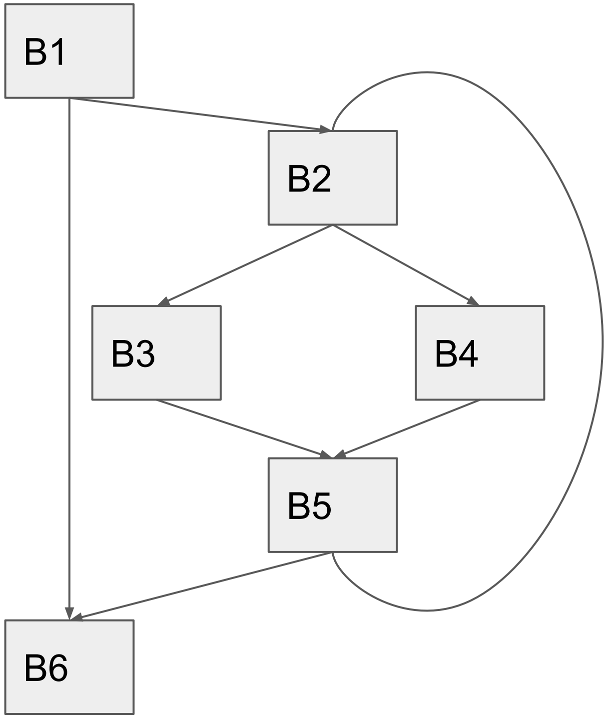
Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ 3 }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ 3 }
B6	5	{ 7 }
B7	5	{ 3 }
B8	5	{ 7 }



Basic Block	IDOM	DF
B0	-	{ }
B1	0	{ 1 }
B2	1	{ 3 }
B3	1	{ 1 }
B4	3	{ }
B5	1	{ 3 }
B6	5	{ 7 }
B7	5	{ 3 }
B8	5	{ 7 }

Class Activity

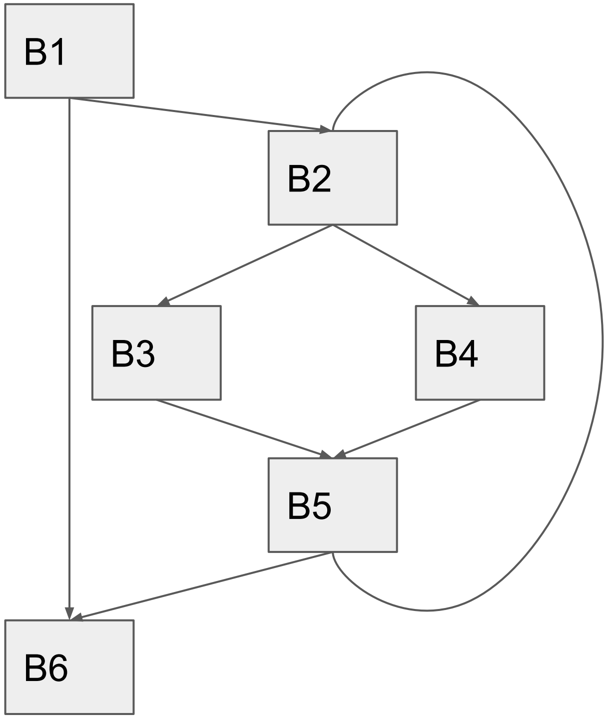
Calculate the DF for each node in the CFG below using the DF algorithm.



Block	DOM	IDOM	DF
B1			
B2			
B3			
B4			
B5			
B6			

Class Activity: Solution

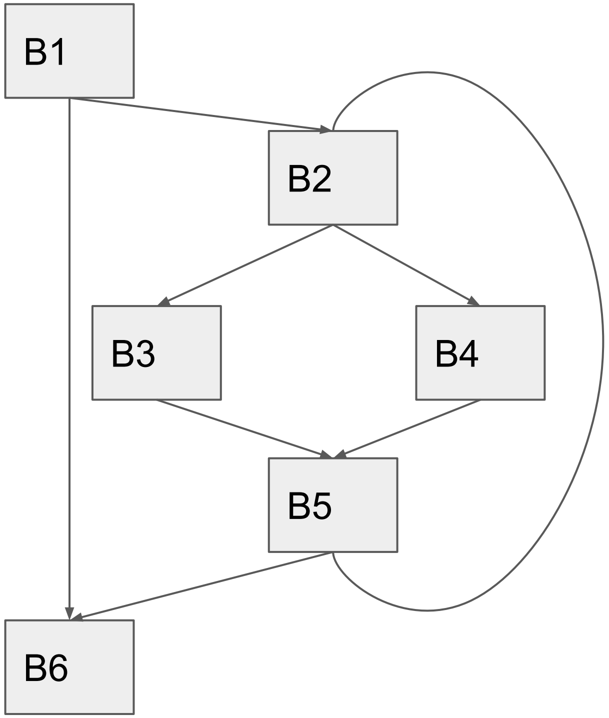
Calculate the DF for each node in the CFG below using the DF algorithm.



Block	DOM	IDOM	DF
B1	{ }		
B2	{ 1 , 2 }		
B3	{ 1, 2, 3 }		
B4	{ 1, 2, 4 }		
B5	{ 1, 2, 5 }		
B6	{ 1, 6 }		

Class Activity: Solution

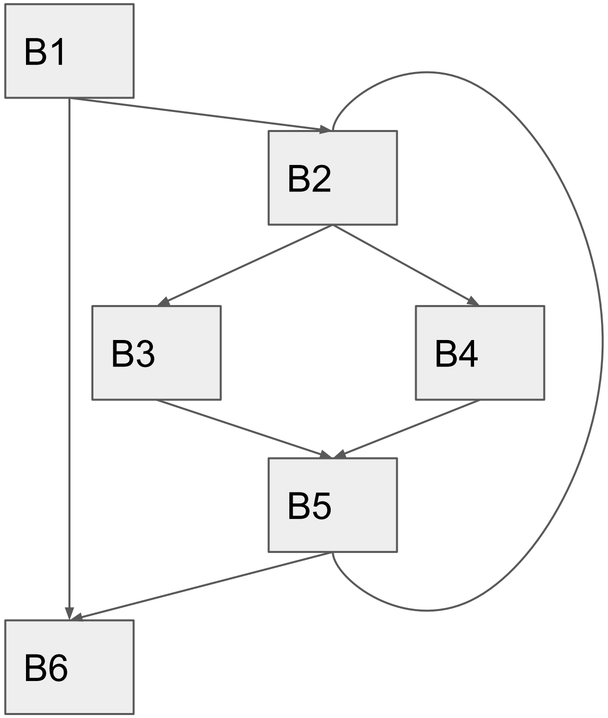
Calculate the DF for each node in the CFG below using the DF algorithm.



Block	DOM	IDOM	DF
B1	{ }	-	{
B2	{ 1 , 2 }	1	
B3	{ 1, 2, 3 }	2	
B4	{ 1, 2, 4 }	2	
B5	{ 1, 2, 5 }	2	
B6	{ 1, 6 }	1	

Class Activity: Solution

Calculate the DF for each node in the CFG below using the DF algorithm.



Block	DOM	IDOM	DF
B1	{ }	-	{ }
B2	{ 1 , 2 }	1	{ 2, 6 }
B3	{ 1, 2, 3 }	2	{ 5 }
B4	{ 1, 2, 4 }	2	{ 5 }
B5	{ 1, 2, 5 }	2	{ 2, 6 }
B6	{ 1, 6 }	1	{ }

Outline

Def-Use Chains

Why do we need SSA form ?

Dominator Trees and Dominance Frontiers

Inserting Phi-Nodes

Renaming Variables

Converting Out of SSA form

Now that we have DFs, how do we insert the Phi-Nodes ?

Insert phi nodes in all blocks that are in the DF(b), where b is the block in which the variable ‘x’ is defined

- DF(b) represents the first join-point in the CFG in which ‘x’ does NOT dominate any downstream uses of ‘x’.
- Phi-node ensures that the invariant of SSA is preserved, every def dominates all its uses
- Can be pruned if the variable ‘x’ is not Live across multiple basic blocks (any variable that is live across multiple blocks is added to *Globals)*
- Variables that are live across basic blocks can be computed as UEVar(b)

Pseudo-code for algorithm

For each name *x* in *Globals*:

    WorkList ← Blocks(*x*)   // Set of Basic Blocks in which 'x' is defined

    For each block b in the WorkList:

        For each block *d* in *DF(b)*:

            If *d* has no ϕ for *x* in *d* then:

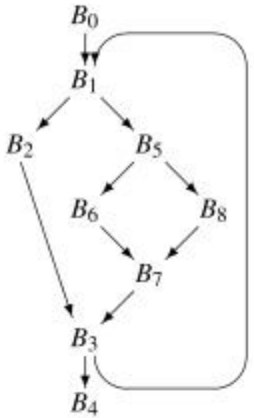
                Insert a ϕ function for *x* in *d*

                WorkList ← WorkList ∪ { d }   // Why do we need this ?

Running Example (Same as Class Activity in DFA lecture)

```
B0:  i ← 1
     → B1
B1:  a ← ···
     c ← ···
     (a < c) → B2,B5
B2:  b ← ···
     c ← ···
     d ← ···
     → B3
B3:  y ← a + b
     z ← c + d
     i ← i + 1
     (i ≤ 100) → B1,B4

B4:  return
B5:  a ← ···
     d ← ···
     (a ≤ d) → B6,B8
B6:  d ← ···
     → B7
B7:  b ← ···
     → B3
B8:  c ← ···
     → B7
```

(a) Code for the Basic Blocks



(b) Control-Flow Graph

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
UEVAR	∅	∅	∅	{a,b,c,d,i}	∅	∅	∅	∅	∅
VARKILL	{i}	{a,c}	{b,c,d}	{y,z,i}	∅	{a,d}	{d}	{b}	{c}

(c) Initial Information

Block	IDOM	DF	Variables def
B0	-	{ }	{ i }
B1	0	{ 1 }	{ a, c }
B2	1	{ 3 }	{ b, c, d }
B3	1	{ 1 }	{ y, z, i }
B4	3	{ }	{ }
B5	1	{ 3 }	{ a, d }
B6	5	{ 7 }	{ d }
B7	5	{ 3 }	{ b }
B8	5	{ 7 }	{ c }

Algorithm for inserting Phi Nodes

Globals = { a, b, c, d, i }

Blocks

Var	Blocks	Phi Nodes
a	{ 1, 5 }	
b	{ 2, 7 }	
c	{ 1, 2, 8 }	
d	{ 2, 5, 6 }	
i	{ 0, 3 }	

WorkList: [ ]

Block	IDOM	DF	Variables def
B0	-	{ }	{ i }
B1	0	{ 1 }	{ a, c }
B2	1	{ 3 }	{ b, c, d }
B3	1	{ 1 }	{ y, z, i }
B4	3	{ }	{ }
B5	1	{ 3 }	{ a, d }
B6	5	{ 7 }	{ d }
B7	5	{ 3 }	{ b }
B8	5	{ 7 }	{ c }

Algorithm for inserting Phi Nodes

Globals = { a, b, c, d, i }

Blocks

Var	Blocks	Phi Nodes
a	{ 1, 5 }	
b	{ 2, 7 }	
c	{ 1, 2, 8 }	
d	{ 2, 5, 6 }	
i	{ 0, 3 }	

WorkList: [1, 5]

Block	IDOM	DF	Variables def
B0	-	{ }	{ i }
B1	0	{ 1 }	{ a, c }
B2	1	{ 3 }	{ b, c, d }
B3	1	{ 1 }	{ y, z, i }
B4	3	{ }	{ }
B5	1	{ 3 }	{ a, d }
B6	5	{ 7 }	{ d }
B7	5	{ 3 }	{ b }
B8	5	{ 7 }	{ c }

Algorithm for inserting Phi Nodes

Globals = { a, b, c, d, i }

Blocks

Var	Blocks	Phi Nodes
a	{ 1, 5 }	{ 1 }
b	{ 2, 7 }	
c	{ 1, 2, 8 }	
d	{ 2, 5, 6 }	
i	{ 0, 3 }	

WorkList: [5, 1]

Block	IDOM	DF	Variables def
B0	-	{ }	{ i }
B1	0	{ 1 }	{ a, c }
B2	1	{ 3 }	{ b, c, d }
B3	1	{ 1 }	{ y, z, i }
B4	3	{ }	{ }
B5	1	{ 3 }	{ a, d }
B6	5	{ 7 }	{ d }
B7	5	{ 3 }	{ b }
B8	5	{ 7 }	{ c }

Algorithm for inserting Phi Nodes

Globals = { a, b, c, d, i }

Blocks

Var	Blocks	Phi Nodes
a	{ 1, 5 }	{ 1, 3 }
b	{ 2, 7 }	
c	{ 1, 2, 8 }	
d	{ 2, 5, 6 }	
i	{ 0, 3 }	

WorkList: [1, 3]

Block	IDOM	DF	Variables def
B0	-	{ }	{ i }
B1	0	{ 1 }	{ a, c }
B2	1	{ 3 }	{ b, c, d }
B3	1	{ 1 }	{ y, z, i }
B4	3	{ }	{ }
B5	1	{ 3 }	{ a, d }
B6	5	{ 7 }	{ d }
B7	5	{ 3 }	{ b }
B8	5	{ 7 }	{ c }

Algorithm for inserting Phi Nodes

Globals = { a, b, c, d, i }

Blocks

Var	Blocks	Phi Nodes
a	**{ 1, 5 }**	**{ 1, 3 }**
b	{ 2, 7 }	
c	{ 1, 2, 8 }	
d	{ 2, 5, 6 }	
i	{ 0, 3 }	

WorkList: [ ]

Block	IDOM	DF	Variables def
B0	-	{ }	{ i }
B1	0	{ 1 }	{ a, c }
B2	1	{ 3 }	{ b, c, d }
B3	1	{ 1 }	{ y, z, i }
B4	3	{ }	{ }
B5	1	{ 3 }	{ a, d }
B6	5	{ 7 }	{ d }
B7	5	{ 3 }	{ b }
B8	5	{ 7 }	{ c }

Algorithm for inserting Phi Nodes

Globals = { a, b, c, d, i }

Blocks

Var	Blocks	Phi Nodes
a	**{ 1, 5 }**	**{ 1, 3 }**
b	{ 2, 7 }	
c	{ 1, 2, 8 }	
d	{ 2, 5, 6 }	
i	{ 0, 3 }	

WorkList: [2, 7]

Block	IDOM	DF	Variables def
B0	-	{ }	{ i }
B1	0	{ 1 }	{ a, c }
B2	1	{ 3 }	{ b, c, d }
B3	1	{ 1 }	{ y, z, i }
B4	3	{ }	{ }
B5	1	{ 3 }	{ a, d }
B6	5	{ 7 }	{ d }
B7	5	{ 3 }	{ b }
B8	5	{ 7 }	{ c }

Algorithm for inserting Phi Nodes

Globals = { a, b, c, d, i }

Blocks

Var	Blocks	Phi Nodes
a	**{ 1, 5 }**	**{ 1, 3 }**
b	{ 2, 7 }	**{ 1, 3 }**
c	{ 1, 2, 8 }	
d	{ 2, 5, 6 }	
i	{ 0, 3 }	

WorkList: [1, 2, 8]

Block	IDOM	DF	Variables def
B0	-	{ }	{ i }
B1	0	{ 1 }	{ a, c }
B2	1	{ 3 }	{ b, c, d }
B3	1	{ 1 }	{ y, z, i }
B4	3	{ }	{ }
B5	1	{ 3 }	{ a, d }
B6	5	{ 7 }	{ d }
B7	5	{ 3 }	{ b }
B8	5	{ 7 }	{ c }

Algorithm for inserting Phi Nodes

Globals = { a, b, c, d, i }

Blocks

Var	Blocks	Phi Nodes
a	**{ 1, 5 }**	**{ 1, 3 }**
b	{ 2, 7 }	**{ 1, 3 }**
c	{ 1, 2, 8 }	**{ 1, 3, 7 }**
d	{ 2, 5, 6 }	**{ 1, 3, 7 }**
i	{ 0, 3 }	

WorkList: [0, 3]

Block	IDOM	DF	Variables def
B0	-	{ }	{ i }
B1	0	{ 1 }	{ a, c }
B2	1	{ 3 }	{ b, c, d }
B3	1	{ 1 }	{ y, z, i }
B4	3	{ }	{ }
B5	1	{ 3 }	{ a, d }
B6	5	{ 7 }	{ d }
B7	5	{ 3 }	{ b }
B8	5	{ 7 }	{ c }

Algorithm for inserting Phi Nodes

Globals = { a, b, c, d, i }

Blocks

Var	Blocks	Phi Nodes
a	**{ 1, 5 }**	**{ 1, 3 }**
b	{ 2, 7 }	**{ 1, 3 }**
c	{ 1, 2, 8 }	**{ 1, 3, 7 }**
d	{ 2, 5, 6 }	**{ 1, 3, 7 }**
i	{ 0, 3 }	**{ 1 }**

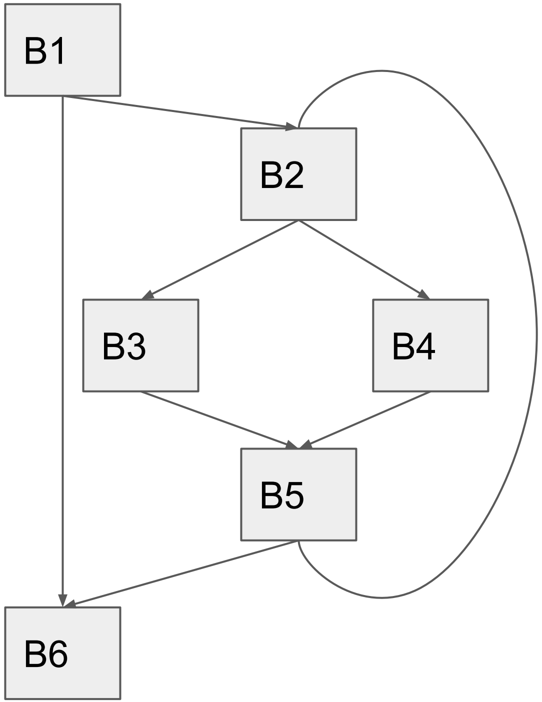
WorkList: [ ]

Block	IDOM	DF	Variables def
B0	-	{ }	{ i }
B1	0	{ 1 }	{ a, c }
B2	1	{ 3 }	{ b, c, d }
B3	1	{ 1 }	{ y, z, i }
B4	3	{ }	{ }
B5	1	{ 3 }	{ a, d }
B6	5	{ 7 }	{ d }
B7	5	{ 3 }	{ b }
B8	5	{ 7 }	{ c }

Class Activity

Insert the Phi-nodes for the example below by identifying the blocks for each variable (same example as in the class activity earlier in this lecture, and in lecture 2: SSA form)

```
B1: x ← …
    y ← …
    a ← y + 2
    b ← 0
    if (x >= a) jump B5

B2: if (y >= x) jump B4

B3:  x ← y + 1
     y ← b * 2
     jump B5

B4:  x ← y + 2
     y ← a + 1

B5:  if (x >= a) jump B2

B6: w ← x + 2
    z ← y * a
    y ← y + 1
```



Block	IDOM	DF
B1	-	{ }
B2	1	{ 2, 6 }
B3	2	{ 5 }
B4	2	{ 5 }
B5	2	{ 2, 6 }
B6	1	{ }

Class Activity: Solution

Insert the Phi-nodes for the example below by identifying the blocks for each variable (same example as in the class activity earlier in this lecture, and in lecture 2: SSA form)

Globals = { a, b, x, y, w }

Blocks

	Blocks	Phi-Nodes
x		
y		
a		
b		
w		

Block	IDOM	DF
B1	-	{ }
B2	1	{ 2, 6 }
B3	2	{ 5 }
B4	2	{ 5 }
B5	2	{ 2, 6 }
B6	1	{ }

Class Activity: Solution

Insert the Phi-nodes for the example below by identifying the blocks for each variable (same example as in the class activity earlier in this lecture, and in lecture 2: SSA form)

Globals = { a, b, x, y, w }

Blocks

	Blocks	Phi-Nodes
x	{ 1, 3, 4 }	
y	{ 1, 3, 4, 6 }	
a	{ 1 }	
b	{ 1 }	
w	{ 6 }	

Block	IDOM	DF
B1	-	{ }
B2	1	{ 2, 6 }
B3	2	{ 5 }
B4	2	{ 5 }
B5	2	{ 2, 6 }
B6	1	{ }

Class Activity: Solution

Insert the Phi-nodes for the example below by identifying the blocks for each variable (same example as in the class activity earlier in this lecture, and in lecture 2: SSA form)

Globals = { a, b, x, y, w }

Blocks

	Blocks	Phi-Nodes
x	{ 1, 3, 4 }	{ 2, 5, 6 }
y	{ 1, 3, 4, 6 }	
a	{ 1 }	
b	{ 1 }	
w	{ 6 }	

Block	IDOM	DF
B1	-	{ }
B2	1	{ 2, 6 }
B3	2	{ 5 }
B4	2	{ 5 }
B5	2	{ 2, 6 }
B6	1	{ }

Class Activity: Solution

Insert the Phi-nodes for the example below by identifying the blocks for each variable (same example as in the class activity earlier in this lecture, and in lecture 2: SSA form)

Globals = { a, b, x, y, w }

Blocks

	Blocks	Phi-Nodes
x	{ 1, 3, 4 }	{ 2, 5, 6 }
y	{ 1, 3, 4, 6 }	{ 2, 5, 6 }
a	{ 1 }	
b	{ 1 }	
w	{ 6 }	

Block	IDOM	DF
B1	-	{ }
B2	1	{ 2, 6 }
B3	2	{ 5 }
B4	2	{ 5 }
B5	2	{ 2, 6 }
B6	1	{ }

Class Activity: Solution

Insert the Phi-nodes for the example below by identifying the blocks for each variable (same example as in the class activity earlier in this lecture, and in lecture 2: SSA form)

Globals = { a, b, x, y, w }

Blocks

	Blocks	Phi-Nodes
x	{ 1, 3, 4 }	{ 2, 5, 6 }
y	{ 1, 3, 4, 6 }	{ 2, 5, 6 }
a	{ 1 }	{ }
b	{ 1 }	
w	{ 6 }	

Block	IDOM	DF
B1	-	{ }
B2	1	{ 2, 6 }
B3	2	{ 5 }
B4	2	{ 5 }
B5	2	{ 2, 6 }
B6	1	{ }

Class Activity: Solution

Insert the Phi-nodes for the example below by identifying the blocks for each variable (same example as in the class activity earlier in this lecture, and in lecture 2: SSA form)

Globals = { a, b, x, y, w }

Blocks

	Blocks	Phi-Nodes
x	{ 1, 3, 4 }	{ 2, 5, 6 }
y	{ 1, 3, 4, 6 }	{ 2, 5, 6 }
a	{ 1 }	{ }
b	{ 1 }	{ }
w	{ 6 }	

Block	IDOM	DF
B1	-	{ }
B2	1	{ 2, 6 }
B3	2	{ 5 }
B4	2	{ 5 }
B5	2	{ 2, 6 }
B6	1	{ }

Class Activity: Solution

Insert the Phi-nodes for the example below by identifying the blocks for each variable (same example as in the class activity earlier in this lecture, and in lecture 2: SSA form)

Globals = { a, b, x, y, w }

Blocks

	Blocks	Phi-Nodes
x	{ 1, 3, 4 }	{ 2, 5, 6 }
y	{ 1, 3, 4, 6 }	{ 2, 5, 6 }
a	{ 1 }	{ }
b	{ 1 }	{ }
w	{ 6 }	{ }

Block	IDOM	DF
B1	-	{ }
B2	1	{ 2, 6 }
B3	2	{ 5 }
B4	2	{ 5 }
B5	2	{ 2, 6 }
B6	1	{ }

Outline

Def-Use Chains

Why do we need SSA form ?

Dominator Trees and Dominance Frontiers

Inserting Phi-Nodes

Renaming Variables

Converting Out of SSA form

Intuition behind Renaming Algorithm

Renaming proceeds in a top-down fashion keeping track of variable lifetimes

- Use a stack to keep track of the latest name of a variable
- Push a new name onto the stack when seeing a new definition
- Use the version at the top of the stack in the uses
- Pop the name from the stack when exiting block and def goes out of scope

Renaming Variables: Algorithm Sketch

Once the Phi-Nodes have been inserted, we need to rename both Phi-nodes and regular variables (each def should have a unique index, and dominate its uses)

- Keep an array of stacks, one per variable - name at top of stack most recent
- Generate unique names for each Phi-function, and push them onto the stack
- Rewrite each operation in the block with version from top of stack
- Rewrite definition by inventing and pushing a new name
- Pop the names generated in block upon exit from the block

Perform the above operations in a recursive manner in a pre-order walk over the DOM tree (i.e., go from the root node to the children)

Renaming Algorithm: Pseudo-code

Generating new names…

```
for each global name i
    counter[i] ← 0
    stack[i] ← Ø
call Rename(n0)

NewName(n)
    i ← counter[n]
    counter[n] ← counter[n] + 1
    push n_i onto stack[n]
    return n_i
```

```
Rename(b)
    for each ϕ-function in b, x ← ϕ(…)
        rename x as NewName(x)

    for each operation "x ← y op z" in b
        rewrite y as top(stack[y])
        rewrite z as top(stack[z])
        rewrite x as NewName(x)

    for each successor of b in the CFG
        rewrite appropriate ϕ parameters

    for each successor s of b in dom. tree
        Rename(s)

    for each operation "x ← y op z" in b
        pop(stack[x])
```

Outline

Def-Use Chains

Why do we need SSA form ?

Dominator Trees and Dominance Frontiers

Inserting Phi-Nodes

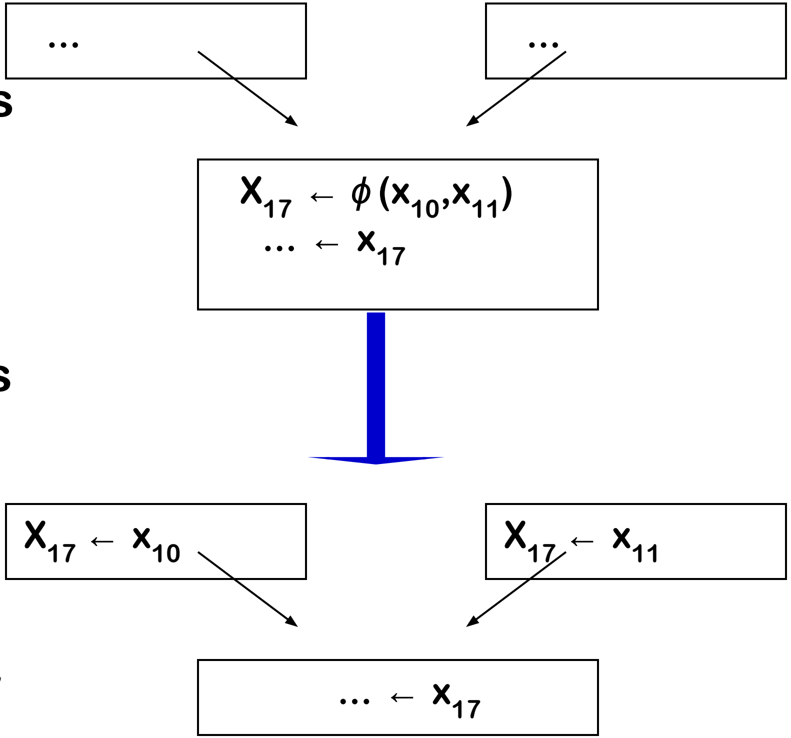
Renaming Variables

Converting Out of SSA form

SSA Deconstruction

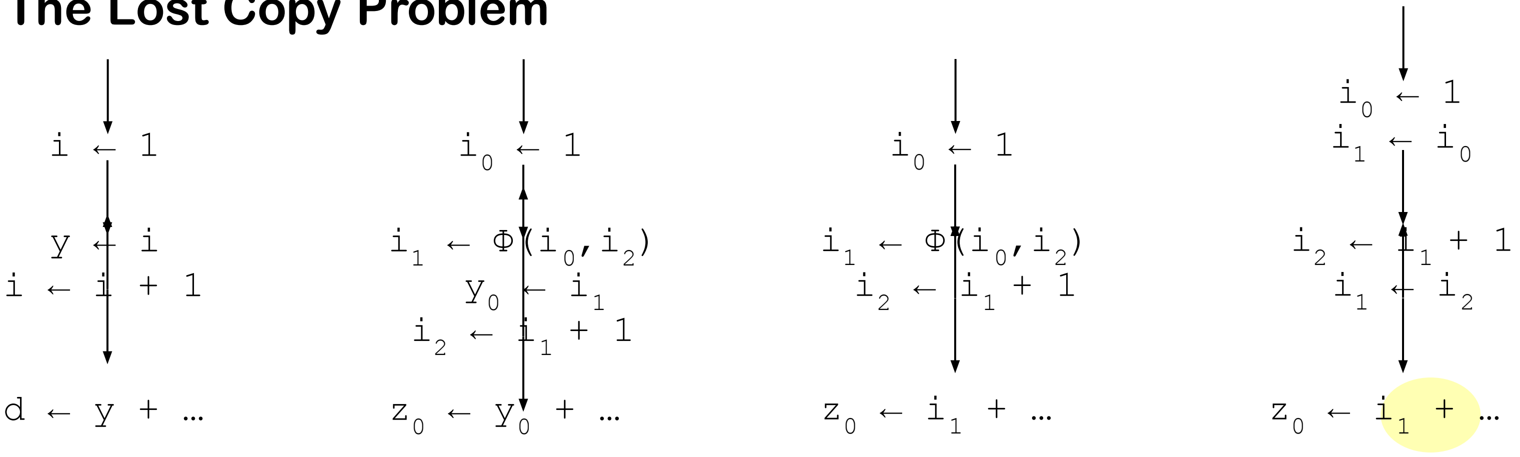
At some point, we need executable code

- Few machines implement ϕ operations
- Need to fix up the flow of values
- Basic idea
  - > Insert copies in ϕ-function pred's
  - > Simple algorithm
    - Works in most cases
  - > Adds lots of copies
    - Most of them coalesce away



Translation Out of SSA Form

The Lost Copy Problem



Original code | **In SSA form** | **With copies folded** | **Copies naïvely inserted**

Copy folding has nothing to do with either i or SSA form.

The assignment to z now receives the wrong value.

To fix this problem, the compiler needs to create a temporary name to hold the penultimate value of i.

Translation Out of SSA Form

The Swap Problem

```
x ← ...
y ← ...

t ← x
x ← y
y ← t
```

Original code

```
x0 ← ...
y0 ← ...

x1 ← Φ(x0,x2)
y1 ← Φ(y0,y2)
   t0 ← x1
   x2 ← y1
   y2 ← t0
```

In SSA Form

```
x0 ← ...
y0 ← ...

x1 ← Φ(x0,y1)
y1 ← Φ(y0,x1)
```

Copies folded

```
x0 ← ...
y0 ← ...
x1 ← x0
y1 ← y0

  x1 ← y1
  y1 ← x1
```

Copies naïvely inserted

Code is incorrect

This problem arises when a *Φ*-function argument is defined by a *Φ*-function in the same block. Requires one or more copies & temporary names.

Outline

Def-Use Chains

Why do we need SSA form ?

Dominator Trees and Dominance Frontiers

Inserting Phi-Nodes

Renaming Variables

Converting Out of SSA form