Dataflow Analysis + Intro to SSA

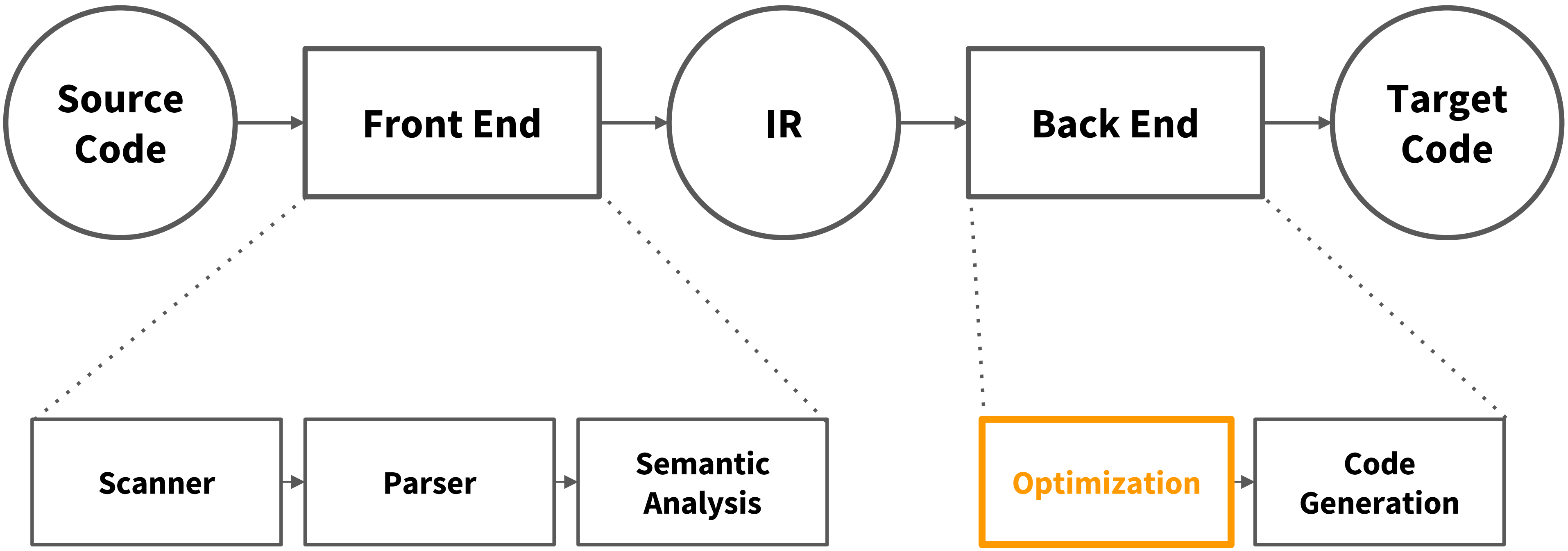
CSE 401/M501

Announcements

- 401 CodeGen due date pushed to **Saturday**
    - Contact us if you need more time, but be careful not to spill over to your report/hw4/final review
- 401 report due next Tuesday; M501 project/report due next Weekend.
- HW4 due next Thursday

23 14:30-15:20 Lecture CSE2 G10 *Dataflow (concl.); start SSA (9.3)* SSA slides 18:00-19:00 OH (Robert) CSE2 153 and Zoom	24 16:00-17:00 OH (Larry) Zoom	25 14:30-15:20 Lecture CSE2 G10 *SSA* 17:00-18:00 OH (Apollo) CSE2 153 and Zoom	26 WE ARE HERE Section *Dataflow & SSA* 15:30-16:30 OH (Jack) CSE2 151 and Zoom 20:00-21:00 OH (Morel) Zoom	27 14:30-15:20 Lecture CSE2 G10 *SSA (concl.)* 23:00 Project: code gen due **Saturday** 11 pm
30 Memorial Day	31 16:00-17:00 OH (Larry) Zoom 23:00 Project: CSE 401 project reports due	01 14:30-15:20 Lecture CSE2 G10 *Back end overview; instruction selection* 17:00-18:00 OH (Apollo) CSE2 153 and Zoom	02 Section *SSA; hw4 last-minute questions; wrapup* 15:30-16:30 OH (Jack) CSE2 151 and Zoom 20:00-21:00 OH (Morel) Zoom 23:00 hw4 due (dataflow & SSA) (SSA diagram - optional),	03 14:30-15:20 Lecture CSE2 G10 *Instruction scheduling & register allocation (no new slides); Wrapup* 23:00 CSE M 501 project due **Saturday** 11 pm 23:00 CSE M 501 report due **Sunday** 11 pm

Review of Optimizations



Review of Optimizations

Peephole

Local

Intraprocedural / Global

Interprocedural

Review of Optimizations

Peephole	A few Instructions
Local	
Intraprocedural / Global	
Interprocedural	

Review of Optimizations

Peephole	A few Instructions
Local	A Basic Block
Intraprocedural / Global	
Interprocedural	

Review of Optimizations

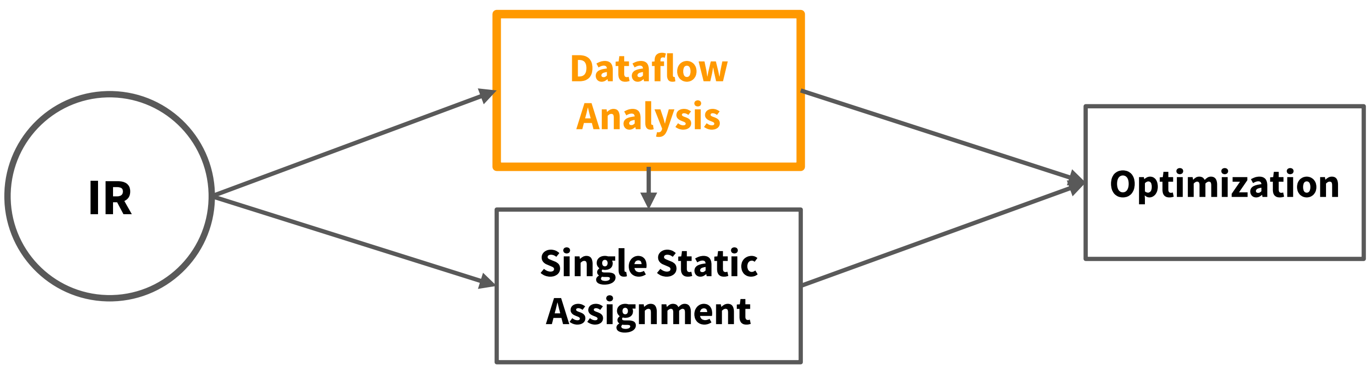
Peephole	A few Instructions
Local	A Basic Block
Intraprocedural / Global	A Function/Method
Interprocedural	

Review of Optimizations

Peephole	A few Instructions
Local	A Basic Block
Intraprocedural / Global	A Function/Method
Interprocedural	A Program

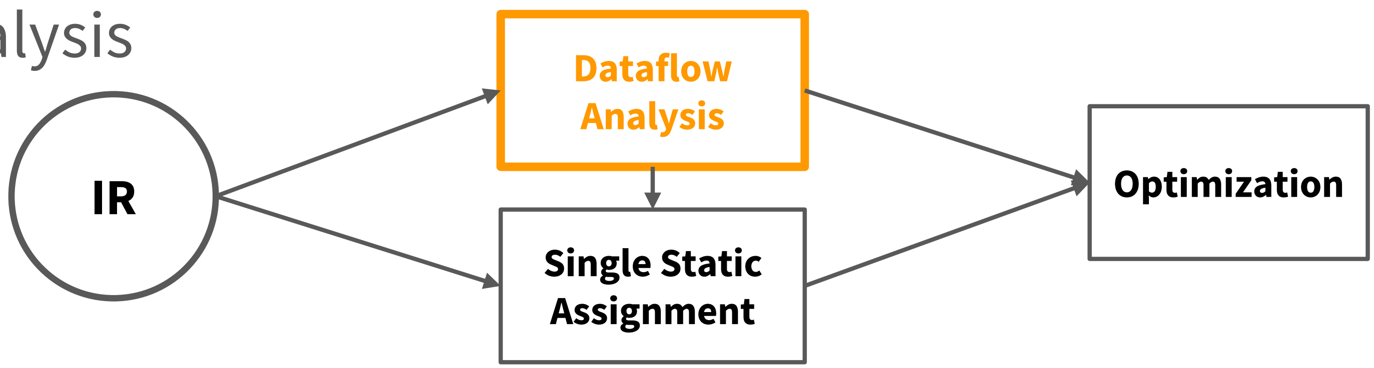
Overview of Dataflow Analysis

- A framework for exposing properties about programs
- Operates using sets of “facts”



Overview of Dataflow Analysis

- A framework for exposing properties about programs
- Operates using sets of “facts”
- Just the initial discovery phase
  - Changes can then be made to optimize based on the analysis



Overview of Dataflow Analysis

- Basic Framework of Set Definitions (for a Basic Block *b*):
  - IN(*b*): facts true on entry to *b*
  - OUT(*b*): facts true on exit from *b*
  - GEN(*b*): facts created (and not killed) in *b*
  - KILL(*b*): facts killed in *b*

Reaching Definitions (*A Dataflow Problem*)

"What definitions of each variable might reach this point"

- Could be used for:
  - Constant Propagation
  - Uninitialized Variables

```
int x;

if (y > 0) {
  x = y;
} else {
  x = 0;
}

System.out.println(x);
```

"x=y", "x=0"

Reaching Definitions (*A Dataflow Problem*)

“What definitions of each variable might reach this point”

- **Be careful**: Does not involve the *value* of the definition
  - The dataflow problem “Available Expressions” is designed for that

```
int x;

if (y > 0) {
  x = y;
} else {
  x = 0;
}

y = -1;
System.out.println(x);
```

still: “x=y”, “x=0”

Equations for Reaching Definitions

- IN(b): the definitions reaching upon entering block b
- OUT(b): the definitions reaching upon exiting block b
- GEN(b): the definitions assigned and not killed in block b
- KILL(b): the definitions of variables overwritten in block b

$$\mathrm{IN}(b) = \bigcup_{p \in \mathrm{pred}(b)} \mathrm{OUT}(p)$$

$$\mathrm{OUT}(b) = \mathrm{GEN}(b) \cup \left(\mathrm{IN}(b) - \mathrm{KILL}(b)\right)$$

Another *Equivalent* Set of Equations (from Lecture):

- Sets:
  - DEFOUT(b): set of definitions in b that reach the end of b (i.e., not subsequently redefined in b)
  - SURVIVED(b): set of all definitions not obscured by a definition in b
  - REACHES(b): set of definitions that reach b
- Equations:

REACHES(b) =

$$\bigcup_{p \in preds(b)} \text{DEFOUT}(p) \cup \left(\text{REACHES}(p) \cap \text{SURVIVED}(p)\right)$$

Problems 1_a and 1_b

L0: a = 0
L1: b = a + 1
L2: c = c + b
L3: a = b * 2
L4: if a < N goto L1
L5: return c

Block	GEN	KILL	IN (1)	OUT (1)	IN (2)	OUT (2)
L0	L0					
L1	L1					
L2	L2					
L3	L3					
L4						
L5						

L0: a = 0
L1: b = a + 1
L2: c = c + b
L3: a = b * 2
L4: if a < N goto L1
L5: return c

Block	GEN	KILL	IN (1)	OUT (1)	IN (2)	OUT (2)
L0	L0					
L1	L1					
L2	L2					
L3	L3	L0				
L4						
L5						

L0: a = 0
L1: b = a + 1
L2: c = c + b
L3: a = b * 2
L4: if a < N goto L1
L5: return c

Block	GEN	KILL	IN (1)	OUT (1)	IN (2)	OUT (2)
L0	L0					
L1	L1		L0			
L2	L2		L0, L1			
L3	L3	L0	L0, L1, L2			
L4			L1, L2, L3			
L5			L1, L2, L3			

L0: a = 0
L1: b = a + 1
L2: c = c + b
L3: a = b * 2
L4: if a < N goto L1
L5: return c

Block	GEN	KILL	IN (1)	OUT (1)	IN (2)	OUT (2)
L0	L0			L0		
L1	L1		L0	L0, L1		
L2	L2		L0, L1	L0, L1, L2		
L3	L3	L0	L0, L1, L2	L1, L2, L3		
L4			L1, L2, L3	L1, L2, L3		
L5			L1, L2, L3	L1, L2, L3		

```
L0:  a = 0
L1:  b = a + 1
L2:  c = c + b
L3:  a = b * 2
L4:  if a < N goto L1
L5:  return c
```

Block	GEN	KILL	IN (1)	OUT (1)	IN (2)	OUT (2)
L0	L0			L0		L0
L1	L1		L0	L0, L1	L0, L1, L2, L3	L0, L1, L2, L3
L2	L2		L0, L1	L0, L1, L2	L0, L1, L2, L3	L0, L1, L2, L3
L3	L3	L0	L0, L1, L2	L1, L2, L3	L0, L1, L2, L3	L1, L2, L3
L4			L1, L2, L3	L1, L2, L3	L1, L2, L3	L1, L2, L3
L5			L1, L2, L3	L1, L2, L3	L1, L2, L3	L1, L2, L3

```
L0:  a = 0
L1:  b = a + 1
L2:  c = c + b
L3:  a = b * 2
L4:  if a < N goto L1
L5:  return c
```

Convergence!

Block	GEN	KILL	IN (1)	OUT (1)	IN (2)	OUT (2)
L0	L0			L0		L0
L1	L1		L0	L0, L1	L0, L1, L2, L3	L0, L1, L2, L3
L2	L2		L0, L1	L0, L1, L2	L0, L1, L2, L3	L0, L1, L2, L3
L3	L3	L0	L0, L1, L2	L1, L2, L3	L0, L1, L2, L3	L1, L2, L3
L4			L1, L2, L3	L1, L2, L3	L1, L2, L3	L1, L2, L3
L5			L1, L2, L3	L1, L2, L3	L1, L2, L3	L1, L2, L3

```
L0:  a = 0
L1:  b = a + 1
L2:  c = c + b
L3:  a = b * 2
L4:  if a < N goto L1
L5:  return c
```

Is it possible to replace the use of *a* in block L1 with the constant 0?

Block	GEN	KILL	IN (1)	OUT (1)	IN (2)	OUT (2)
L0	L0			L0		L0
L1	L1		L0	L0, L1	L0, L1, L2, L3	L0, L1, L2, L3
L2	L2		L0, L1	L0, L1, L2	L0, L1, L2, L3	L0, L1, L2, L3
L3	L3	L0	L0, L1, L2	L1, L2, L3	L0, L1, L2, L3	L1, L2, L3
L4			L1, L2, L3	L1, L2, L3	L1, L2, L3	L1, L2, L3
L5			L1, L2, L3	L1, L2, L3	L1, L2, L3	L1, L2, L3

```
L0:  a = 0
L1:  b = a + 1
L2:  c = c + b
L3:  a = b * 2
L4:  if a < N goto L1
L5:  return c
```

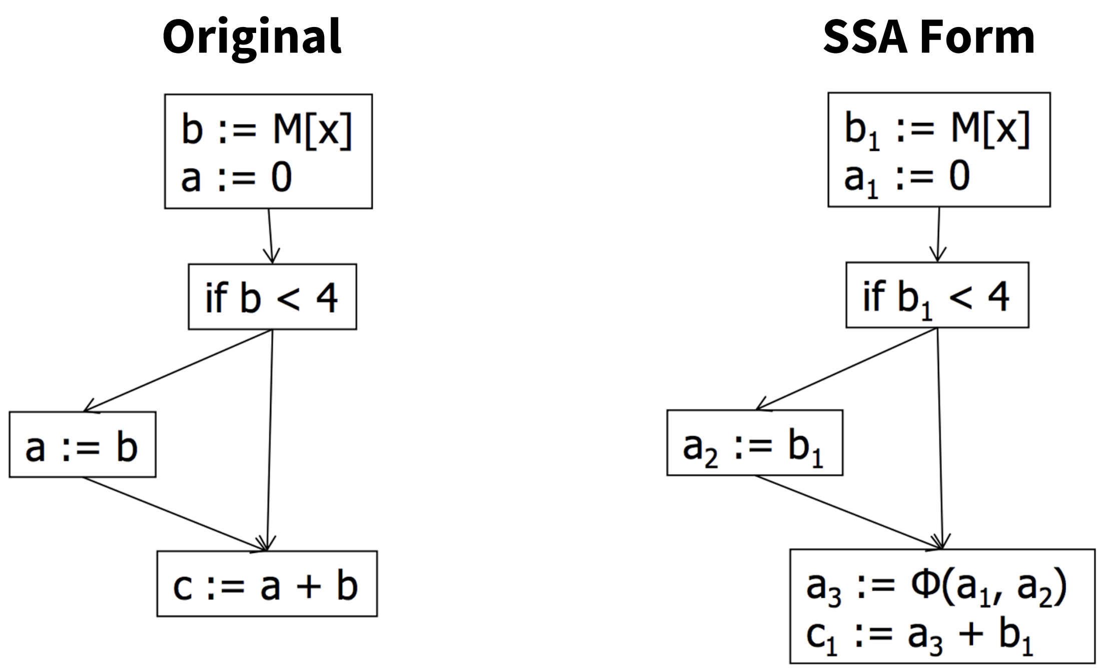
Is it possible to replace the use of *a* in block **L1** with the constant 0?

No. To determine this, we would look at the IN set for block L1 -- the fact that the IN set contains two definitions of ‘a’ (L0 and L3) means we cannot perform this constant propagation. In other words, more than one definition of ‘a’ is a reaching definition to block L1, and therefore performing constant propagation would only preserve one possible value of ‘a’ and the generated code would not be equivalent.

Block	GEN	KILL	IN (1)	OUT (1)	IN (2)	OUT (2)
L0	L0			L0		L0
L1	L1		L0	L0, L1	L0, L1, L2, L3	L0, L1, L2, L3
L2	L2		L0, L1	L0, L1, L2	L0, L1, L2, L3	L0, L1, L2, L3
L3	L3	L0	L0, L1, L2	L1, L2, L3	L0, L1, L2, L3	L1, L2, L3
L4			L1, L2, L3	L1, L2, L3	L1, L2, L3	L1, L2, L3
L5			L1, L2, L3	L1, L2, L3	L1, L2, L3	L1, L2, L3

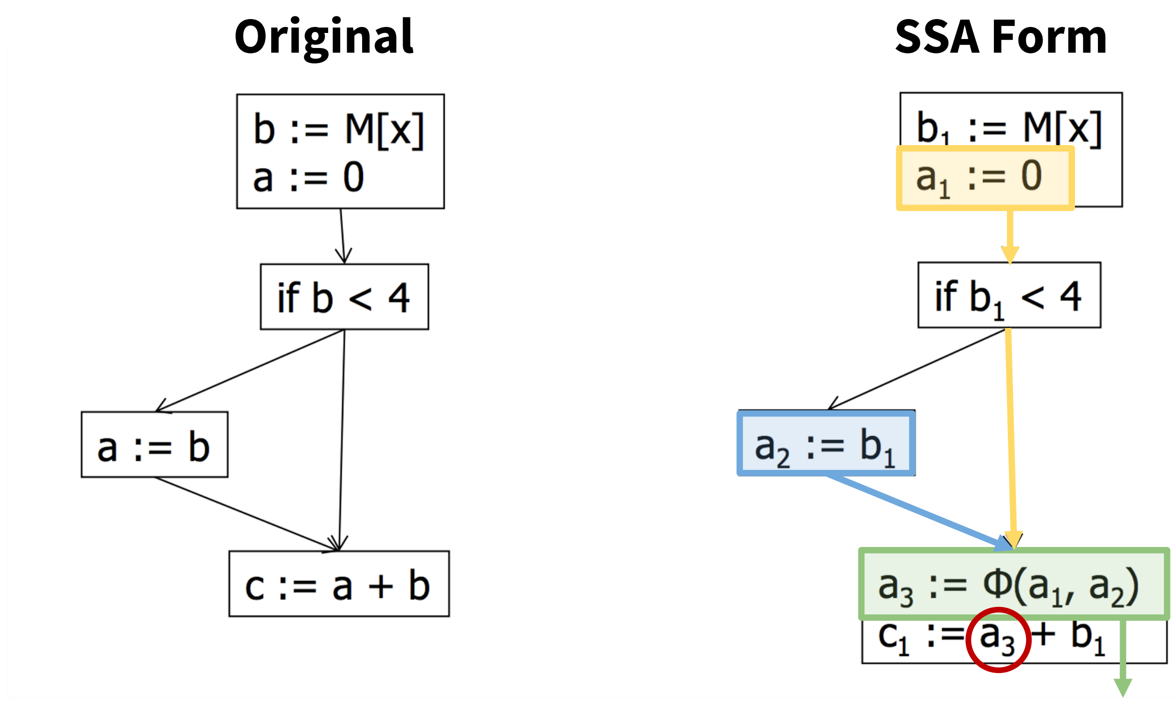
Phi-Functions

- A way to represent <u>multiple possible values</u> for a certain definition
  - Not a "real" instruction – just a form of bookkeeping needed for SSA



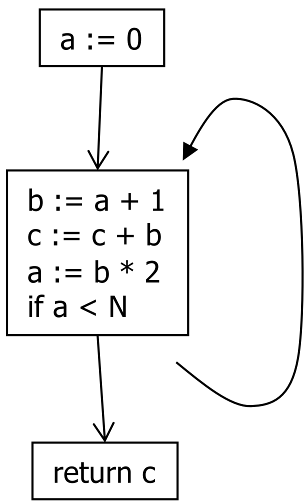
Where to place Phi-Functions?

- Wherever a variable has multiple possible definitions entering a block
  - Inefficient (and unnecessary!) to consider all possible phi-functions at the start of each block

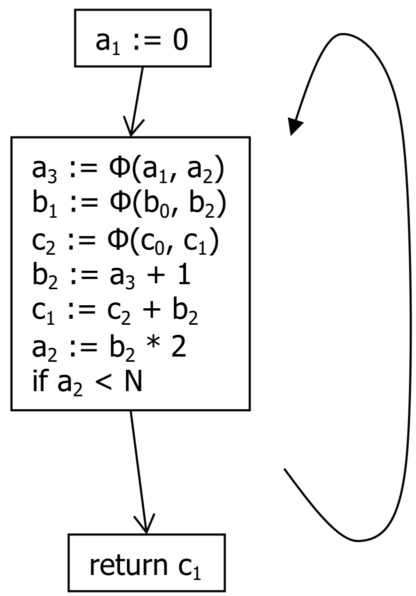


Example With a Loop

Original



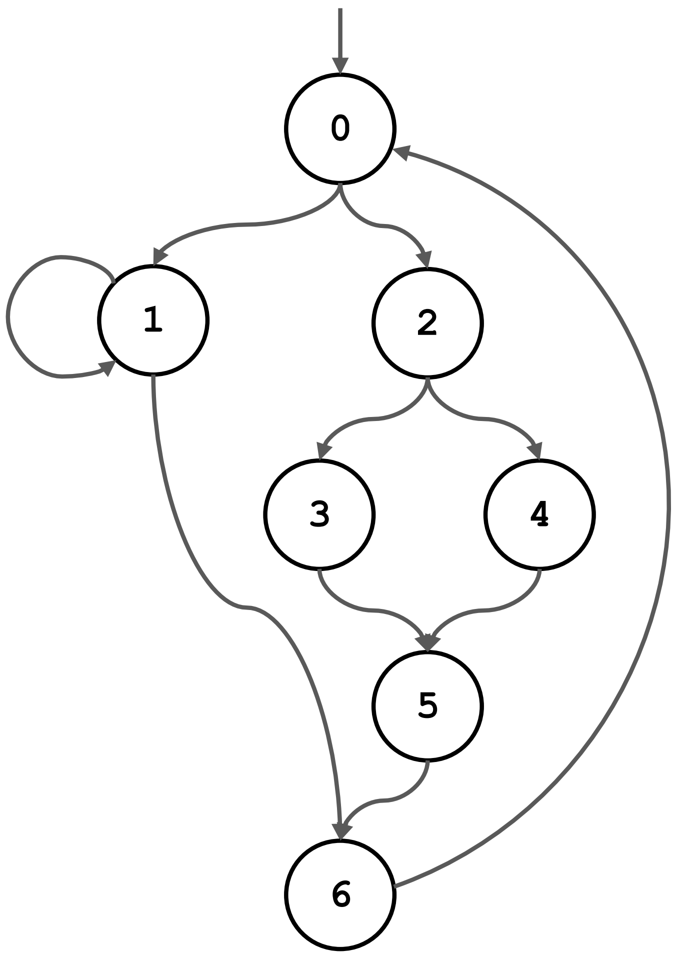
SSA



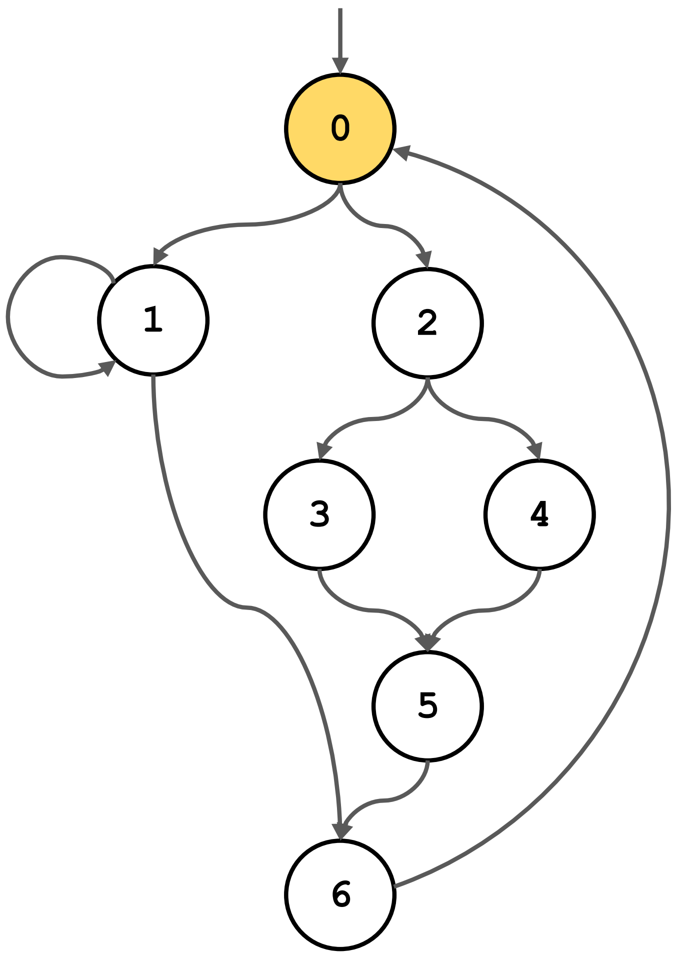
Notes:

- Loop-back edges are also merge points, so require Φ-functions
- a_0, b_0, c_0 are initial values of a, b, c on entry to initial block
- b_1 is dead – can delete later
- c is live on entry – either input parameter or uninitialized

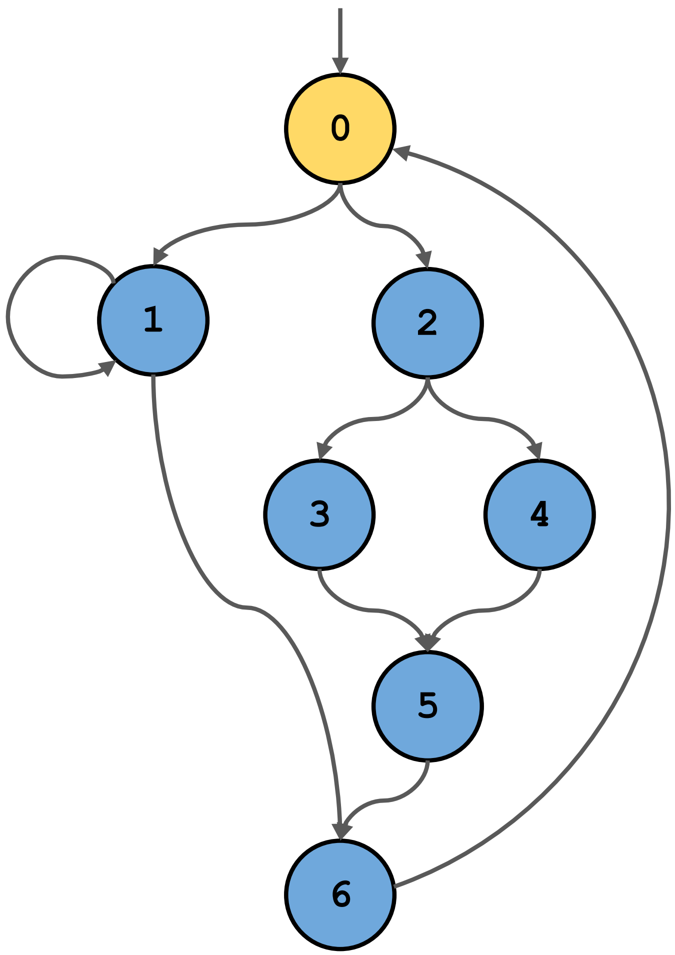
Problem 2(a)



NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0		
1		
2		
3		
4		
5		
6		



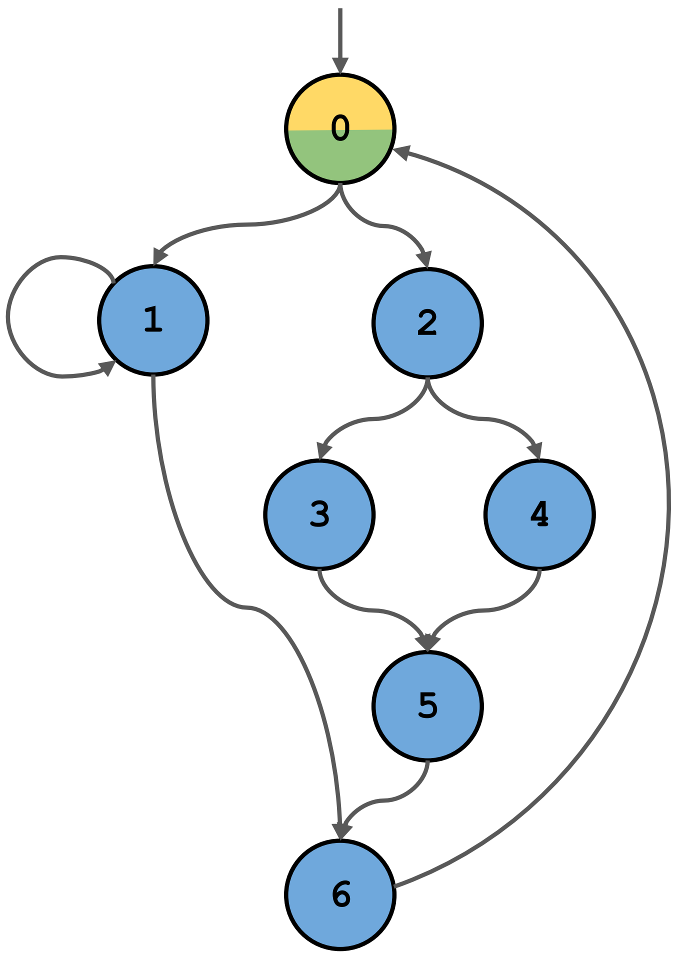
NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0		
1		
2		
3		
4		
5		
6		



NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	
1		
2		
3		
4		
5		
6		

A node **X** ***dominates*** a node **Y** iff every path from the entry point of the control flow graph to **Y** includes **X**.
A node **X** ***strictly dominates*** a node **Y** iff **X** dominates **Y** and $X \neq Y$

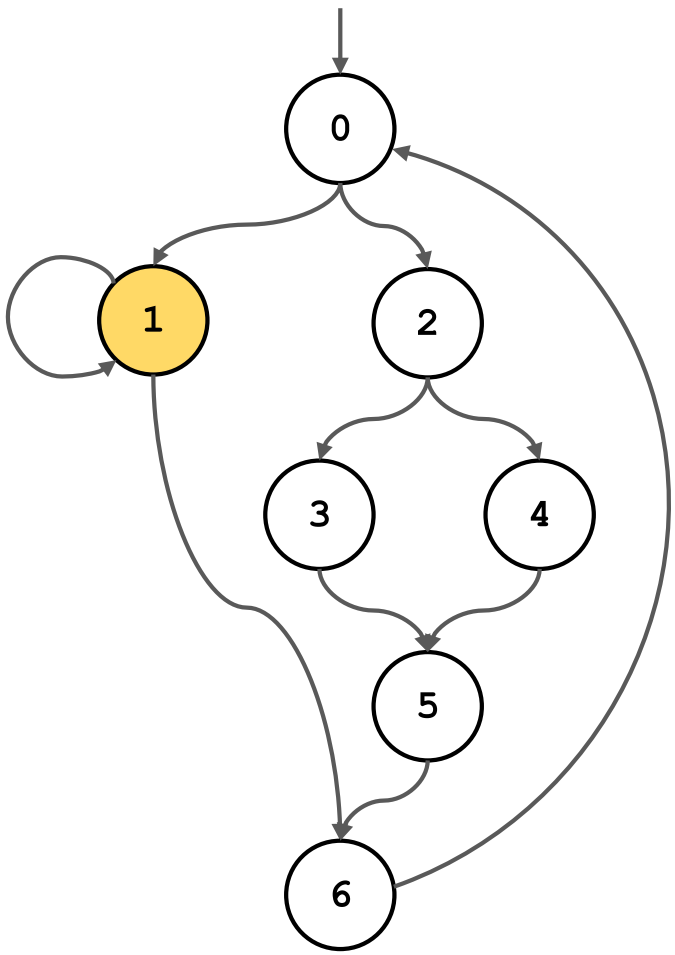
Need to go through 0 to get through 1, 2, 3, 4, 5, 6 and 0 cannot strictly dominate itself



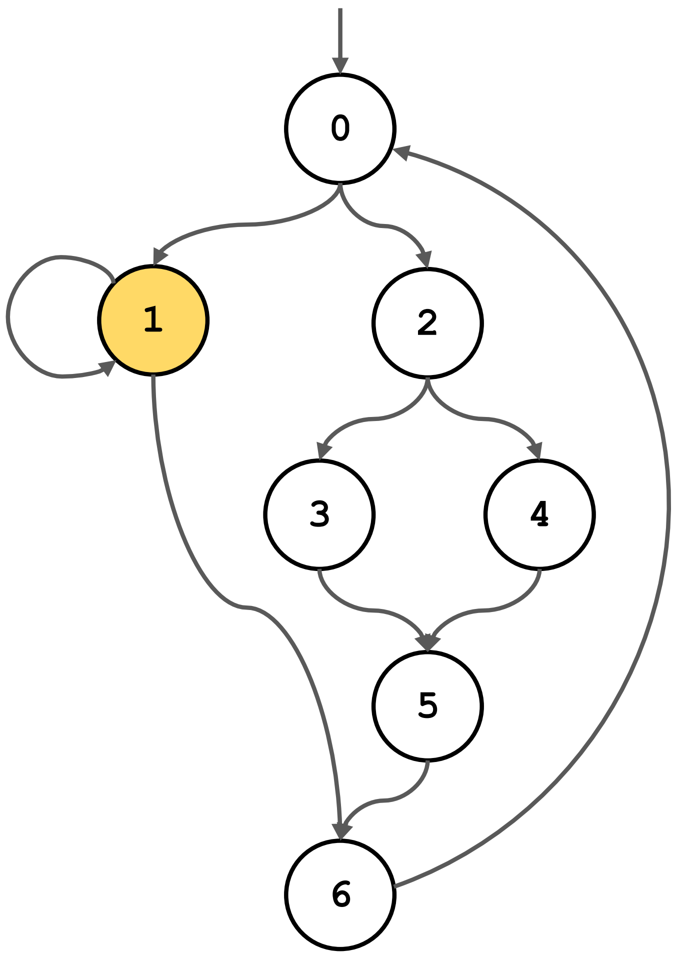
NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1		
2		
3		
4		
5		
6		

A node **Y** is in the ***dominance frontier*** of node **X** iff **X** dominates an immediate predecessor of **Y** but **X** does not strictly dominate **Y** .
A node **0** is in the ***dominance frontier*** of node **0** iff **0** dominates an immediate predecessor **(6)** of **0** but **0** does not strictly dominate 0

0 dominates 6, 6 is an immediate predecessor of 0, 0 does not strictly dominate 0



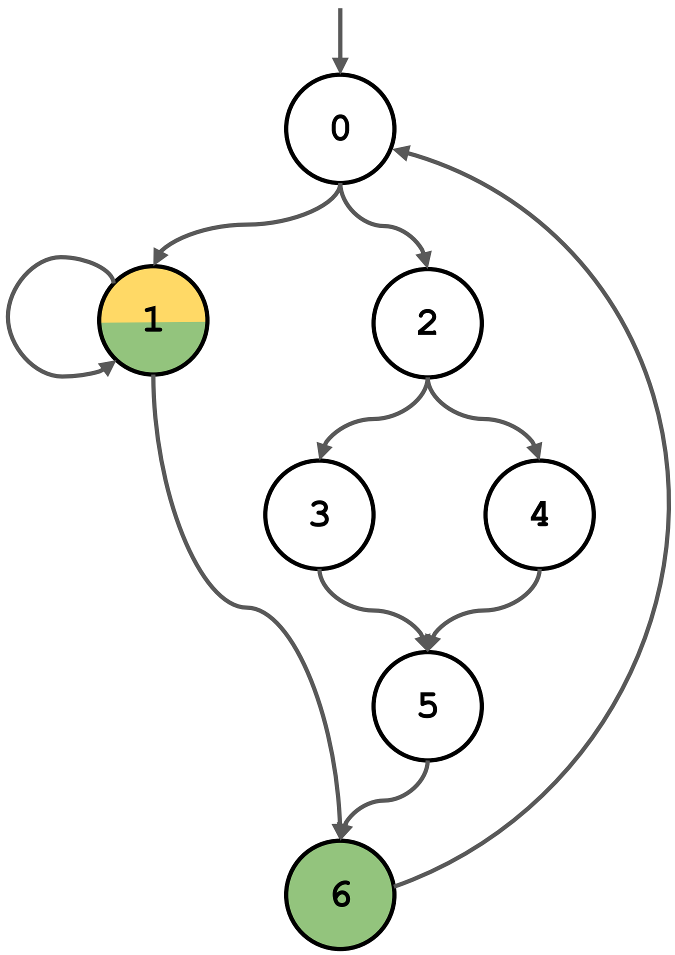
NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1		
2		
3		
4		
5		
6		



NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1	∅	
2		
3		
4		
5		
6		

A node **X** ***dominates*** a node **Y** iff every path from the entry point of the control flow graph to **Y** includes **X** .
A node **X** ***strictly dominates*** a node **Y** iff **X** dominates **Y** and **X** ≠ **Y**

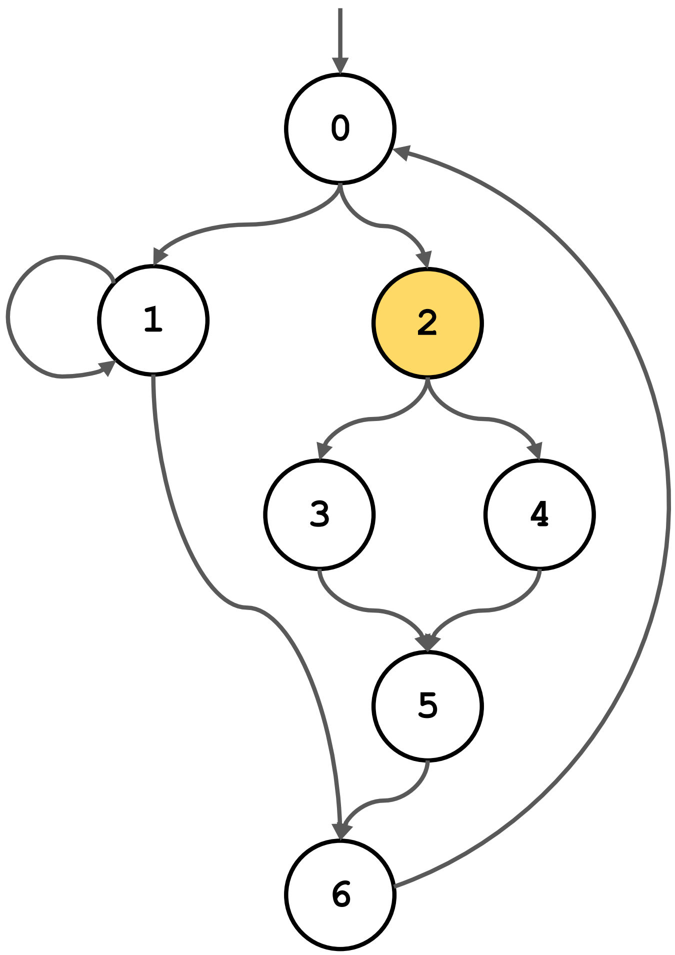
1 does not dominate 6 because there is a path from 5 that doesn't include 1. 1 does not strictly dominate itself



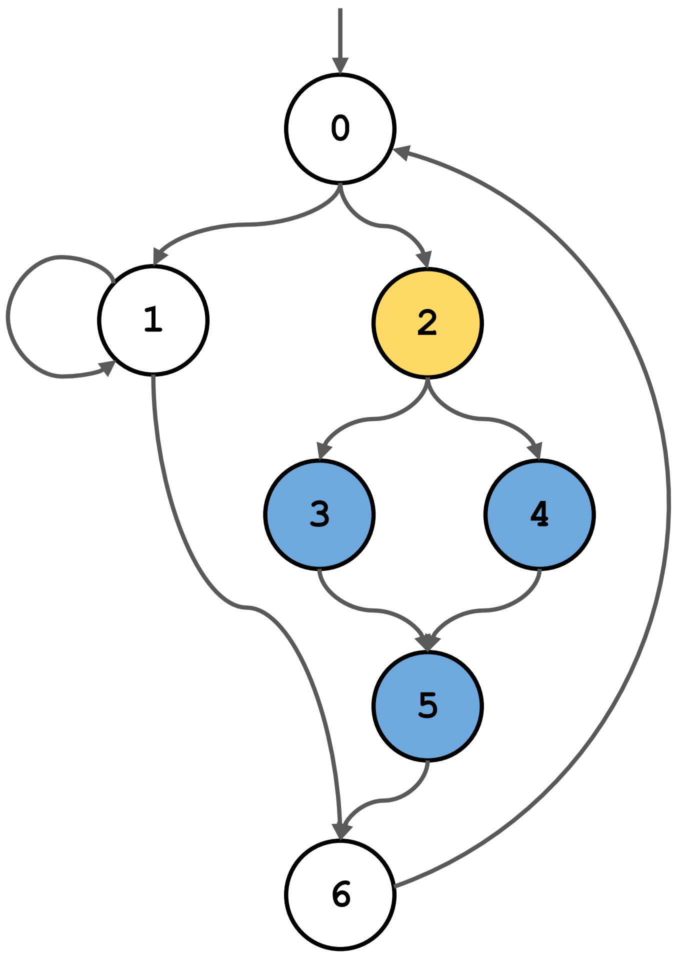
NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1	∅	1, 6
2		
3		
4		
5		
6		

A node **Y** is in the ***dominance frontier*** of node **X** iff **X** dominates an immediate predecessor of **Y** but **X** does not strictly dominate **Y** .

X = 1, Y = 6, 1 dominates 1, 1 is an immediate predecessor of 6, 1 does not strictly dominate 6
X = 1, Y = 1, 1 dominates 1, 1 is an immediate predecessor of 1, 1 does not strictly dominate 1



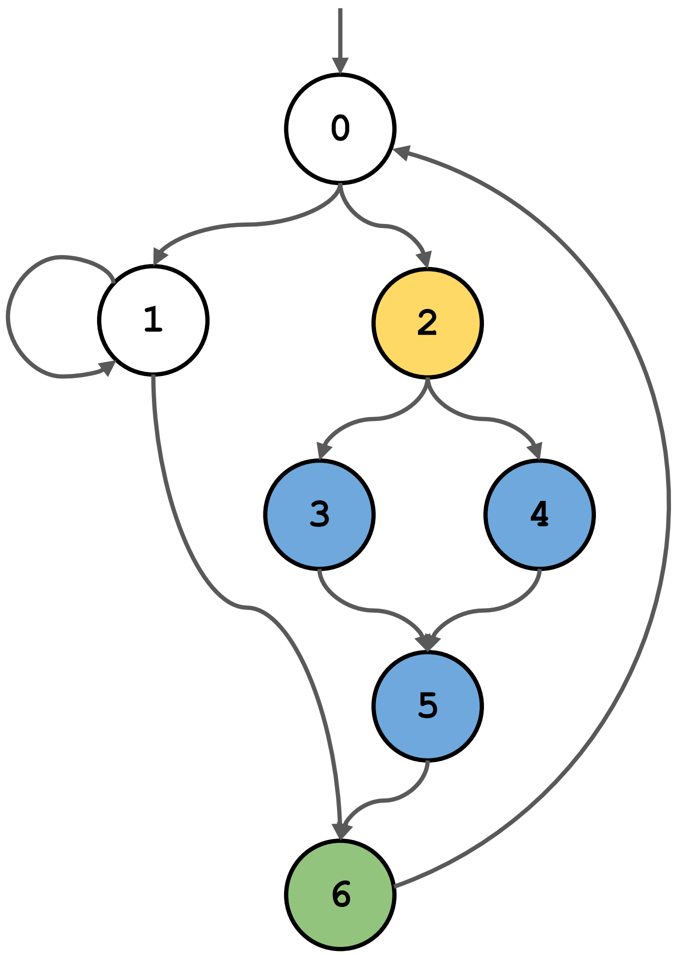
NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1	∅	1, 6
2		
3		
4		
5		
6		



NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1	ø	1, 6
2	3, 4, 5	
3		
4		
5		
6		

A node **X** ***dominates*** a node **Y** iff every path from the entry point of the control flow graph to **Y** includes **X** .
A node **X** ***strictly dominates*** a node **Y** iff **X** dominates **Y** and **X** ≠ **Y**

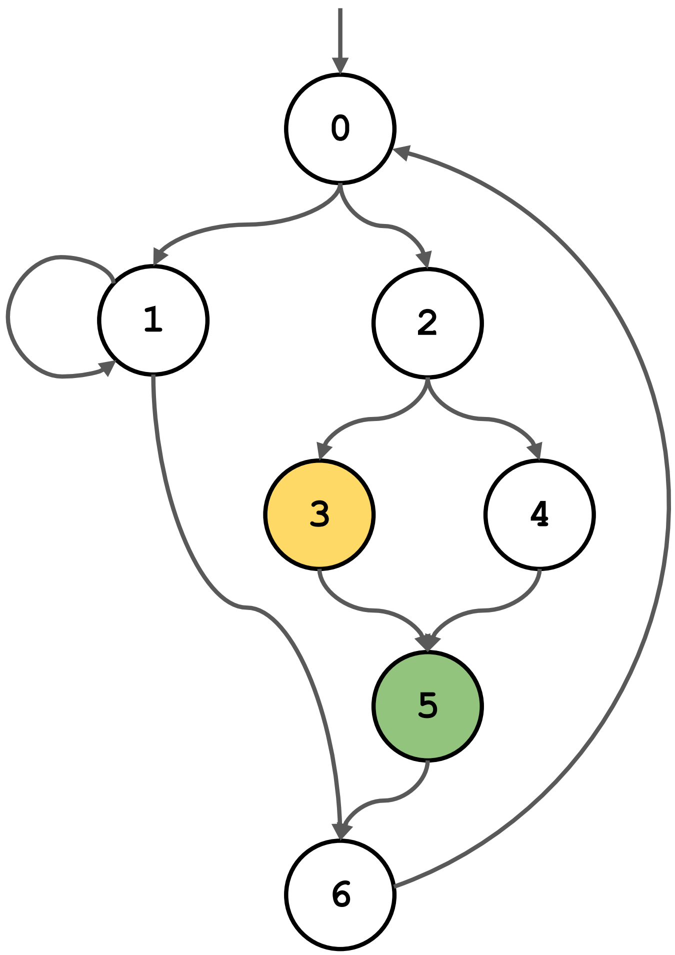
Need to go through 2 to get through 3, 4, 5 and 2 cannot strictly dominate itself



NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1	∅	1, 6
2	3, 4, 5	6
3		
4		
5		
6		

A node **Y** is in the ***dominance frontier*** of node **X** iff **X** dominates an immediate predecessor of **Y** but **X** does not strictly dominate **Y** .

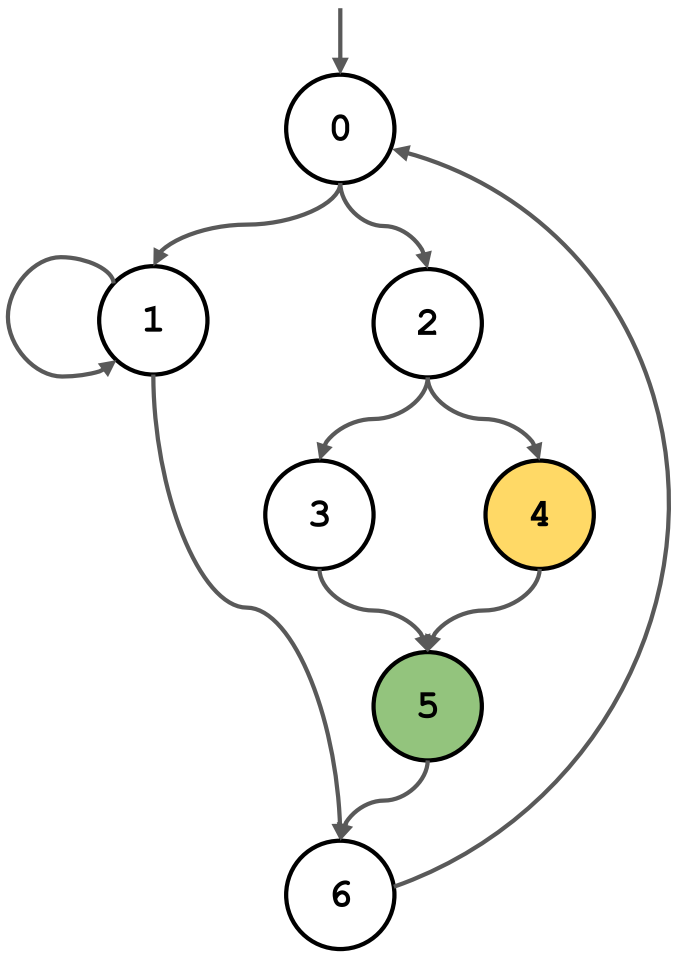
X = 2, Y = 6, 2 dominates 5, 5 is an immediate predecessor of 6, 2 does not strictly dominate 6



NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1	∅	1, 6
2	3, 4, 5	6
3	∅	5
4		
5		
6		

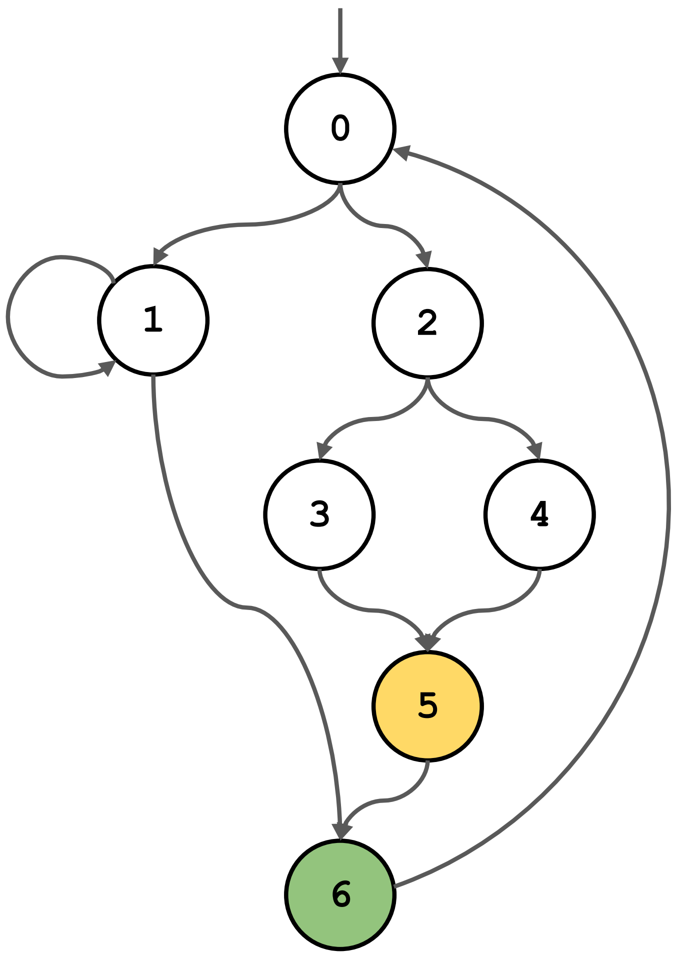
3 does not strictly dominate 5 (path through 4) and therefore does not strictly dominate anything else

3 dominates 3, 3 is an immediate predecessor of 5, 3 does not strictly dominate 5



NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1	∅	1, 6
2	3, 4, 5	6
3	∅	5
4	∅	5
5		
6		

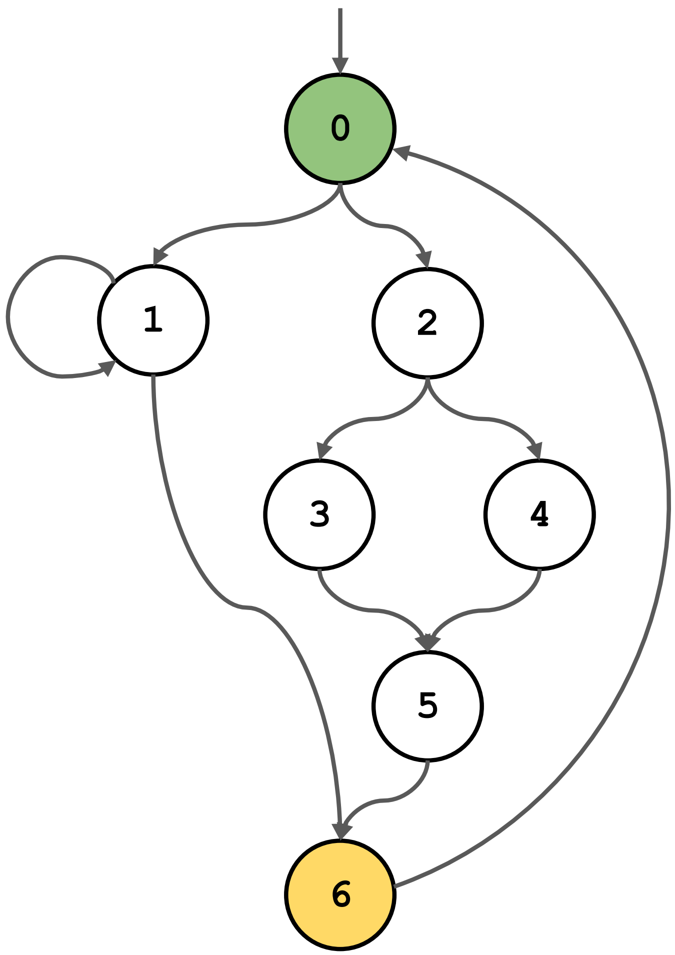
Same as previous slide but with 4 instead of 3



NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1	∅	1, 6
2	3, 4, 5	6
3	∅	5
4	∅	5
5	∅	6
6		

5 does not strictly dominate 6 (path through 1) and therefore does not strictly dominate anything else

5 dominates 5, 5 is an immediate predecessor of 6, 5 does not strictly dominate 6



NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1	∅	1, 6
2	3, 4, 5	6
3	∅	5
4	∅	5
5	∅	6
6	∅	0

6 does not strictly dominate 0 (path through 0) and therefore does not strictly dominate anything else

6 dominates 6, 6 is an immediate predecessor of 0, 6 does not strictly dominate 0

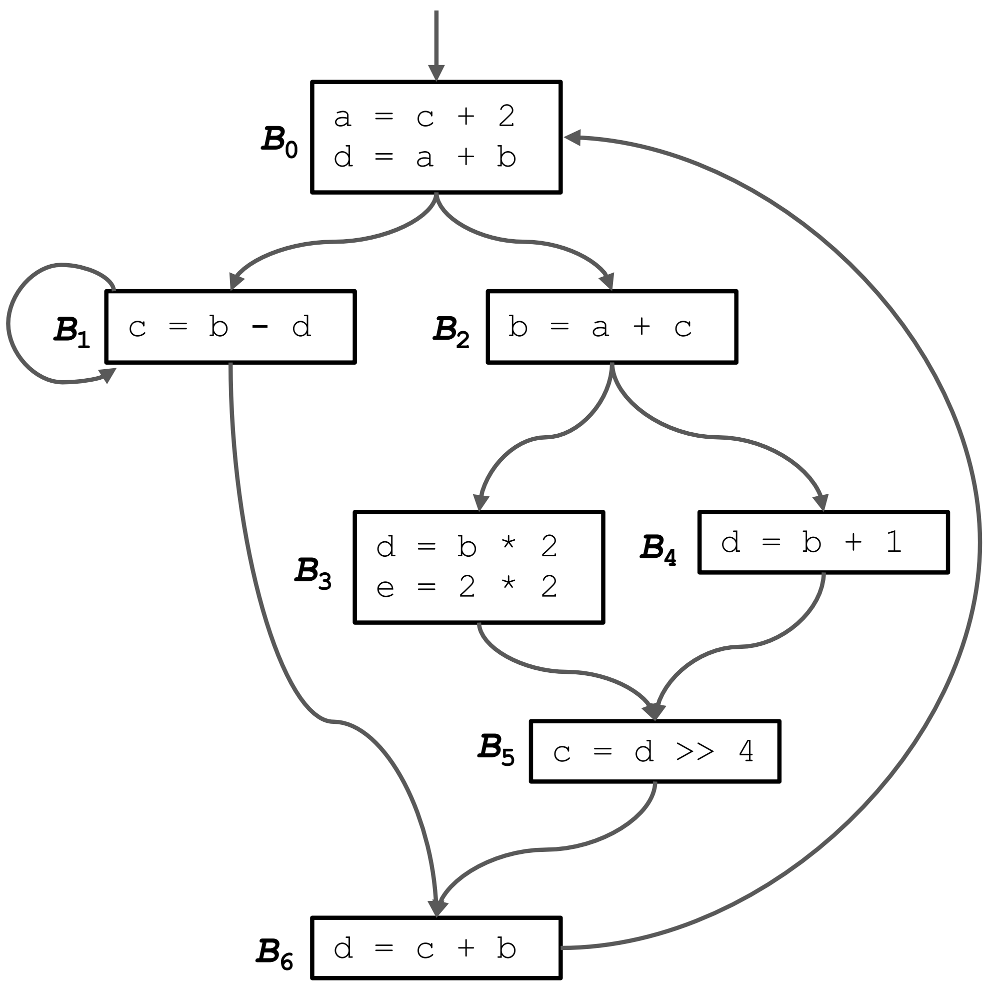
Problem 2(b)

Converting to SSA

1. **Compute the dominance frontier of each node** ⇐ Already done (in problem 2a)

2. Determine which variables need merging in each node

3. Assign numbers to definitions and add phi functions



Step 1: Dominance Frontiers

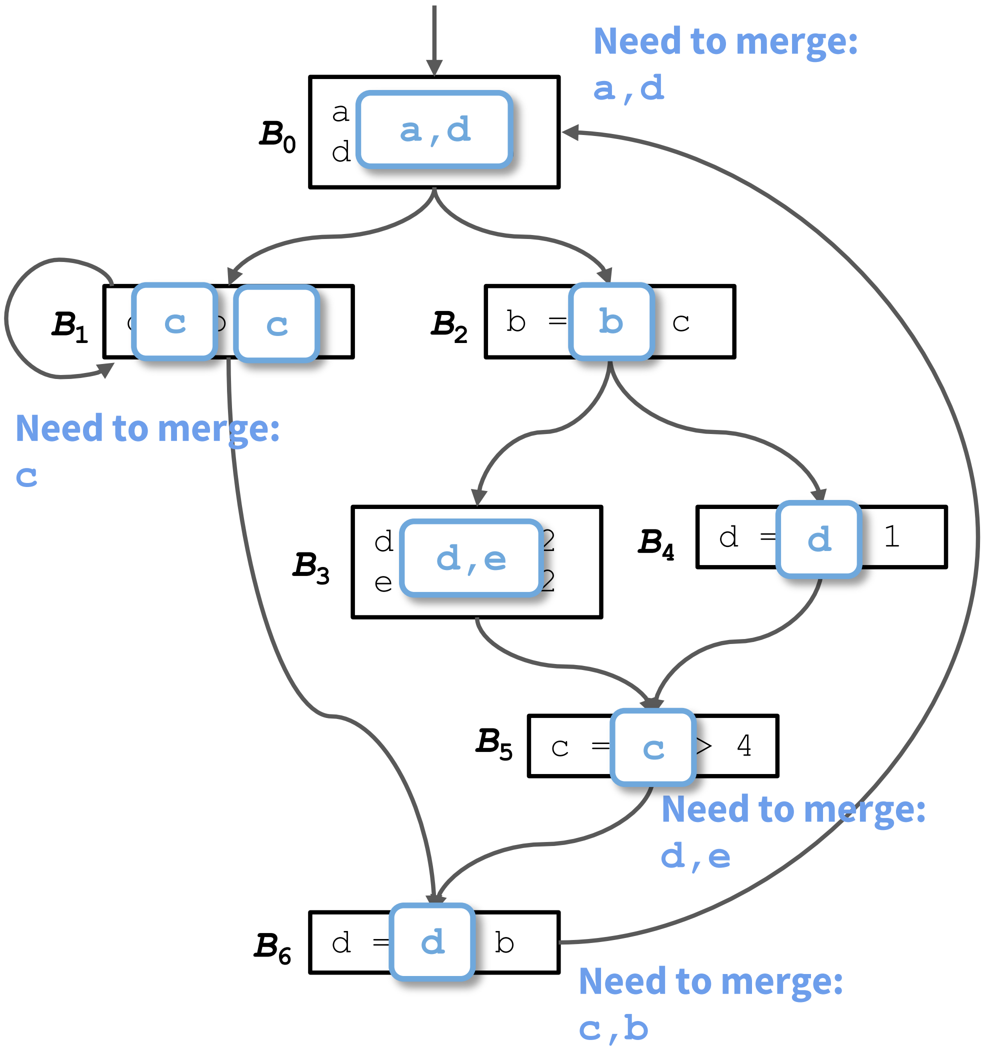
NODE	STRICTLY DOMINATES	DOMINANCE FRONTIER
0	1, 2, 3, 4, 5, 6	0
1	∅	1, 6
2	3, 4, 5	6
3	∅	5
4	∅	5
5	∅	6
6	∅	0

Converting to SSA

1. Compute the dominance frontier of each node ☑

2. **Determine which variables need merging in each node** ⇐ **We will compute using the dominance frontiers**

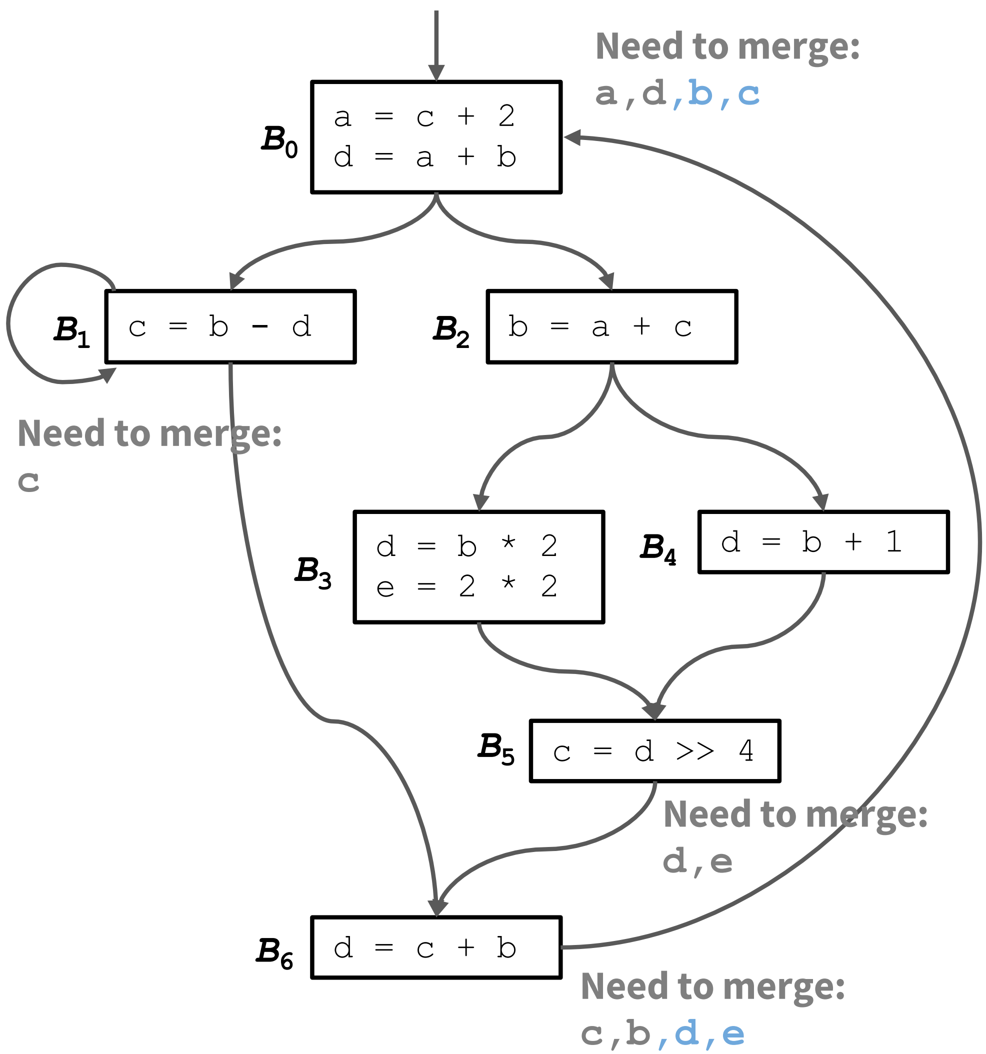
3. Assign numbers to definitions and add phi functions



Step 2: Determine Necessary Merges

ITERATION 1: Each node in the dominance frontier of node X will merge any definitions created in node X.

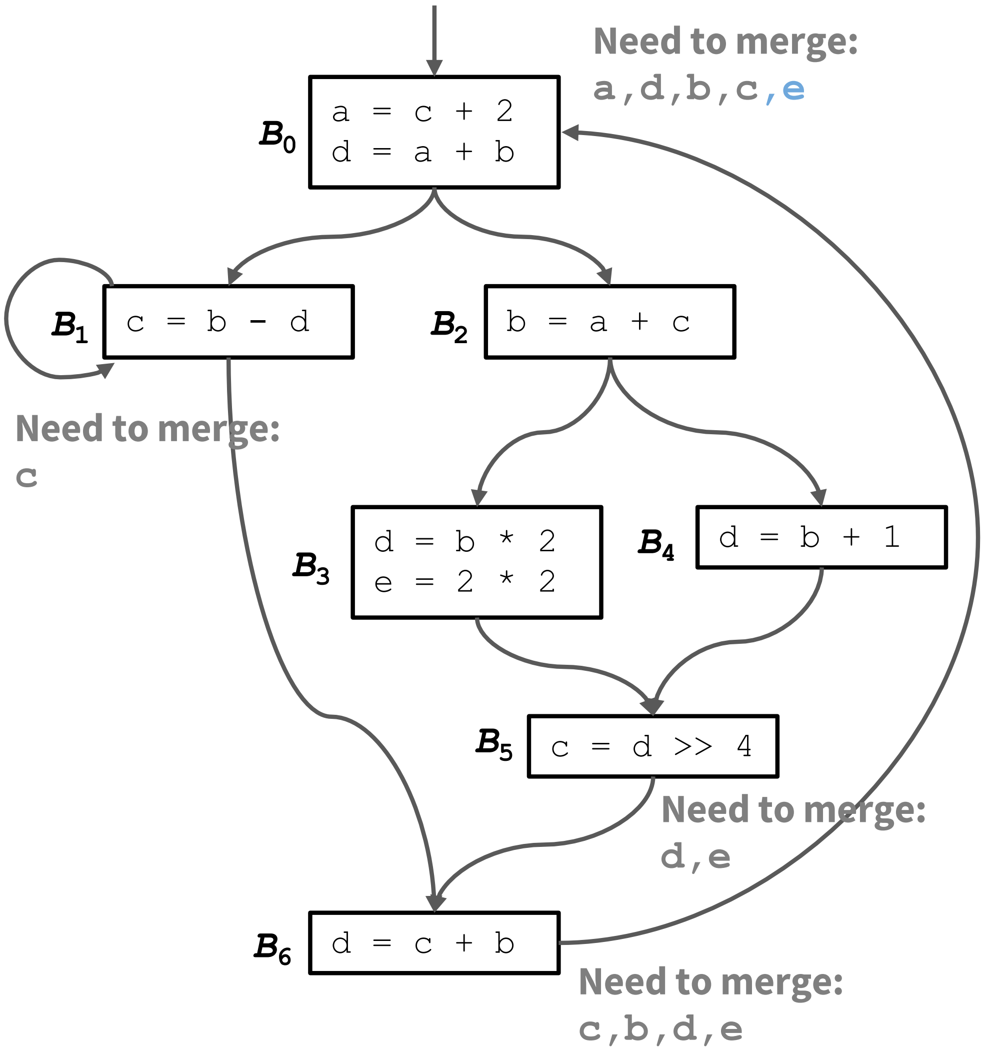
NODE		DOMINANCE FRONTIER
0	a,d	0
1	c	1, 6
2	b	6
3	d,e	5
4	d	5
5	c	6
6	d	0



Step 2: Determine Necessary Merges

ITERATION 2: Each merge will create a new definition, which may need merging again.

NODE		DOMINANCE FRONTIER
0		0
1		1, 6
2		6
3		5
4		5
5	d,e	6
6	b,c	0



Step 2: Determine Necessary Merges

ITERATION 3: Each merge will create a new definition, which may need merging again.

NODE	DOMINANCE FRONTIER
0	0
1	1, 6
2	6
3	5
4	5
5	6
6	0

d,e

Converting to SSA

1. Compute the dominance frontier of each node ✓

2. Determine which variables need merging in each node ✓

3. **Assign numbers to definitions and add phi functions** ← **Place phi functions first, then increment subscripts**

Step 3: Assign Definition Numbers

Merges go first, and each successive definition of a variable should increment its index by 1.

B_0

```
a = c + 2
d = a + b
```

Need to merge:
`a,b,c,d,e`

⇒

B_0

$$
\begin{aligned}
a_1 &= \Phi(a_0,\ a_2)\\
b_1 &= \Phi(b_0,\ b_3)\\
c_1 &= \Phi(c_0,\ c_5)\\
d_1 &= \Phi(d_0,\ d_7)\\
e_1 &= \Phi(e_0,\ e_4)\\
a_2 &= c_1 + 2\\
d_2 &= a_2 + b_1
\end{aligned}
$$

Note: these subscripts determined after doing the rest of the CFG!

Step 3: Assign Definition Numbers

Merges go first, and each successive definition of a variable should increment its index by 1.

B_1

```
c = b - d
```

Need to merge:
`c`

⇒

B_1

$c_2 = \Phi(c_1, c_3)$
$c_3 = b_1 - d_2$

Note: must merge its own (later) definition because of the back-edge!

Step 3: Assign Definition Numbers

Merges go first, and each successive definition of a variable should increment its index by 1.

B_2 `b = a + c` ⇒ B_2 $b_2 = a_2 + c_1$

Nothing to merge

Step 3: Assign Definition Numbers

Merges go first, and each successive definition of a variable should increment its index by 1.

B_3

```
d = b * 2
e = 2 * 2
```

Nothing to merge

⇒

B_3

$d_3 = b_2 * 2$

$e_2 = 2 * 2$

Step 3: Assign Definition Numbers

Merges go first, and each successive definition of a variable should increment its index by 1.

B_4

```
d = b + 1
```

Nothing to merge

⇨

B_4

$d_4 = b_2 + 1$

Step 3: Assign Definition Numbers

Merges go first, and each successive definition of a variable should increment its index by 1.

***B*₅**

```
c = d >> 4
```

Need to merge:
`d, e`

⇒

***B*₅**

$$d_5 = \Phi(d_3,\ d_4)$$
$$e_3 = \Phi(e_1,\ e_2)$$
$$c_4 = d_5 \gg 4$$

Step 3: Assign Definition Numbers

Merges go first, and each successive definition of a variable should increment its index by 1.

B_6

```
d = c + b
```

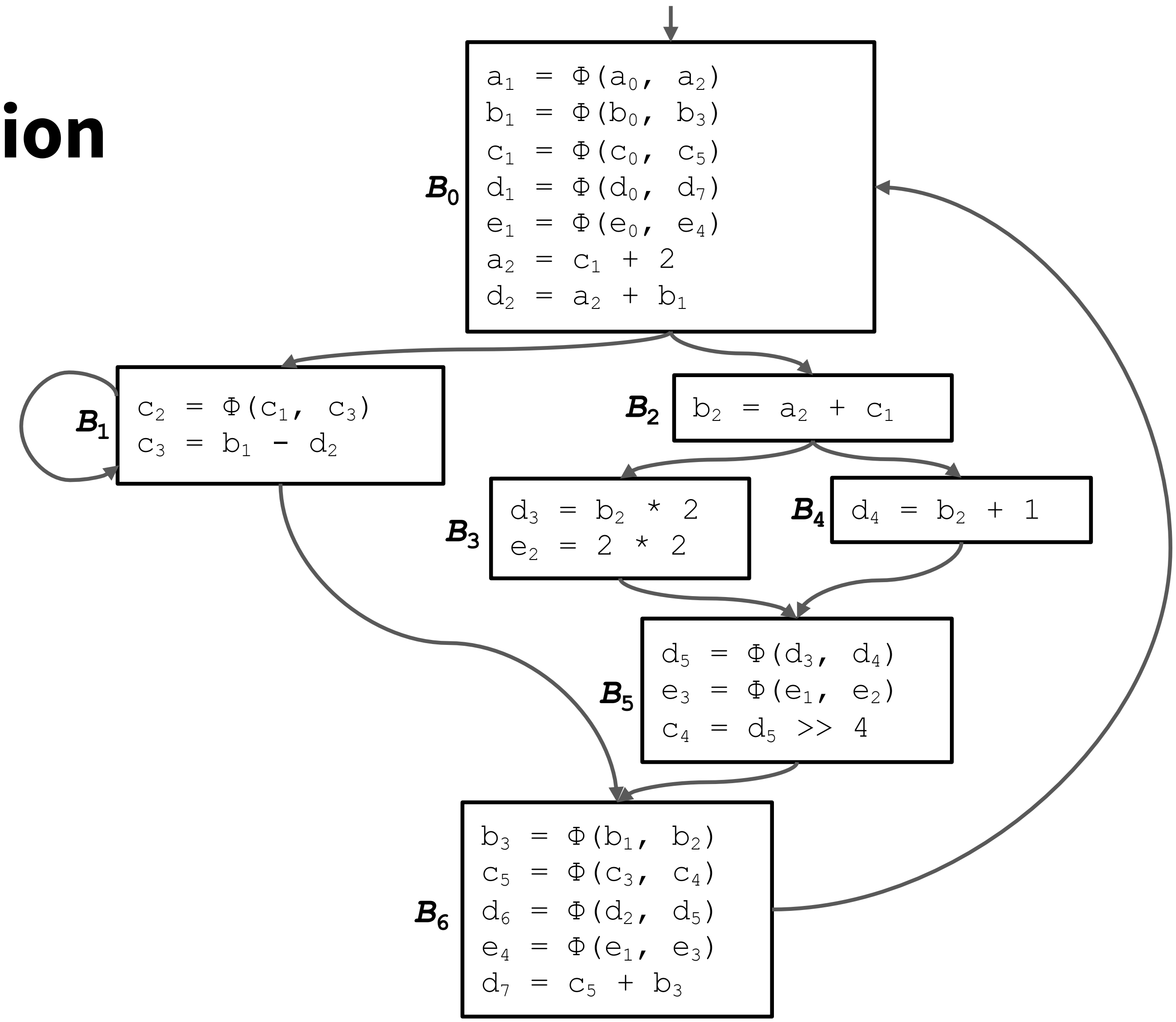
Need to merge:
`b,c,d,e`

⇒

B_6

$$
\begin{aligned}
b_3 &= \Phi(b_1,\ b_2) \\
c_5 &= \Phi(c_3,\ c_4) \\
d_6 &= \Phi(d_2,\ d_5) \\
e_4 &= \Phi(e_1,\ e_3) \\
d_7 &= c_5 + b_3
\end{aligned}
$$

Solution



B0
a1 = Φ(a0, a2)
b1 = Φ(b0, b3)
c1 = Φ(c0, c5)
d1 = Φ(d0, d7)
e1 = Φ(e0, e4)
a2 = c1 + 2
d2 = a2 + b1
B1
c2 = Φ(c1, c3)
c3 = b1 - d2
B2
b2 = a2 + c1
B3
d3 = b2 * 2
e2 = 2 * 2
B4
d4 = b2 + 1
B5
d5 = Φ(d3, d4)
e3 = Φ(e1, e2)
c4 = d5 >> 4
B6
b3 = Φ(b1, b2)
c5 = Φ(c3, c4)
d6 = Φ(d2, d5)
e4 = Φ(e1, e3)
d7 = c5 + b3