

# CSE211: Compiler Design

Nov. 3, 2023

- **Topic:** Converting into SSA

- **Questions:**

- *What is SSA form?*
- *What is the phi instruction?*

```
0
7 3:                                     ; preds = %1
8  %4 = tail call i32 @_Z14first_functionv(), !dbg !19
9  call void @llvm.dbg.value(metadata i32 %4, metadata !14, metadata
10 br label %7, !dbg !21
11
12 5:                                     ; preds = %1
13  %6 = tail call i32 @_Z15second_functionv(), !dbg !22
14  call void @llvm.dbg.value(metadata i32 %6, metadata !14, metadata
15 br label %7
16
17 7:                                     ; preds = %5, %3
18  %8 = phi i32 [ %4, %3 ], [ %6, %5 ], !dbg !24
19  call void @llvm.dbg.value(metadata i32 %8, metadata !14, metadata
20 ret i32 %8, !dbg !25
21 }
```

# Announcements

- Homework 2 is out
  - Due Nov. 13
  - Work on the assignment!
    - Performance differences?
- Homework 3 will be released on the 13<sup>th</sup>
- 2 more lectures in module 2

# Announcements

- We are working on grading your assignments ASAP.  
Stay tuned!
- Start thinking about next paper review
- Start thinking about final project if you are interested!

# Review SSA

# Intermediate representations

- What have we seen so far?
  - 3 address code
  - AST
  - data-dependency graphs
  - control flow graphs
- At a high-level:
  - 3 address code is good for **data-flow** reasoning
  - control flow graphs are good for... **control flow** reasoning

*What we want: an IR that can reasonably capture both control and data flow*

# Static Single-Assignment Form (SSA)

- Every variable is defined and written to *once*
  - We have seen this in local value numbering!
- Control flow is captured using  $\phi$  instructions

# $\phi$ instructions

- Example: how to convert this code into SSA?

```
int x;
```

```
if (<some_condition>) {  
    x = 5;  
}
```

```
else {  
    x = 7;  
}
```

```
print(x)
```

# $\phi$ instructions

- Example: how to convert this code into SSA?

```
int x;
```

```
if (<some_condition>) {  
    x = 5;  
}
```

Start with numbering

```
else {  
    x = 7;  
}
```

```
print(x)
```



# $\phi$ instructions

- Example: how to convert this code into SSA?

```
int x;
```

```
if (<some_condition>) {  
    x0 = 5;  
}
```

Start with numbering

```
else {  
    x1 = 7;  
}
```

```
print(x)
```

# $\phi$ instructions

- Example: how to convert this code into SSA?

```
int x;
```

```
if (<some_condition>) {  
    x0 = 5;  
}
```

Start with numbering

```
else {  
    x1 = 7;  
}
```

```
print(x)
```

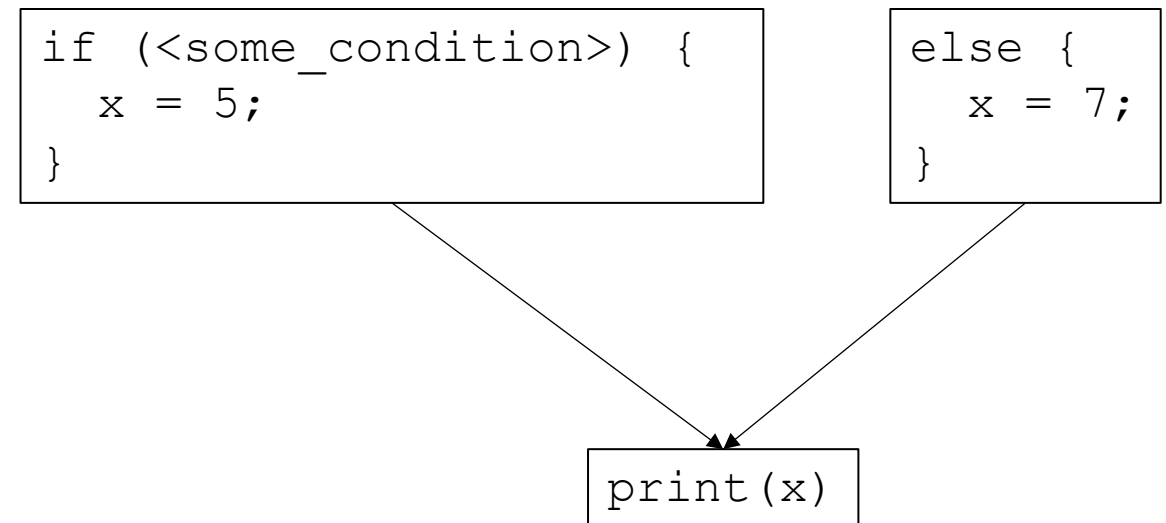
What here?

# $\phi$ instructions

- Example: how to convert this code into SSA?

```
int x;  
  
if (<some_condition>) {  
    x = 5;  
}  
  
else {  
    x = 7;  
}  
  
print(x)
```

let's make a CFG

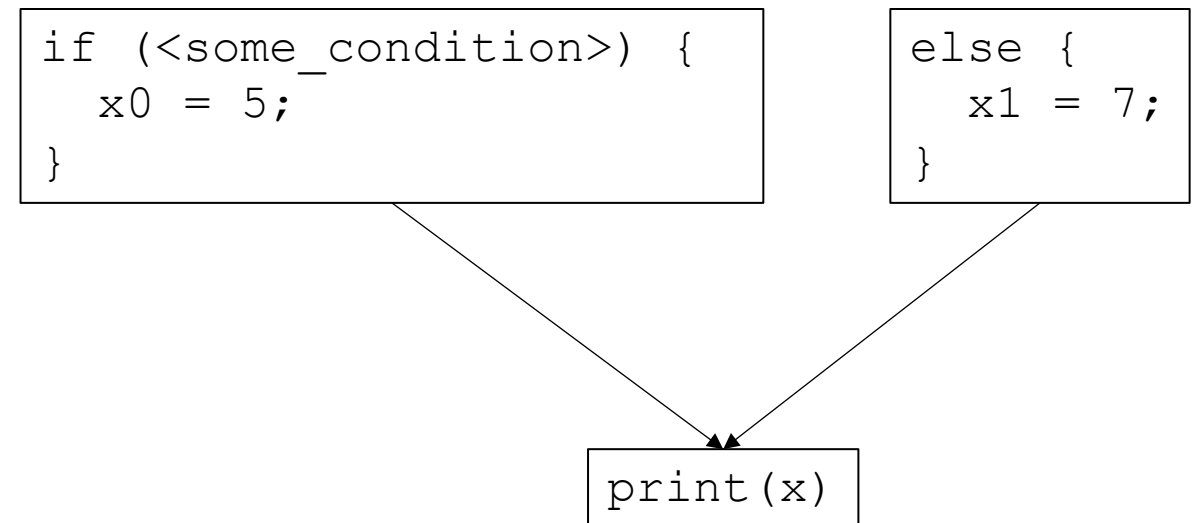


# $\phi$ instructions

- Example: how to convert this code into SSA?

```
int x;  
  
if (<some_condition>) {  
    x0 = 5;  
}  
  
else {  
    x1 = 7;  
}  
  
print(x)
```

number the variables

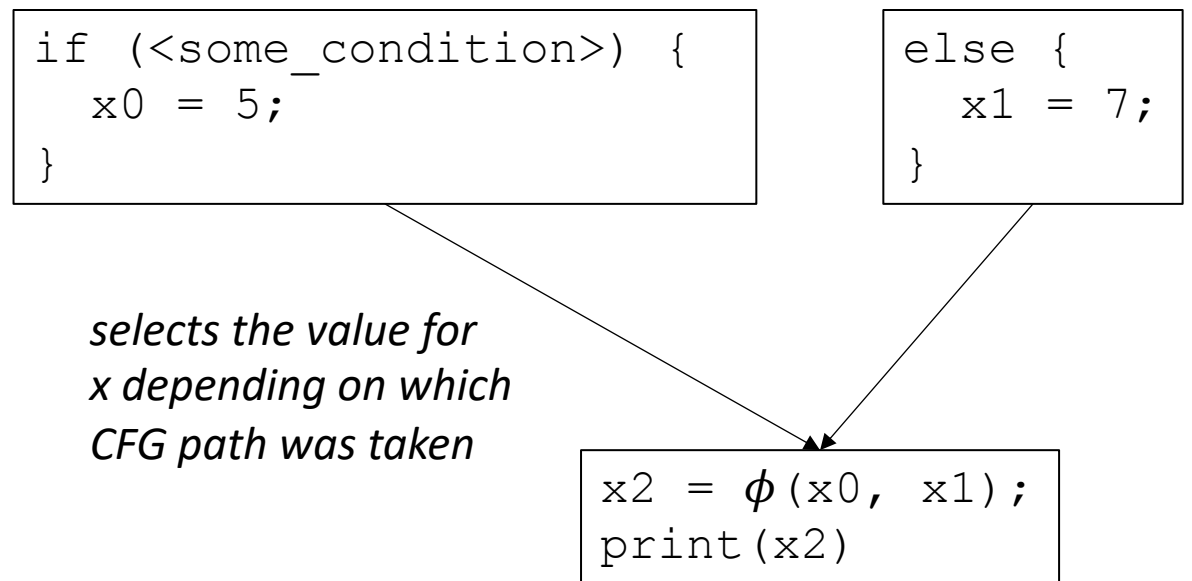


# $\phi$ instructions

- Example: how to convert this code into SSA?

```
int x;  
  
if (<some_condition>) {  
    x0 = 5;  
}  
  
else {  
    x1 = 7;  
}  
  
x2 =  $\phi$ (x0, x1);  
print(x2)
```

number the variables



# $\phi$ instructions

- $x_n = \phi(x_0, x_1, x_2, x_3 \dots);$
- selects one of the values depending on the previously executed basic block. Implementations will define how the value is selected:
  - LLVM: couples values with labels
  - EAC book: uses left-to-right ordering of parents in visual CFG

New material

# Conversion into SSA

Different algorithms depending on how many  $\phi$  instructions

The fewer  $\phi$  instructions, the more efficient analysis will be

Two phases:

- inserting  $\phi$  instructions

- variable naming



# Maximal SSA

*Straightforward:*

- For each variable, for each basic block: insert a  $\phi$  instruction with placeholders for arguments
- local numbering for each variable using a global counter
- instantiate  $\phi$  arguments

# Maximal SSA

## Example

```
x = 1;  
y = 2;  
  
if (<condition>) {  
    x = y;  
}  
  
else {  
    x = 6;  
    y = 100;  
}  
  
print(x)
```

# Maximal SSA

## Example

```
x = 1;
y = 2;

if (<condition>) {
    x = y;
}

else {
    x = 6;
    y = 100;
}

print(x)
```

Insert  $\phi$  with argument placeholders

```
x = 1;
y = 2;

if (<condition>) {
    x =  $\phi(\dots)$ ;
    y =  $\phi(\dots)$ ;
    x = y;
}

else {
    x =  $\phi(\dots)$ ;
    y =  $\phi(\dots)$ ;
    x = 6;
    y = 100;
}

x =  $\phi(\dots)$ ;
y =  $\phi(\dots)$ ;
print(x)
```

# Maximal SSA

## Example

```
x = 1;
y = 2;

if (<condition>) {
    x = y;
}

else {
    x = 6;
    y = 100;
}

print(x)
```

Insert  $\phi$  with argument placeholders

```
x = 1;
y = 2;

if (<condition>) {
    x =  $\phi(\dots)$ ;
    y =  $\phi(\dots)$ ;
    x = y;
}

else {
    x =  $\phi(\dots)$ ;
    y =  $\phi(\dots)$ ;
    x = 6;
    y = 100;
}

x =  $\phi(\dots)$ ;
y =  $\phi(\dots)$ ;
print(x)
```

Rename variables  
iterate through basic  
blocks with a global  
counter

```
x0 = 1;
y1 = 2;

if (<condition>) {
    x3 =  $\phi(\dots)$ ;
    y4 =  $\phi(\dots)$ ;
    x5 = y4;
}

else {
    x6 =  $\phi(\dots)$ ;
    y7 =  $\phi(\dots)$ ;
    x8 = 6;
    y9 = 100;
}

x10 =  $\phi(\dots)$ ;
y11 =  $\phi(\dots)$ ;
print(x10)
```

# Maximal SSA

## Example

```
x = 1;
y = 2;

if (<condition>) {
    x = y;
}

else {
    x = 6;
    y = 100;
}

print(x)
```

Insert  $\phi$  with argument placeholders

```
x = 1;
y = 2;

if (<condition>) {
    x =  $\phi(\dots)$ ;
    y =  $\phi(\dots)$ ;
    x = y;
}

else {
    x =  $\phi(\dots)$ ;
    y =  $\phi(\dots)$ ;
    x = 6;
    y = 100;
}

x =  $\phi(\dots)$ ;
y =  $\phi(\dots)$ ;
print(x)
```

Rename variables  
iterate through basic  
blocks with a global  
counter

```
x0 = 1;
y1 = 2;

if (<condition>) {
    x3 =  $\phi(\dots)$ ;
    y4 =  $\phi(\dots)$ ;
    x5 = y4;
}

else {
    x6 =  $\phi(\dots)$ ;
    y7 =  $\phi(\dots)$ ;
    x8 = 6;
    y9 = 100;
}

x10 =  $\phi(\dots)$ ;
y11 =  $\phi(\dots)$ ;
print(x10)
```

fill in  $\phi$  arguments  
by considering CFG

```
x0 = 1;
y1 = 2;

if (<condition>) {
    x3 =  $\phi(x0)$ ;
    y4 =  $\phi(y1)$ ;
    x5 = y4;
}

else {
    x6 =  $\phi(x0)$ ;
    y7 =  $\phi(y1)$ ;
    x8 = 6;
    y9 = 100;
}

x10 =  $\phi(x5, x8)$ ;
y11 =  $\phi(y4, y9)$ ;
print(x10)
```

# More efficient translation?

## Example

```
x = 1;
y = 2;

if (...) {
    x = y;
}

else {
    x = 6;
    y = 100;
}

print(x)
```

## maximal SSA

```
x0 = 1;
y1 = 2;

if (...) {
    x3 =  $\phi$ (x0);
    y4 =  $\phi$ (y1);
    x5 = y4;
}

else {
    x6 =  $\phi$ (x0);
    y7 =  $\phi$ (y1);
    x8 = 6;
    y9 = 100;
}

x10 =  $\phi$ (x5, x8);
y11 =  $\phi$ (y4, y9);
print(x10)
```

## Optimized?

```
x0 = 1;
y1 = 2;

if (...) {
    x3 =  $\phi$ (x0);
    y4 =  $\phi$ (y1);
    x5 = y4;
}

else {
    x6 =  $\phi$ (x0);
    y7 =  $\phi$ (y1);
    x8 = 6;
    y9 = 100;
}

x10 =  $\phi$ (x5, x8);
y11 =  $\phi$ (y4, y9);
print(x10)
```

# More efficient translation?

## Example

```
x = 1;
y = 2;

if (...) {
    x = y;
}

else {
    x = 6;
    y = 100;
}

print(x)
```

## maximal SSA

```
x0 = 1;
y1 = 2;

if (...) {
    x3 =  $\phi$ (x0);
    y4 =  $\phi$ (y1);
    x5 = y4;
}

else {
    x6 =  $\phi$ (x0);
    y7 =  $\phi$ (y1);
    x8 = 6;
    y9 = 100;
}

x10 =  $\phi$ (x5, x8);
y11 =  $\phi$ (y4, y9);
print(x10)
```

## Hand Optimized SSA

```
x0 = 1;
y1 = 2;

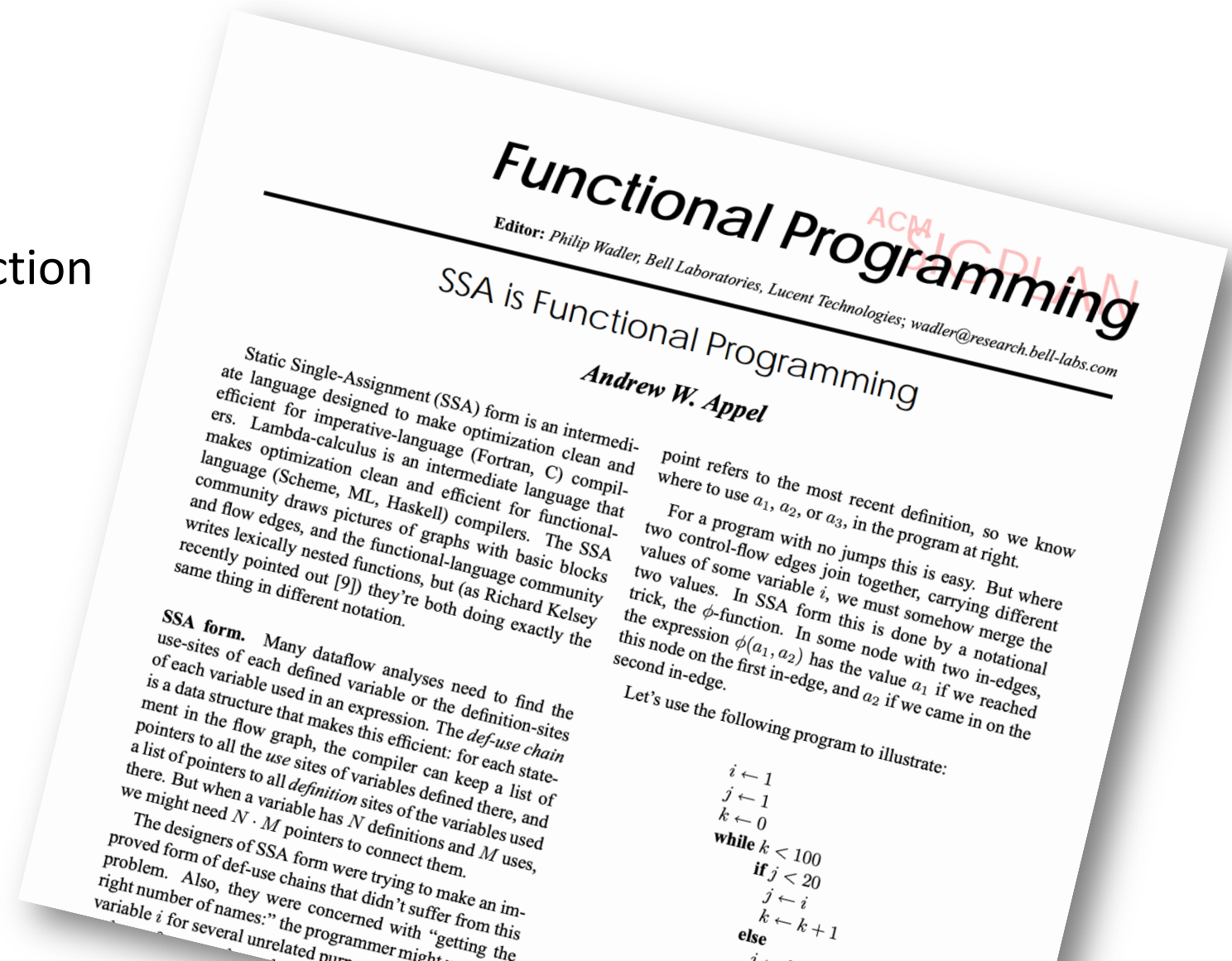
if (...) {
    x5 = y1;
}

else {
    x8 = 6;
    y9 = 100;
}

x10 =  $\phi$ (x5, x8);
y11 =  $\phi$ (y1, y9);
print(x10)
```

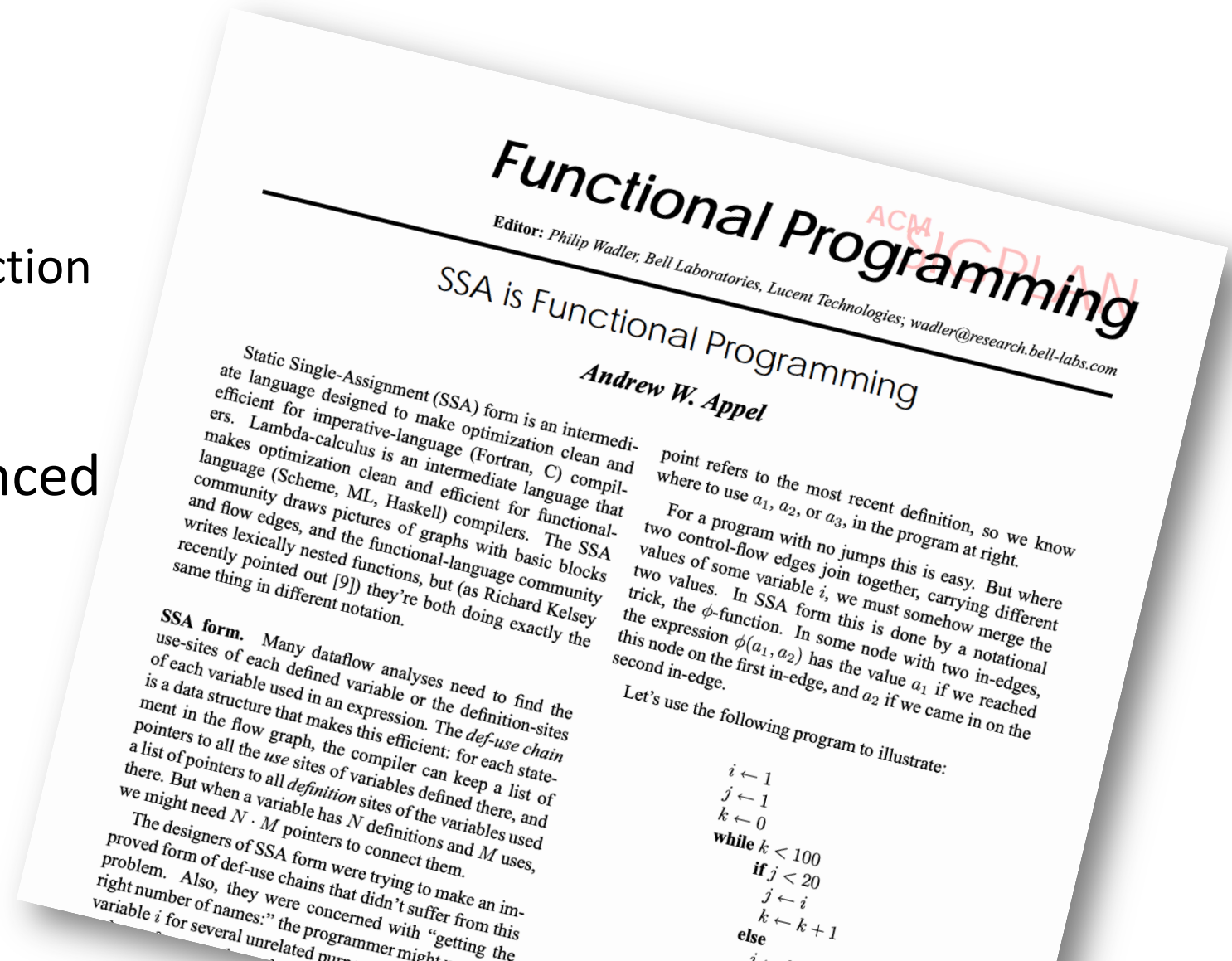
# A note on SSA variants:

- “Really Crude Approach”:
  - Just like our example:
  - Every block has a  $\phi$  instruction for every variable





- “Really Crude Approach”:
  - Just like our example:
  - Every block has a  $\phi$  instruction for every variable
- This approach was referenced in a later paper as “Maximal SSA”



# A note on SSA variants:

- EAC book describes a different “Maximal SSA”
  - Insert  $\phi$  instruction at every join node
  - Naming becomes more difficult

## Appel Maximal SSA

```
x0 = 1;  
y1 = 2;  
  
if (<condition>) {  
    x3 =  $\phi$ (x0);  
    y4 =  $\phi$ (y1);  
    x5 = y4;  
}  
  
else {  
    x6 =  $\phi$ (x0);  
    y7 =  $\phi$ (y1);  
    x8 = 6;  
    y9 = 100;  
}  
  
x10 =  $\phi$ (x5, x8);  
y11 =  $\phi$ (y4, y9);  
print(x10)
```

## EAC Maximal SSA

```
x0 = 1;  
y1 = 2;  
  
if (...) {  
    x5 = y1;  
}  
  
else {  
    x8 = 6;  
    y9 = 100;  
}  
  
x10 =  $\phi$ (x5, x8);  
y11 =  $\phi$ (y1, y9);  
print(x10)
```

# A note on SSA variants:

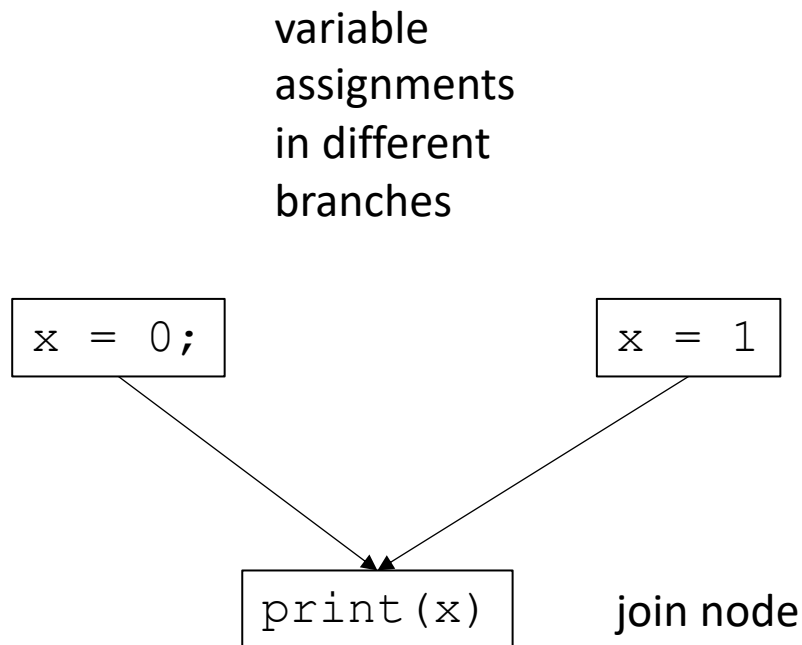
- EAC book describes:
  - Minimal SSA
  - Pruned SSA
  - **Semipruned SSA: We will discuss this one**

# A more optimal approach for $\phi$ placements

- When is a  $\phi$  needed?

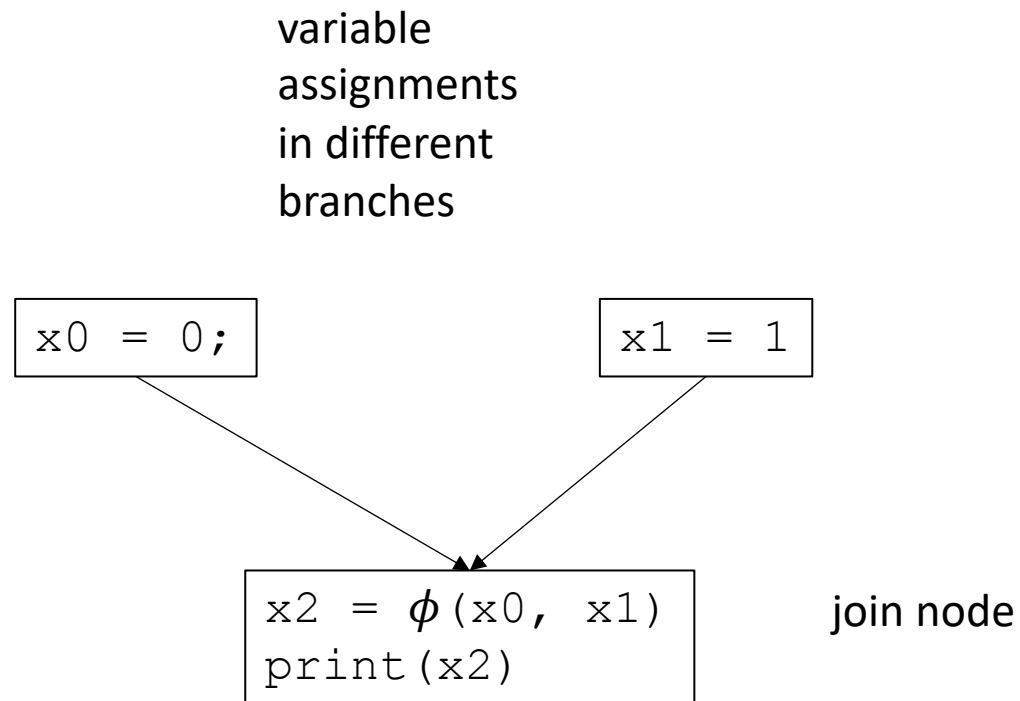
# A more optimal approach for $\phi$ placements

- When is a  $\phi$  needed?



# A more optimal approach for $\phi$ placements

- When is a  $\phi$  needed?



# A more optimal approach for $\phi$ placements

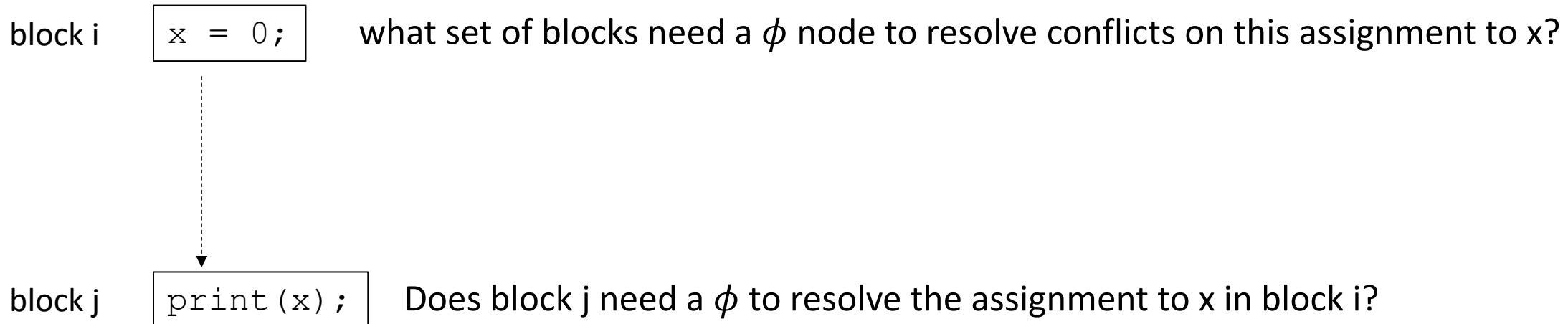
- When is a  $\phi$  needed?
- More specific question: given a block  $i$ , find the set of blocks  $B$  which may need a  $\phi$  instruction for a definition in block  $i$ .

$x = 0;$
----------

what set of blocks need a  $\phi$  node to resolve conflicts on this assignment to  $x$ ?

# A more optimal approach for $\phi$ placements

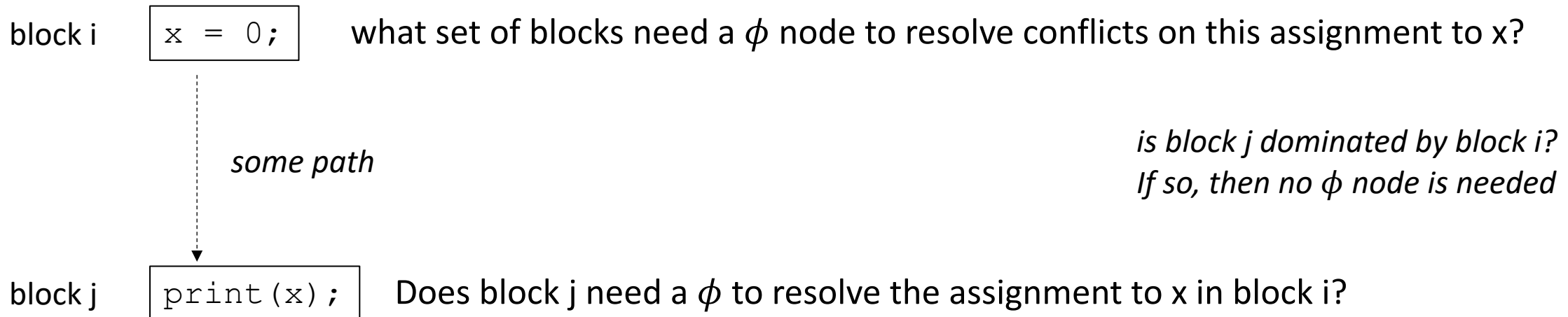
- When is a  $\phi$  needed?
- More specific question: given a block  $i$ , find the set of blocks  $B$  which may need a  $\phi$  instruction for a definition in block  $i$ .





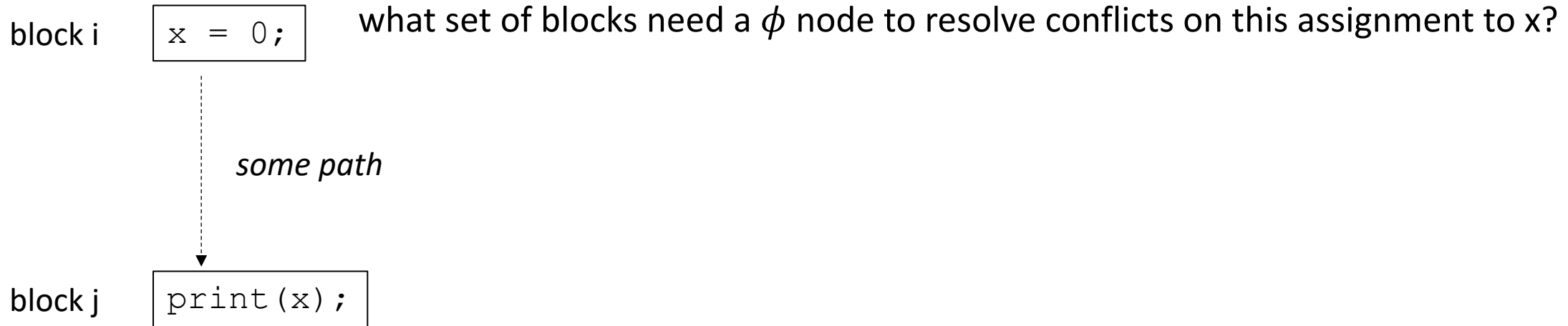
# A more optimal approach for $\phi$ placements

- When is a  $\phi$  needed?
- More specific question: given a block  $i$ , find the set of blocks  $B$  which may need a  $\phi$  instruction for a definition in block  $i$ .



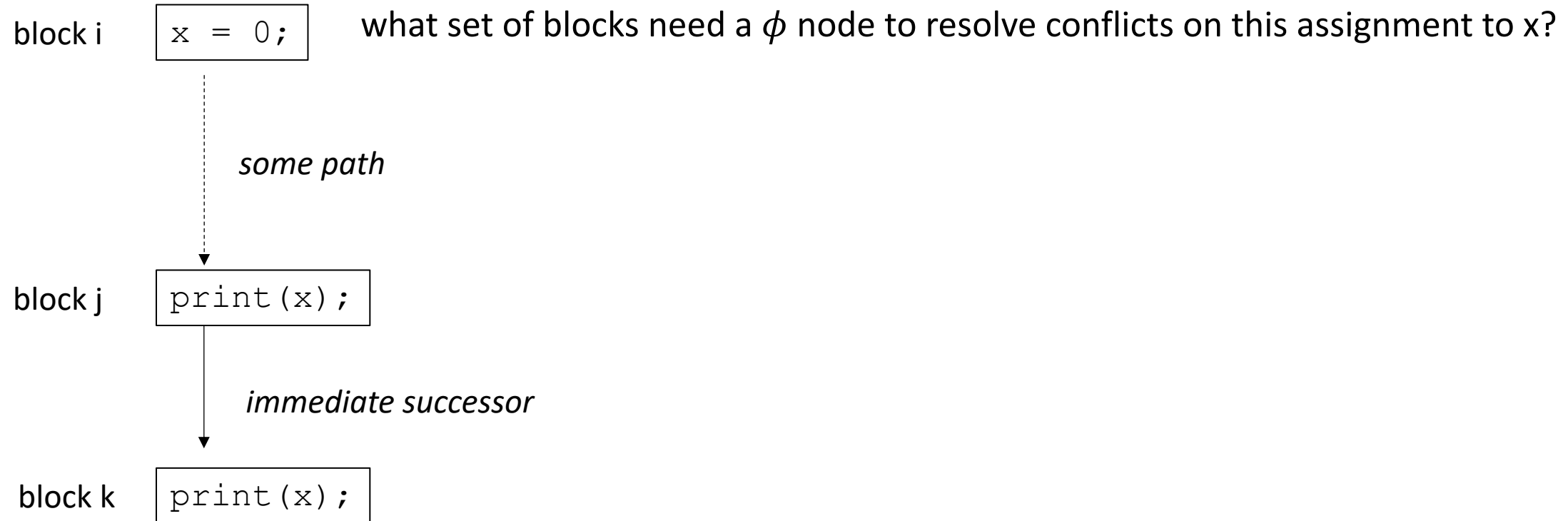
# A more optimal approach for $\phi$ placements

- say j is dominated by i. Thus, no  $\phi$  node is needed in block j



# A more optimal approach for $\phi$ placements

- say  $j$  is dominated by  $i$ . Thus, no  $\phi$  node is needed in block  $j$



# A more optimal approach for $\phi$ placements

- say j is dominated by i. Thus, no  $\phi$  node is needed in block j

block i    `x = 0;`    what set of blocks need a  $\phi$  node to resolve conflicts on this assignment to x?

*some path*

block j    `print(x);`

*immediate successor*

block k    `print(x);`

path that doesn't go through block i and assigns to x

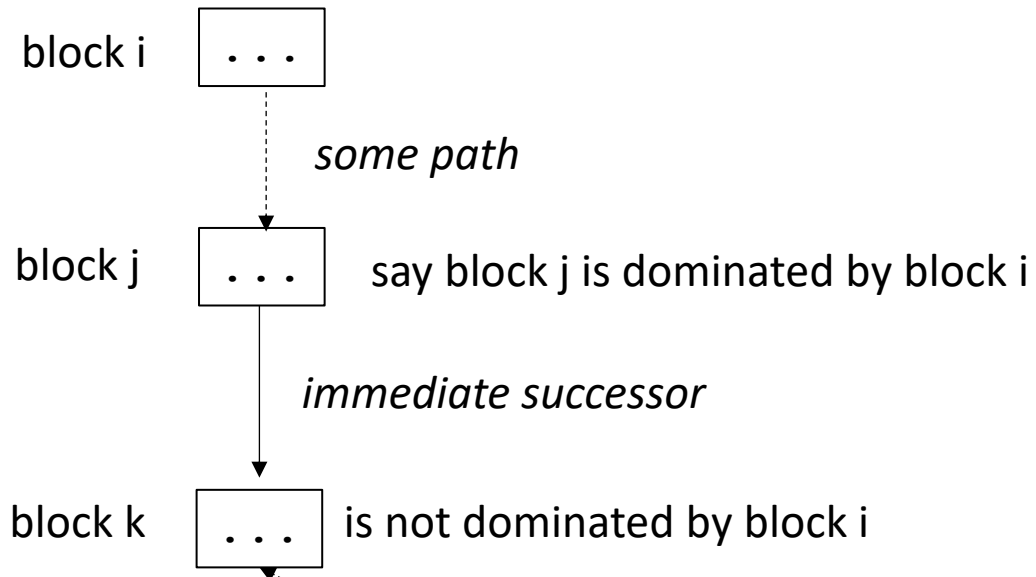
Say block k is not dominated by block i.  
Then there exists another in-edge to block k.

If x is assigned along a path not through block i,  
then a  $\phi$  node is needed

Dominance frontier

# Dominance frontier

- For a block  $i$ , the set of blocks  $B$  in  $i$ 's dominance frontier lie just “outside” the blocks that  $i$  dominates.



*example: block k is in the dominance frontier of block i*

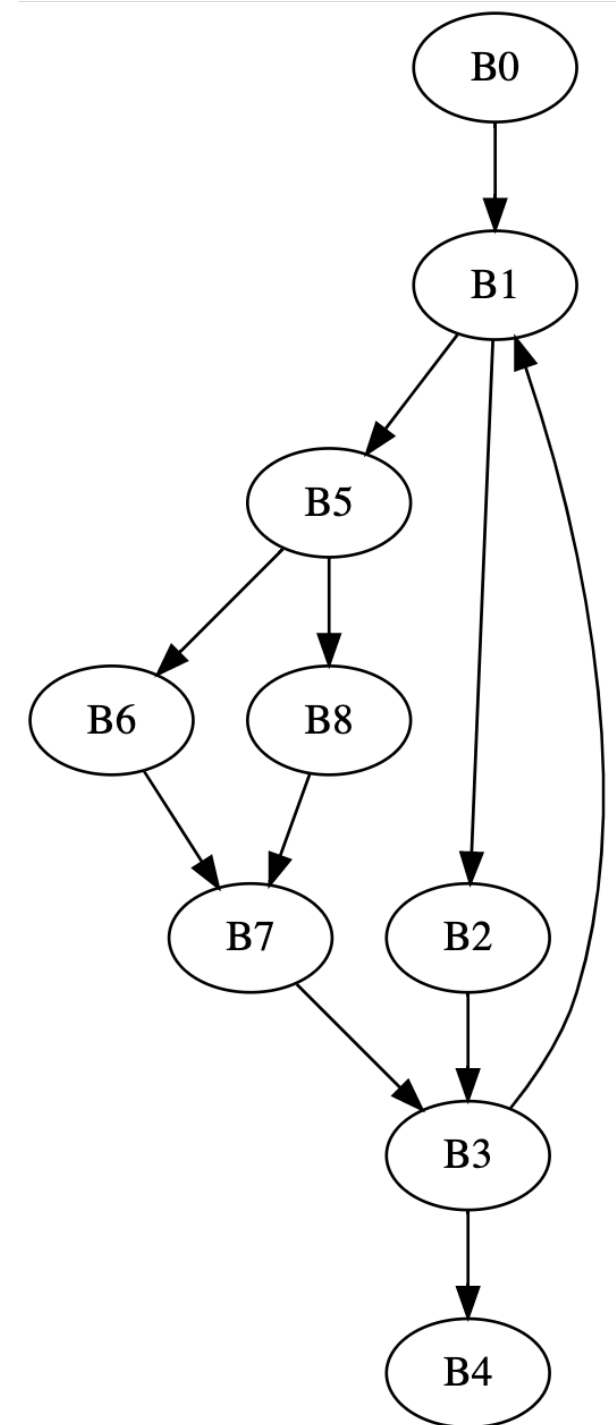
*There will be some path into block k that does not go through block i*

# Dominance frontier

- a viz using coloring (thanks to Chris Liu!)
- Efficient algorithm for computing in EAC section 9.3.2 using a dominator tree.

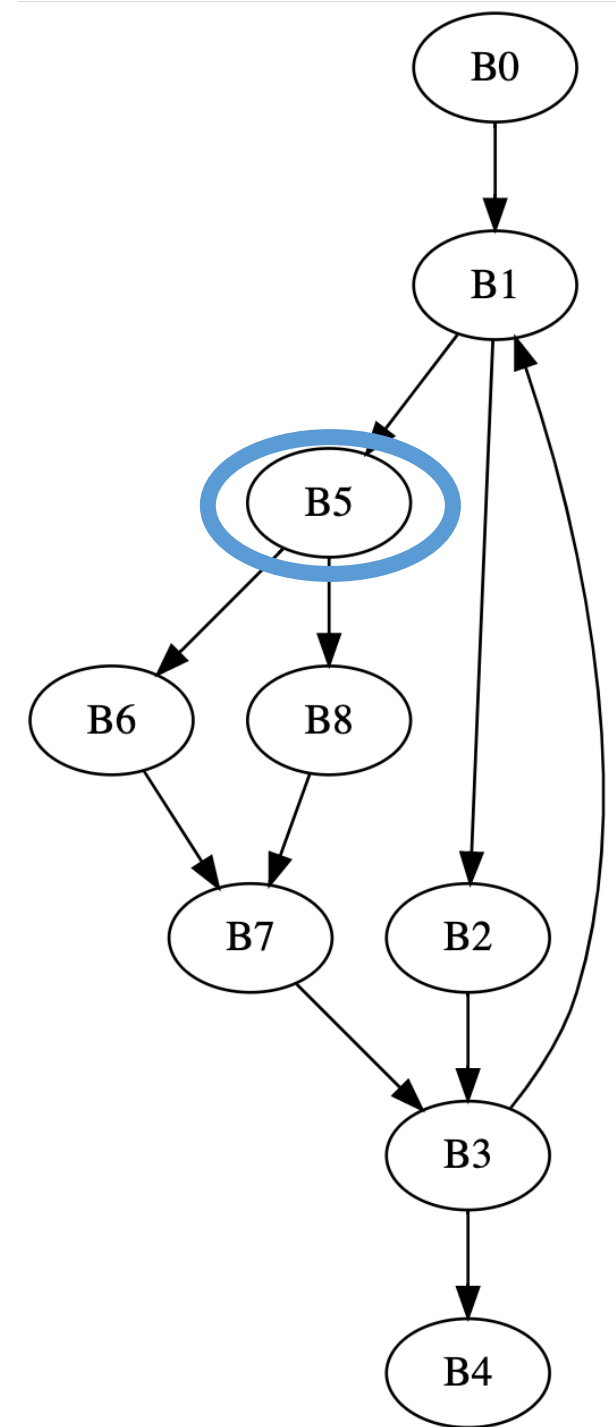
*Note that we are using strict dominance:  
nodes don't dominate themselves!*

Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

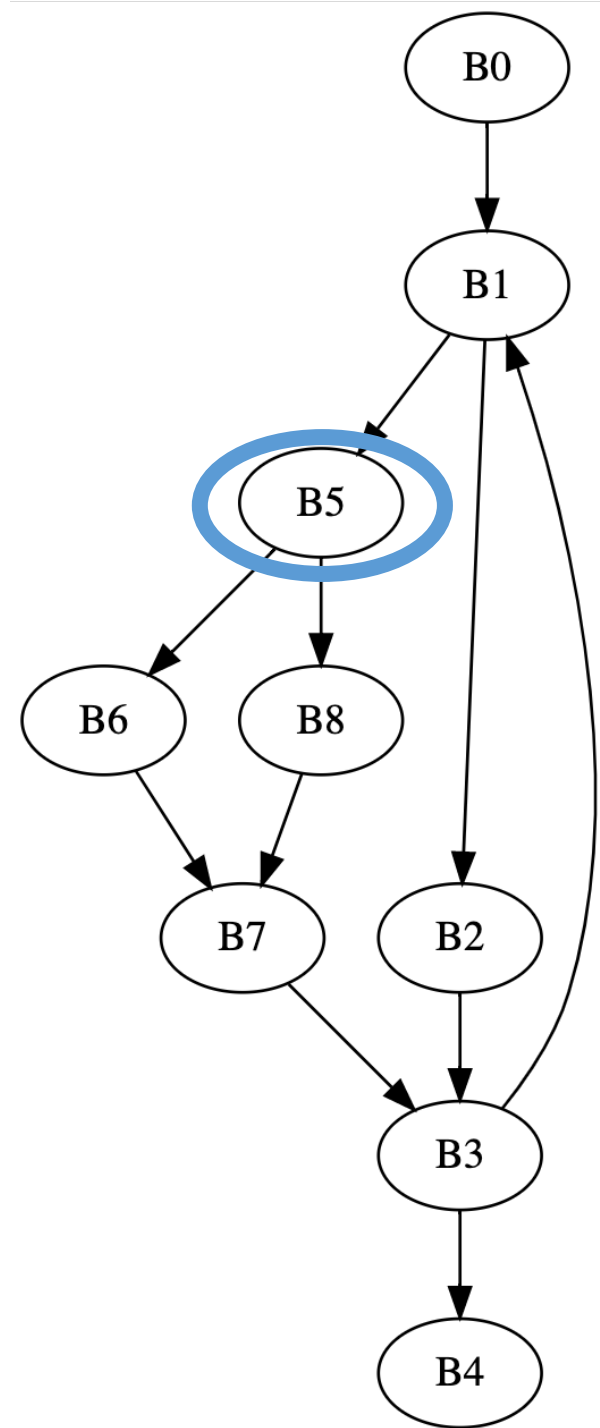




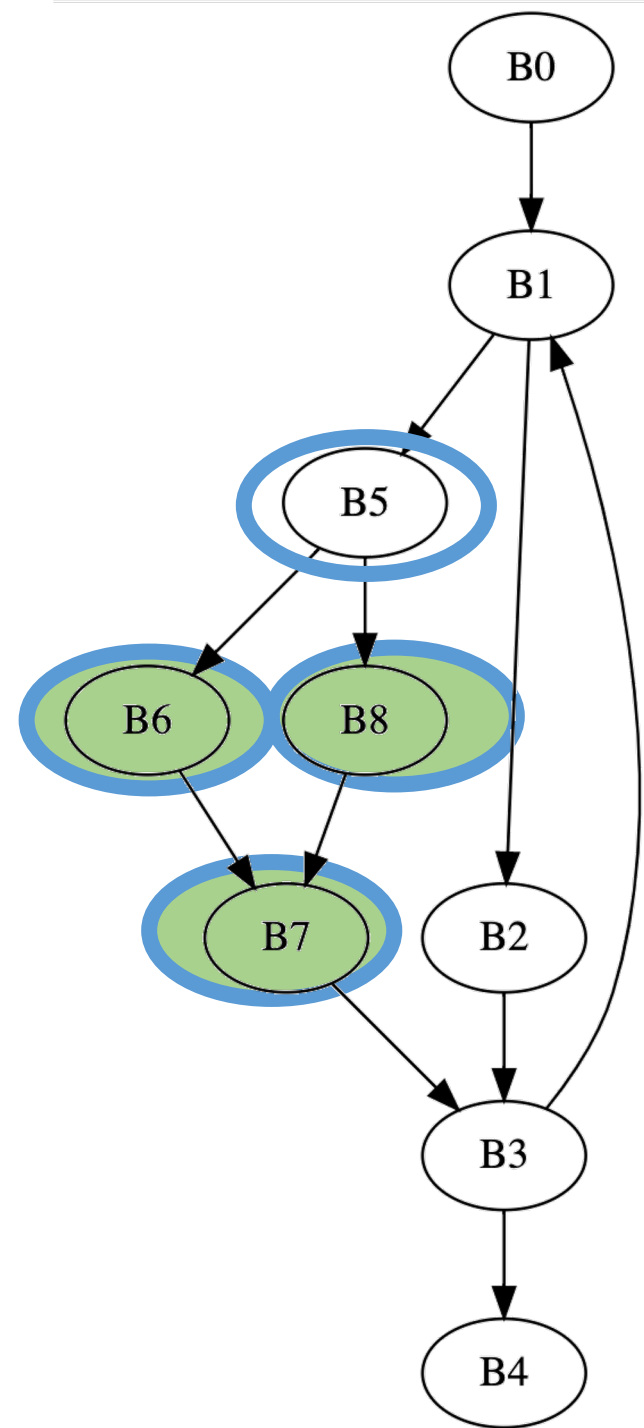
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



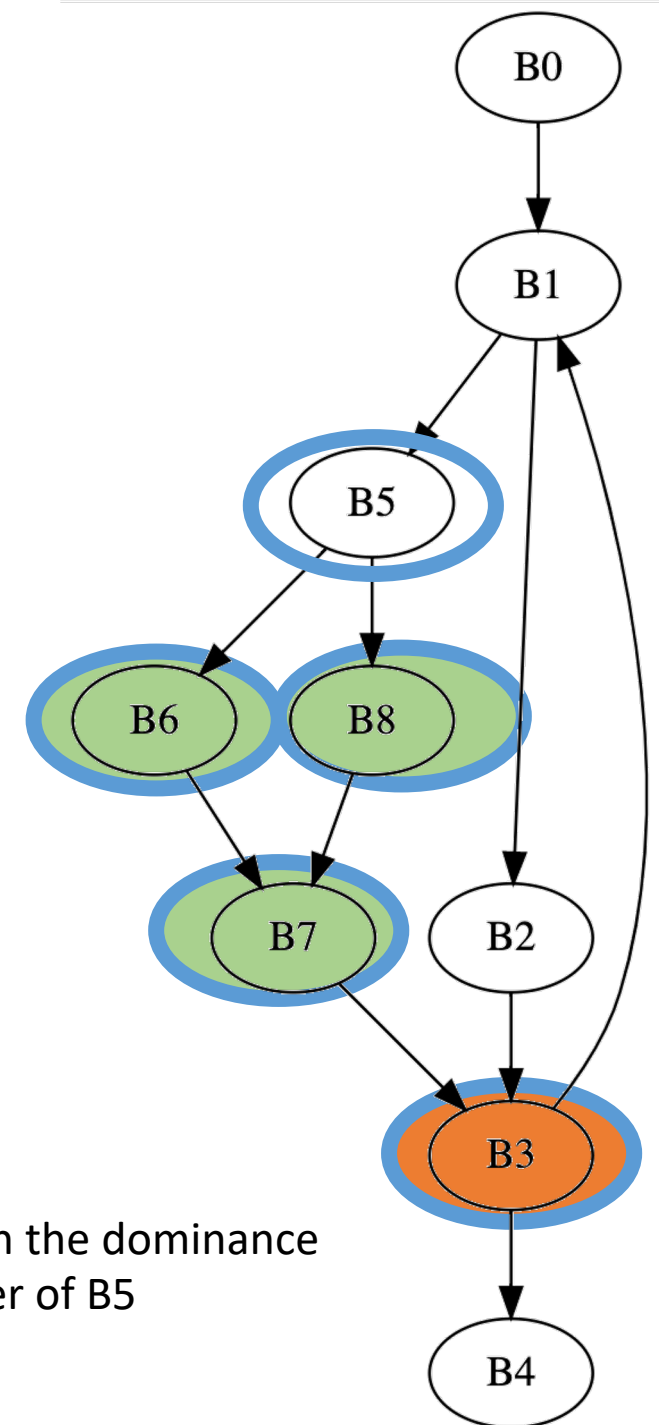
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

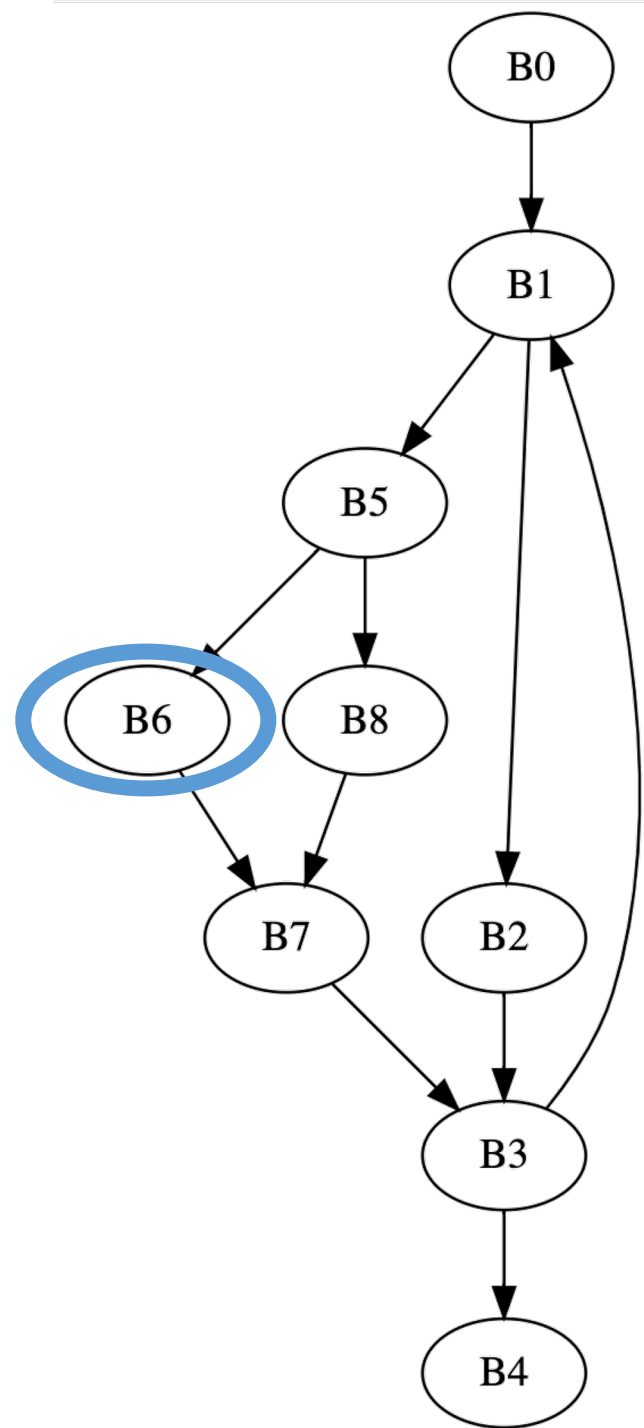


Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

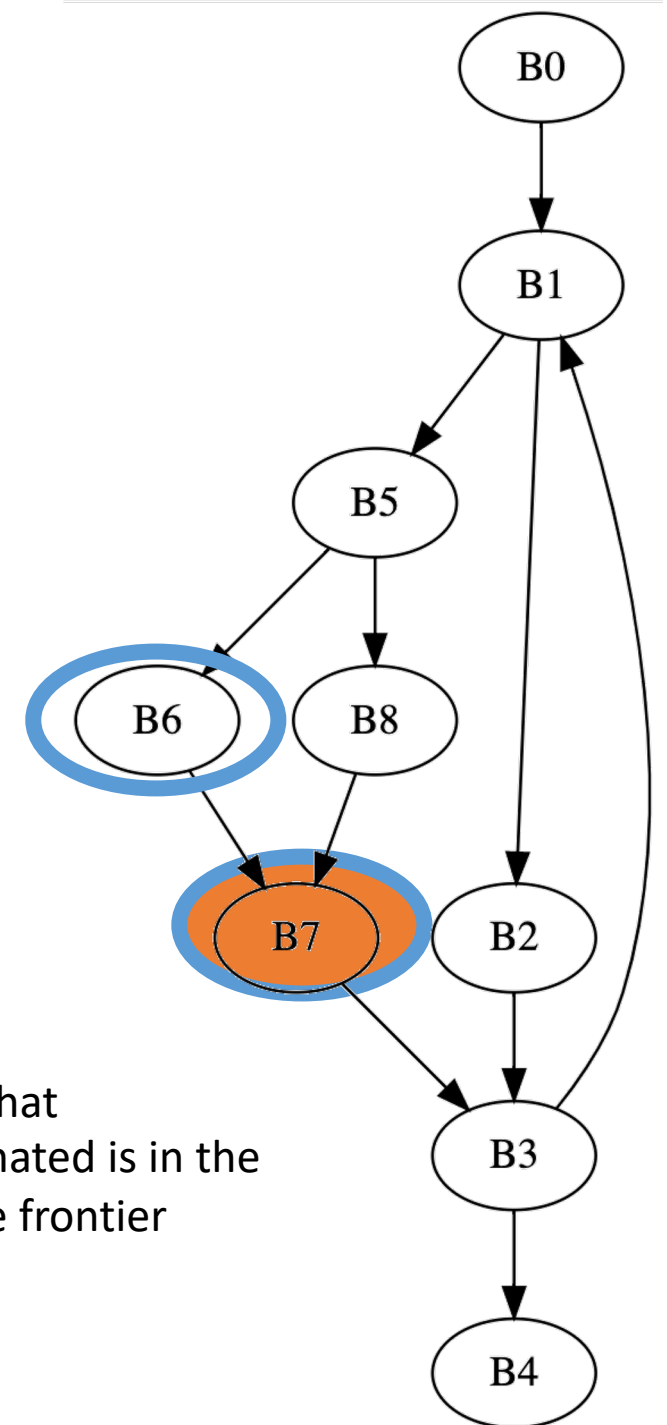


B3 is in the dominance frontier of B5

Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

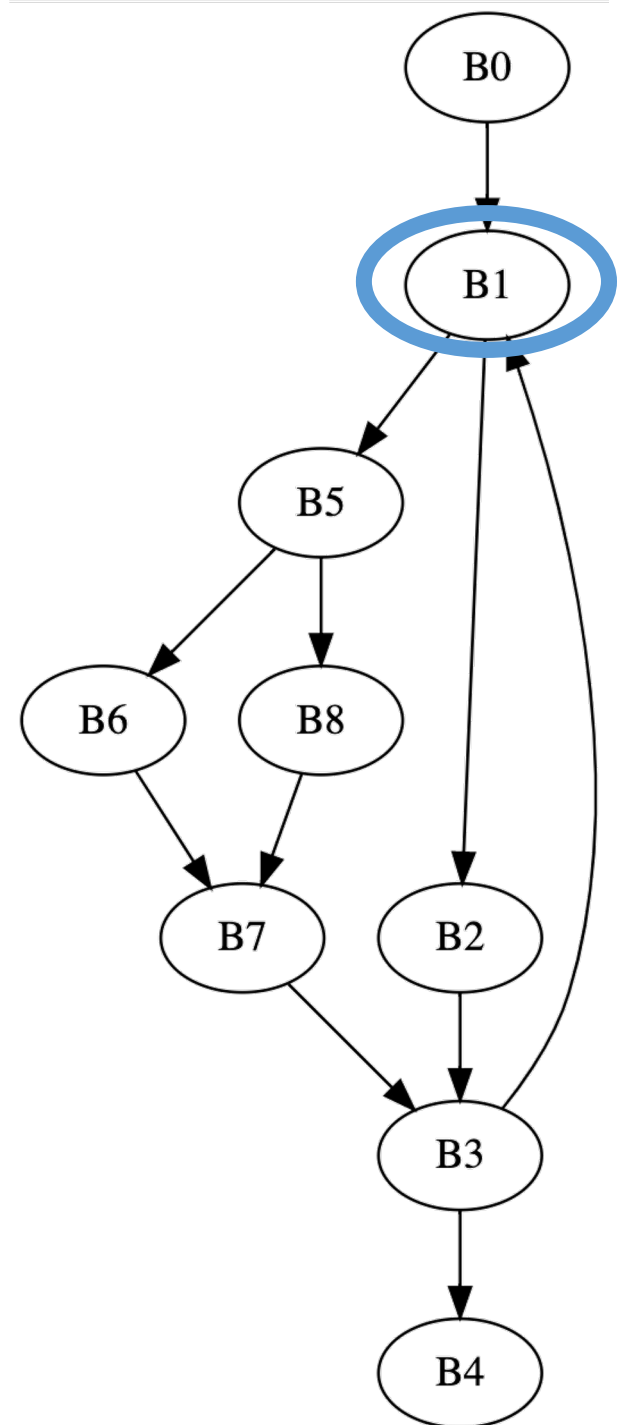


Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

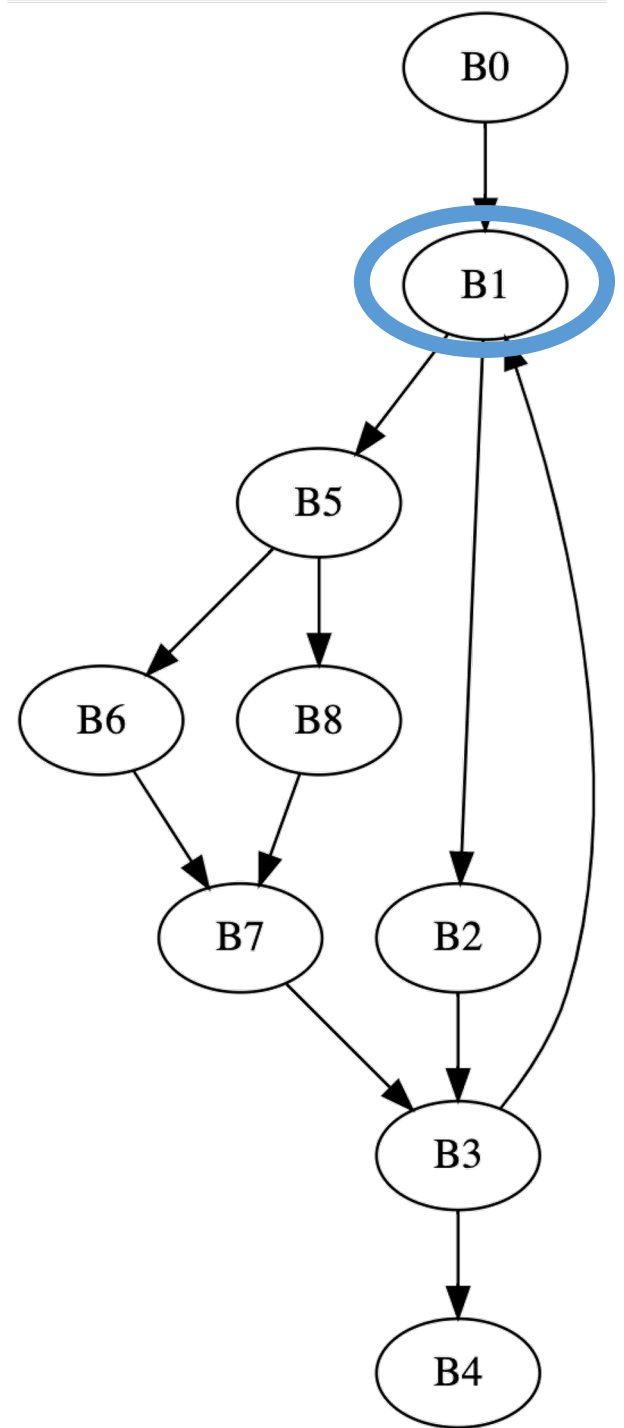


Any child that  
isn't dominated is in the  
dominance frontier

Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

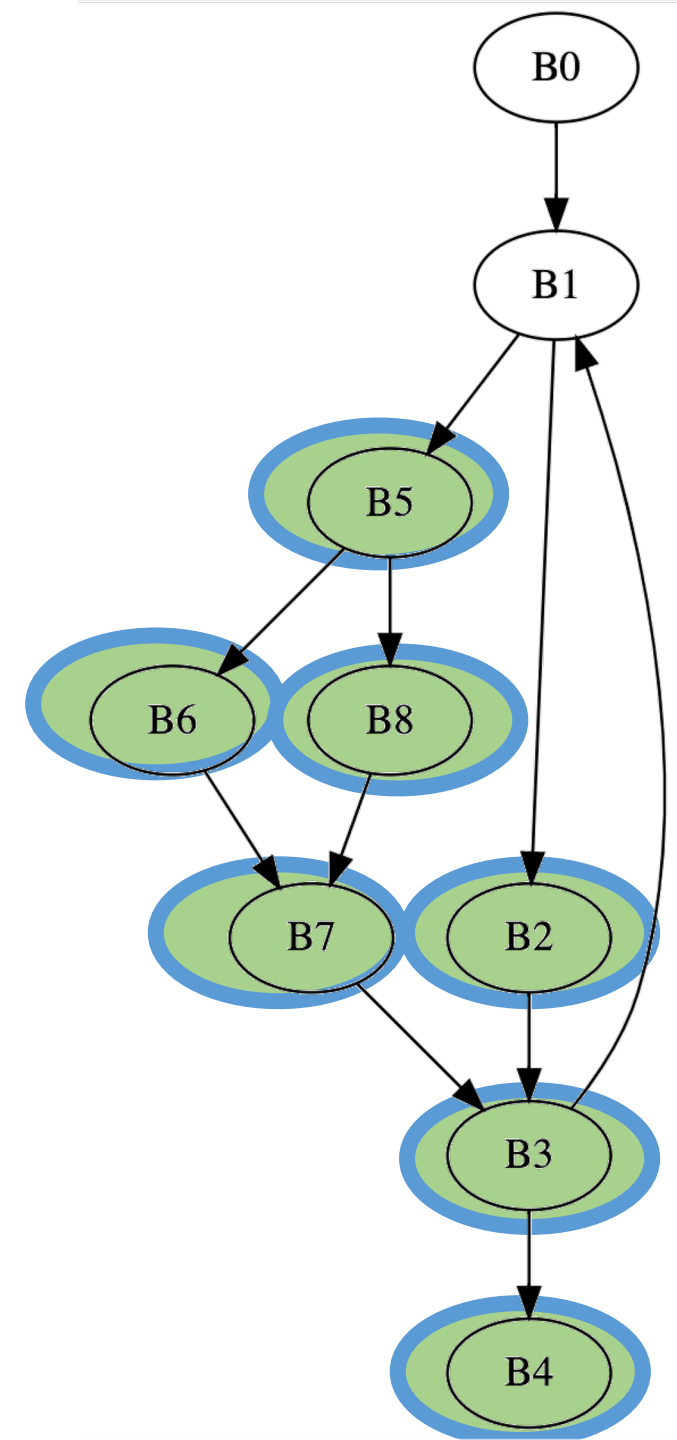


Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

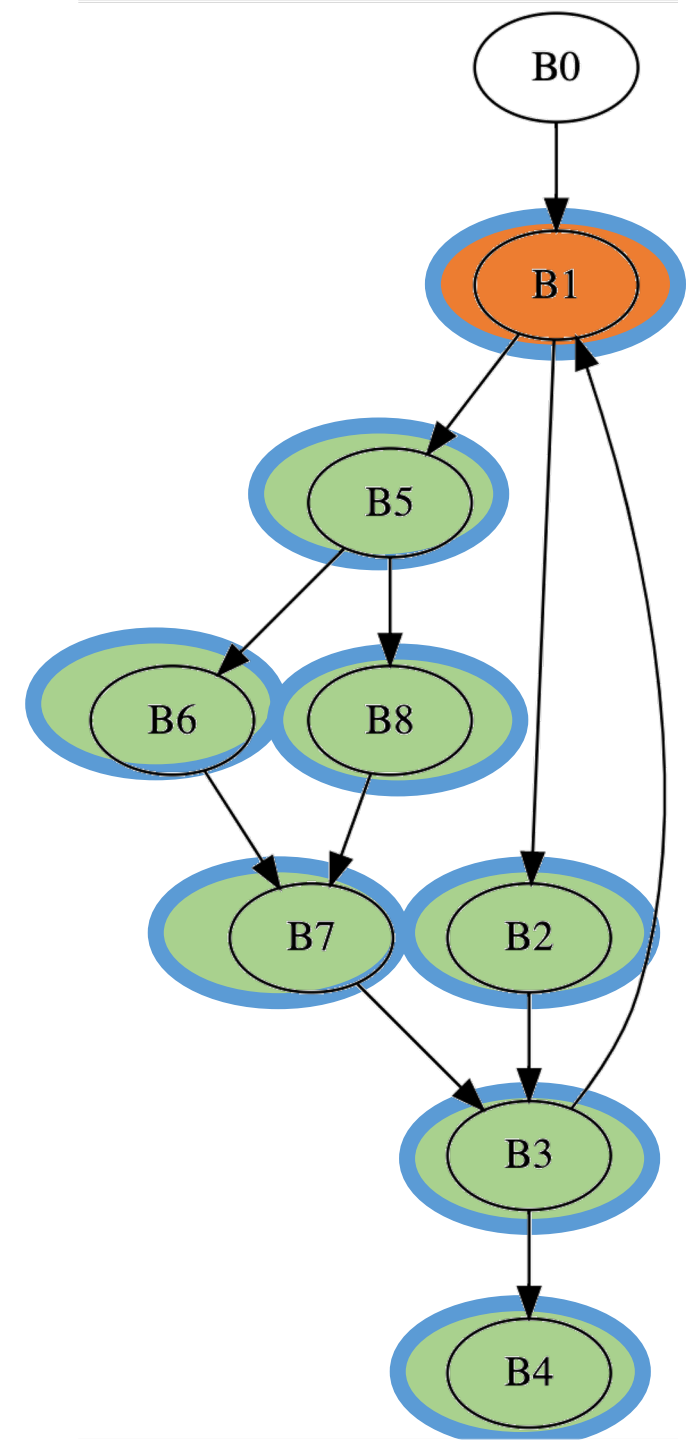




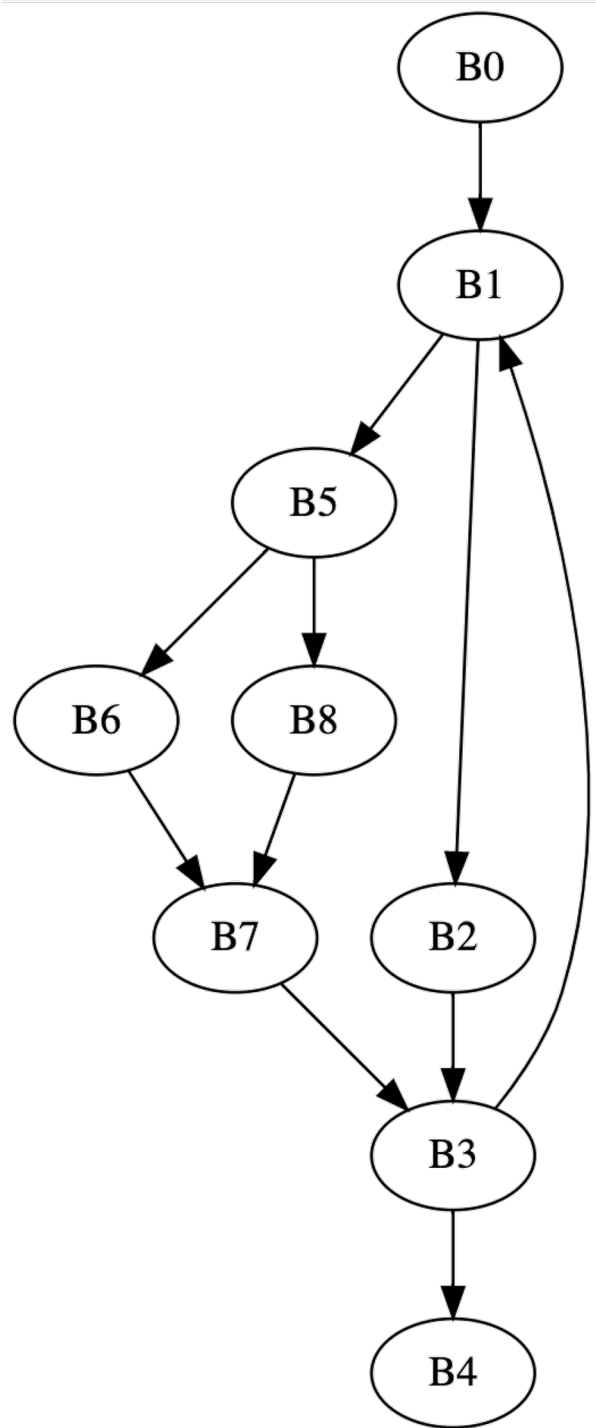
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7



# Dominance Frontier

- Intuition: a variable declared in block  $b$  may need to resolve a conflict in the dominance frontier of  $b$ 
  - Because it may have been assigned a new value in another path

```
B0: i = ...;

B1: a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;

B4: return;
```

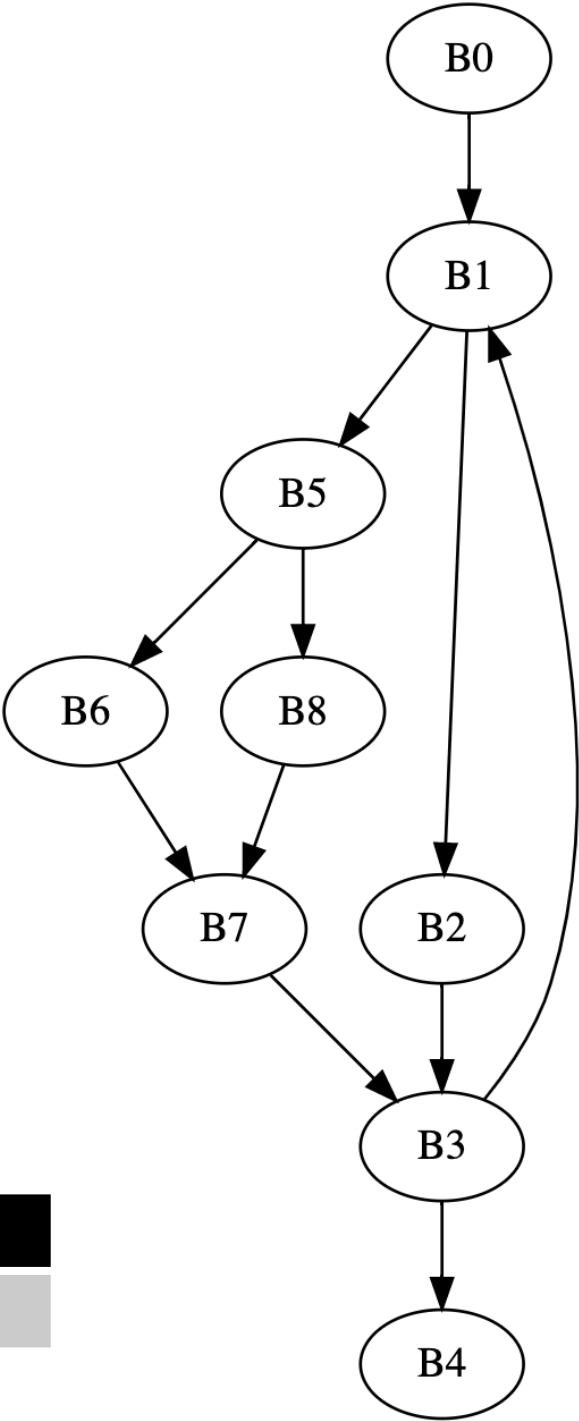
```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```

Var	a	b	c	d	i	y	z
Blocks							



```
B0: i = ...;

B1: a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;

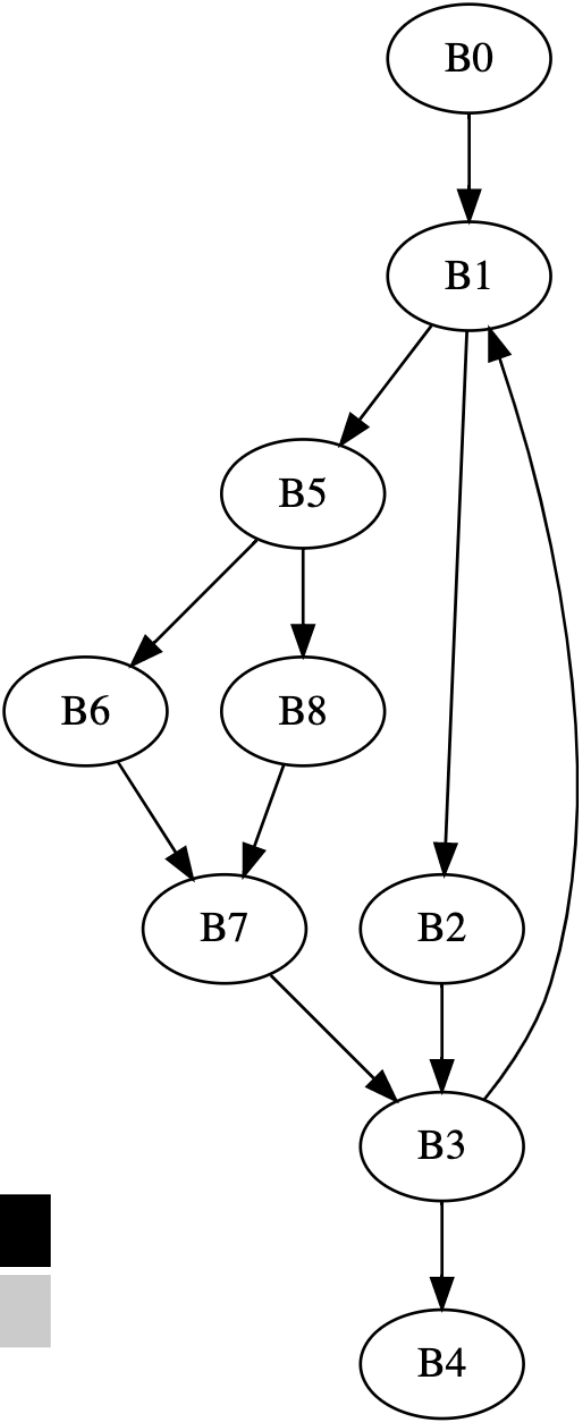
B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```



Var	a	b	c	d	i	y	z
Blocks	B1, B5	B2, B7	B1,B2,B8	B2,B5,B6	B0, B3	B3	B3

```
B0: i = ...;

B1: a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;

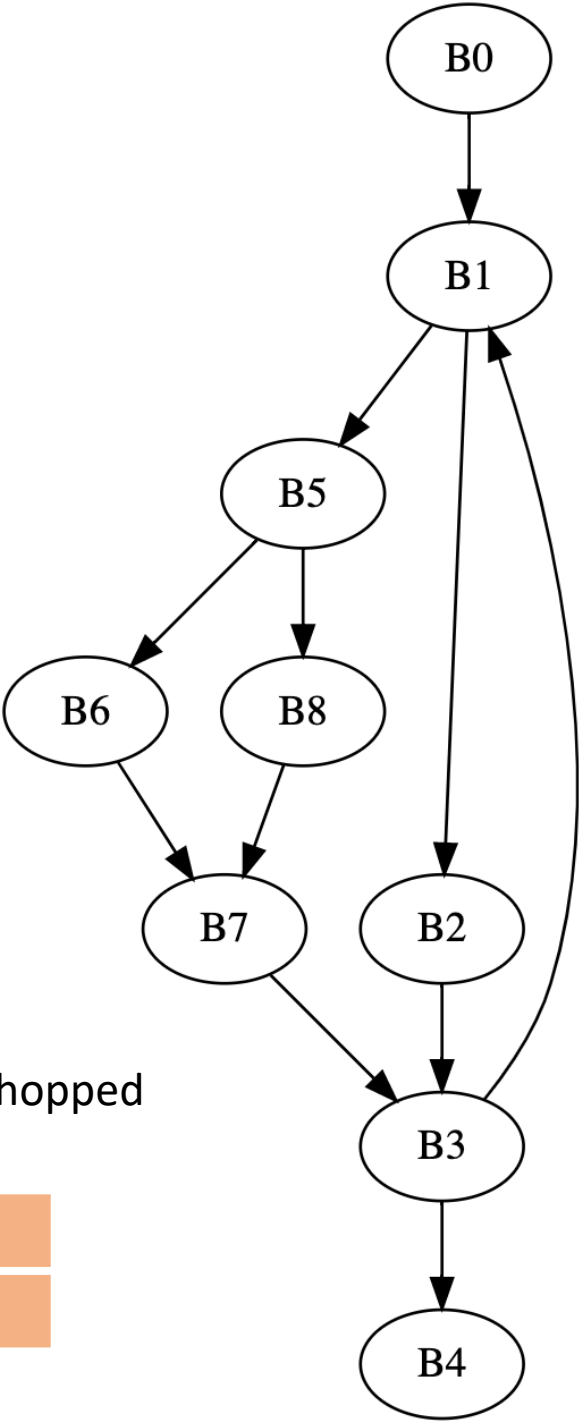
B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```



local variables can be chopped

Var	a	b	c	d	i	y	z
Blocks	B1, B5	B2, B7	B1,B2,B8	B2,B5,B6	B0, B3	B3	B3

B0: i = ...;

B1: a = ...;  
c = ...;  
br ... B2, B5;

B2: b = ...;  
c = ...;  
d = ...;

B3: y = ...;  
z = ...;  
i = ...;  
br ... B1, B4;

B4: return;

B5: a = ...;  
d = ...;  
br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;  
br B7;

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a	b	c	d	i
Blocks	B1,B5	B2,B7	B1,B2,B8	B2,B5,B6	B0,B3



```
B0: i = ...;

B1: a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;

B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a
Blocks	B1,B5

for each variable v:  
  for each block b that writes to v:  
     $\phi$  is needed in the DF of b

```
B0: i = ...;

B1: a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;

B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a
Blocks	B1,B5

for each variable v:  
  for each block b that writes to v:  
     $\phi$  is needed in the DF of b

```
B0: i = ...;

B1: a =  $\phi$ (...);
    a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;

B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a
Blocks	B1,B5

for each variable v:  
  for each block b that writes to v:  
     $\phi$  is needed in the DF of b

```
B0: i = ...;

B1: a =  $\phi$ (...);
    a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;

B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a
Blocks	B1,B5

for each variable v:  
  for each block b that writes to v:  
     $\phi$  is needed in the DF of b

```
B0: i = ...;

B1: a =  $\phi$ (...);
    a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: a =  $\phi$ (...);
    y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;
```

```
B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a
Blocks	B1,B5

for each block b:  
 $\phi$  is needed in the DF of b

```
B0: i = ...;

B1: a =  $\phi$ (...);
    a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: a =  $\phi$ (...);
    y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;
```

```
B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a
Blocks	B1,B5

We’ve now added new definitions of ‘a’!

```
B0: i = ...;

B1: a = ϕ(...);
    a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: a = ϕ(...);
    y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;
```

```
B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a
Blocks	B1,B5,B1,B3

We’ve now added new definitions of ‘a’!

```
B0: i = ...;

B1: a =  $\phi$ (...);
    a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: a =  $\phi$ (...);
    y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;
```

```
B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a
Blocks	B1,B5,B3

We’ve now added new definitions of ‘a’!



B0: i = ...;

B1: a =  $\phi$ (...);  
a = ...;  
c = ...;  
**br** ... B2, B5;

B2: b = ...;  
c = ...;  
d = ...;

B3: a =  $\phi$ (...);  
y = ...;  
z = ...;  
i = ...;  
**br** ... B1, B4;

B4: **return**;

B5: a = ...;  
d = ...;  
**br** ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;  
**br** B7;

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a	b
Blocks	B1,B5,B3	B2,B7

```
B0: i = ...;

B1: a =  $\phi$ (...);
    a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: a =  $\phi$ (...);
    y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;
```

```
B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a	b
Blocks	B1,B5,B3	B2,B7

B0: i = ...;

B1: a =  $\phi$ (...);  
a = ...;  
c = ...;  
**br** ... B2, B5;

B2: b = ...;  
c = ...;  
d = ...;

B3: a =  $\phi$ (...);  
b =  $\phi$ (...);  
y = ...;  
z = ...;  
i = ...;  
**br** ... B1, B4;

B4: **return**;

B5: a = ...;  
d = ...;  
**br** ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;  
**br** B7;

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a	b
Blocks	B1,B5,B3	B2,B7

B0: i = ...;

B1: a =  $\phi$ (...);  
a = ...;  
c = ...;  
**br** ... B2, B5;

B2: b = ...;  
c = ...;  
d = ...;

B3: a =  $\phi$ (...);  
b =  $\phi$ (...);  
y = ...;  
z = ...;  
i = ...;  
**br** ... B1, B4;

B4: **return**;

B5: a = ...;  
d = ...;  
**br** ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;  
**br** B7;

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a	b
Blocks	B1,B5,B3	B2,B7

B0: i = ...;

B1: a =  $\phi$ (...);  
a = ...;  
c = ...;  
**br** ... B2, B5;

B2: b = ...;  
c = ...;  
d = ...;

B3: a =  $\phi$ (...);  
b =  $\phi$ (...);  
y = ...;  
z = ...;  
i = ...;  
**br** ... B1, B4;

B4: **return**;

B5: a = ...;  
d = ...;  
**br** ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;  
**br** B7;

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a	b
Blocks	B1,B5,B3	B2,B7,B3

B0: i = ...;

B1: a =  $\phi$ (...);  
b =  $\phi$ (...);  
a = ...;  
c = ...;  
**br** ... B2, B5;

B2: b = ...;  
c = ...;  
d = ...;

B3: a =  $\phi$ (...);  
b =  $\phi$ (...);  
y = ...;  
z = ...;  
i = ...;  
**br** ... B1, B4;

B4: **return**;

B5: a = ...;  
d = ...;  
**br** ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;  
**br** B7;

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a	b
Blocks	B1,B5,B3	B2,B7,B3,B1

```
B0: i = ...;

B1: a = ϕ(...);
    b = ϕ(...);
    a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: a = ϕ(...);
    b = ϕ(...);
    y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;
```

```
B4: return;
```

```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

B8: c = ...;
    br B7;
```

Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a	b
Blocks	B1,B5,B3	B2,B7,B3.B1

```

B0: i0 = ...;

B1: a =  $\phi$ (...);
    b =  $\phi$ (...);
    c =  $\phi$ (...);
    d =  $\phi$ (...);
    i =  $\phi$ (...);
    a = ...;
    c = ...;
    br ... B2, B5;

B2: b = ...;
    c = ...;
    d = ...;

B3: a =  $\phi$ (...);
    b =  $\phi$ (...);
    c =  $\phi$ (...);
    d =  $\phi$ (...);
    y = ...;
    z = ...;
    i = ...;
    br ... B1, B4;

```

```

B4: return

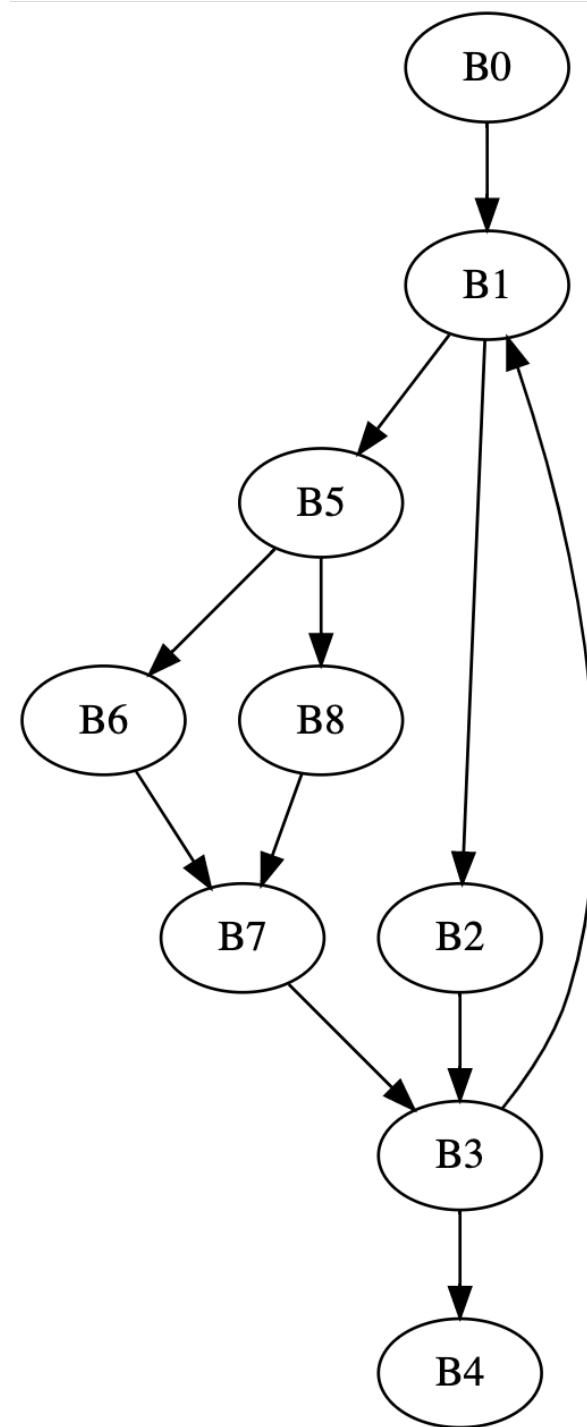
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: d =  $\phi$ (...);
    c =  $\phi$ (...);
    b = ...;

B8: c = ...;
    br B7;

```





```

B0: i0 = ...;

B1: a0 =  $\phi$ (...);
    b1 =  $\phi$ (...);
    c2 =  $\phi$ (...);
    d3 =  $\phi$ (...);
    i4 =  $\phi$ (...);
    a5 = ...;
    c6 = ...;
    br ... B2, B5;

B2: b7 = ...;
    c8 = ...;
    d9 = ...;

B3: a10 =  $\phi$ (...);
    b11 =  $\phi$ (...);
    c12 =  $\phi$ (...);
    d13 =  $\phi$ (...);
    y14 = ...;
    z15 = ...;
    i16 = ...;
    br ... B1, B4;

```

```

B4: return

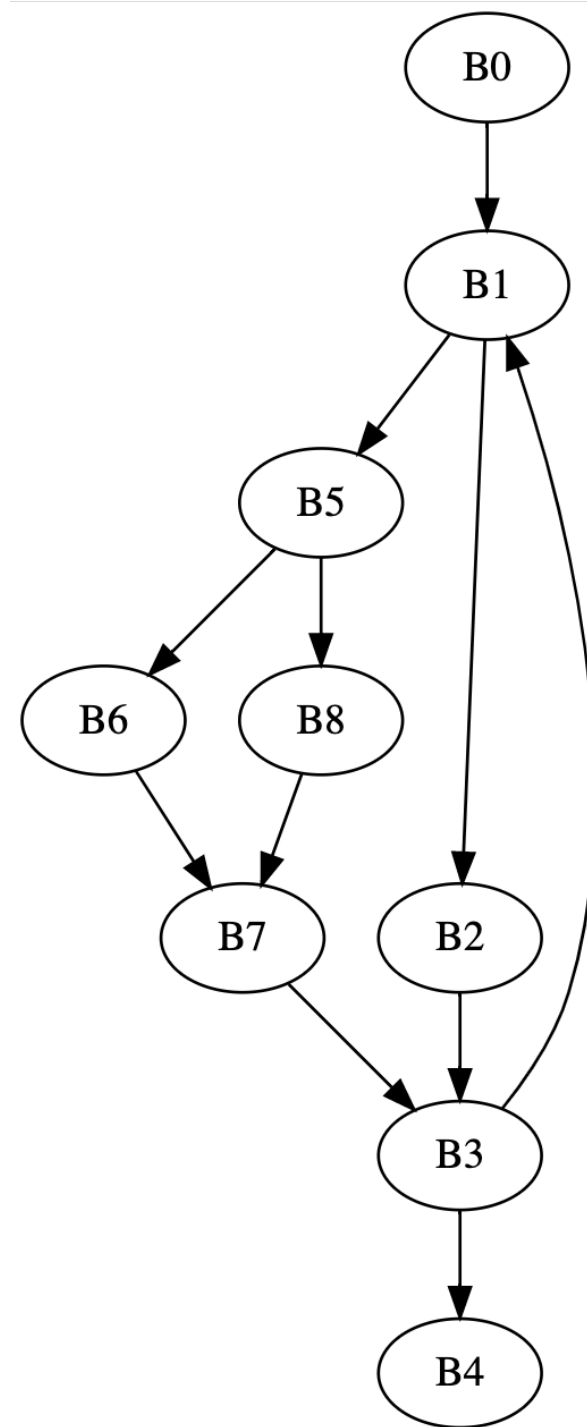
B5: a17 = ...;
    d18 = ...;
    br ... B6, B8;

B6: d19 = ...;

B7: d20 =  $\phi$ (...);
    c21 =  $\phi$ (...);
    b22 = ...;

B8: c23 = ...;
    br B7;

```



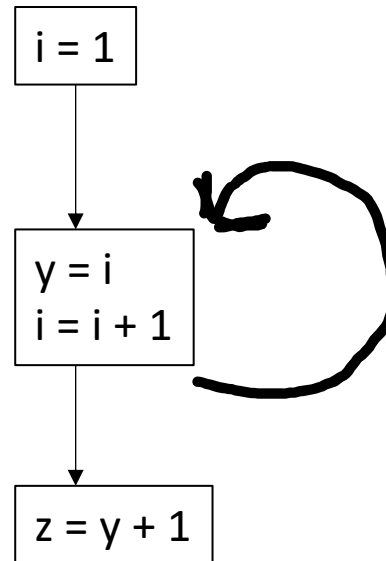
# How to convert back to 3 address code from SSA?

- Can a processor execute phi instructions?

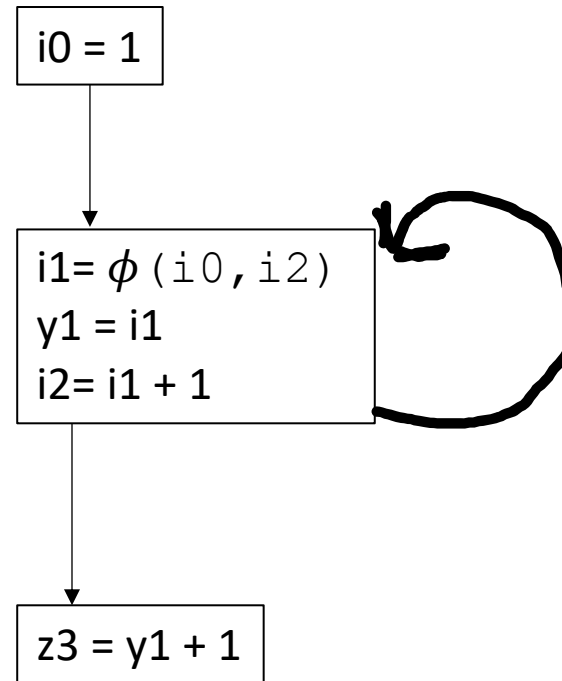
# How to convert back to 3 address code from SSA?

- Can a processor execute phi instructions?
- Just assign to the new variable in the parent?

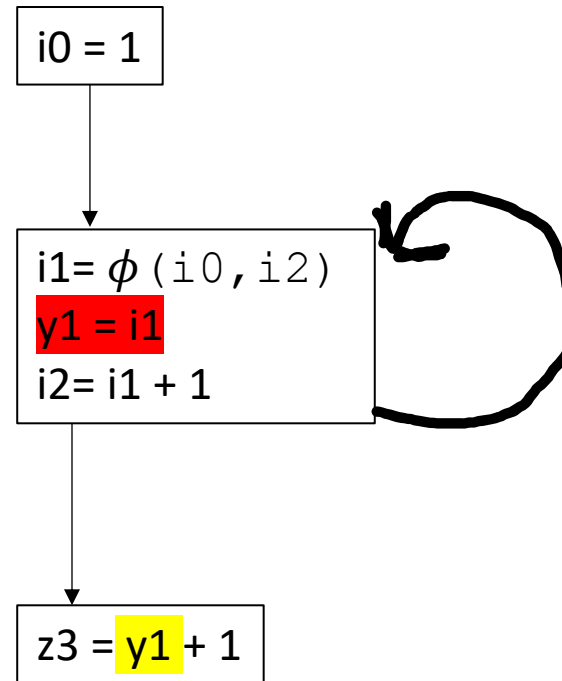
# Lost copy issue



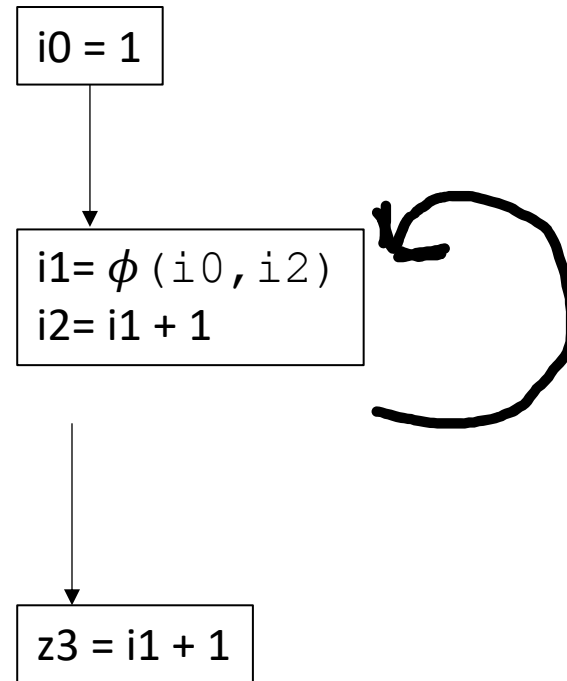
# Lost copy issue



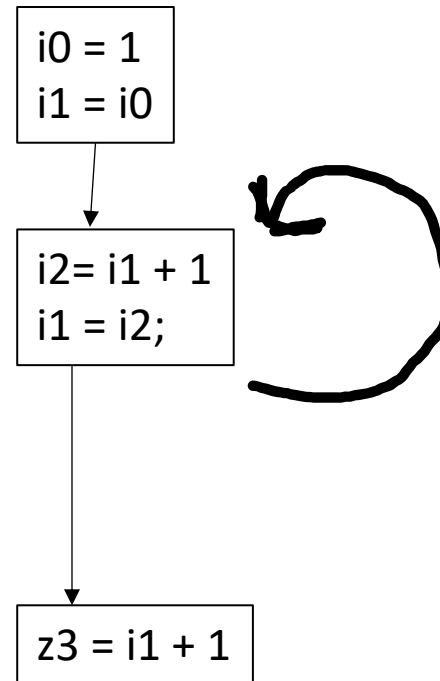
# Lost copy issue



# Lost copy issue

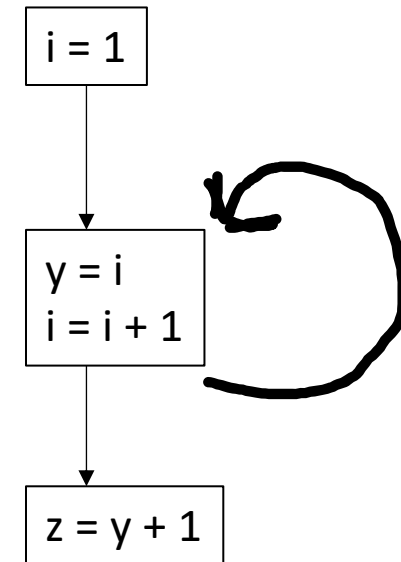
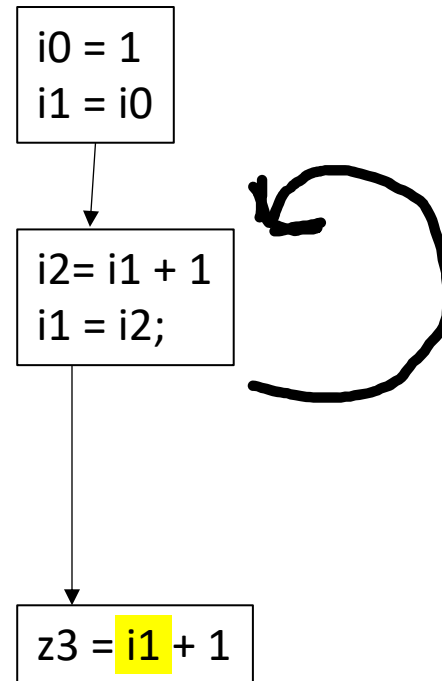


# Lost copy issue



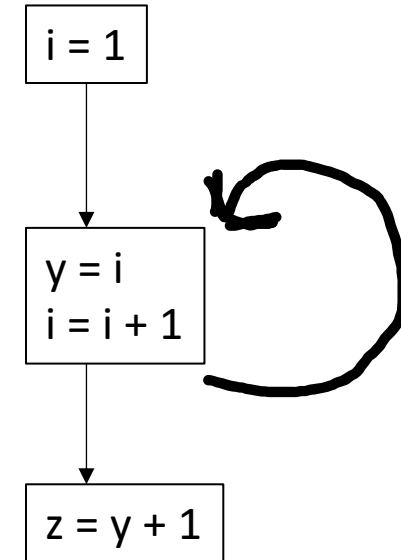
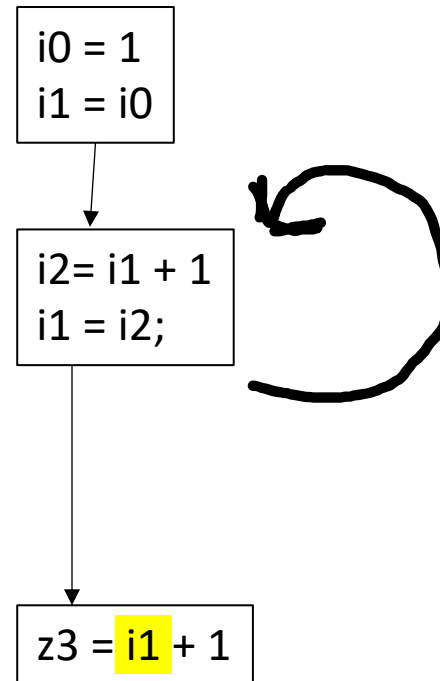


# Lost copy issue



# Lost copy issue

*Known as the lost-copy problem  
there are algorithms for handling this (see book)*



*Similar problem called the Swap problem*

# Let's back up

- Converting to SSA is difficult!
- Converting out of SSA is difficult!
- Why do we use SSA?

# Optimizations using SSA

# Constant Propagation

- Perform certain operations at compile time if the values are known
- Flow the information of known values throughout the program

# Constant Folding

If values are constant:

```
x = 128 * 2 * 5;
```

# Constant Folding

If values are constant:

```
x = 128 * 2 * 5;
```

```
x = 1280;
```

# Constant Folding

If values are constant:

```
x = 128 * 2 * 5;
```

```
x = 1280;
```

Using identities

```
x = z * 0;
```



# Constant Folding

If values are constant:

```
x = 128 * 2 * 5;
```

```
x = 1280;
```

Using identities

```
x = z * 0;
```

```
x = 0;
```

# Constant Folding

If values are constant:

```
x = 128 * 2 * 5;
```

```
x = 1280;
```

Using identities

```
x = z * 0;
```

```
x = 0;
```

Operations on other data structures

```
x = "CSE" + "211";
```

# Constant Folding

If values are constant:

```
x = 128 * 2 * 5;
```

```
x = 1280;
```

Using identities

```
x = z * 0;
```

```
x = 0;
```

Operations on other data structures

```
x = "CSE" + "211";
```

```
x = "CSE211";
```

*local to expressions!*

# Constant Propagation

multiple expressions:

```
x = 42;  
y = x + 5;
```

# Constant Propagation

multiple expressions:

```
x = 42;  
y = x + 5;
```

```
y = 47;
```

# Constant Propagation

multiple expressions:

```
x = 42;  
y = x + 5;
```

```
y = 47;
```

Within a basic block, you can use local value numbering

# Constant Propagation

multiple expressions:

```
x = 42;  
y = x + 5;
```

```
y = 47;
```

What about across basic blocks?

```
x = 42;  
z = 5;  
if (<some condition> {  
    y = 5;  
}  
else {  
    y = z;  
}  
w = y;
```

# To do this, we're going to use a lattice

- An object in abstract algebra
- Unique to each analysis you want to implement
  - Kind of like the flow function



# A simple lattice

- A set of symbols:  $\{c_1, c_2, c_3 \dots\}$
- Special symbols:
  - Top :  $\top$
  - Bottom :  $\perp$
- Meet operator:  $\wedge$

# A simple lattice

- A set of symbols:  $\{c_1, c_2, c_3 \dots\}$
- Special symbols:
  - Top :  $\top$
  - Bottom :  $\perp$
- Meet operator:  $\wedge$

Lattices are an abstract algebra construct, with a few properties:

$$\perp \wedge x = \perp$$

$$\top \wedge x = x$$

Where  $x$  is any symbol

# A simple lattice

- A set of symbols:  $\{c_1, c_2, c_3 \dots\}$
- Special symbols:
  - Top :  $\top$
  - Bottom :  $\perp$
- Meet operator:  $\wedge$

Lattices are an abstract algebra construct, with a few properties:

$$\perp \wedge x = \perp$$

$$\top \wedge x = x$$

Where  $x$  is any symbol

For each analysis, we get to define symbols and the meet operation over them.

# A simple lattice

- A set of symbols:  $\{c_1, c_2, c_3 \dots\}$
- Special symbols:
  - Top :  $\top$
  - Bottom :  $\perp$
- Meet operator:  $\wedge$

Lattices are an abstract algebra construct, with a few properties:

$$\perp \wedge x = \perp$$

$$\top \wedge x = x$$

Where  $x$  is any symbol

## **For constant propagation:**

take the symbols to be integers

Simple meet operations for integers:

if  $c_i \neq c_j$ :

$$c_i \wedge c_j = \perp$$

else:

$$c_i \wedge c_j = c_j$$

# Constant propagation

- Map each SSA variable  $x$  to a lattice value:
  - $\text{Value}(x) = \top$  if the analysis has not made a judgment
  - $\text{Value}(x) = c_i$  if the analysis found that variable  $x$  holds value  $c_i$
  - $\text{Value}(x) = \perp$  if the analysis has found that the value cannot be known

# Constant propagation algorithm

Initially:

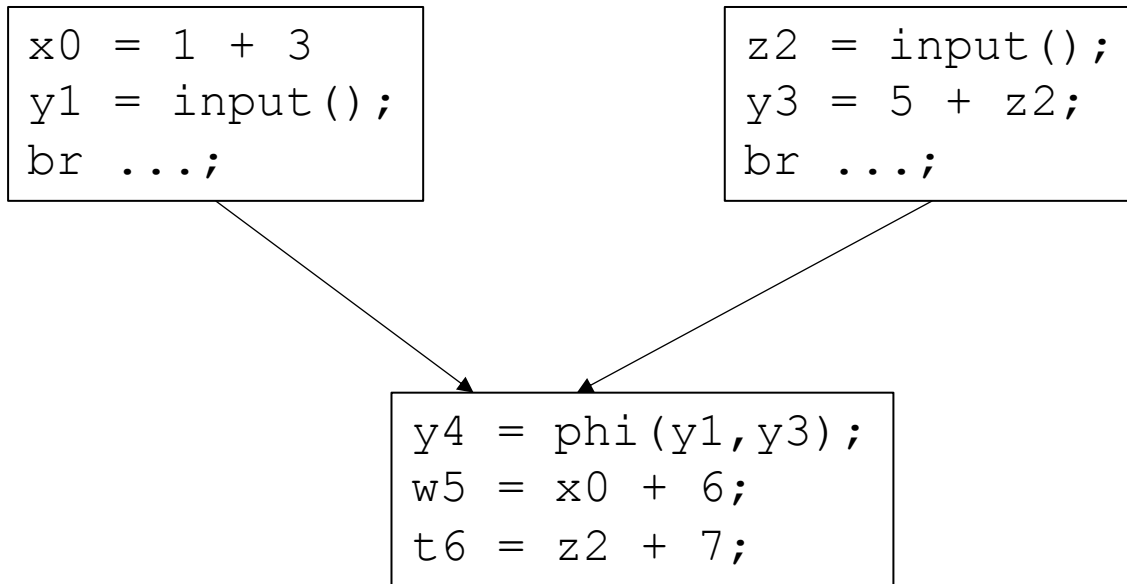
Assign each SSA variable a value  $c$  based on its expression:

- a constant  $c_i$  if the value can be known
- $\perp$  if the value comes from an argument or input
- $T$  otherwise, e.g. if the value comes from a  $\phi$  node

Then, create a “uses” map

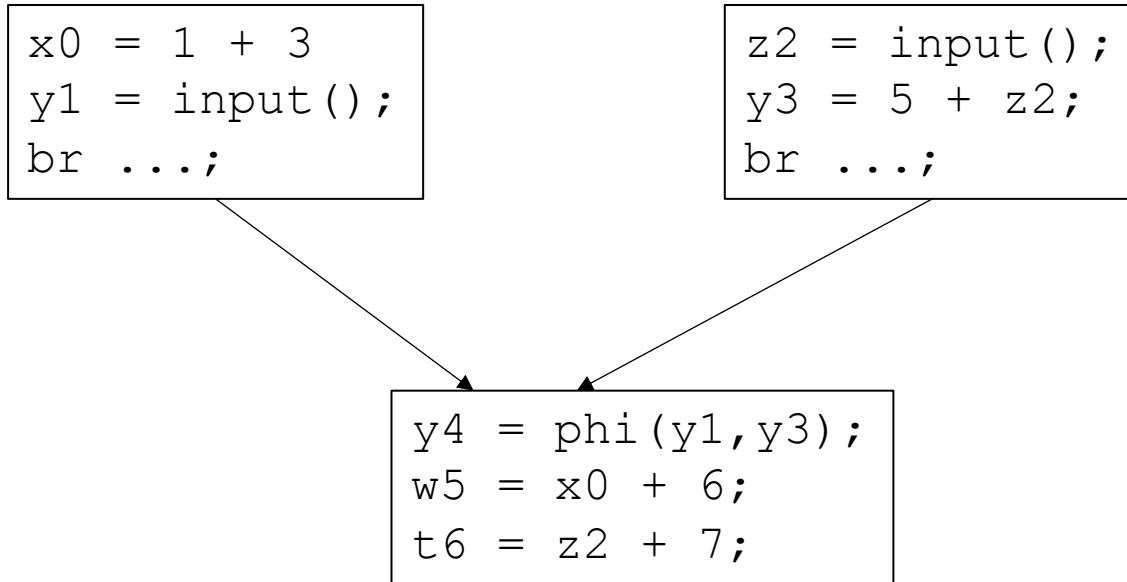
*This can be done in a single pass*

# Example:



```
Value {  
  x0 : 4  
  y1 : B  
  z2 : B  
  y3 : T  
  y4 : T  
  w5 : T  
  t6 : T  
}
```

# Example:



```
Value {  
  x0 : 4  
  y1 : B  
  z2 : B  
  y3 : T  
  y4 : T  
  w5 : T  
  t6 : T  
}
```

```
Uses {  
  x0 : [w5]  
  y1 : [y4]  
  z2 : [y3, t6]  
  y3 : [y4]  
  y4 : []  
  w5 : []  
  t6 : []  
}
```



# Constant propagation algorithm

worklist based algorithm:

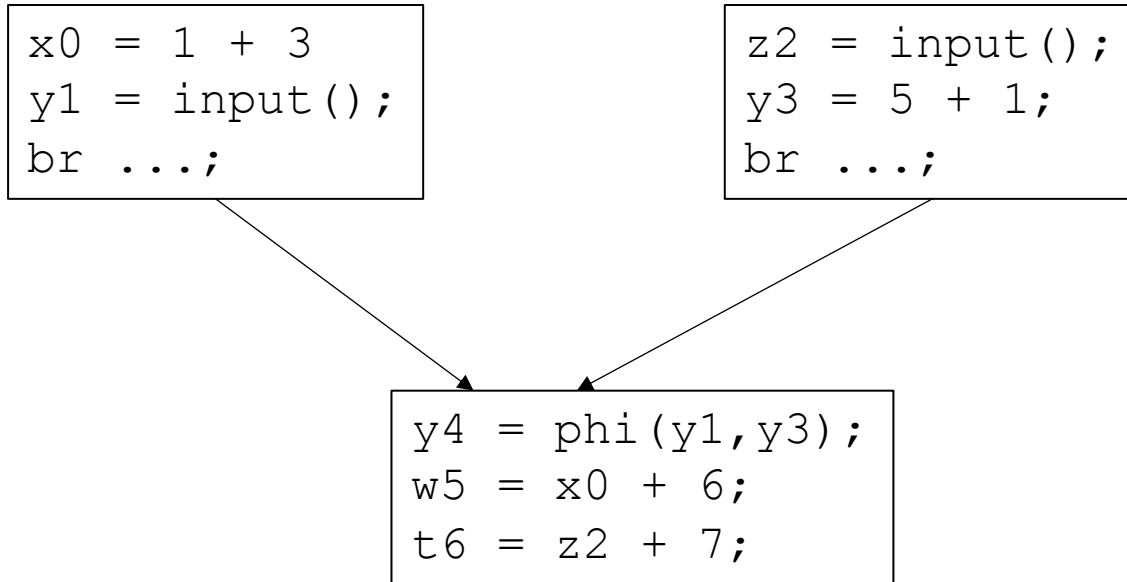
All variables **NOT** assigned to  $T$  get put on a worklist

iterate through the worklist:

For every item  $n$  in the worklist, we can look up the uses of  $n$

evaluate each use  $m$  over the lattice

# Example:



Worklist: `[x0, y1, z2, y3]`

```
Value {  
  x0 : 4  
  y1 : B  
  z2 : B  
  y3 : 6  
  y4 : T  
  w5 : T  
  t6 : T  
}
```

```
Uses {  
  x0 : [w5]  
  y1 : [y4]  
  z2 : [t6]  
  y3 : [y4]  
  y4 : []  
  w5 : []  
  t6 : []  
}
```

# Constant propagation algorithm

for each item in the worklist, evaluate all of its uses  $m$  over the lattice (unique to each optimization)

**Example:**  $m = n * x$

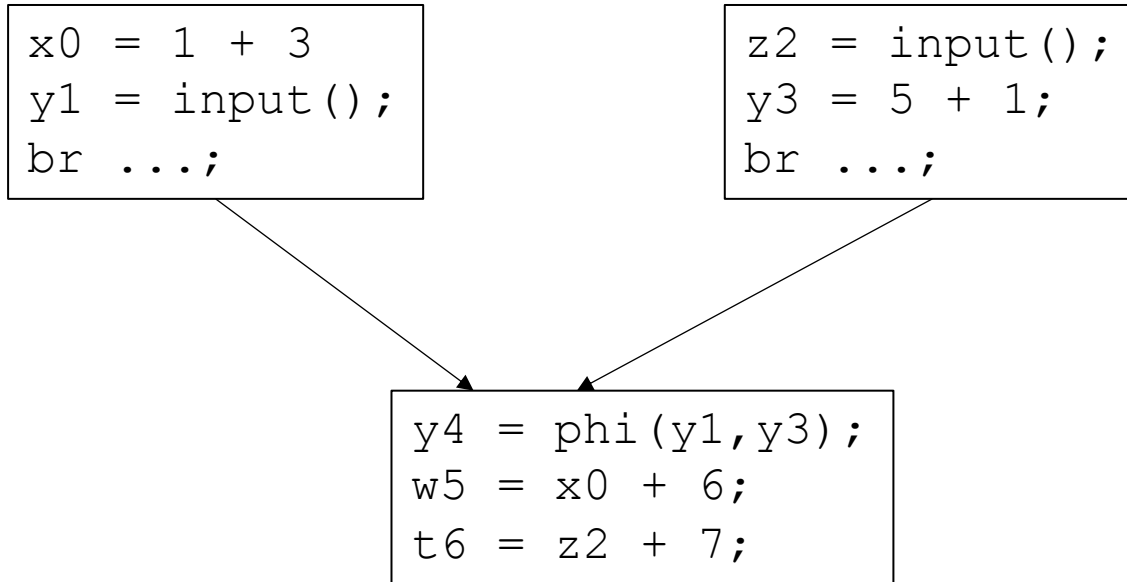
**if** (Value( $n$ ) is  $\perp$  or Value( $x$ ) is  $\perp$ )

Value( $m$ ) =  $\perp$ ;

Add  $m$  to the worklist if Value( $m$ ) has changed;

break;

# Example:

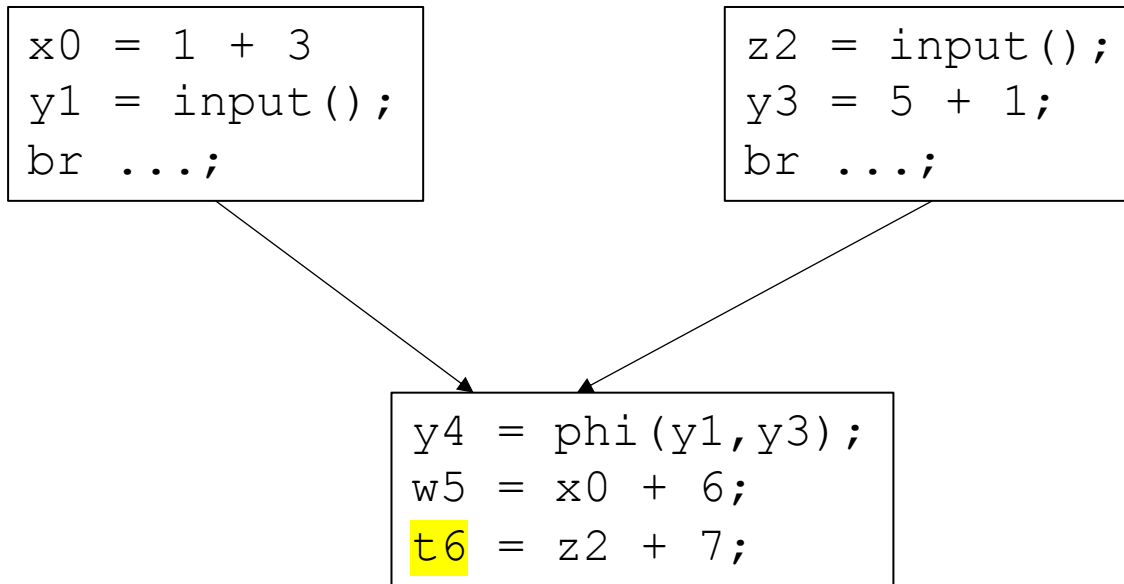


Worklist: [x0, y1, **z2**, y3]

```
Value {
  x0 : 4
  y1 : B
  z2 : B
  y3 : 6
  y4 : T
  w5 : T
  t6 : T
}
```

```
Uses {
  x0 : [w5]
  y1 : [y4]
  z2 : [t6]
  y3 : [y4]
  y4 : []
  w5 : []
  t6 : []
}
```

# Example:



Worklist: [x0, y1, **z2**, y3, t6]

```
Value {
  x0 : 4
  y1 : B
  z2 : B
  y3 : 6
  y4 : T
  w5 : T
  t6 : B
}
```

```
Uses {
  x0 : [w5]
  y1 : [y4]
  z2 : [t6]
  y3 : [y4]
  y4 : []
  w5 : []
  t6 : []
}
```

# Constant propagation algorithm

evaluate  $m$  over the lattice (unique to each optimization)

**Example:**  $m = n * x$

**if** (Value( $n$ ) is  $\perp$  or Value( $x$ ) is  $\perp$ )

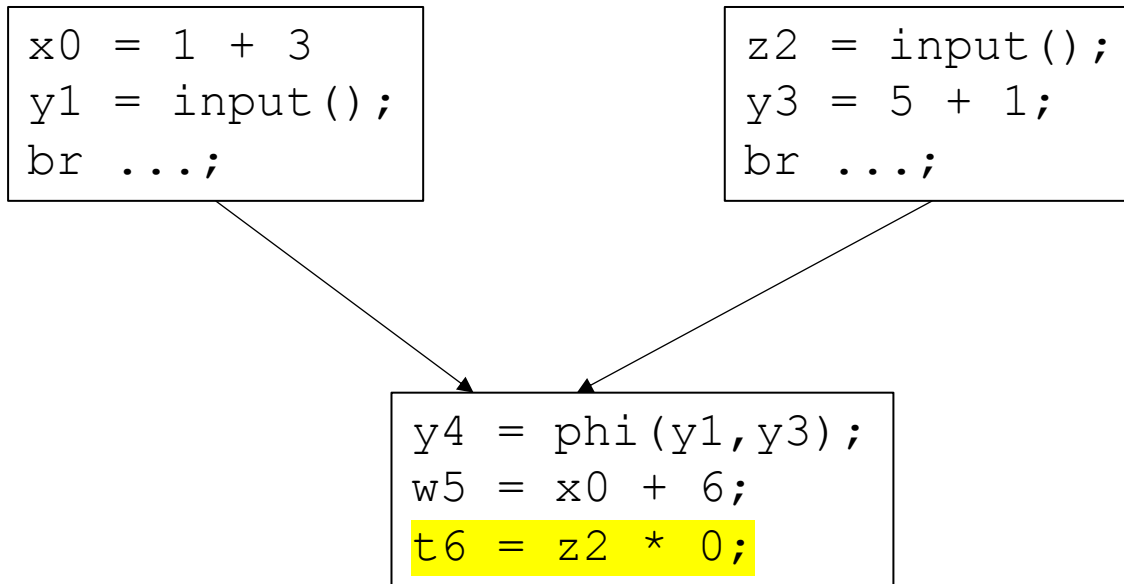
Value( $m$ ) =  $\perp$ ;

Add  $m$  to the worklist if Value( $m$ ) has changed;

break;

*Can we optimize this for special cases?*

# Example:

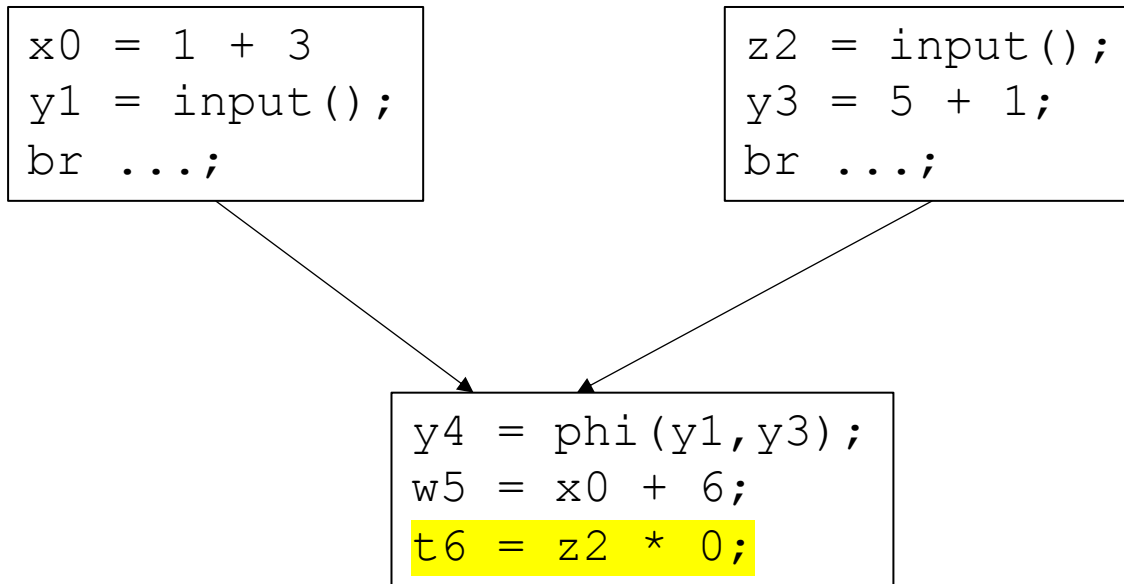


Worklist: [x0, y1, **z2**, y3]

```
Value {  
  x0 : 4  
  y1 : B  
  z2 : B  
  y3 : 6  
  y4 : T  
  w5 : T  
  t6 : T  
}
```

```
Uses {  
  x0 : [w5]  
  y1 : [y4]  
  z2 : [t6]  
  y3 : [y4]  
  y4 : []  
  w5 : []  
  t6 : []  
}
```

# Example:



Worklist: [x0, y1, **z2**, y3]

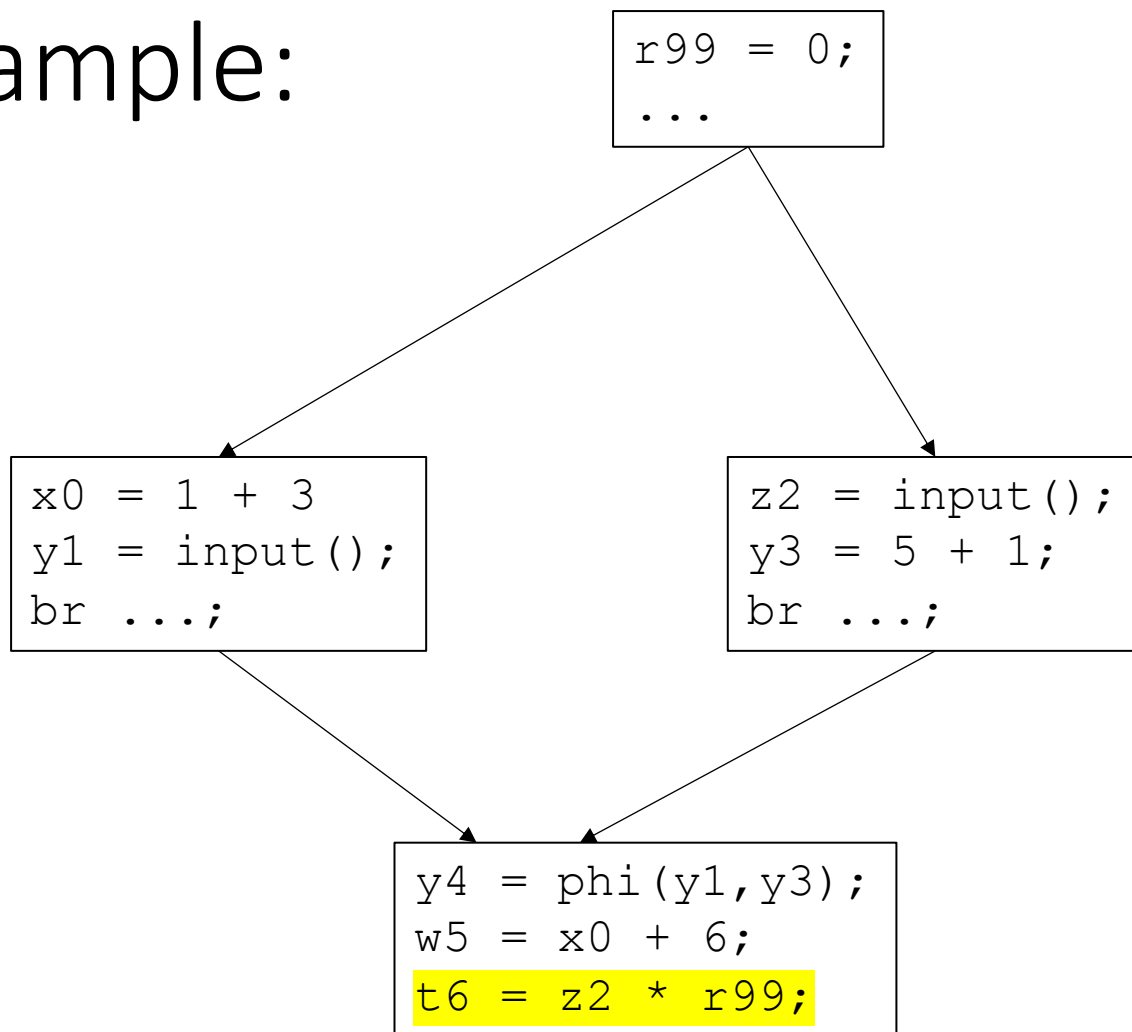
Can't this be done  
at the expression level?

```
Value {
  x0 : 4
  y1 : B
  z2 : B
  y3 : 6
  y4 : T
  w5 : T
  t6 : T
}
```

```
Uses {
  x0 : [w5]
  y1 : [y4]
  z2 : [t6]
  y3 : [y4]
  y4 : []
  w5 : []
  t6 : []
}
```



# Example:



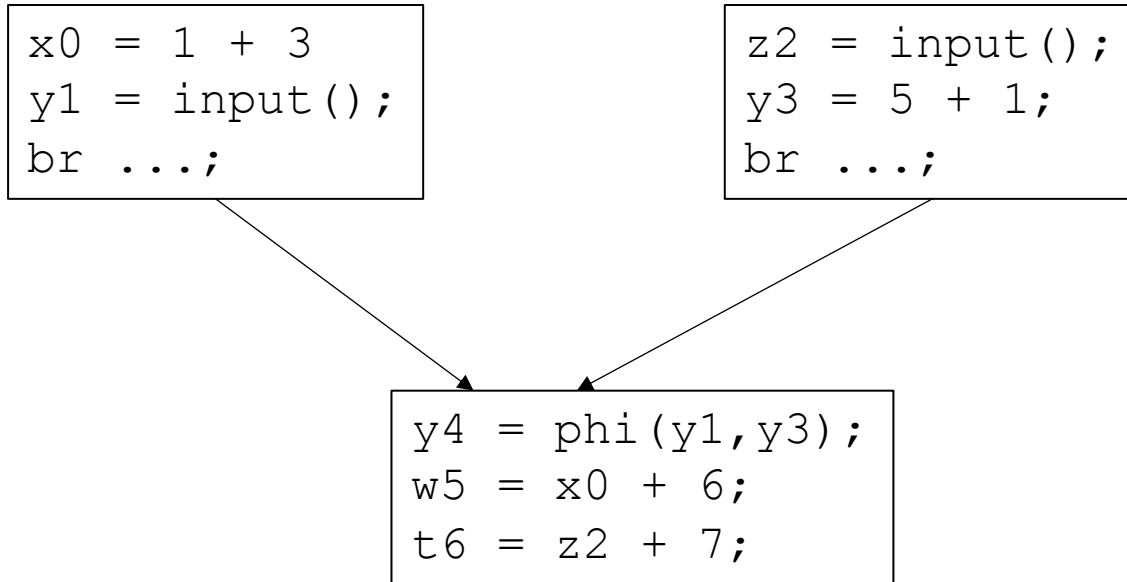
Worklist: [x0, y1, z2, y3]

Can't this be done  
at the expression level?

```
Value {  
  x0 : 4  
  y1 : B  
  z2 : B  
  y3 : 6  
  y4 : T  
  w5 : T  
  t6 : T  
  r99 : 0  
}
```

```
Uses {  
  x0 : [w5]  
  y1 : [y4]  
  z2 : [t6]  
  y3 : [y4]  
  y4 : []  
  w5 : []  
  t6 : []  
}
```

# Example:



Worklist: [x0, y1, z2, y3]

```
Value {  
  x0 : 4  
  y1 : B  
  z2 : B  
  y3 : 6  
  y4 : T  
  w5 : T  
  t6 : T  
}
```

```
Uses {  
  x0 : [w5]  
  y1 : [y4]  
  z2 : [t6]  
  y3 : [y4]  
  y4 : []  
  w5 : []  
  t6 : []  
}
```

# Constant propagation algorithm

evaluate  $m$  over the lattice (unique to each optimization)

**Example:**  $m = n * x$

*// continued from previous slide*

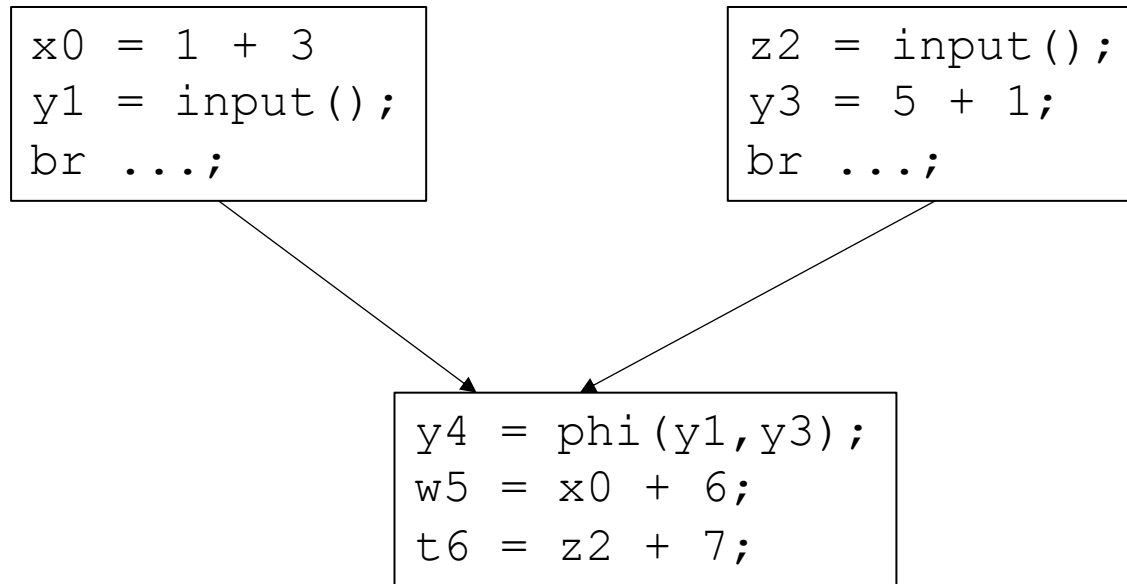
**if** (Value( $n$ ) has a value and Value( $x$ ) has a value)

Value( $m$ ) = **evaluate**(Value( $n$ ), Value( $x$ ));

Add  $m$  to the worklist if Value( $m$ ) has changed;

break;

# Example:

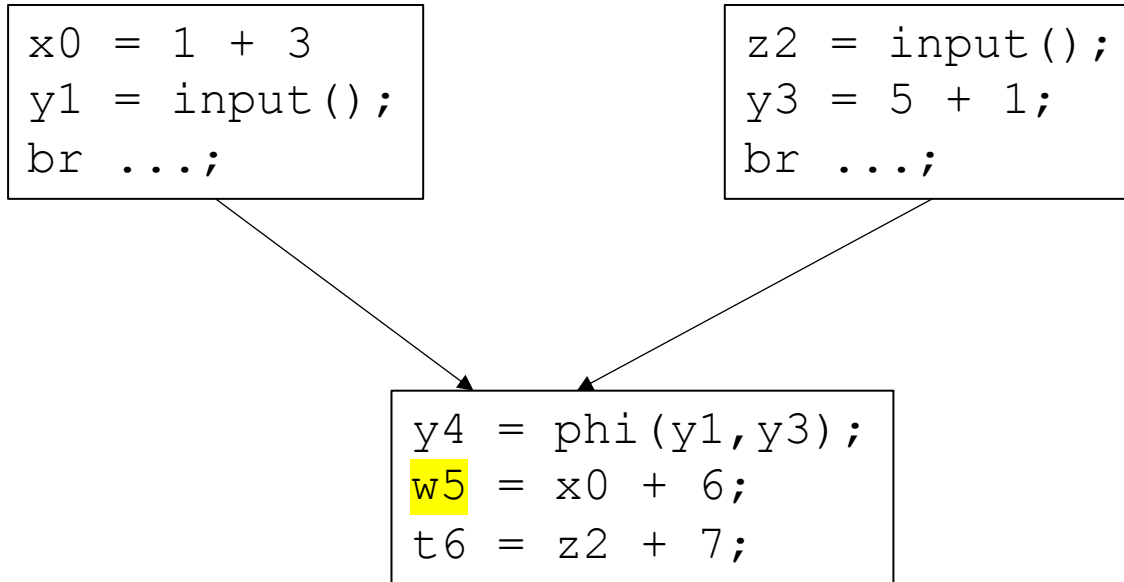


Worklist: [**x0**, y1, y3, w5]

```
Value {  
  x0 : 4  
  y1 : B  
  z2 : B  
  y3 : 6  
  y4 : T  
  w5 : T  
  t6 : T  
}
```

```
Uses {  
  x0 : [w5]  
  y1 : [y4]  
  z2 : [t6]  
  y3 : [y4]  
  y4 : []  
  w5 : []  
  t6 : []  
}
```

# Example:

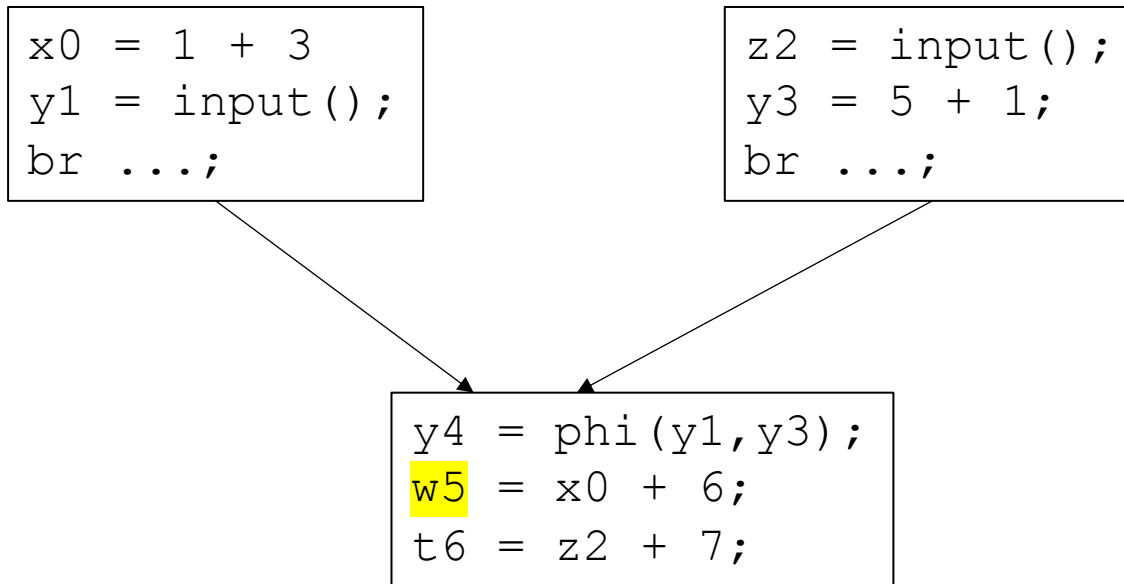


Worklist: [**x0**, y1, y3]

```
Value {  
  x0 : 4  
  y1 : B  
  z2 : B  
  y3 : 6  
  y4 : T  
  w5 : T  
  t6 : T  
}
```

```
Uses {  
  x0 : [w5]  
  y1 : [y4]  
  z2 : [t6]  
  y3 : [y4]  
  y4 : []  
  w5 : []  
  t6 : []  
}
```

# Example:



Worklist: [**x0**, y1, y3]

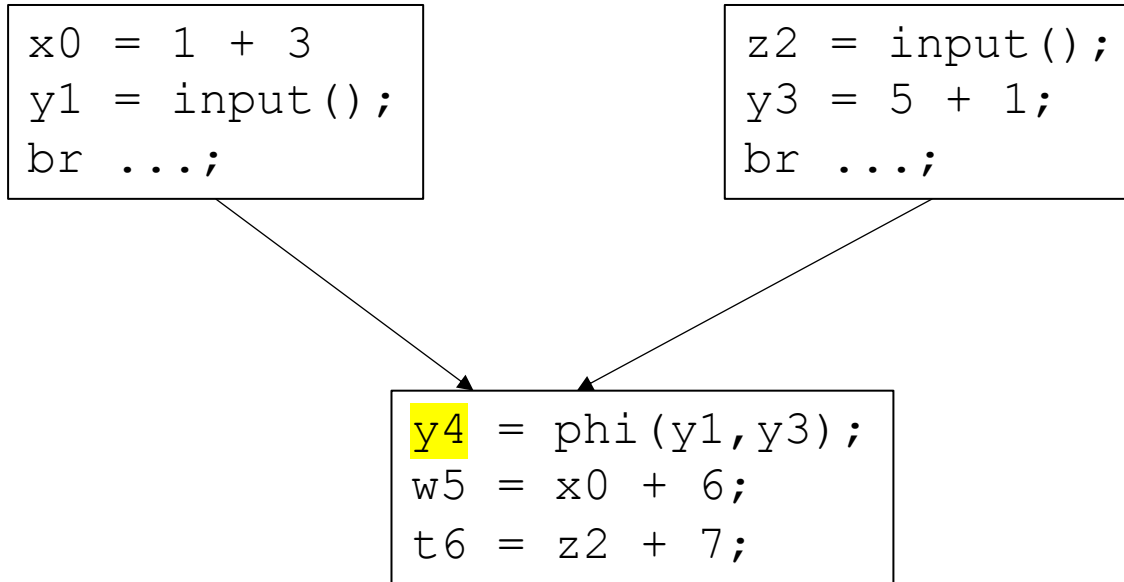
```
Value {
  x0 : 4
  y1 : B
  z2 : B
  y3 : 6
  y4 : T
  w5 : 10
  t6 : T
}
```

```
Uses {
  x0 : [w5]
  y1 : [y4]
  z2 : [t6]
  y3 : [y4]
  y4 : []
  w5 : []
  t6 : []
}
```

# The elephant in the room

...

# Example:



Worklist: [x0, y1, y3]

```
Value {  
  x0 : 4  
  y1 : B  
  z2 : B  
  y3 : 6  
  y4 : T  
  w5 : T  
  t6 : T  
}
```

```
Uses {  
  x0 : [w5]  
  y1 : [y4]  
  z2 : [t6]  
  y3 : [y4]  
  y4 : []  
  w5 : []  
  t6 : []  
}
```



# Constant propagation algorithm

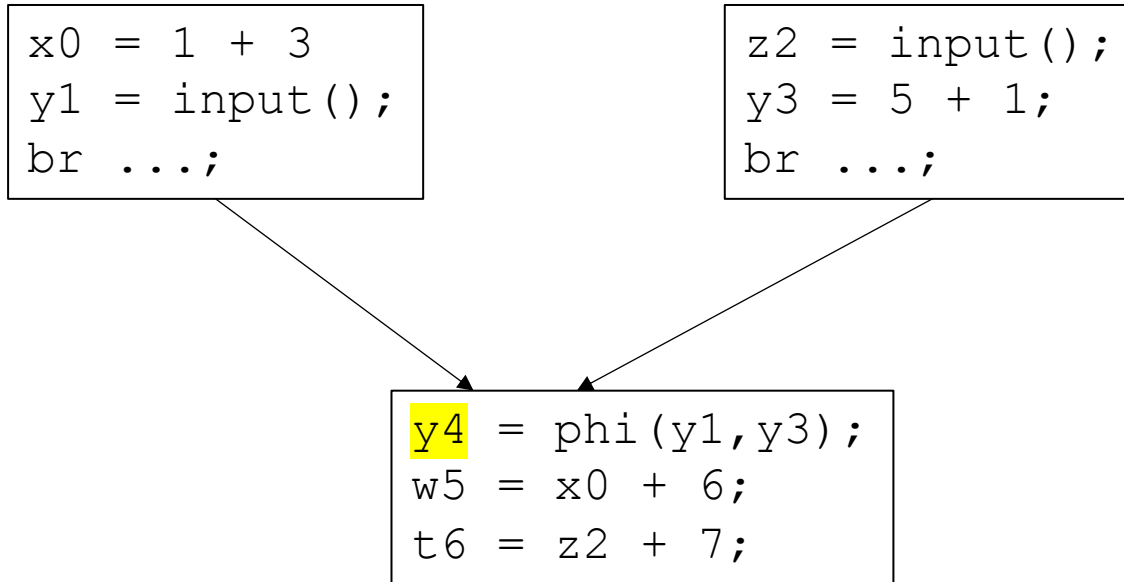
evaluate  $m$  over the lattice:

**Example:**  $m = \phi(x_1, x_2)$

$$\text{Value}(m) = x_1 \wedge x_2$$

if  $\text{Value}(m)$  is not  $\top$  and  $\text{Value}(m)$  has changed, then add  $m$  to the worklist

# Example:

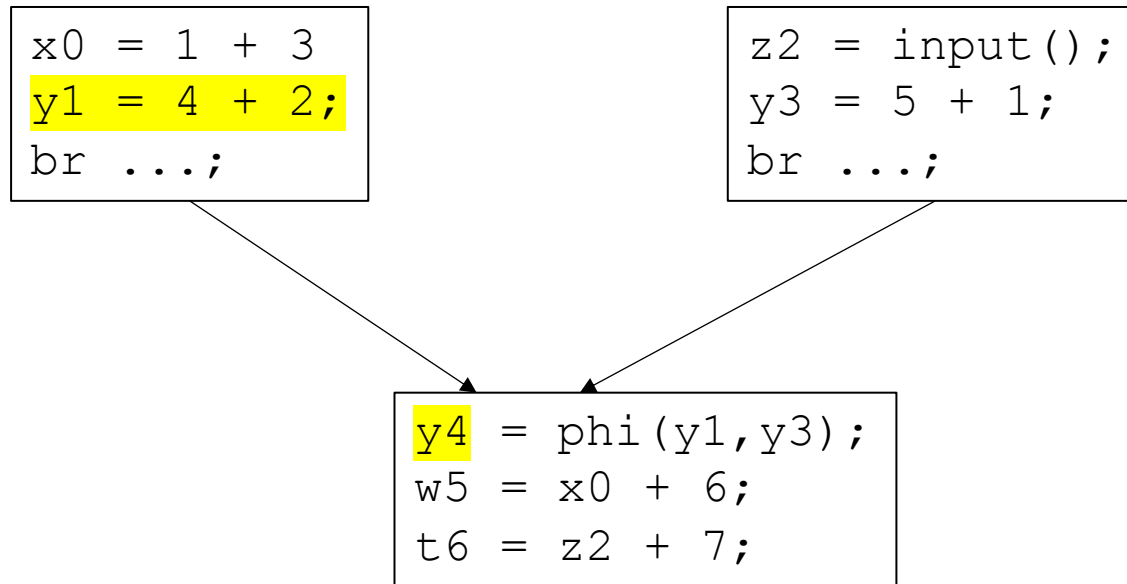


Worklist: [x0, y1, y3]

```
Value {
  x0 : 4
  y1 : B
  z2 : B
  y3 : 6
  y4 : B
  w5 : T
  t6 : T
}
```

```
Uses {
  x0 : [w5]
  y1 : [y4]
  z2 : [t6]
  y3 : [y4]
  y4 : []
  w5 : []
  t6 : []
}
```

# Example:



Worklist: [x0, y1, y3]

```
Value {  
  x0 : 4  
  y1 : B  
  z2 : B  
  y3 : 6  
  y4 : T  
  w5 : T  
  t6 : T  
}
```

```
Uses {  
  x0 : [w5]  
  y1 : [y4]  
  z2 : [t6]  
  y3 : [y4]  
  y4 : []  
  w5 : []  
  t6 : []  
}
```

# Constant propagation algorithm

evaluate  $m$  over the lattice:

**Example:**  $m = \phi(x_1, x_2)$

$$\text{Value}(m) = x_1 \wedge x_2$$

if  $\text{Value}(m)$  is not  $\top$  and  $\text{Value}(m)$  has changed, then add  $m$  to the worklist

# Constant propagation algorithm

evaluate  $m$  over the lattice:

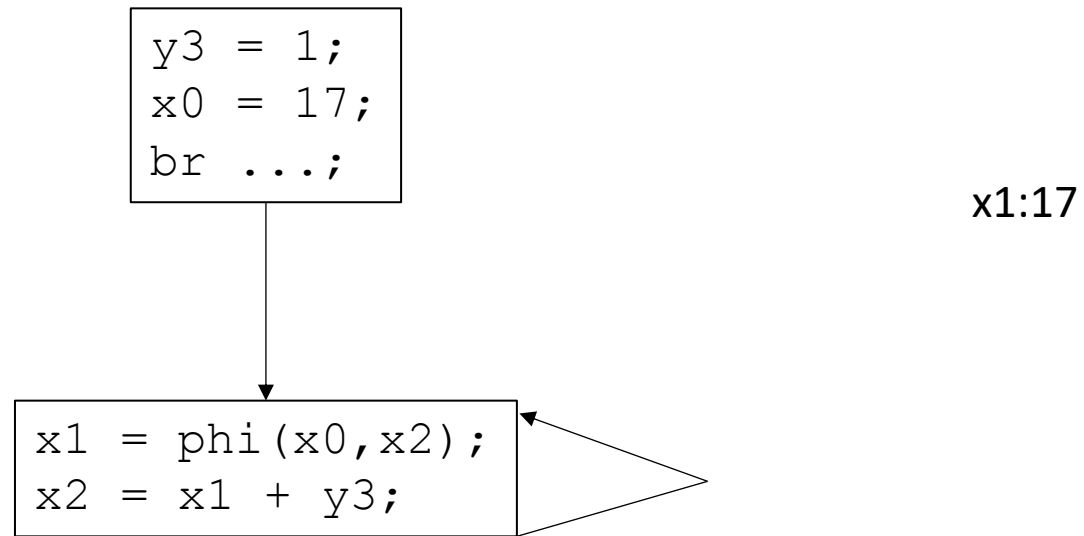
**Example:**  $m = \phi(x_1, x_2)$

$\text{Value}(m) = x_1 \wedge x_2$

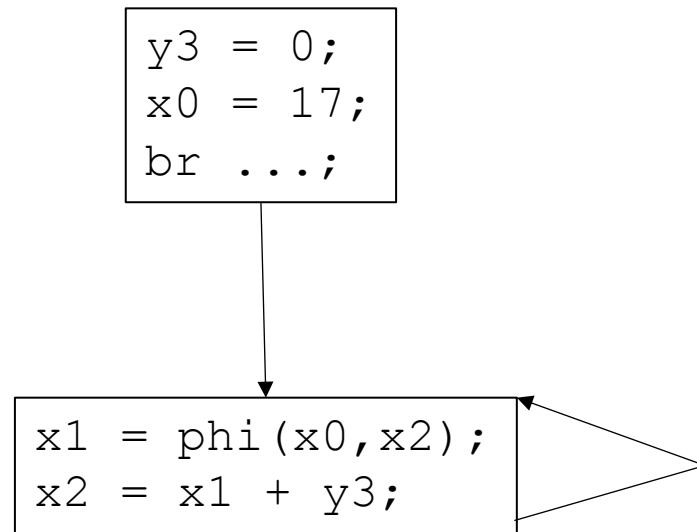
if  $\text{Value}(m)$  is not  $\top$  and  $\text{Value}(m)$  has changed, then add  $m$  to the worklist

Issue here:  
potentially assigning  
a value that might  
not hold

# Example loop:



# Example loop:



*optimistic analysis: Assume unknowns  
will be the target value for the optimization.  
Correct later*

*pessimistic analysis: Assume unknowns will NOT  
be the target value for the optimization.*

*Pros/cons?*

# A simple lattice

- A set of symbols:  $\{c_1, c_2, c_3 \dots\}$
- Special symbols:
  - Top :  $\top$
  - Bottom :  $\perp$
- Meet operator:  $\wedge$

Lattices are an abstract algebra construct, with a few properties:

$$\perp \wedge x = \perp$$

$$\top \wedge x = x$$

Where  $x$  is any symbol

## For Loop unrolling

take the symbols to be integers

Simple meet operations for integers:

if  $c_i \neq c_j$ :

$$c_i \wedge c_j = \perp$$

else:

$$c_i \wedge c_j = c_j$$



# A simple lattice

- A set of symbols:  $\{c_1, c_2, c_3 \dots\}$
- Special symbols:
  - Top :  $\top$
  - Bottom :  $\perp$
- Meet operator:  $\wedge$

Lattices are an abstract algebra construct, with a few properties:

$$\perp \wedge x = \perp$$

$$\top \wedge x = x$$

Where  $x$  is any symbol

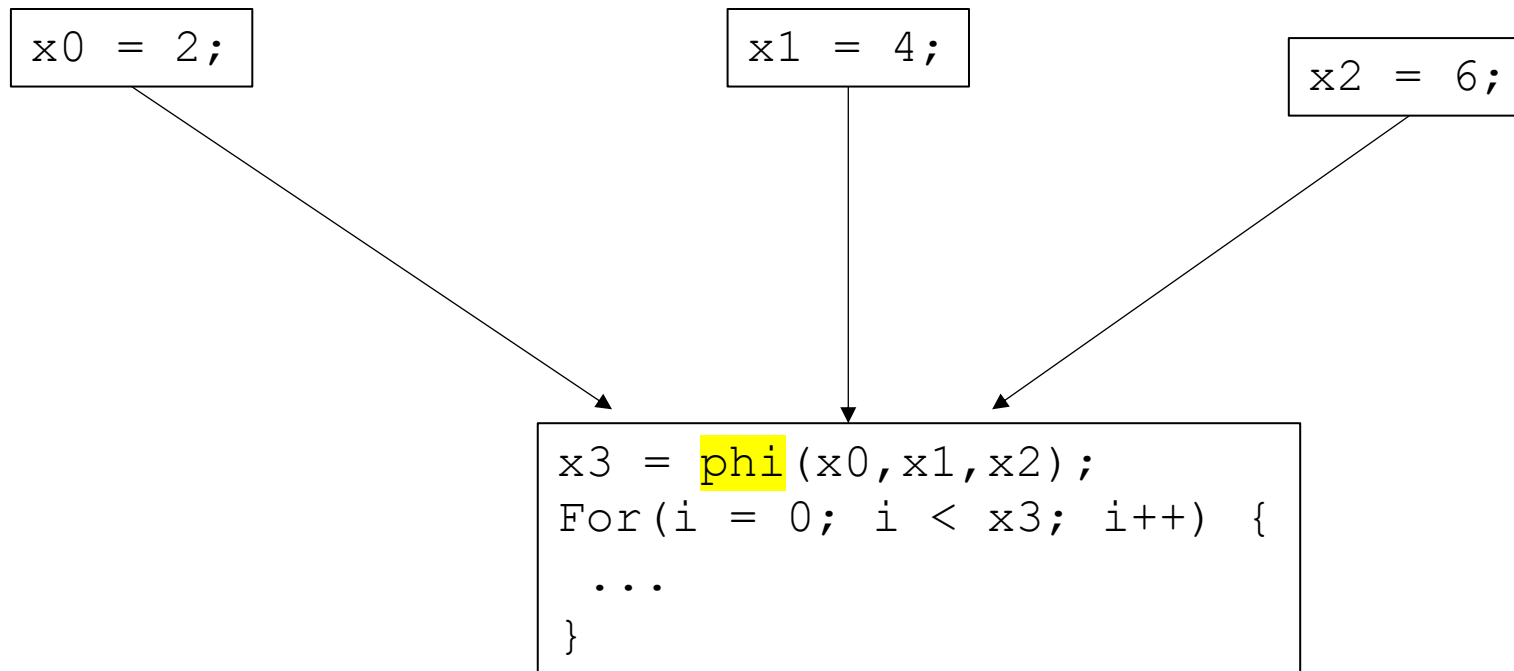
## For Loop unrolling

take the symbols to be integers  
representing the GCD

$$c_i \wedge c_j = \text{GCD}(c_i, c_j)$$

# Another lattice

- Given loop code:
  - Is it possible to unroll the loop N times?



# Another lattice

- Value ranges

*Track if  $i, j, k$  are guaranteed to be between 0 and 1024.*

*Meet operator takes a union of possible ranges.*

```
int * x = int[1024];  
x[i] = x[j] + x[k];
```

# Have a nice weekend!

- See you in office hours or in a week!