CSE211: Compiler Design

Nov. 6, 2023

- **Topic**:
  - Converting out of SSA
  - An SSA optimization

- **Questions**:
  - *Can a processor execute an SSA program?*
  - *How can you convert a program into SSA form?*
  - *How can you convert a program back from SSA form*

```
3:                                                 ; preds = %1
  %4 = tail call i32 @_Z14first_functionv(), !dbg !19
  call void @llvm.dbg.value(metadata i32 %4, metadata !14, metadata
  br label %7, !dbg !21

5:                                                 ; preds = %1
  %6 = tail call i32 @_Z15second_functionv(), !dbg !22
  call void @llvm.dbg.value(metadata i32 %6, metadata !14, metadata
  br label %7

7:                                                 ; preds = %5, %3
  %8 = phi i32 [ %4, %3 ], [ %6, %5 ], !dbg !24
  call void @llvm.dbg.value(metadata i32 %8, metadata !14, metadata
  ret i32 %8, !dbg !25
}
```

Announcements

- Homework 2 is out
  - Due Nov. 13
  - Work on the assignment!
- Homework 3 will be released on the 13th
- Last lecture in module 2
  - Then we move on to parallelism

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!
  - Remember, there are examples on the webpage of previous year’s courses!

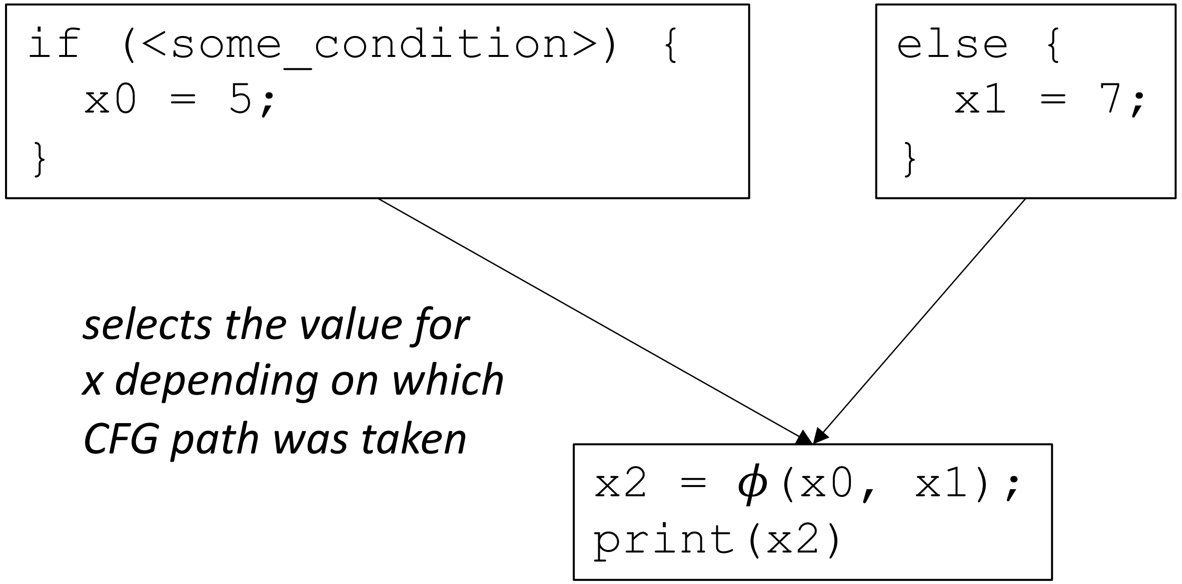
Review converting into SSA

ϕ instructions

- Example: how to convert this code into SSA?

```
int x;

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

else {
  x1 = 7;
}

x2 = ϕ(x0, x1);
print(x2)
```

number the variables



Conversion into SSA

Different algorithms depending on how many ϕ instructions

The fewer ϕ instructions, the more efficient analysis will be

Two phases:

- inserting ϕ instructions
- variable naming

Maximal SSA

Straightforward:

- For each variable, for each basic block: insert a ϕ instruction with placeholders for arguments

- local numbering for each variable using a global counter

- instantiate ϕ arguments

Maximal SSA

Example

```
x = 1;
y = 2;

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

else {
  x = 6;
  y = 100;
}

print(x)
```

Insert ϕ with argument placeholders

```
x = 1;
y = 2;

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

else {
  x = ϕ(...);
  y = ϕ(...);
  x = 6;
  y = 100;
}

x = ϕ(...);
y = ϕ(...);
print(x)
```

Rename variables iterate through basic blocks with a global counter

```
x0 = 1;
y1 = 2;

if (<condition>) {
  x3 = ϕ(...);
  y4 = ϕ(...);
  x5 = y4;
}

else {
  x6 = ϕ(...);
  y7 = ϕ(...);
  x8 = 6;
  y9 = 100;
}

x10 = ϕ(...);
y11 = ϕ(...);
print(x10)
```

fill in ϕ arguments by considering CFG

```
x0 = 1;
y1 = 2;

if (<condition>) {
  x3 = ϕ(x0);
  y4 = ϕ(y1);
  x5 = y4;
}

else {
  x6 = ϕ(x0);
  y7 = ϕ(y1);
  x8 = 6;
  y9 = 100;
}

x10 = ϕ(x5,x8);
y11 = ϕ(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 = ϕ(x0);
  y4 = ϕ(y1);
  x5 = y4;
}

else {
  x6 = ϕ(x0);
  y7 = ϕ(y1);
  x8 = 6;
  y9 = 100;
}

x10 = ϕ(x5,x8);
y11 = ϕ(y4,y9);
print(x10)
```

Hand Optimized SSA

```
x0 = 1;
y1 = 2;

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

else {
  x8 = 6;
  y9 = 100;
}

x10 = ϕ(x5,x8);
y11 = ϕ(y1,y9);
print(x10)
```

A note on SSA variants:

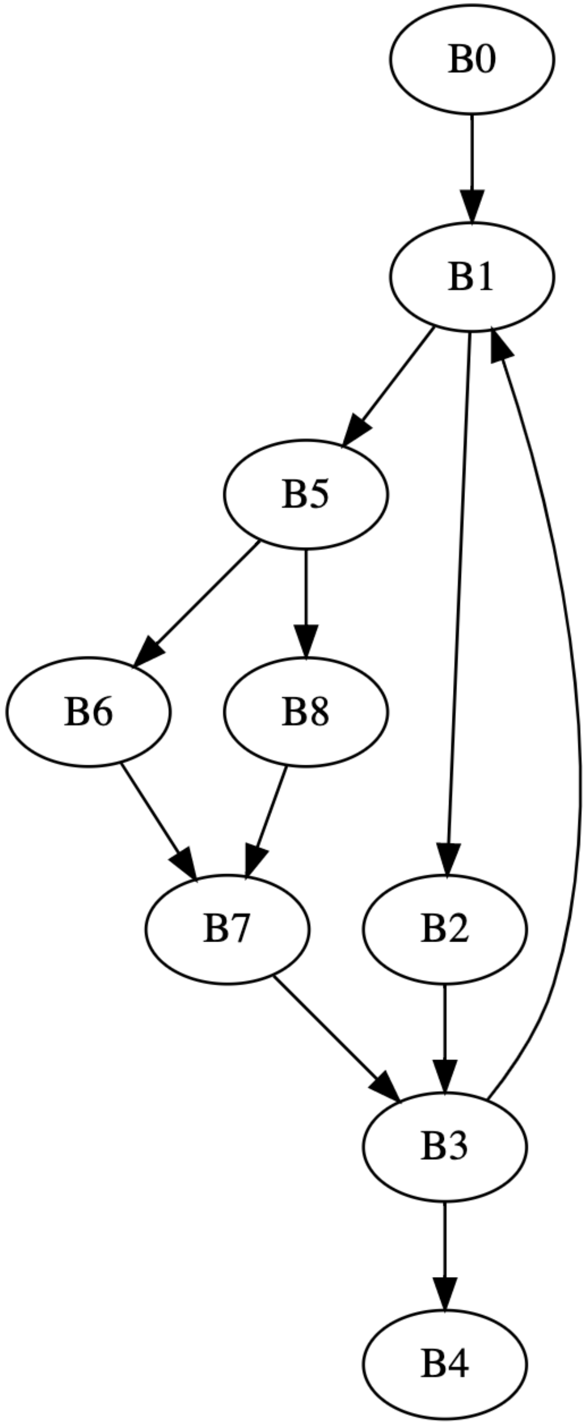
- EAC book describes:
  - Minimal SSA
  - Pruned SSA
  - **Semipruned SSA: We will discuss this one**

Dominance frontier

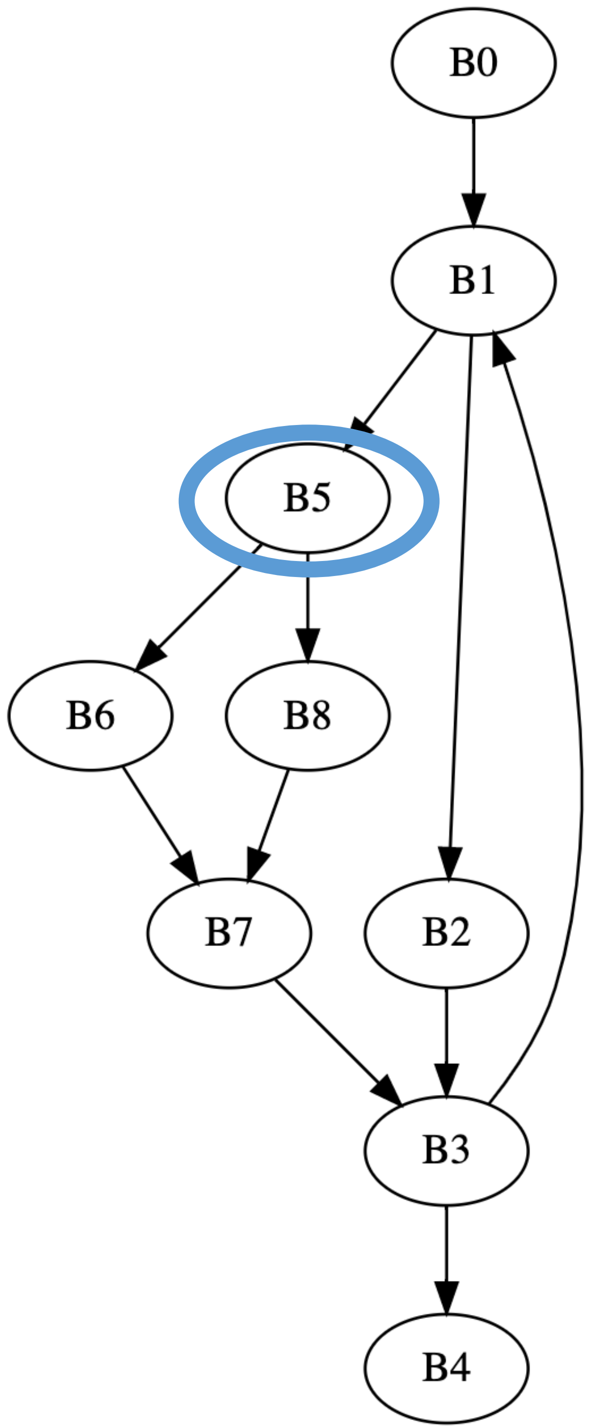
- a viz using coloring (thanks to Chris Liu!)
- Efficient algorithm for computing in EAC section 9.3.2 using a dominator tree.

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

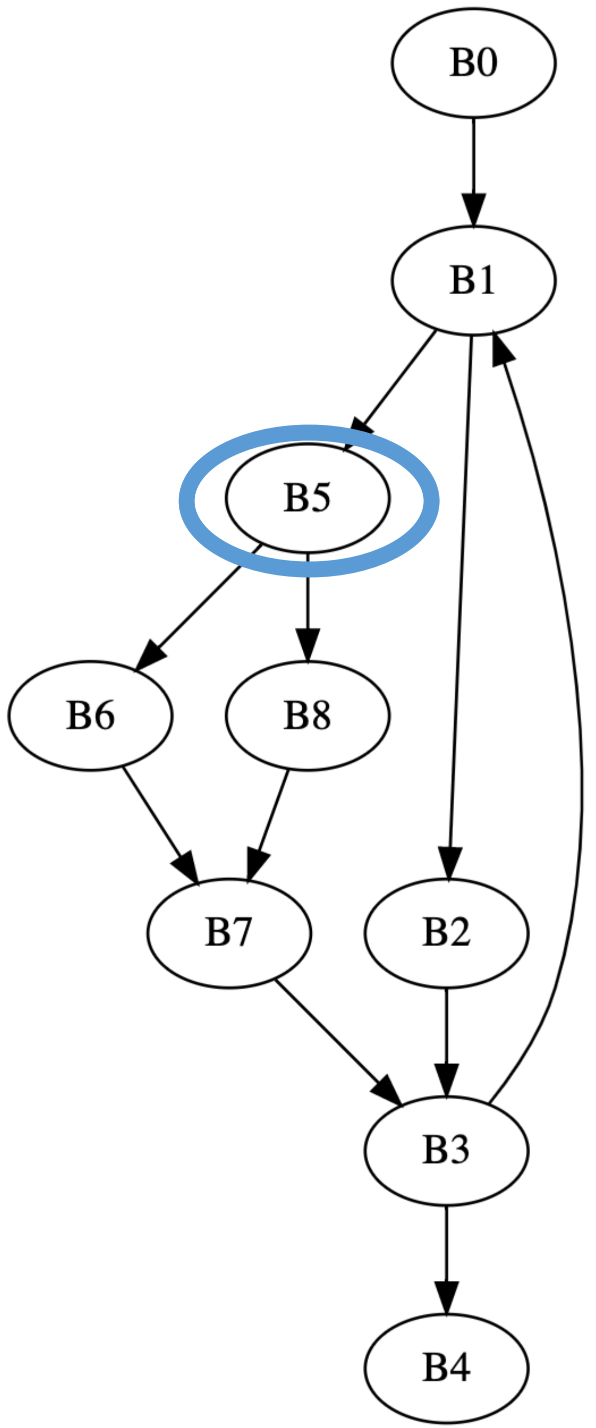
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



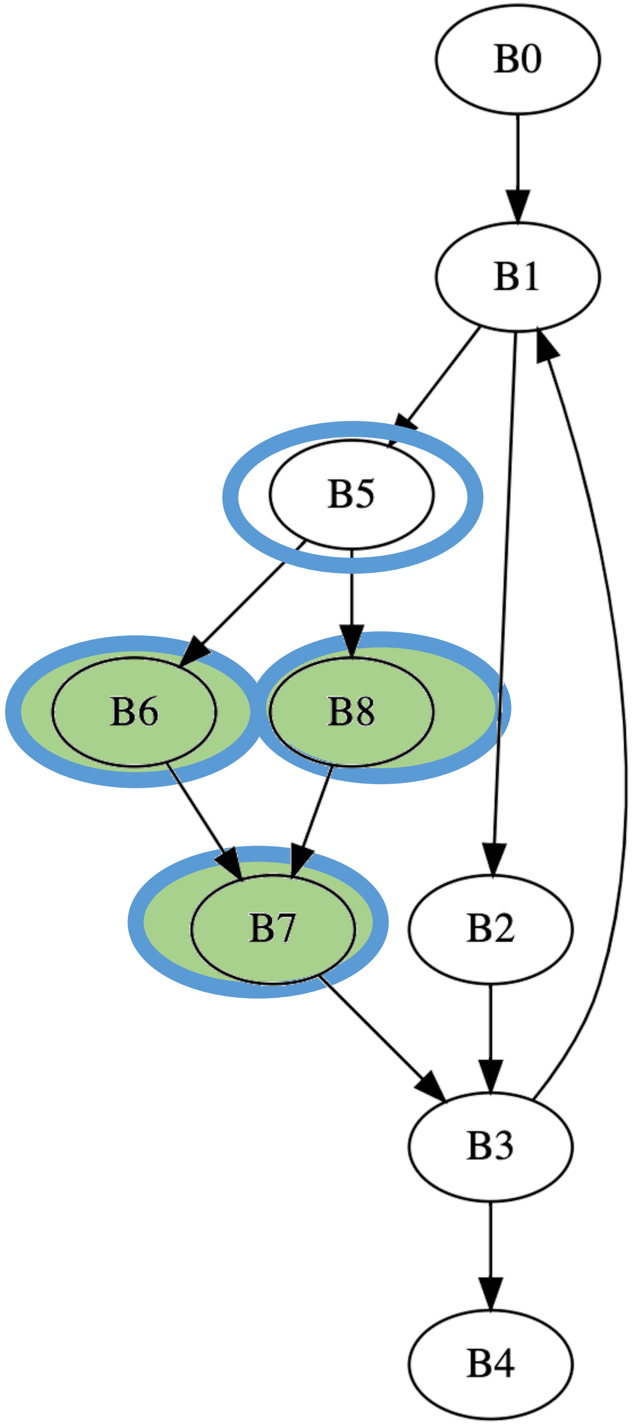
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



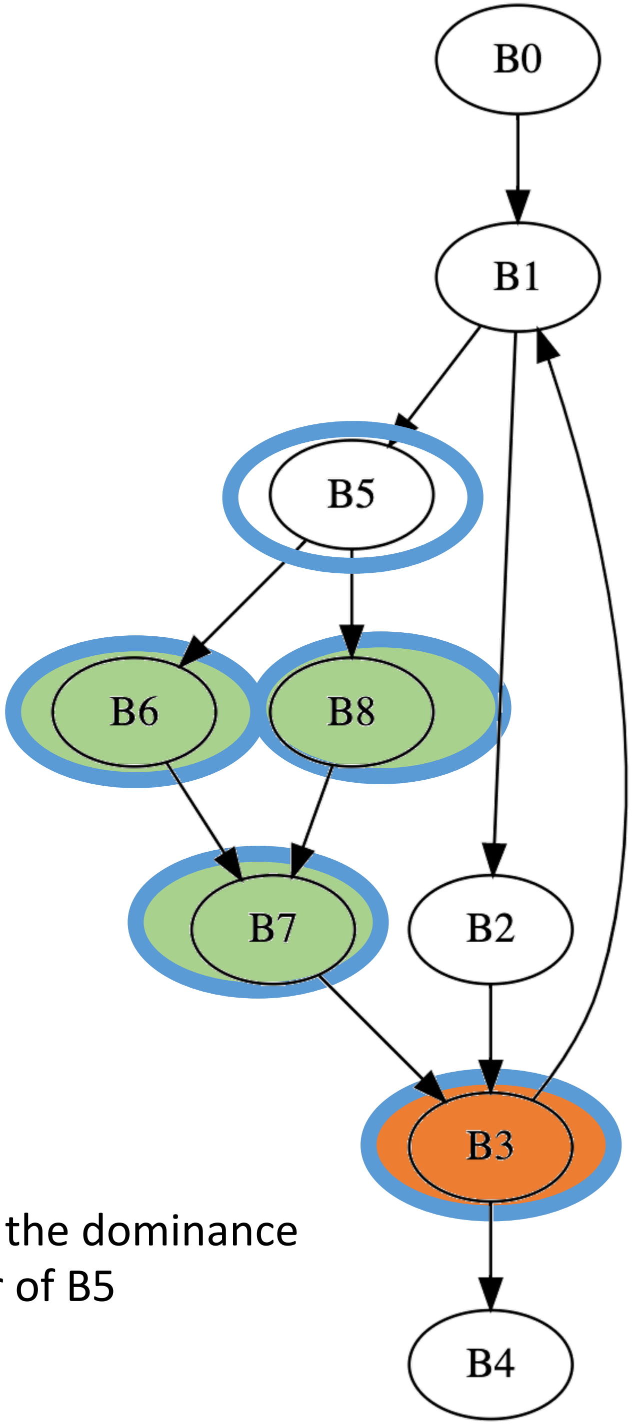
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



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

```
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:
    ϕ 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:
    ϕ is needed in the DF of b

```
B0: i = ...;

B1: a = ϕ(...);
    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:
    ϕ is needed in the DF of b

```
B0: i = ...;

B1: a = φ(...);
    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:
    ϕ is needed in the DF of b

```
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

for each block b:

ϕ is needed in the DF of b

```
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

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 = 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,B3

We've now added new definitions of 'a'!

New matieral

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?

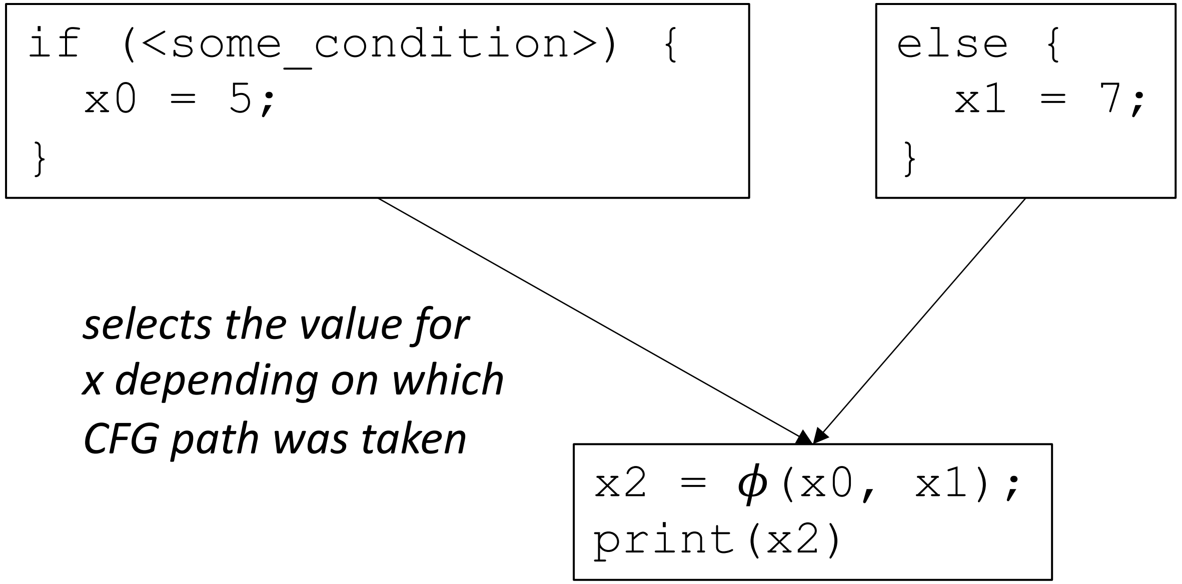
- Can a processor execute phi instructions?
- Just assign to the new variable in the parent?

ϕ instructions

- Example: how to convert this code into SSA?

```
int x;

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

else {
  x1 = 7;
}

x2 = ϕ(x0, x1);
print(x2)
```

number the variables

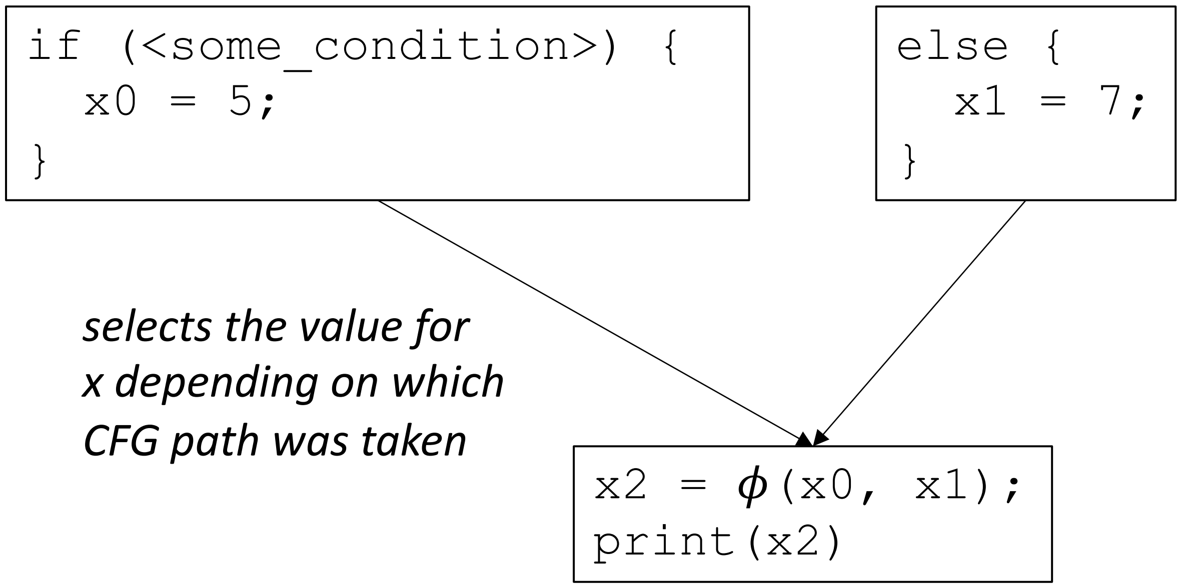


ϕ instructions

- Example: how to convert this code into SSA?

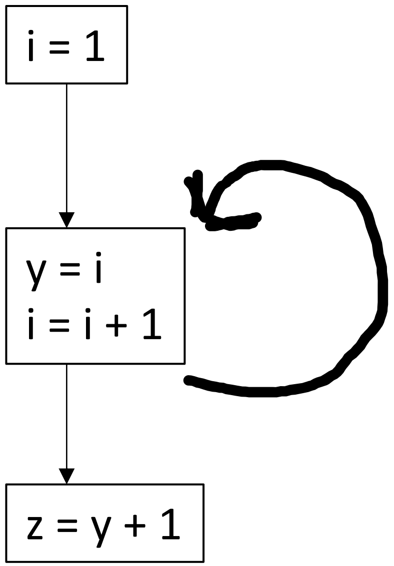
```
int x;

if (<some_condition>) {
  x0 = 5;
  x2 = x0;
}

else {
  x1 = 7;
  x2 = x1;
}

print(x2)
```

number the variables



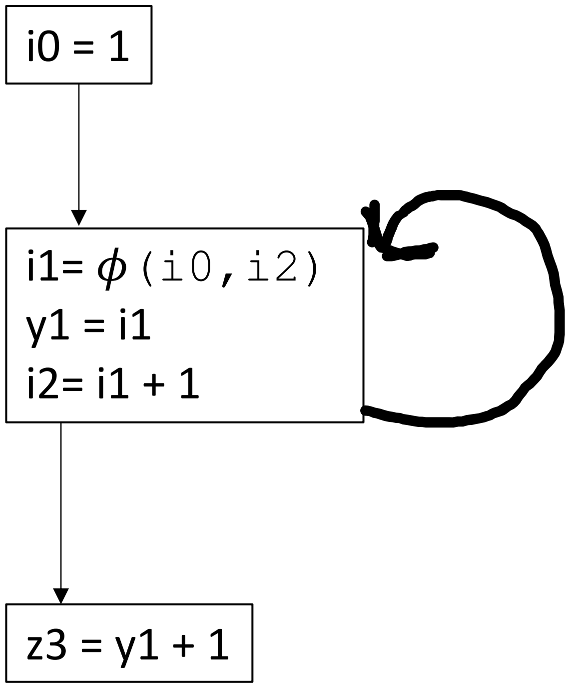
Seems like it works, but…

Lost copy issue



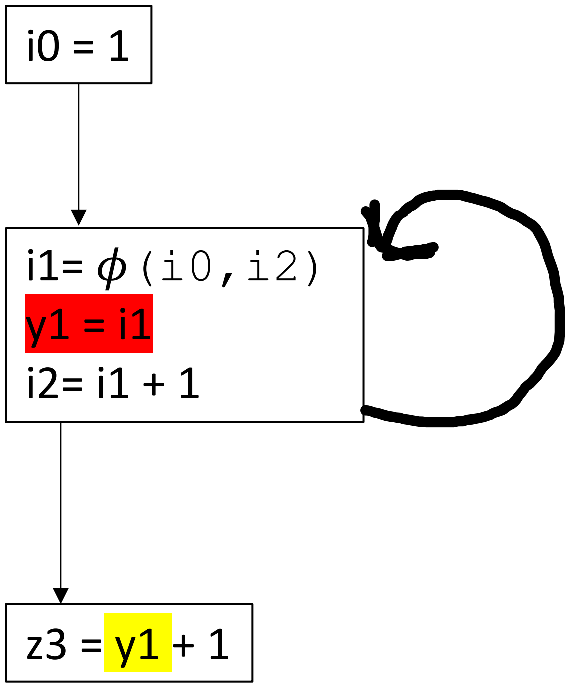
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue



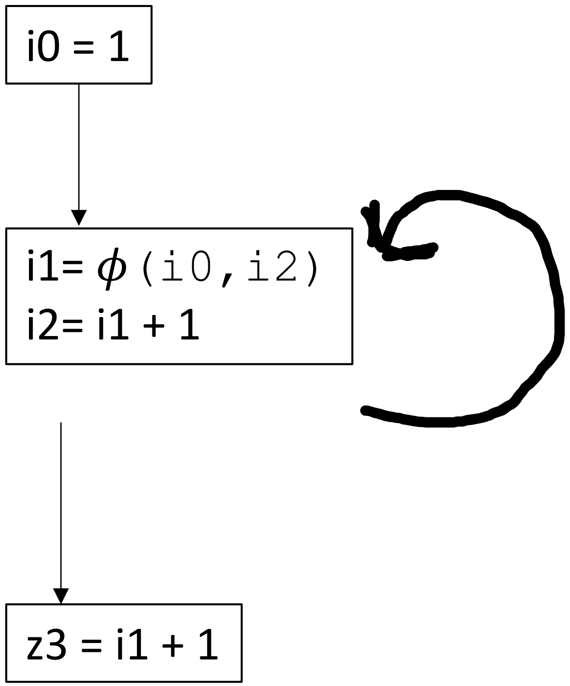
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue



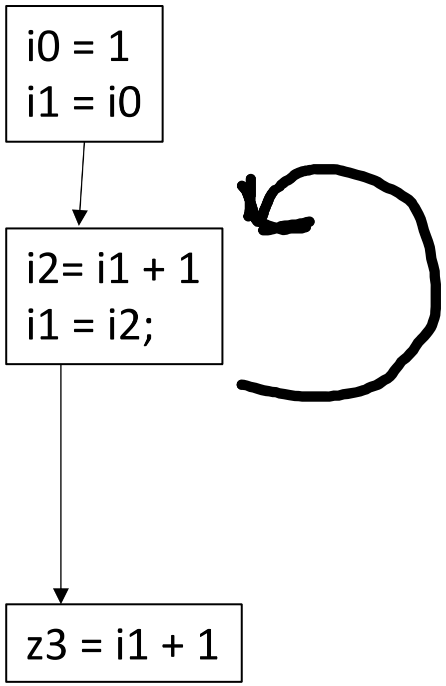
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue



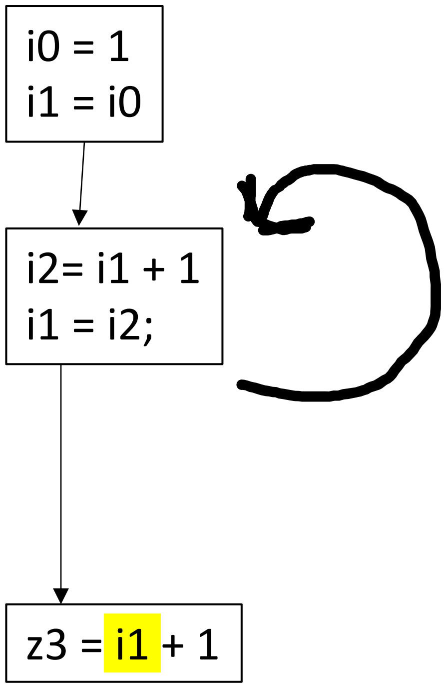
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

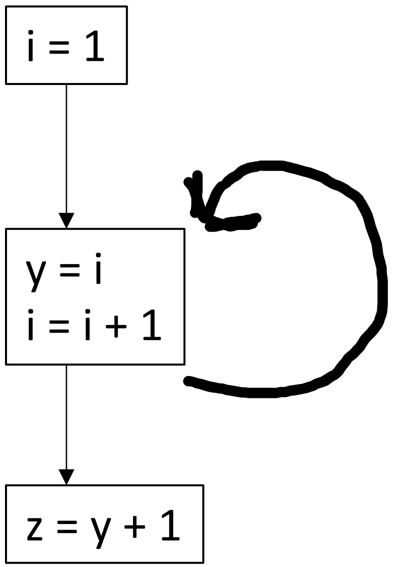
Lost copy issue



Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue

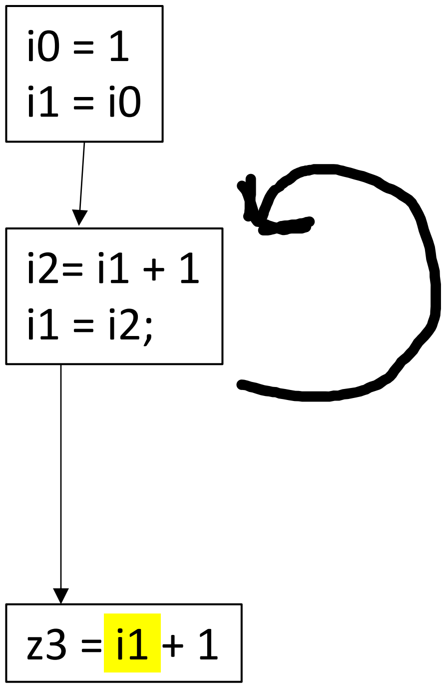


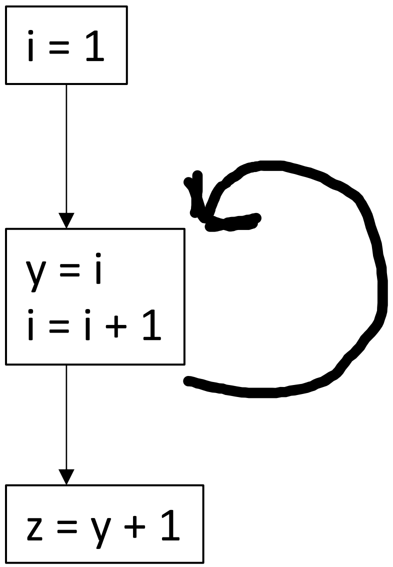


Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue

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





Similar problem called the Swap problem

Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

How to do it then?

- Book gives an algorithm
- Main idea is to introduce *more* temporary registers
- Aggressively do copy propagation to remove them

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: $\{c_1, c_2, c_3 ...\}$
- Special symbols:
  - Top : $\top$
  - Bottom : $\bot$

- Meet operator: $\wedge$

A simple lattice

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

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

$\bot \wedge x = \bot$
$\top \wedge x = x$
Where x is any symbol

A simple lattice

- A set of symbols: $\{c_1, c_2, c_3 ...\}$
- Special symbols:
  - Top : T
  - Bottom : ⊥
- Meet operator: ∧

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

$\bot \wedge x = \bot$
$T \wedge x = x$
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: $\{c_1, c_2, c_3 \ldots\}$
- Special symbols:
  - Top : T
  - Bottom : ⊥
- Meet operator: ∧

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

$\bot \wedge x = \bot$
$T \wedge x = x$
Where x is any symbol

For constant propagation:

take the symbols to be integers

Simple meet operations for integers:
if $c_i \; != c_j$:
  $c_i \wedge c_j = \bot$

else:
$c_i \wedge c_j = c_j$

Constant propagation

- Map each SSA variable x to a lattice value:
  - Value(x) = ⊤ if the analysis has not made a judgment
  - Value(x) = $\mathbf{c_i}$ if the analysis found that variable x holds value c_i
  - Value(x) = ⊥ 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:

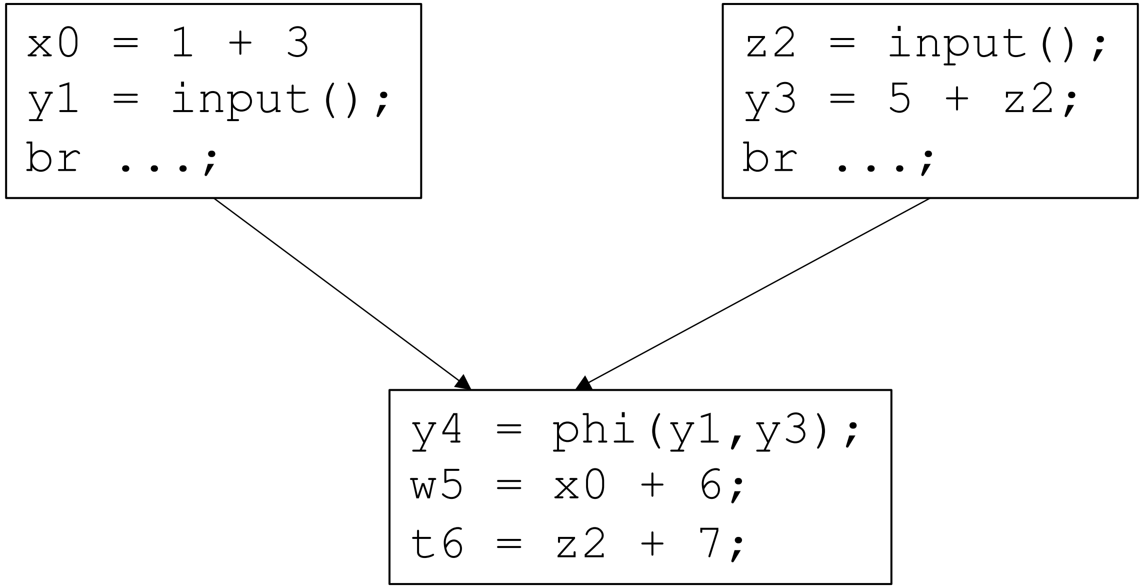
- a constant c_i if the value can be known
- $\bot$ if the value comes from an argument or input
- $\top$ otherwise, e.g. if the value comes from a ϕ node

Then, create a “uses” map

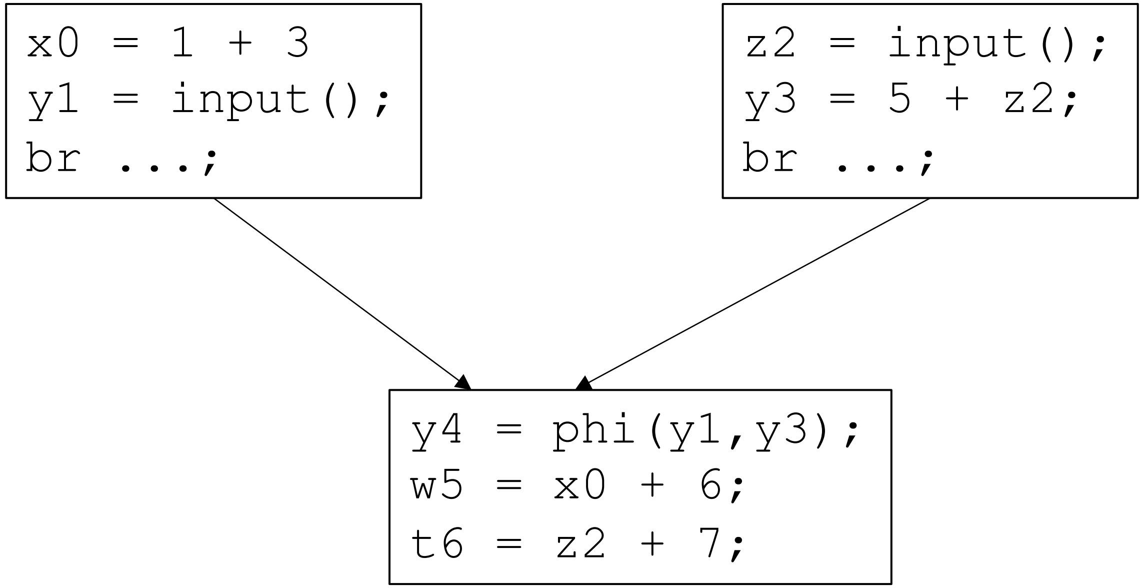
This can be done in a single pass

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

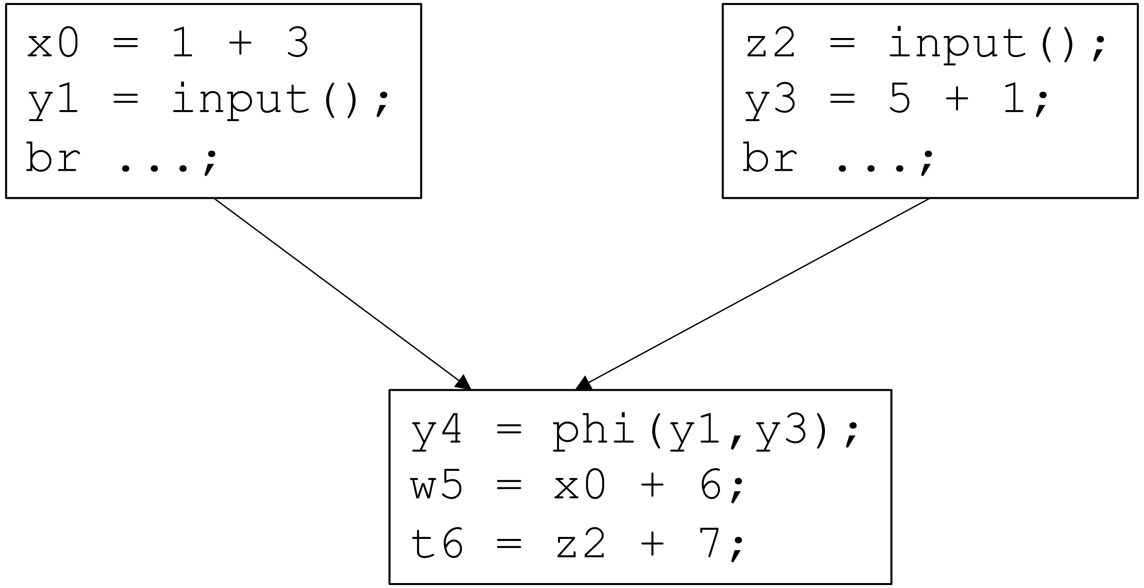
iterate through the worklist:

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

evaluate each use *m* over the lattice

Example:

```
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 : []
}
```

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

Constant propagation algorithm

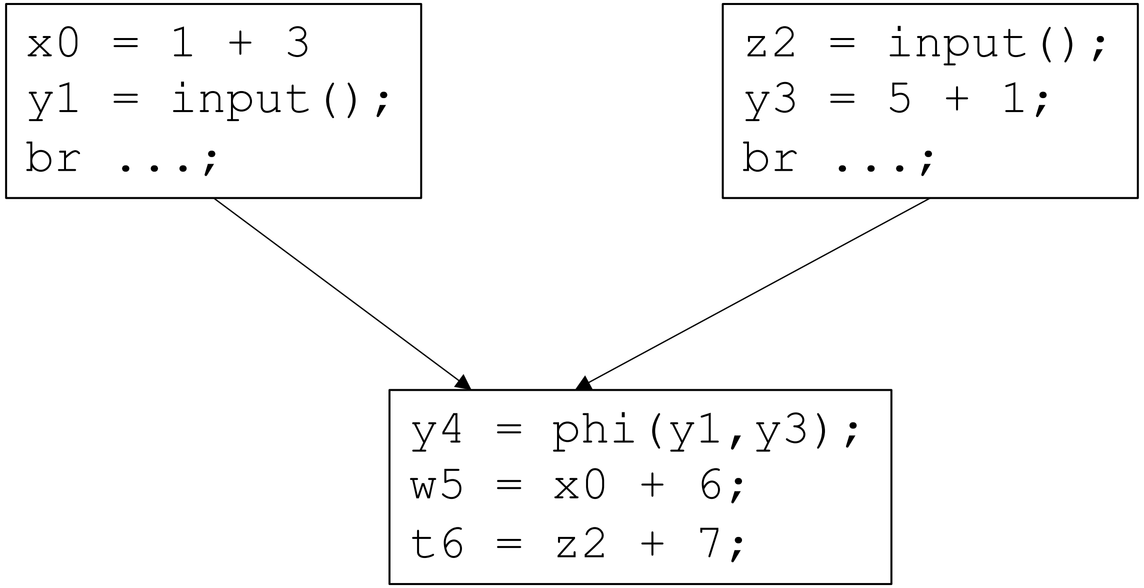
for each item in the worklist, evaluate all of it's uses *m* over the lattice (unique to each optimization)

Example: *m = n*x*

if (Value(n) is $\bot$ or Value(x) is $\bot$)

  Value(m) = $\bot$**;**
  Add m to the worklist if Value(m) has changed;
  break;

Example:



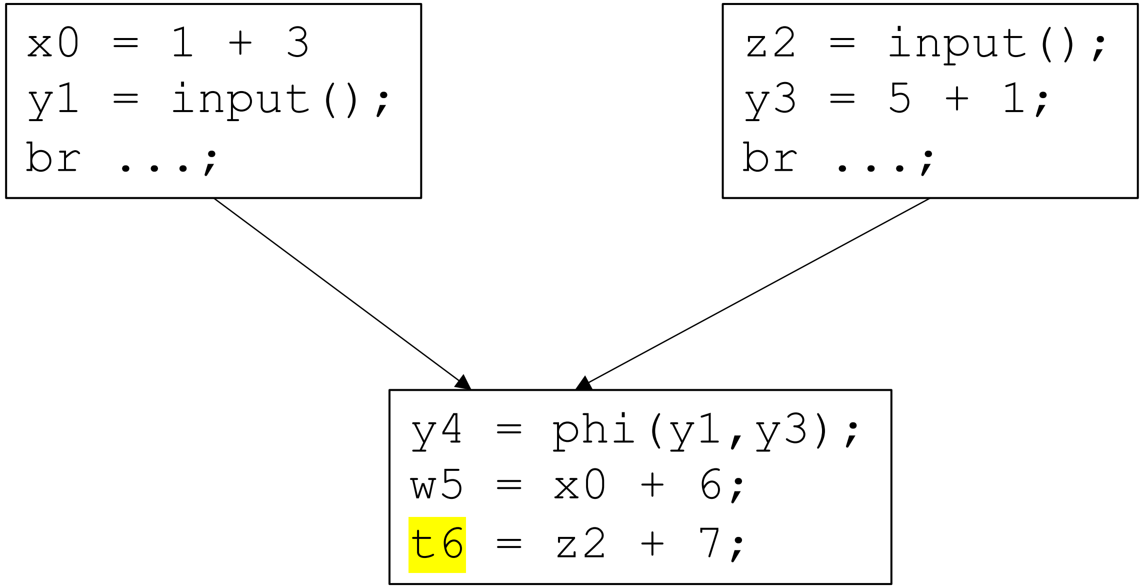
Worklist: `[x0,y1,z2,y3]`

```
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:

```
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 : []
}
```

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

Constant propagation algorithm

evaluate m over the lattice (unique to each optimization)

Example: *m = n*x*

if (Value(n) is ⊥ or Value(x) is ⊥)

  Value(m) = ⊥**;**
  Add m to the worklist if Value(m) has changed;
  break;

Can we optimize this for special cases?

Example:

```
x0 = 1 + 3
y1 = input();
br ...;
```

```
z2 = input();
y3 = 5 + 1;
br ...;
```

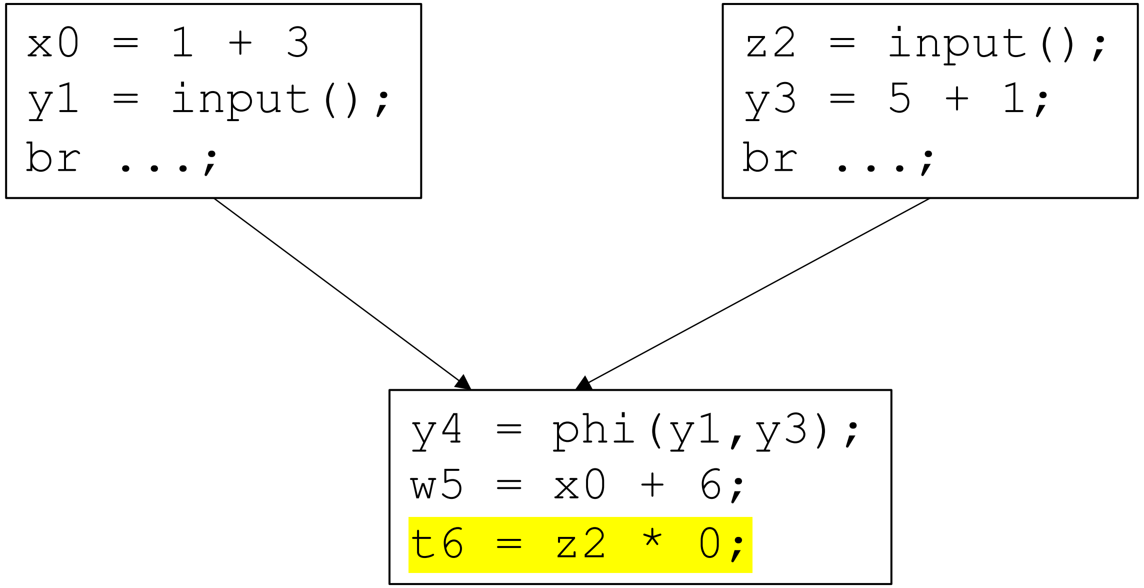
```
y4 = phi(y1,y3);
w5 = x0 + 6;
t6 = z2 * 0;
```

```
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 : []
}
```

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

Example:



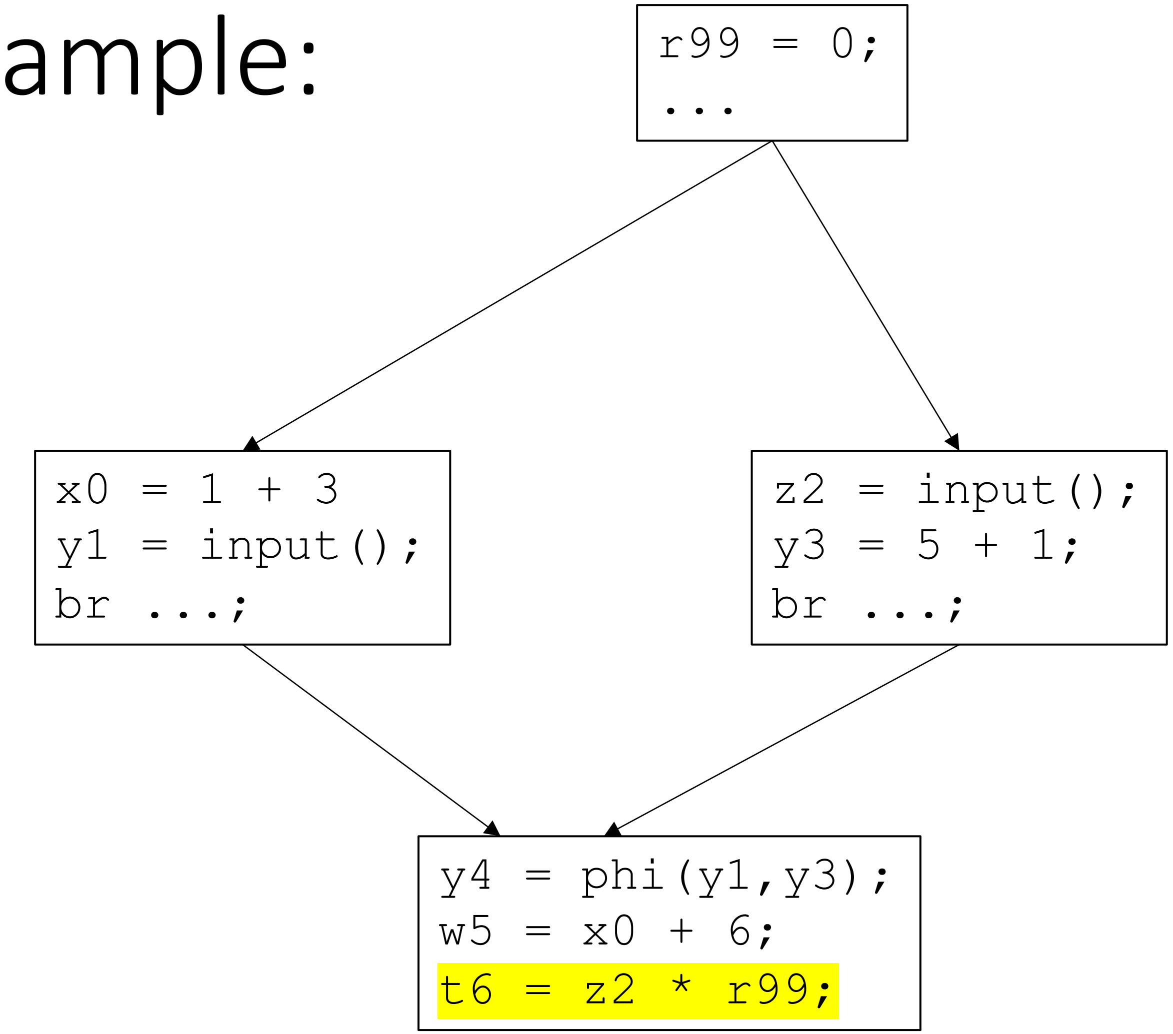

```
Value {
   x0 : 4
   y1 : B
   z2 : B
   y3 : 6
   y4 : T
   w5 : T
   t6 : T
}
```

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

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

Can't this be done
at the expression level?

Example:




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

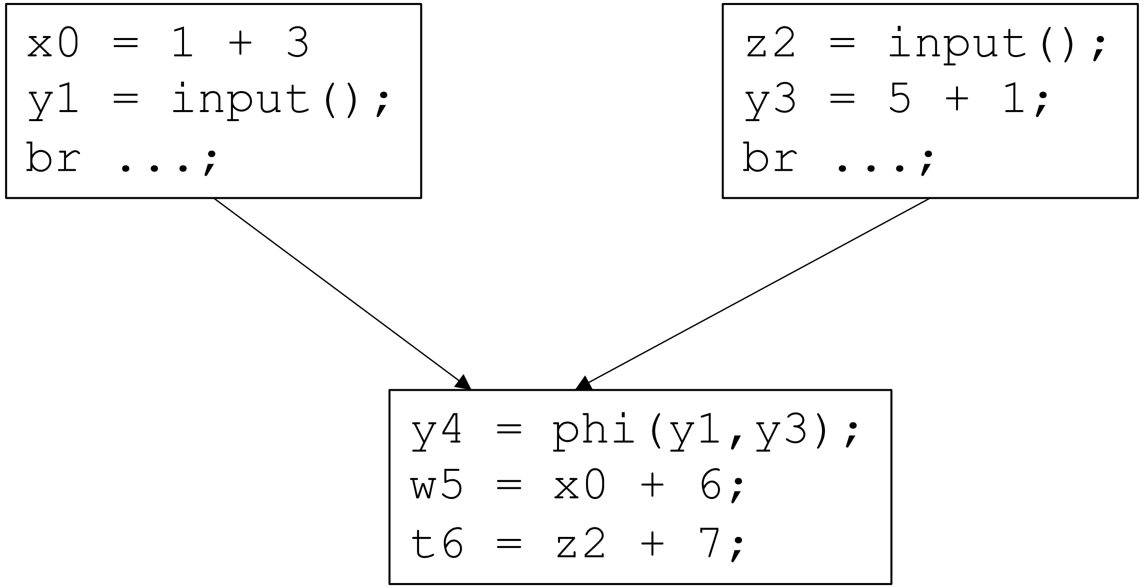
```
Uses {
   x0 : [w5]
   y1 : [y4]
   z2 : [t6]
   y3 : [y4]
   y4 : []
   w5 : []
   t6 : []
}
```

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

Can’t this be done
at the expression level?

Example:

```
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 : []
}
```

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

Constant propagation algorithm

evaluate m over the lattice (unique to each optimization)

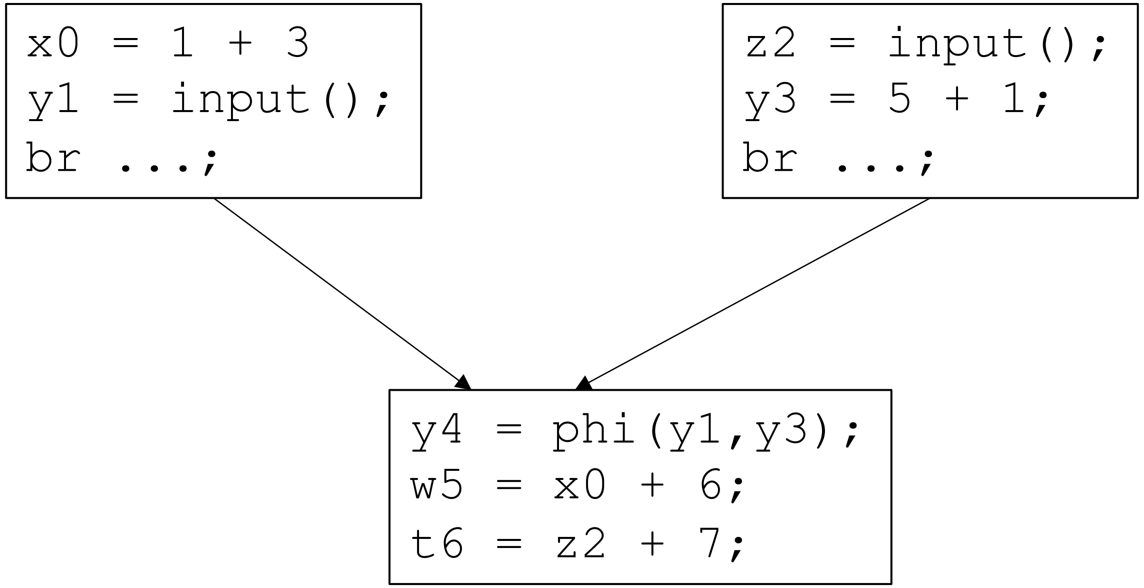
Example: *m = n*x*

// 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:

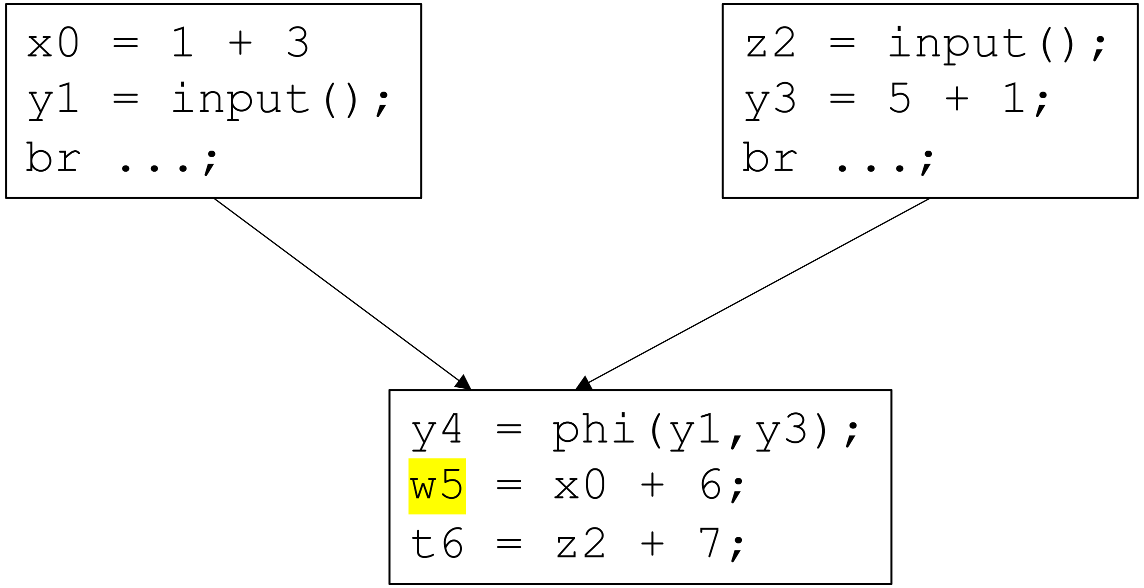



```
Value {
  x0 : 4
  y1 : B
  z2 : B
  y3 : 6
  y4 : T
  w5 : T
  t6 : T
}
```

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

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

Example:




```
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 : []
}
```

Worklist: [x0,y1,y3]

Example:

```
x0 = 1 + 3
y1 = input();
br ...;
```

```
z2 = input();
y3 = 5 + 1;
br ...;
```

```
y4 = phi(y1,y3);
w5 = x0 + 6;
t6 = z2 + 7;
```

Worklist: `[x0,y1,y3]`

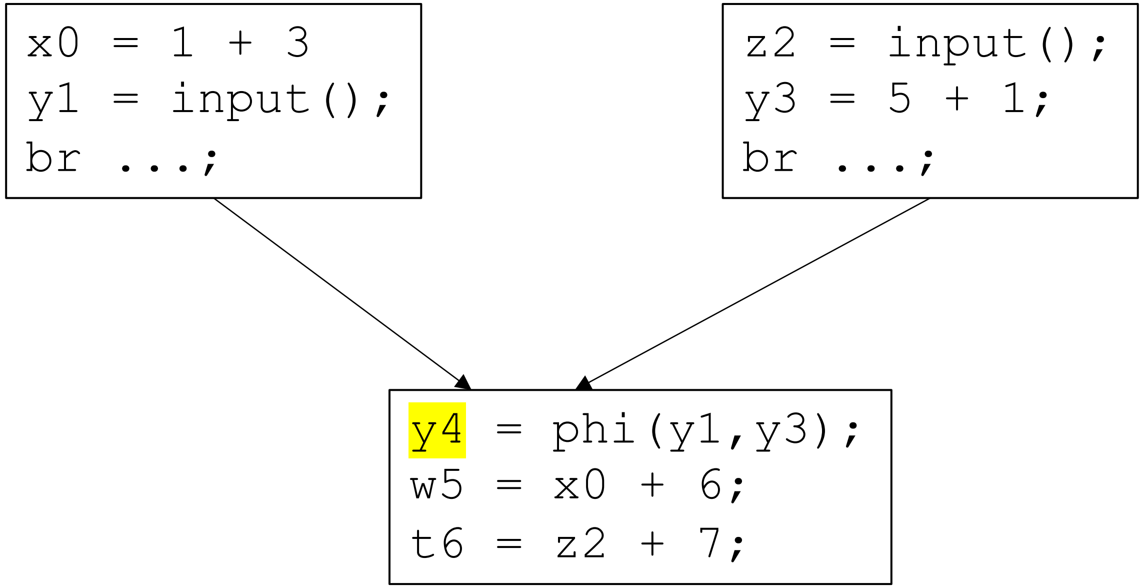
```
Value {
  x0 : 4
  y1 : B
  z2 : B
  y3 : 6
  y4 : T
  w5 : 10
  t6 : T
}
```

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

The elephant in the room

…

Example:




```
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 : []
}
```

Worklist: `[x0,y1,y3]`

Constant propagation algorithm

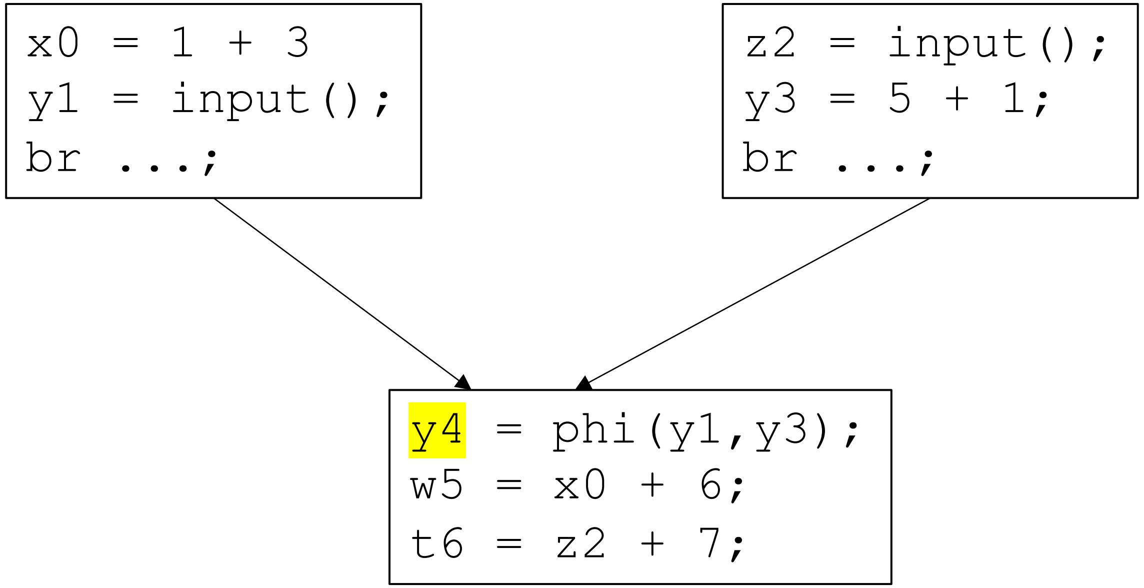
evaluate m over the lattice:

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

Value(m) = $x_1 \wedge x_2$

if Value(m) is not T and Value(m) has changed, then add m to the worklist

Example:

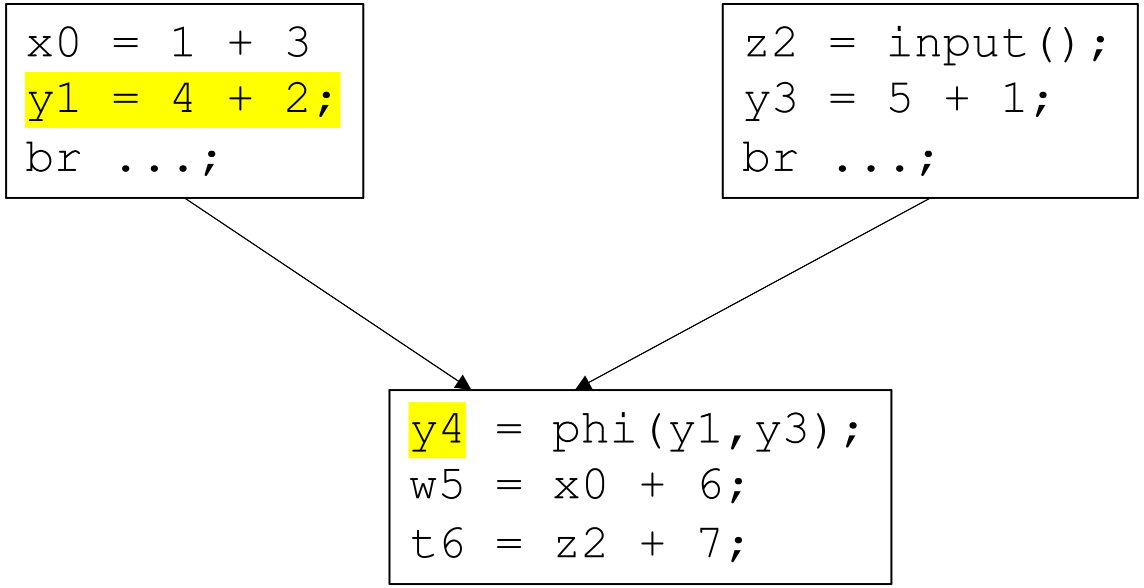


Worklist: `[x0,y1,y3]`

```
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 : 6
   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: $m = \phi(x_1, x_2)$

Value(m) = $x_1 \wedge x_2$

if Value(m) is not T and Value(m) has changed, then add m to the worklist

Constant propagation algorithm

evaluate m over the lattice:

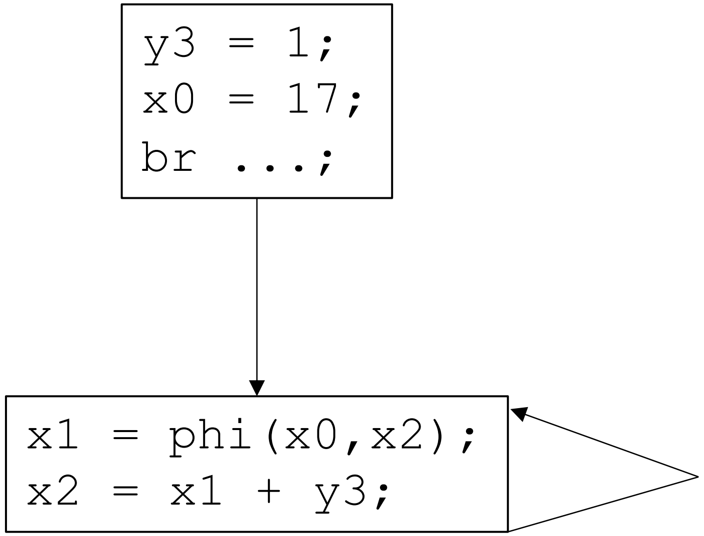
Issue here: potentially assigning a value that might not hold

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

Value(m) = $x_1 \wedge x_2$

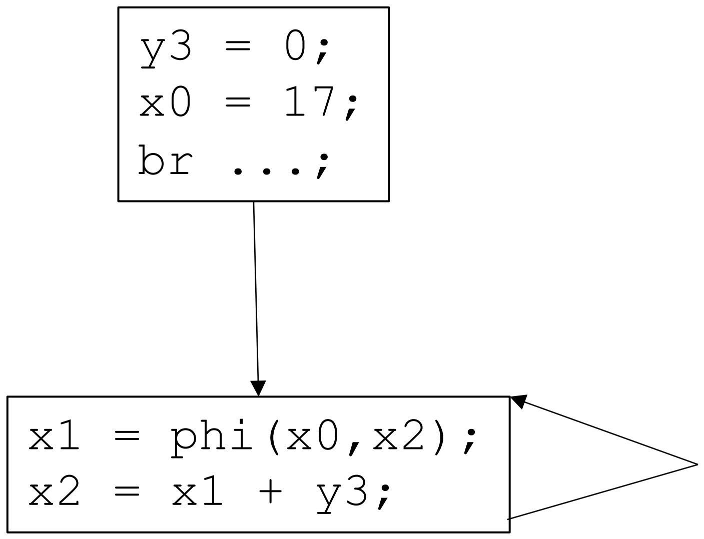
if Value(m) is not T and Value(m) has changed, then add m to the worklist

Example loop:



x1:17

Example loop:



optimistic analysis: Assign unknowns to the earliest possible value.
Correct later

pessimistic analysis: Do not assign unknowns values unless they are known for sure.

Pros/cons?

A simple lattice

- A set of symbols: $\{c_1, c_2, c_3 ...\}$
- Special symbols:
  - Top : T
  - Bottom : ⊥
- Meet operator: ∧

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

$\bot \wedge x = \bot$
$T \wedge x = x$
Where x is any symbol

For Loop unrolling

take the symbols to be integers

Simple meet operations for integers:
if c_i != c_j :
  $c_i \wedge c_j = \bot$

else:
$c_i \wedge c_j = c_j$

A simple lattice

- A set of symbols: $\{c_1, c_2, c_3 \ldots\}$
- Special symbols:
  - Top : T
  - Bottom : ⊥
- Meet operator: $\wedge$

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

$\bot \wedge x = \bot$
$T \wedge x = x$
Where x is any symbol

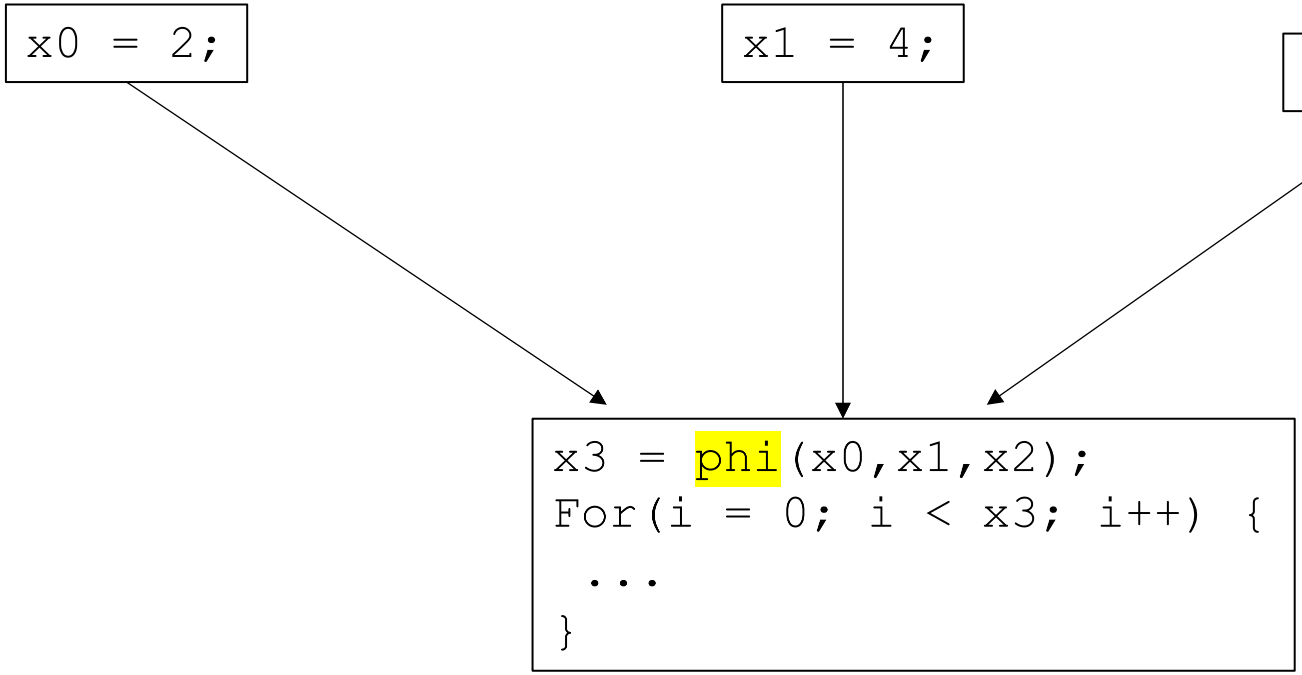
For Loop unrolling

take the symbols to be integers representing the GCD

$c_i \wedge c_j = GCD(c_i, c_j)$

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];
```

See you next time

- Starting module 3