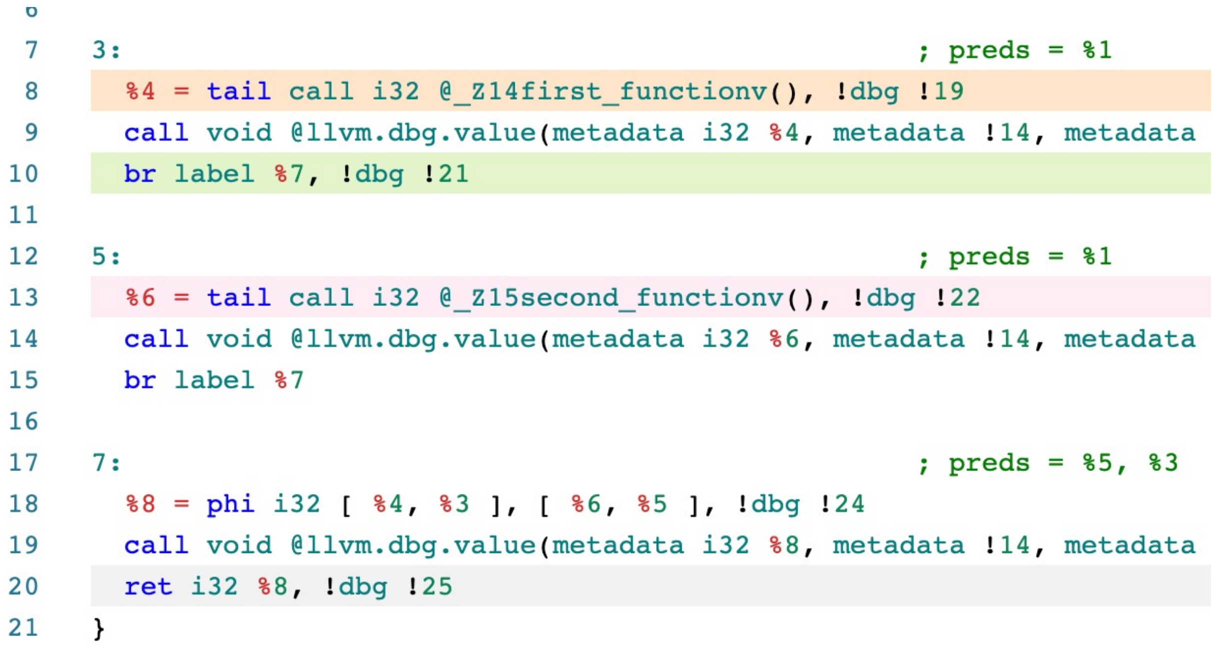
CSE211: Compiler Design

Oct. 25, 2021

- **Topic**: SSA
  - SSA analysis
  - converting back from SSA

- **Questions**:
  - *What are the benefits of SSA?*



```
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

- Remote lecture today
    - Feeling better but testing positive ☹
    - Going to retest on Wednesday night and I’ll let you know
- Remote office hours tomorrow
- Homework 2 is out
    - Please have a partner by the end of day tomorrow
    - Due Nov. 2

Announcements

- Homework 2 is out
  - Everyone should have a partner by now. Please let me know ASAP if not!
  - Please get started soon so that you have time to ask for help if needed

Announcements

- Midterm
  - According to the schedule it was going to be released today. But we’re a little bit behind.
  - I’ll release it on Friday (Oct. 28th) and it will be due the next Friday (Nov. 4).
  - Rules:
    - Open book, open internet, open notes.
    - Do not discuss the test with any other student while it is out.
    - Do not google (or otherwise search) for exact questions. It is fine to search for concepts.
    - Do not post questions to others on the internet (e.g. through discord or reddit)
    - Any question should be asked as a private post on Piazza. If it’s a clarification that needs to be made to the whole class, I will do it in a public Piazza thread

Announcements

- Midterm
    - Designed to take about 2 hours (not including studying)
    - Students report taking longer because they study while taking the test.
    - Students also report taking longer because they double check their answers and make the test nicely formatted.
    - Please look over the test as soon as it is released so that you roughly know how long it will take you.

    - LATE MIDTERMS WILL NOT BE ACCEPTED

Announcements

- Mark your attendance for today after you watch the recording (or if you are attending live)
  - Please try to keep on top of this.
  - We have more attendance put in, please let us know within 1 week if there are any issues

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 ϕ instructions

ϕ instructions

- Example: how to convert this code into SSA?

```
int x;

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

else {
  x = 7;
}

print(x)
```

ϕ instructions

- Example: how to convert this code into SSA?

```
int x;

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

else {
  x = 7;
}

print(x)
```

Start with numbering

ϕ instructions

- Example: how to convert this code into SSA?

```
int x;

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

else {
  x1 = 7;
}

print(x)
```

Start with numbering

ϕ instructions

- Example: how to convert this code into SSA?

```
int x;

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

else {
  x1 = 7;
}

print(x)
```

Start with numbering

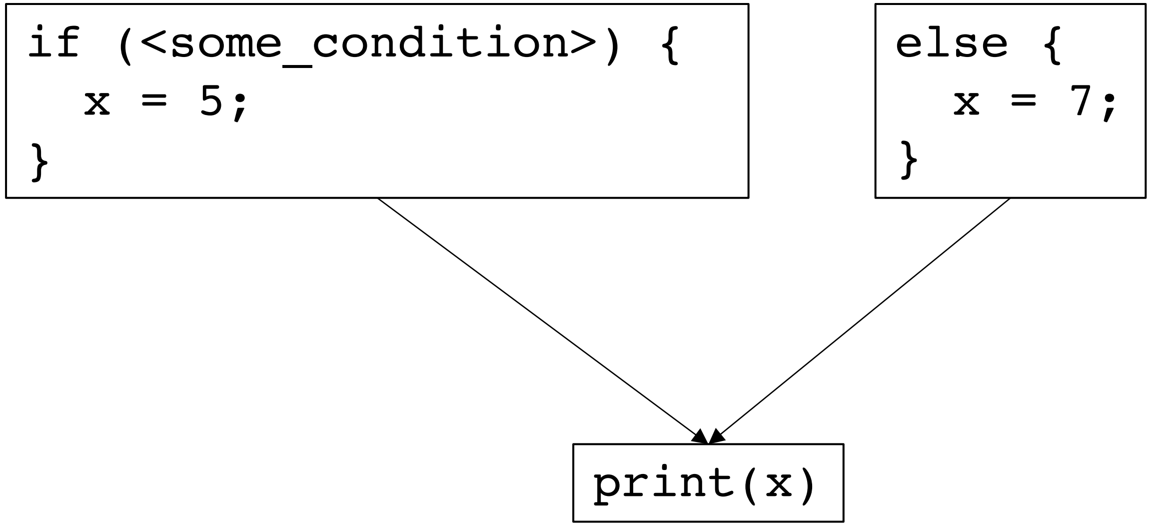
What here?

ϕ 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



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

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

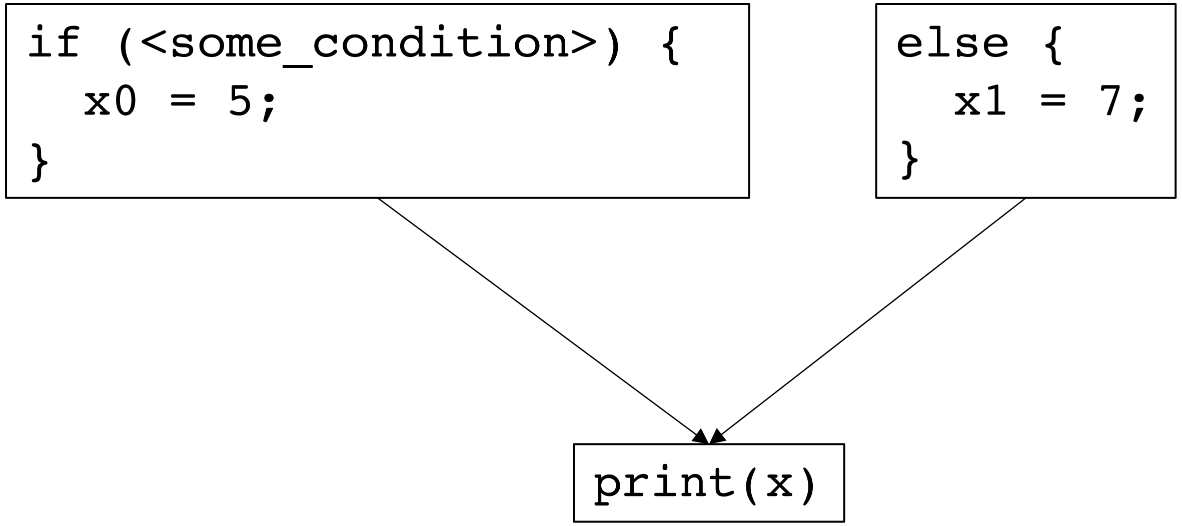
```
print(x)
```

ϕ instructions

- Example: how to convert this code into SSA?

```
int x;

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

else {
  x1 = 7;
}

print(x)
```

number the variables



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

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

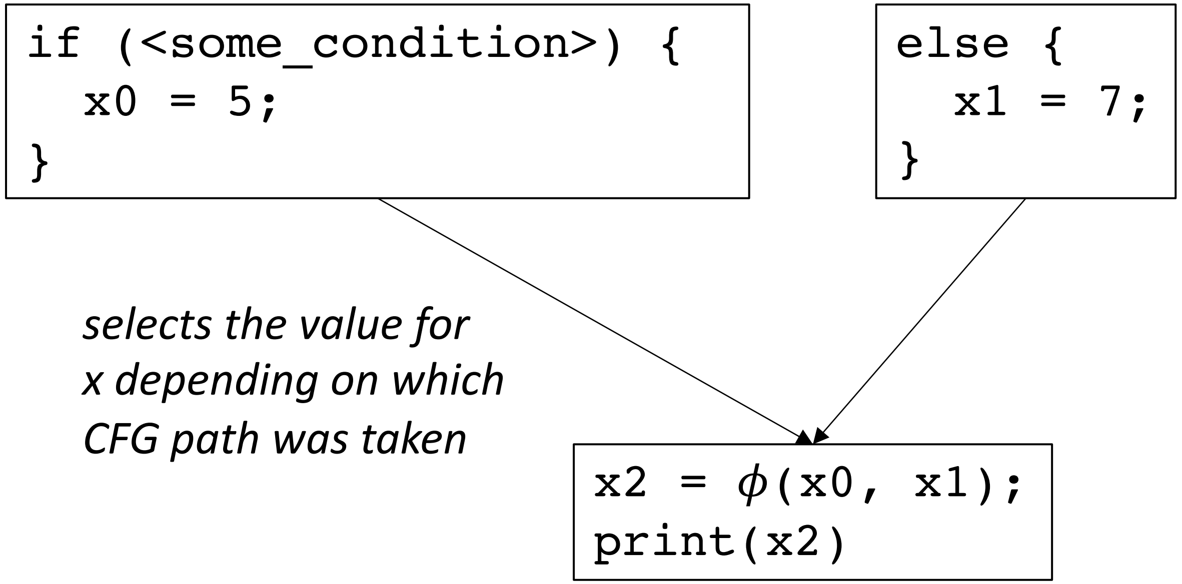
```
print(x)
```

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



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

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

selects the value for x depending on which CFG path was taken

```
x2 = ϕ(x0, x1);
print(x2)
```

Conversion into SSA

Different algorithms depending on how many ϕ instructions

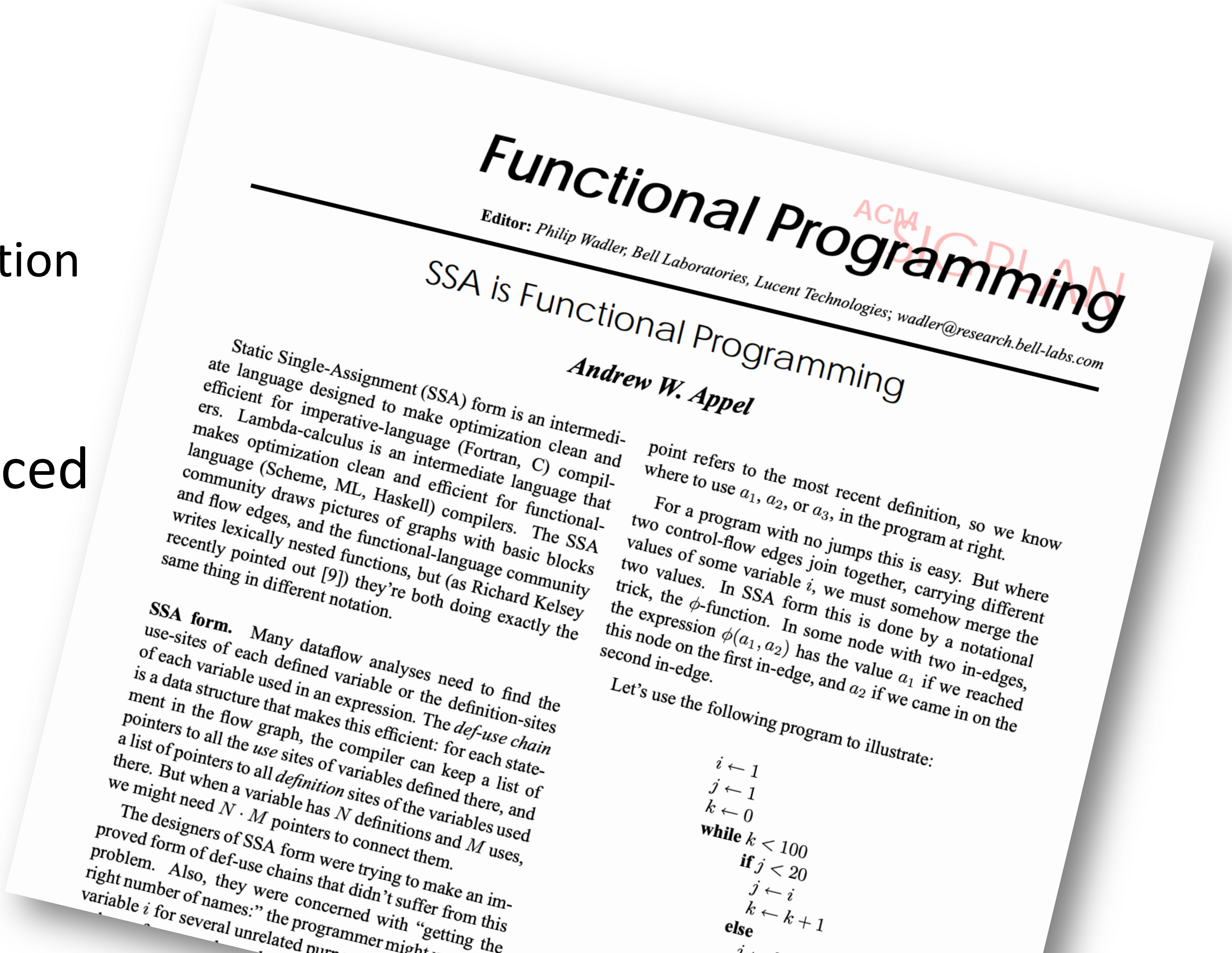
The fewer ϕ instructions, the more efficient analysis will be

Two phases:

- inserting ϕ instructions
- variable naming

A note on SSA variants:

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



Functional Programming

ACM SIGPLAN

Editor: *Philip Wadler, Bell Laboratories, Lucent Technologies; wadler@research.bell-labs.com*

SSA is Functional Programming

Andrew W. Appel

Static Single-Assignment (SSA) form is an intermediate language designed to make optimization clean and efficient for imperative-language (Fortran, C) compilers. Lambda-calculus is an intermediate language that makes optimization clean and efficient for functional-language (Scheme, ML, Haskell) compilers. The SSA community draws pictures of graphs with basic blocks and flow edges, and the functional-language community writes lexically nested functions, but (as Richard Kelsey recently pointed out [9]) they're both doing exactly the same thing in different notation.

SSA form. Many dataflow analyses need to find the use-sites of each defined variable or the definition-sites of each variable used in an expression. The *def-use chain* is a data structure that makes this efficient: for each statement in the flow graph, the compiler can keep a list of pointers to all the *use* sites of variables defined there, and a list of pointers to all *definition* sites of the variables used there. But when a variable has N definitions and M uses, we might need $N \cdot M$ pointers to connect them.

The designers of SSA form were trying to make an improved form of def-use chains that didn't suffer from this problem. Also, they were concerned with “getting the right number of names:” the programmer might

variable i for several unrelated

point refers to the most recent definition, so we know where to use a_1, a_2, or a_3, in the program at right.

For a program with no jumps this is easy. But where two control-flow edges join together, carrying different values of some variable i, we must somehow merge the two values. In SSA form this is done by a notational trick, the ϕ-function. In some node with two in-edges, the expression $\phi(a_1, a_2)$ has the value a_1 if we reached this node on the first in-edge, and a_2 if we came in on the second in-edge.

Let's use the following program to illustrate:

```
$i \leftarrow 1$
$j \leftarrow 1$
$k \leftarrow 0$
while $k < 100$
    if $j < 20$
        $j \leftarrow i$
        $k \leftarrow k + 1$
    else
```

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)
```

A note on SSA variants:

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

Appel Maximal SSA

```
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)
```

EAC Maximal SSA

```
x0 = 1;
y1 = 2;

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

else {
  x8 = 6;
  y9 = 100;
}

x10 = ϕ(x5,x8);
y11 = ϕ(y1,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

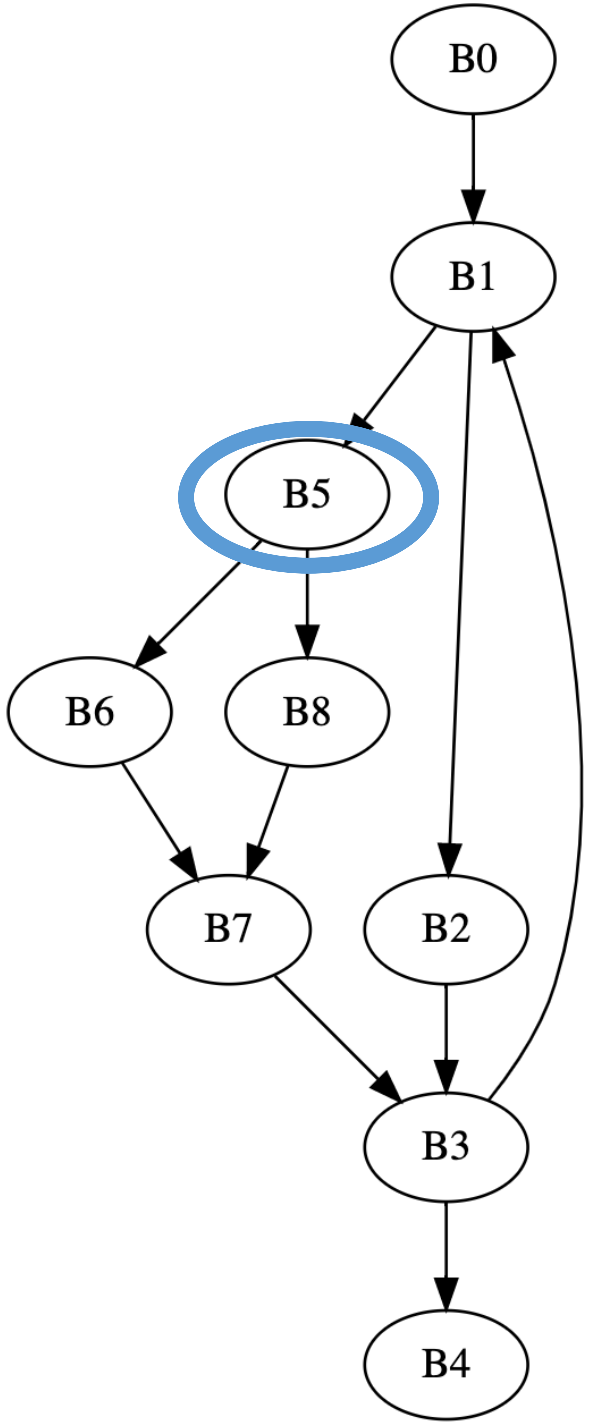
```
x0 = 1;
y1 = 2;

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

else {
  x8 = 6;
  y9 = 100;
}

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

A note on SSA variants:

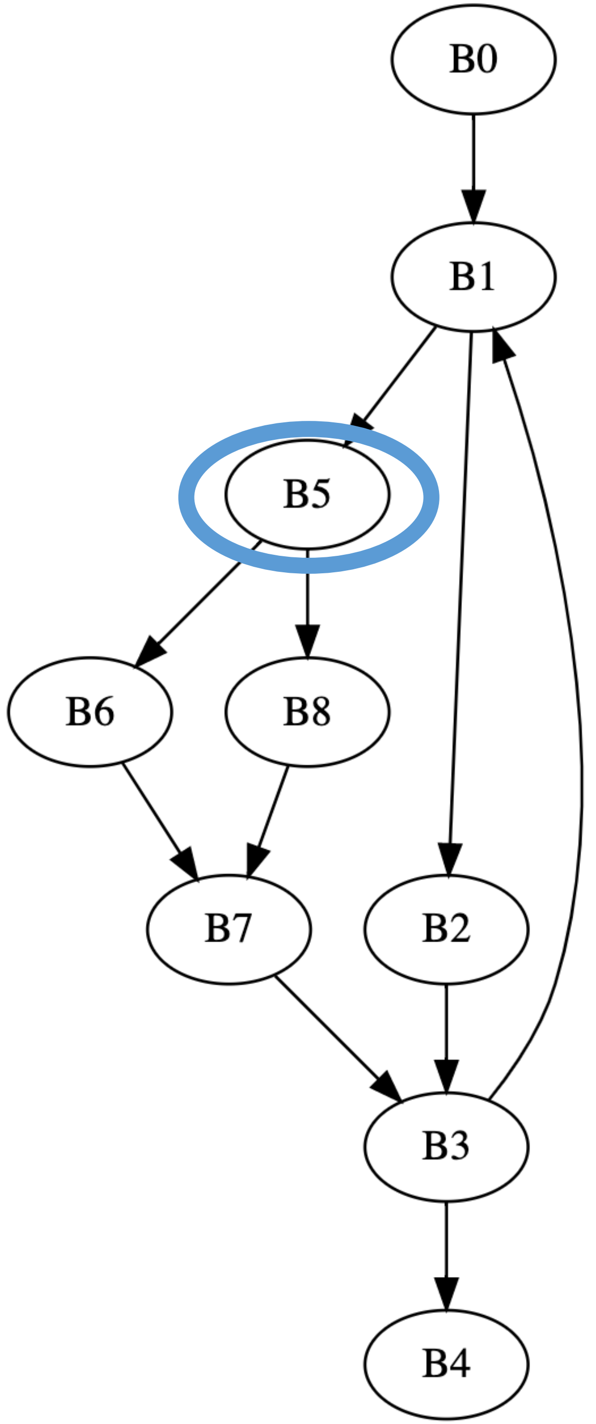
- EAC book describes:
    - Minimal SSA
    - Pruned SSA
    - **Semipruned SSA: We will discuss this one**

Dominance frontier

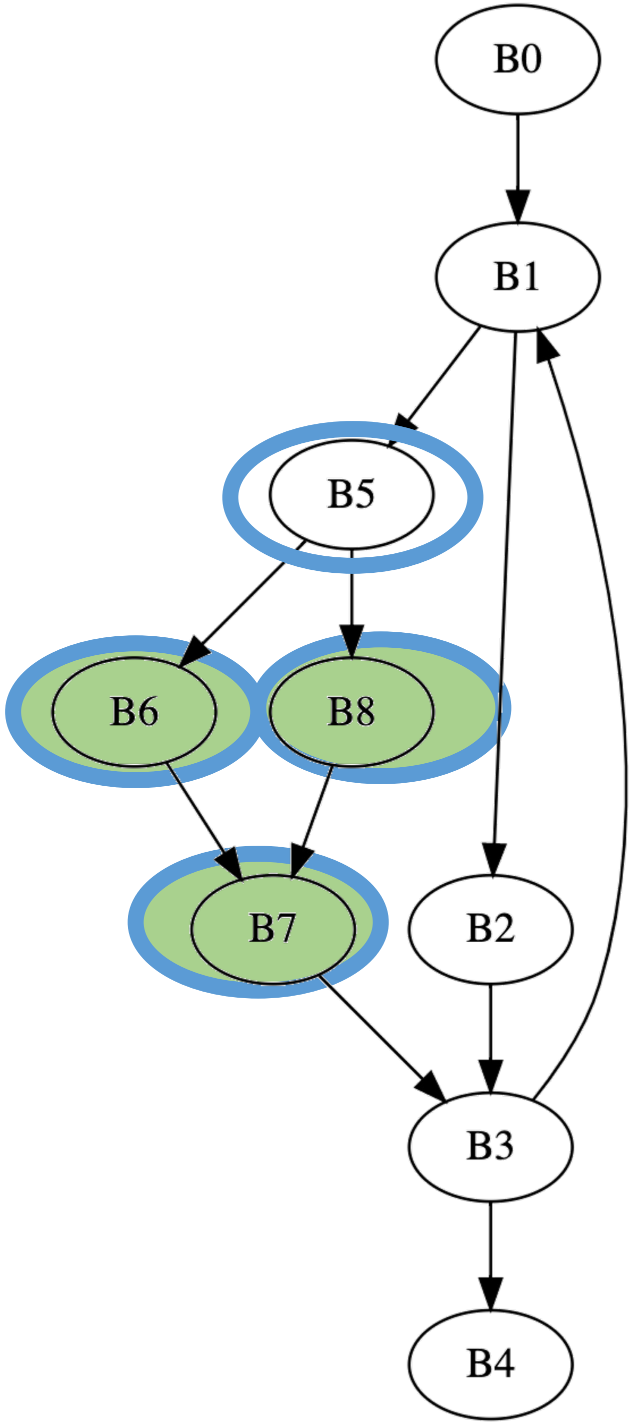
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



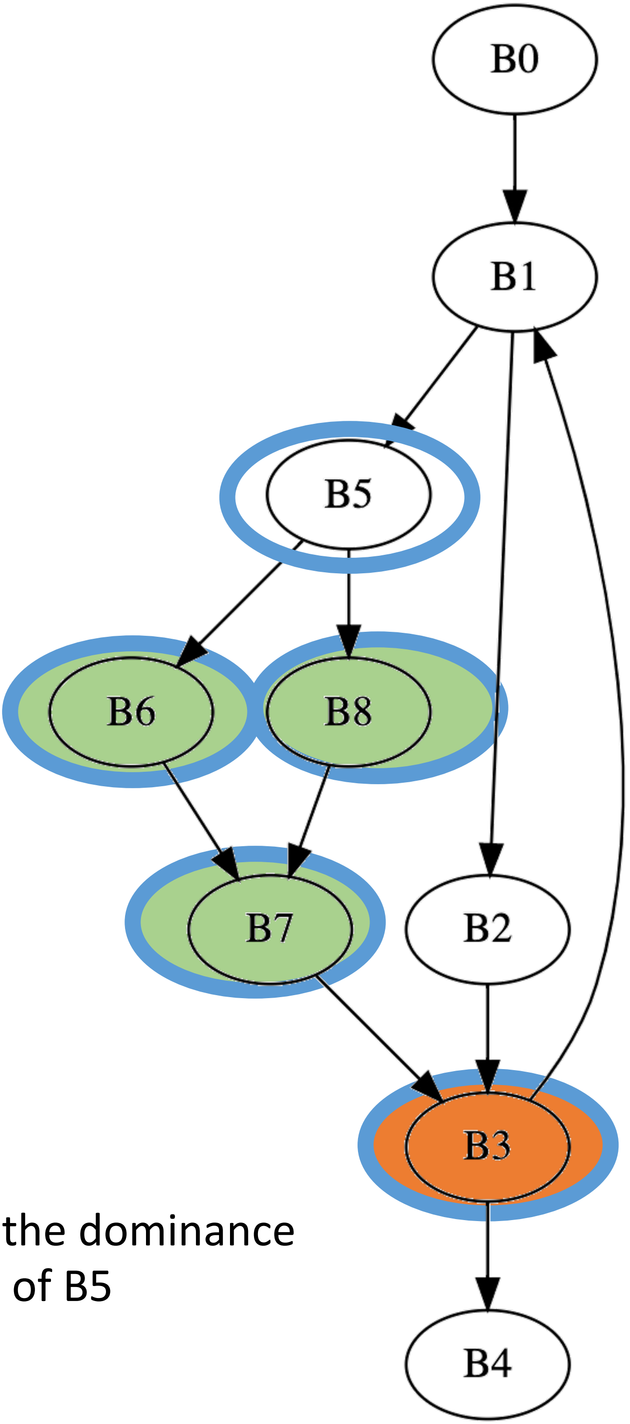
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



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,

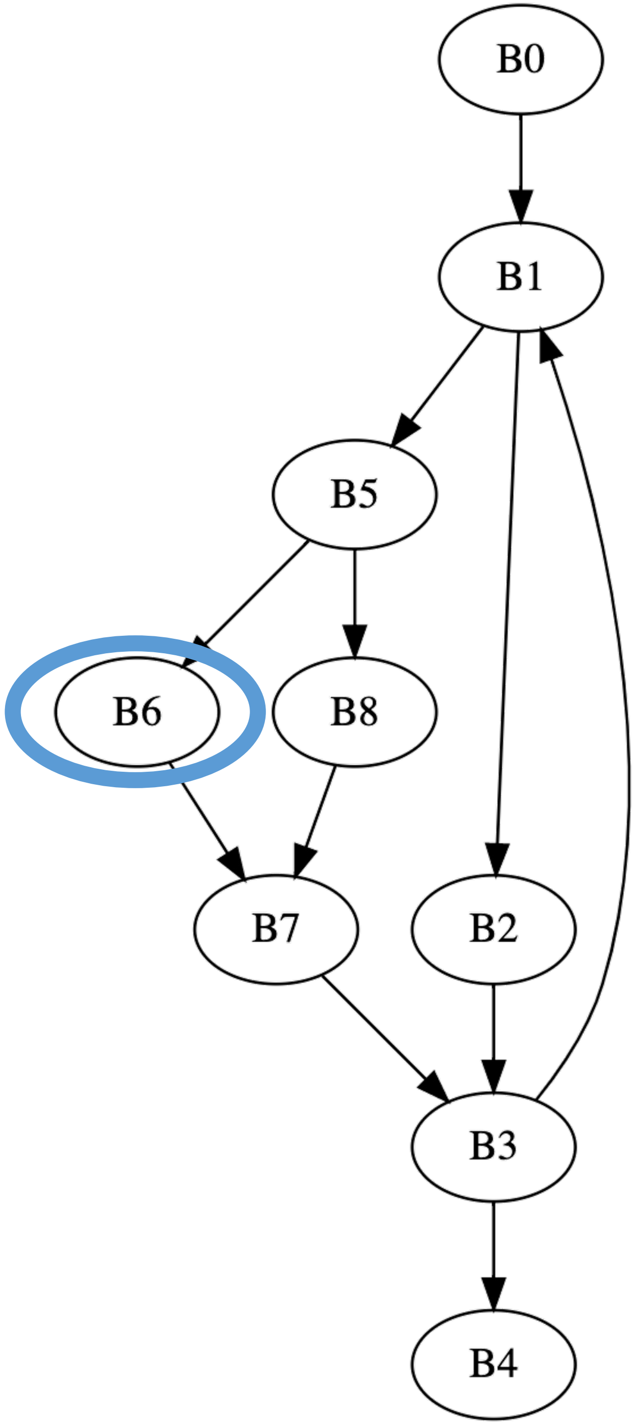


Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

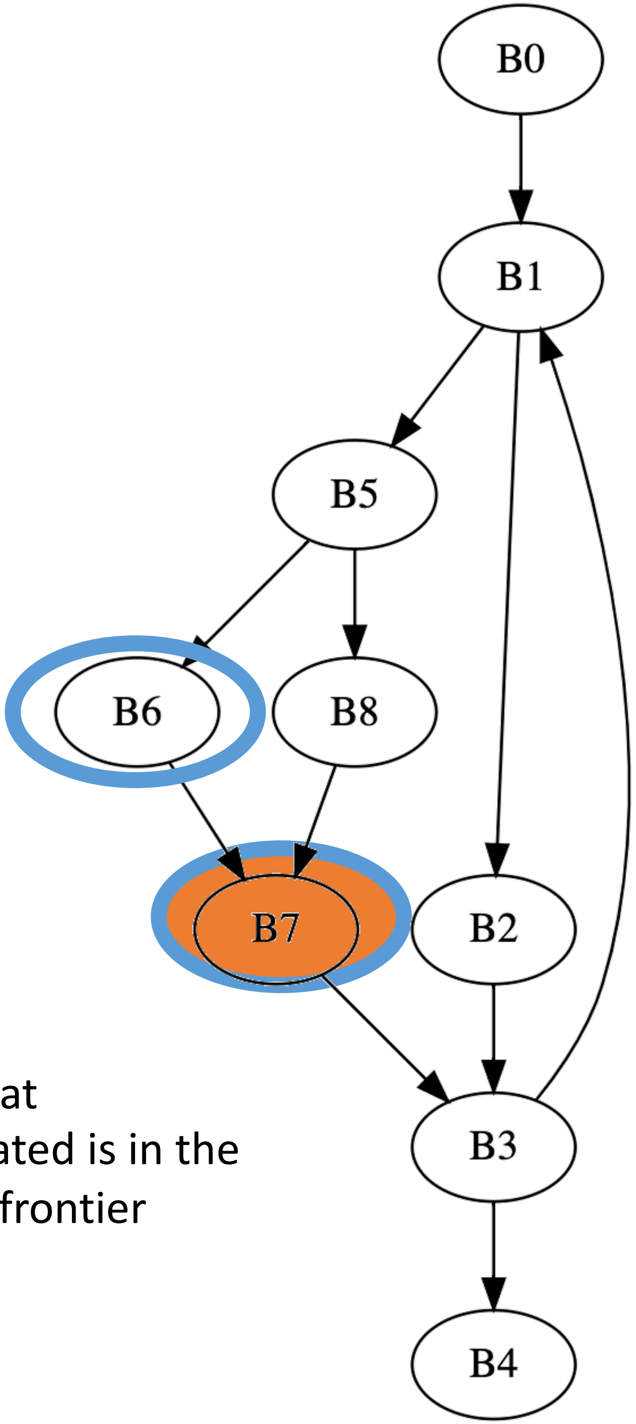


B3 is in the dominance frontier of B5

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,

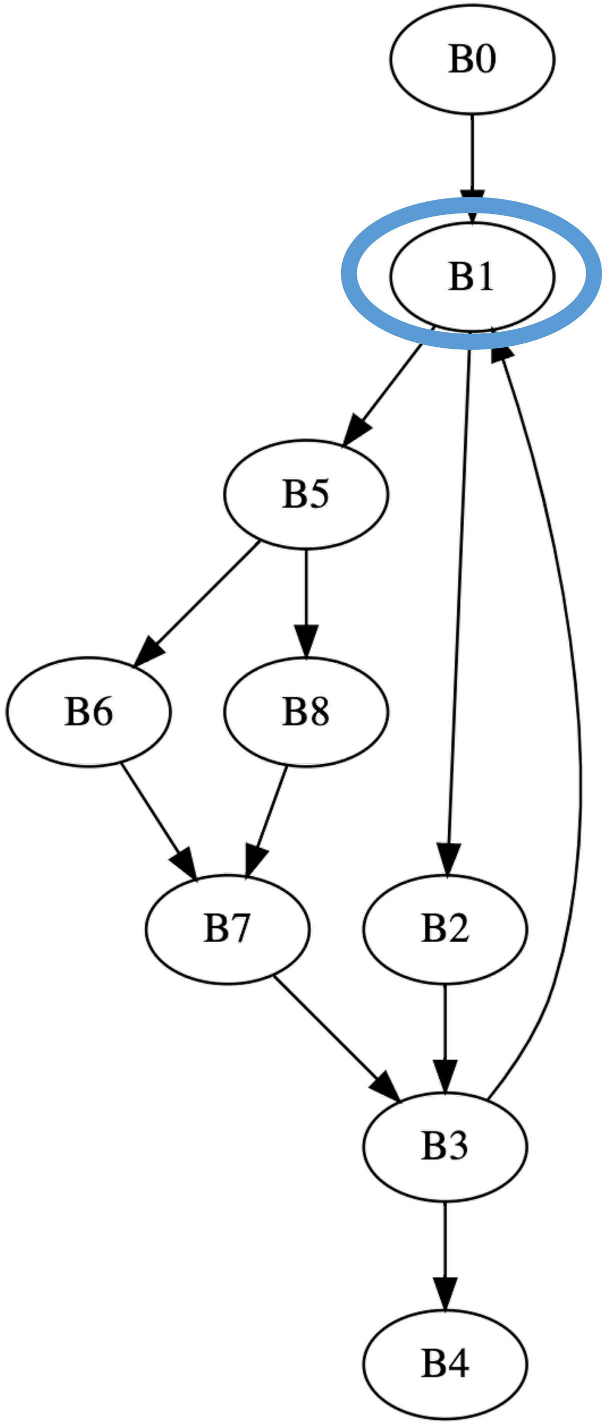


Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

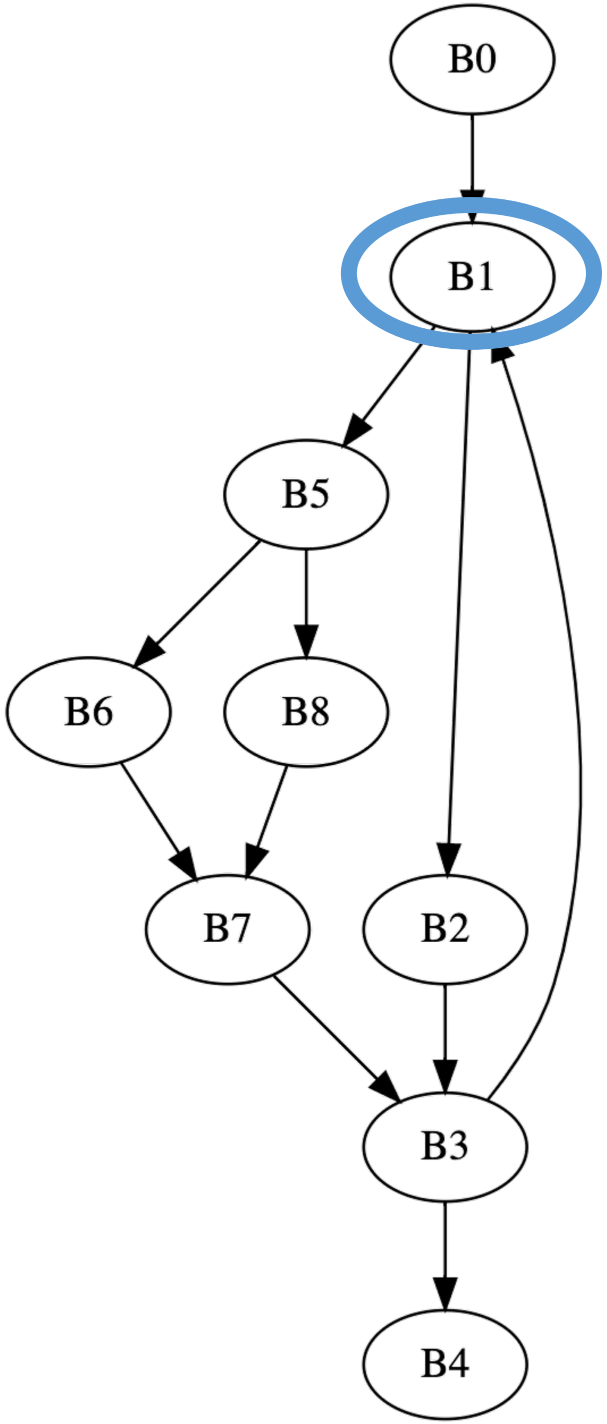


Any child that
isn't dominated is in the
dominance frontier

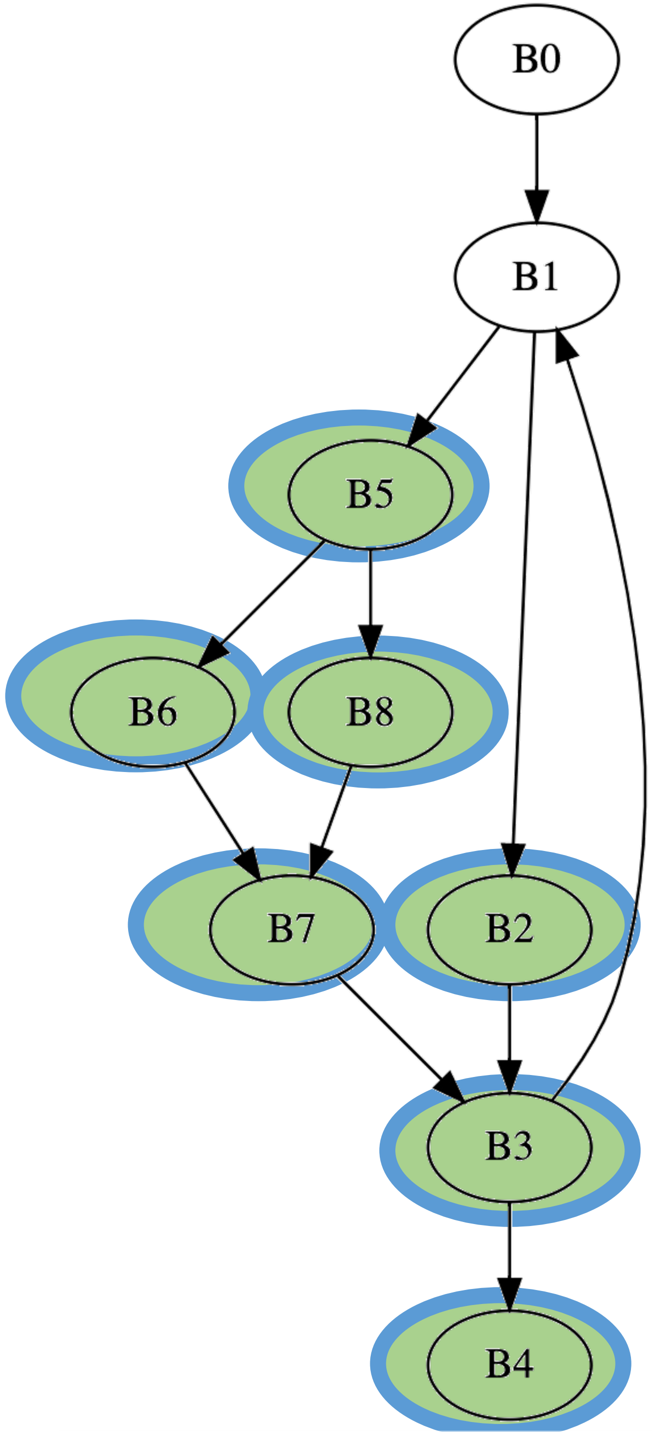
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



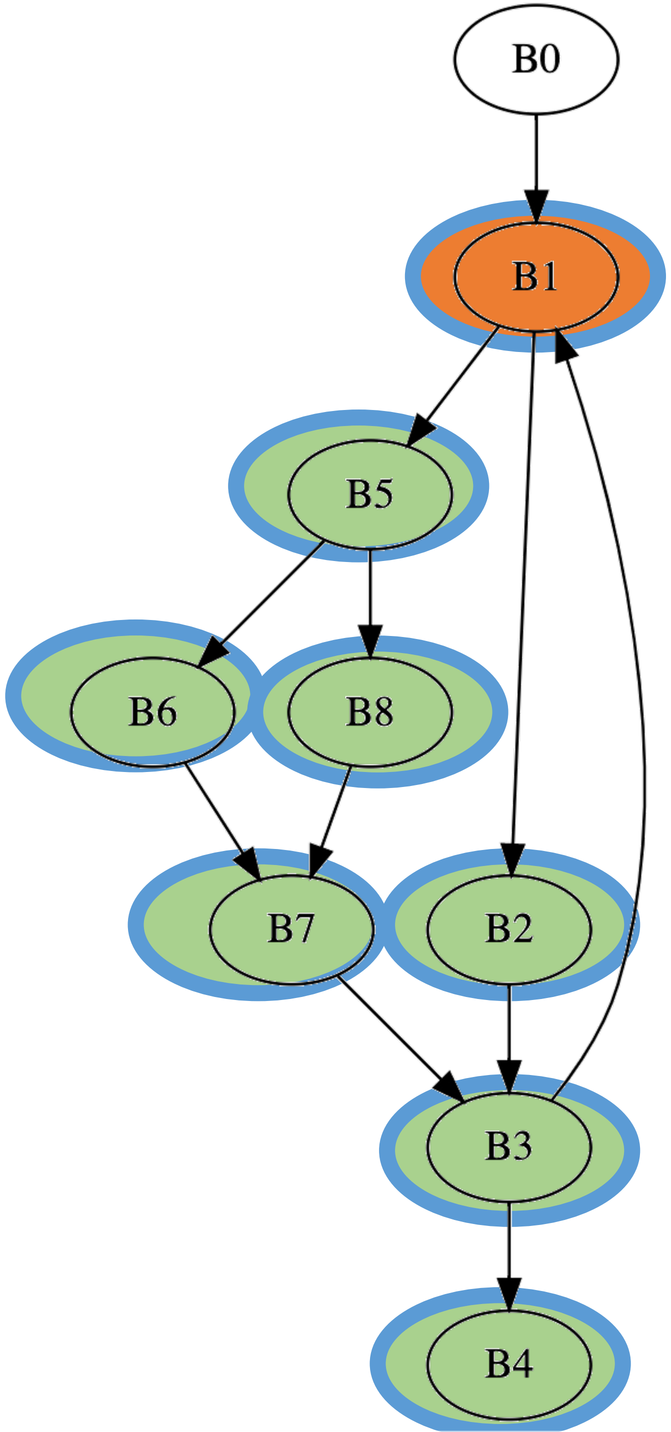
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



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,



```
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 = 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:
    $\phi$ 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;
```

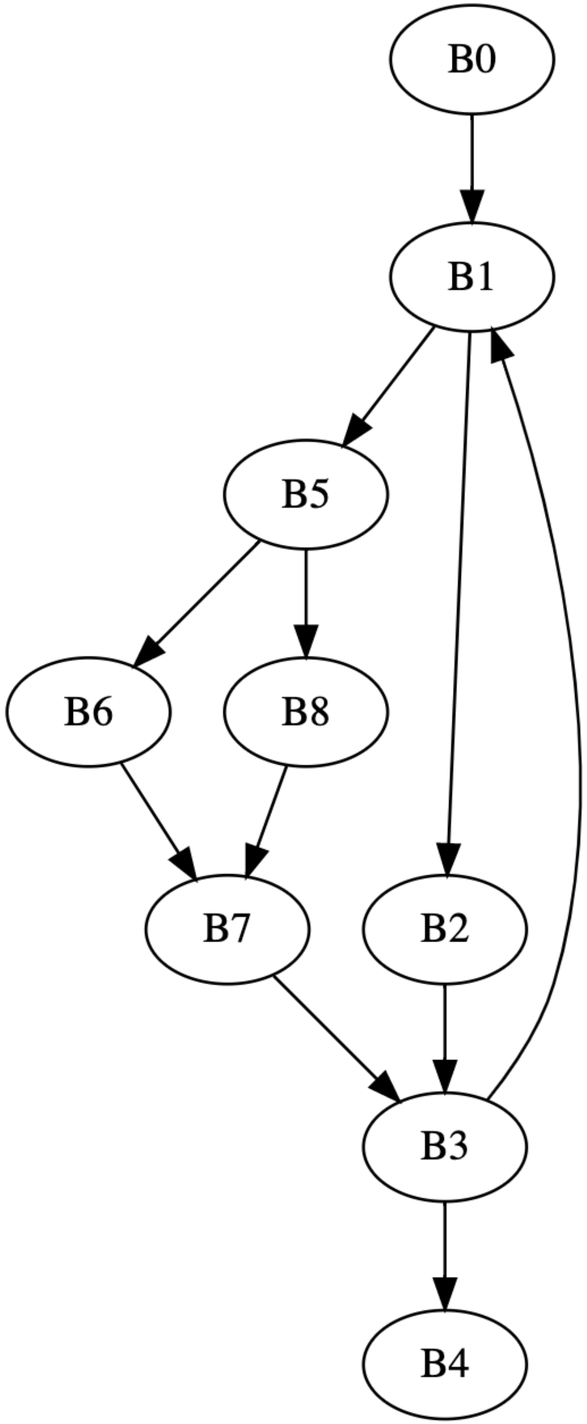
Node	Dominator Frontier
B0	{}
B1	B1
B2	B3
B3	B1
B4	{}
B5	B3
B6	B7
B7	B3
B8	B7

Var	a
Blocks	B1,B5,B3

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

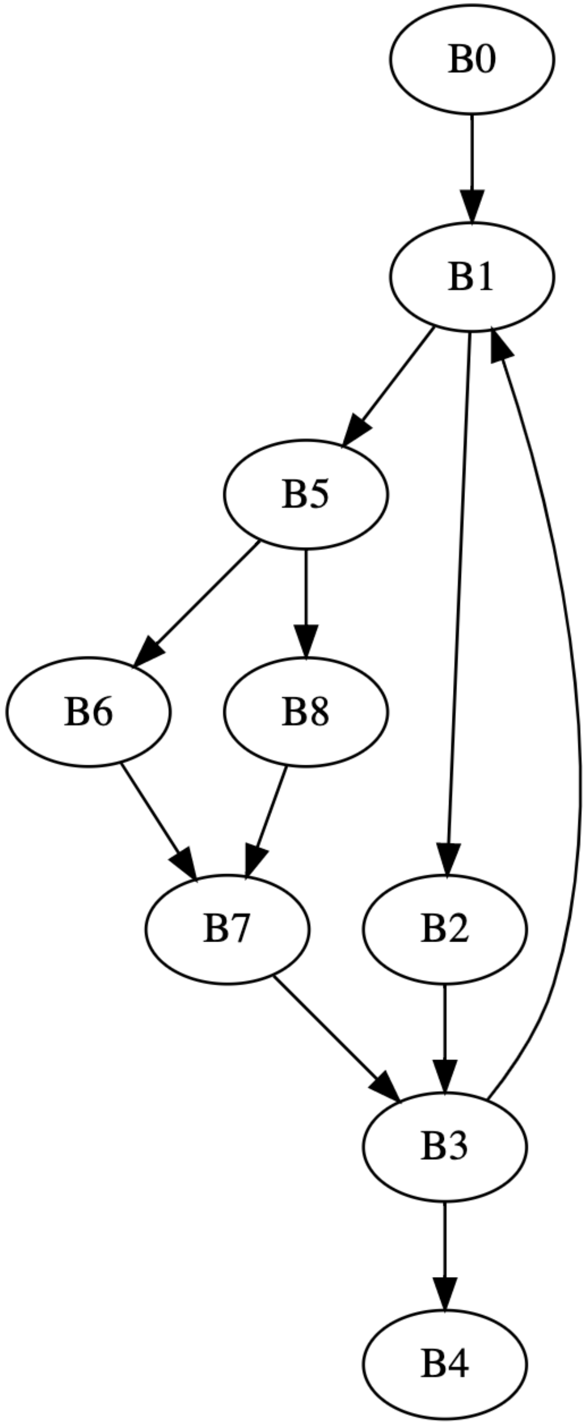
```
B0: i0 = ...;

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

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

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

```
B4: return

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

B6: d = ...;

B7: d = φ(...);
    c = φ(...);
    b = ...;

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



```
B0: i0 = ...;

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

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

B3: a10 = φ(...);
    b11 = φ(...);
    c12 = φ(...);
    d13 = φ(...);
    y14 = ...;
    z15 = ...;
    i16 = ...;
    br ... B1, B4;
```

```
B4: return

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

B6: d19 = ...;

B7: d20 = φ(...);
    c21 = φ(...);
    b22 = ...;

B8: c23 = ...;
    br B7;
```

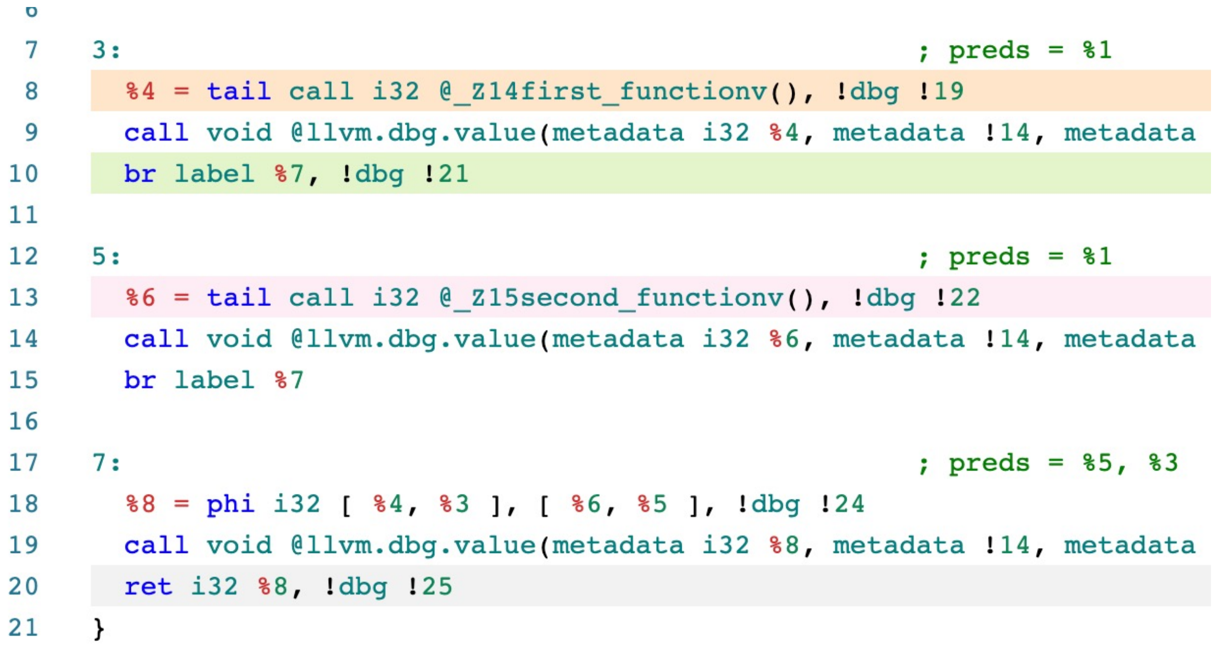


CSE211: Compiler Design

Oct. 25, 2021

- **Topic**: SSA
  - SSA analysis
  - converting back from SSA

- **Questions**:
  - *What are the benefits of SSA?*



```
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
}
```

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 : T
  - Bottom : ⊥

- Meet operator: $\wedge$

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:

⊥ ∧ x = ⊥
T ∧ x = x
Where x is any symbol

A simple lattice

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

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

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

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

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

For constant propagation:

take the symbols to be integers

Simple meet operations for integers:

```
if $c_i$ != $c_j$ :
   $c_i \wedge c_j = \bot$

else:
$c_i \wedge c_j = c_j$
```

Constant propagation

- Map each SSA variable x to a lattice value:
  - Value(x) = $\top$ 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) = $\bot$ 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:

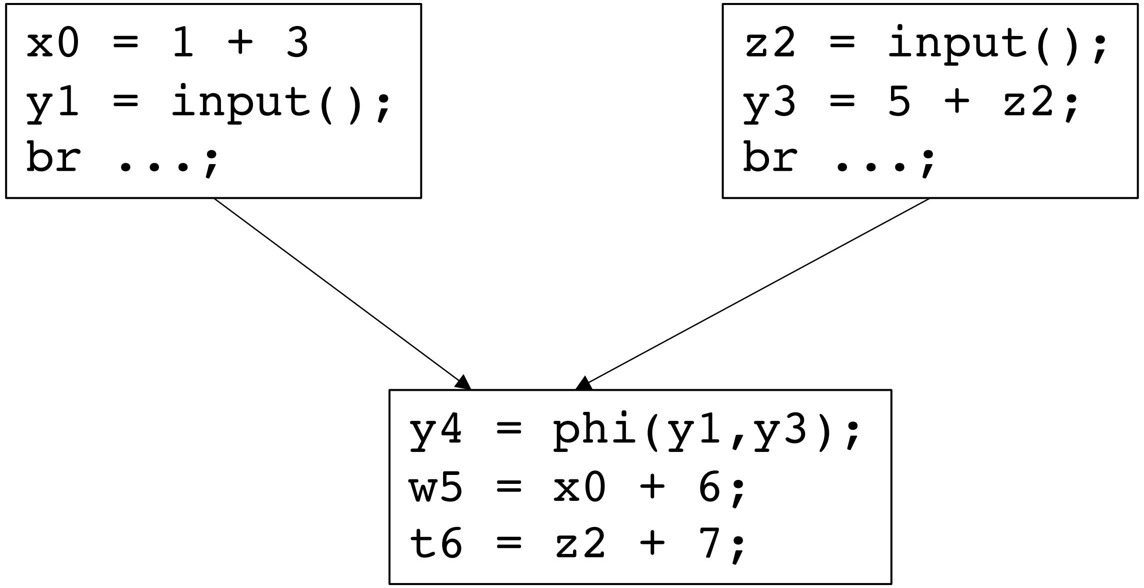
- a constant c_i if the value can be known
- $\perp$ if the value comes from an argument or input
- $\top$ otherwise, e.g. if the value comes from a ϕ node

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
}
```

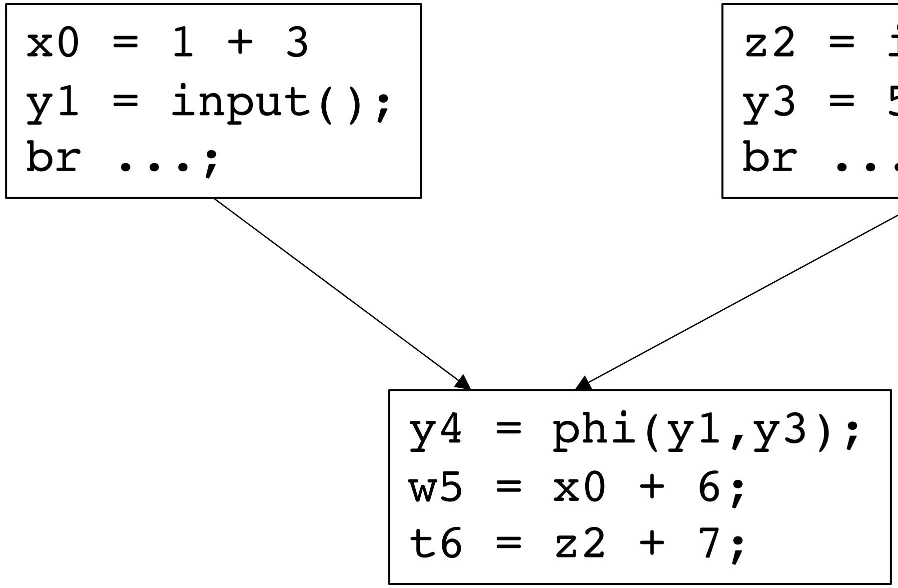


```
x0 = 1 + 3
y1 = input();
br ...;
```

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

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

Example:



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

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

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

```
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:

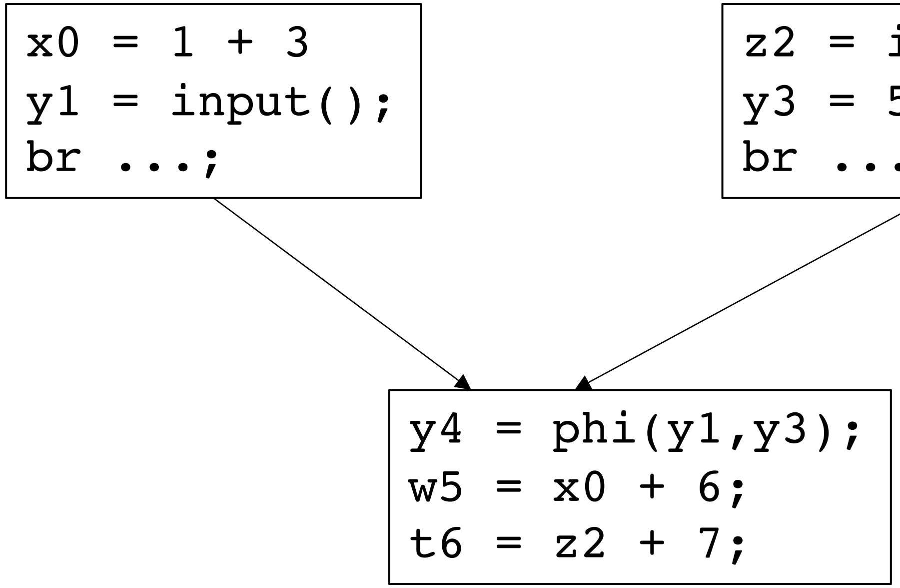
All variables **NOT** assigned to T get put on a worklist

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:



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

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

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

Worklist: `[x0,y1,z2]`

```
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 : []
}
```

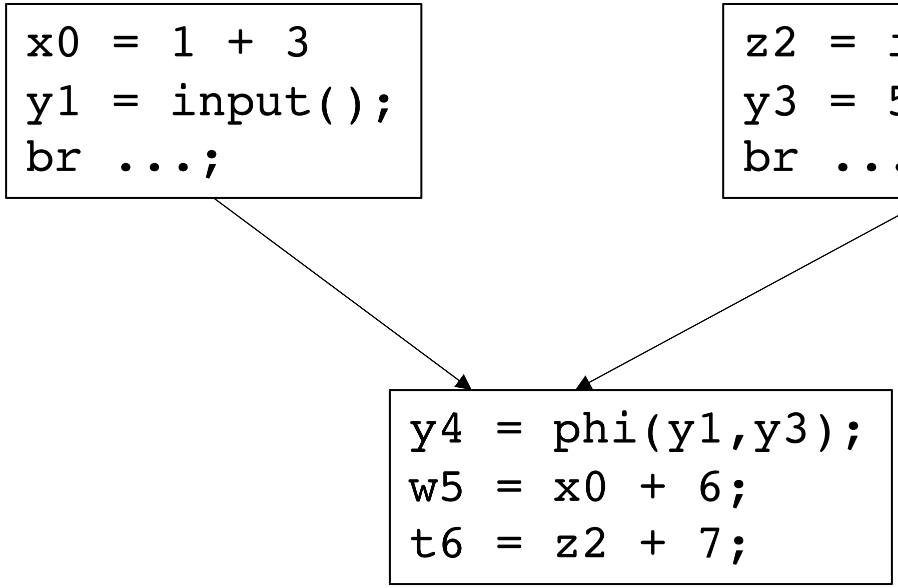
Constant propagation algorithm

evaluate m over the lattice (unique to each optimization)

Example: *m = n*x*

```
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:



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

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

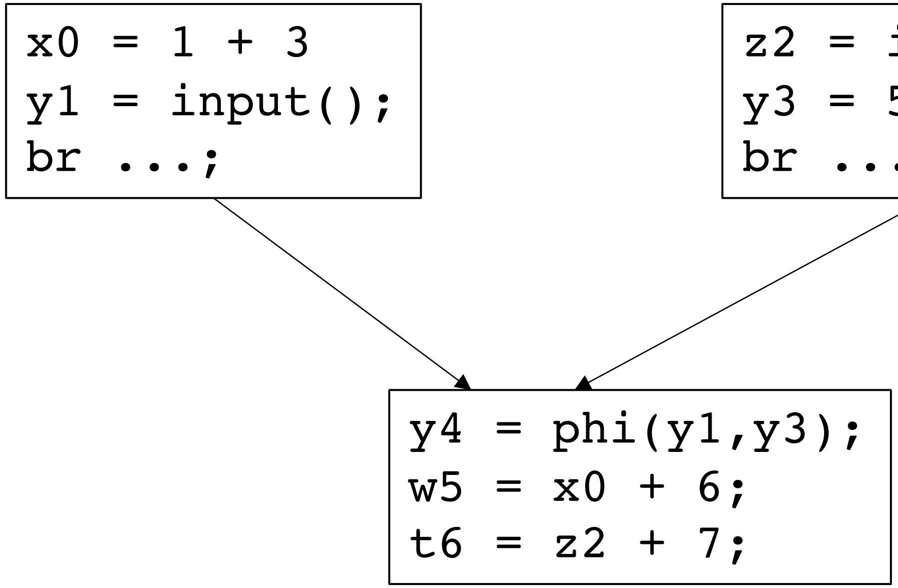
```
y4 = phi(y1,y3);
w5 = x0 + 6;
t6 = z2 + 7;
```

Worklist: `[x0,y1,z2]`

```
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 : []
}
```

Example:



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

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

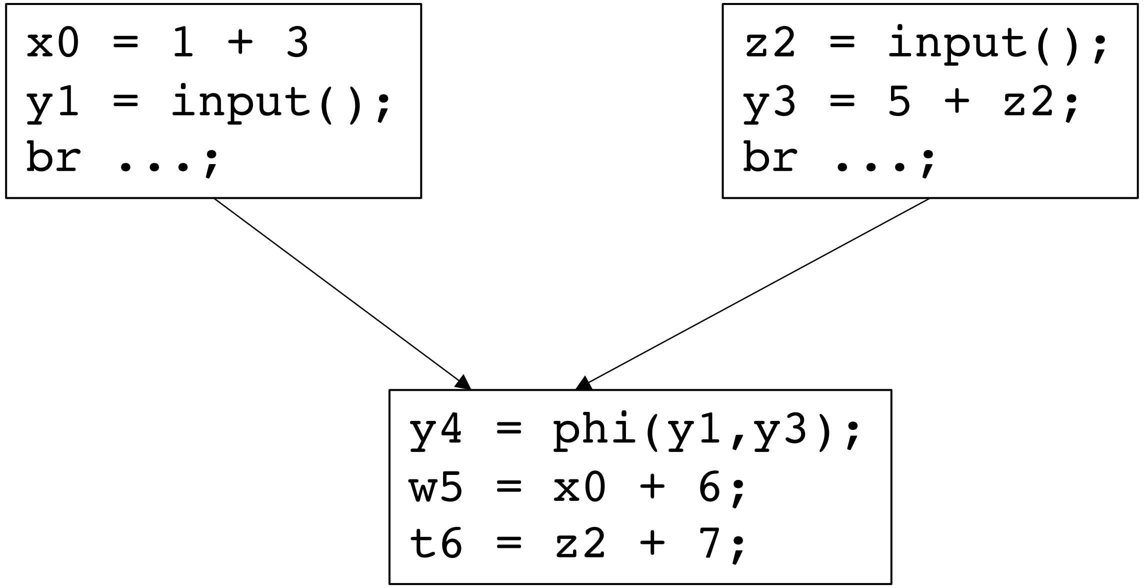
```
y4 = phi(y1,y3);
w5 = x0 + 6;
t6 = z2 + 7;
```

Worklist: `[x0,y1,z2]`

```
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 : []
}
```

Example:



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

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

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

Worklist: `[y1,z2,w5]`

```
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 : []
}
```

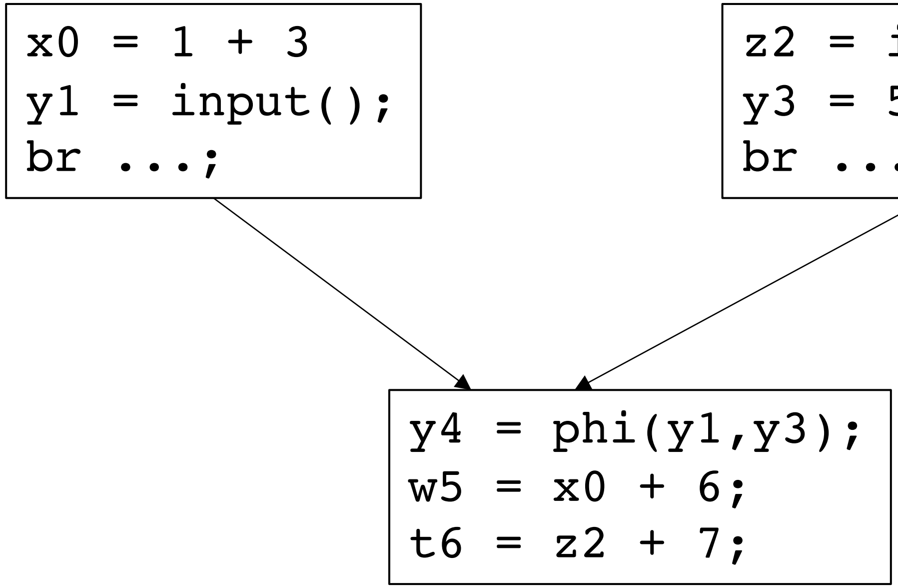
Constant propagation algorithm

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

Example: *m = n*x*

```
// Next case

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:



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

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

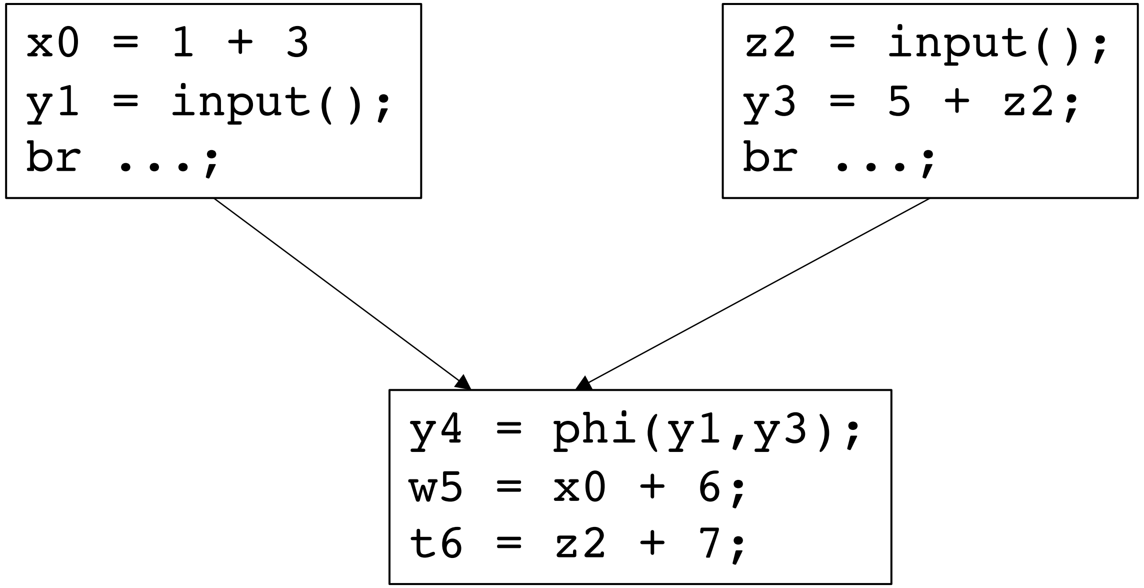
```
y4 = phi(y1,y3);
w5 = x0 + 6;
t6 = z2 + 7;
```

Worklist: `[x0,y1,z2]`

```
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 : []
}
```

Example:



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

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

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

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

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

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

Constant propagation algorithm

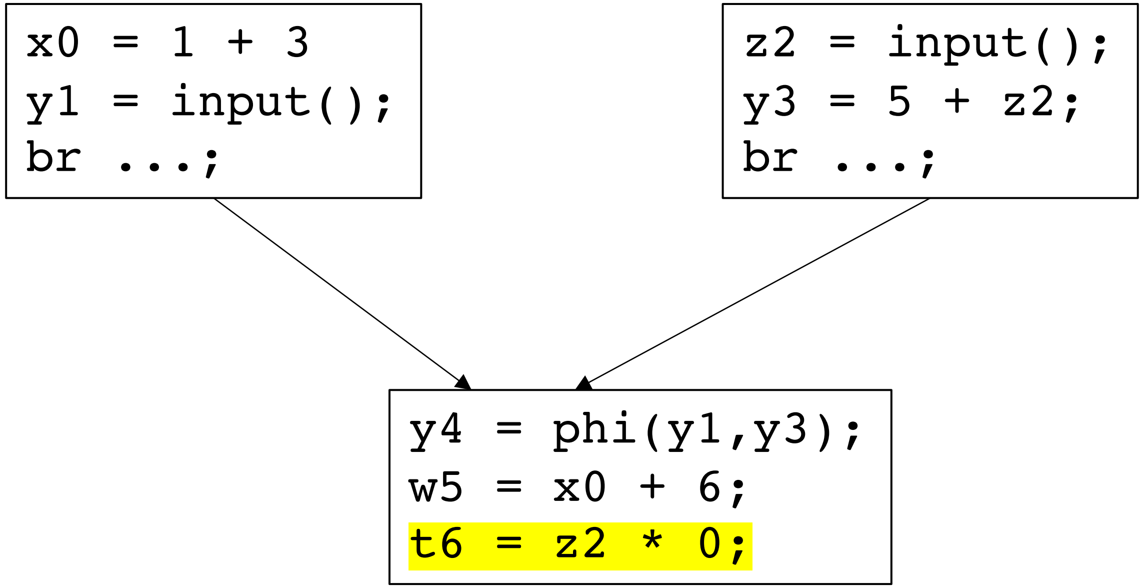
evaluate m over the lattice (unique to each optimization)

Example: *m = n*x*

```
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 + z2;
br ...;
```

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

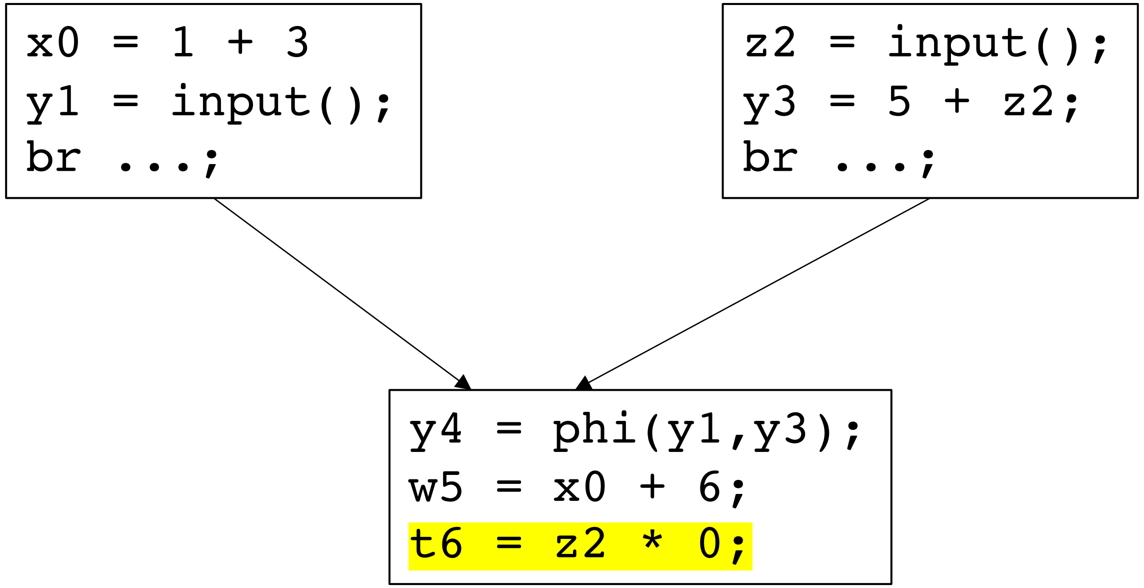
```
Value {
  x0 : 4
  y1 : B
  z2 : B
  y3 : B
  y4 : T
  w5 : T
  t6 : T
}
```

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

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

Example:

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



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

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

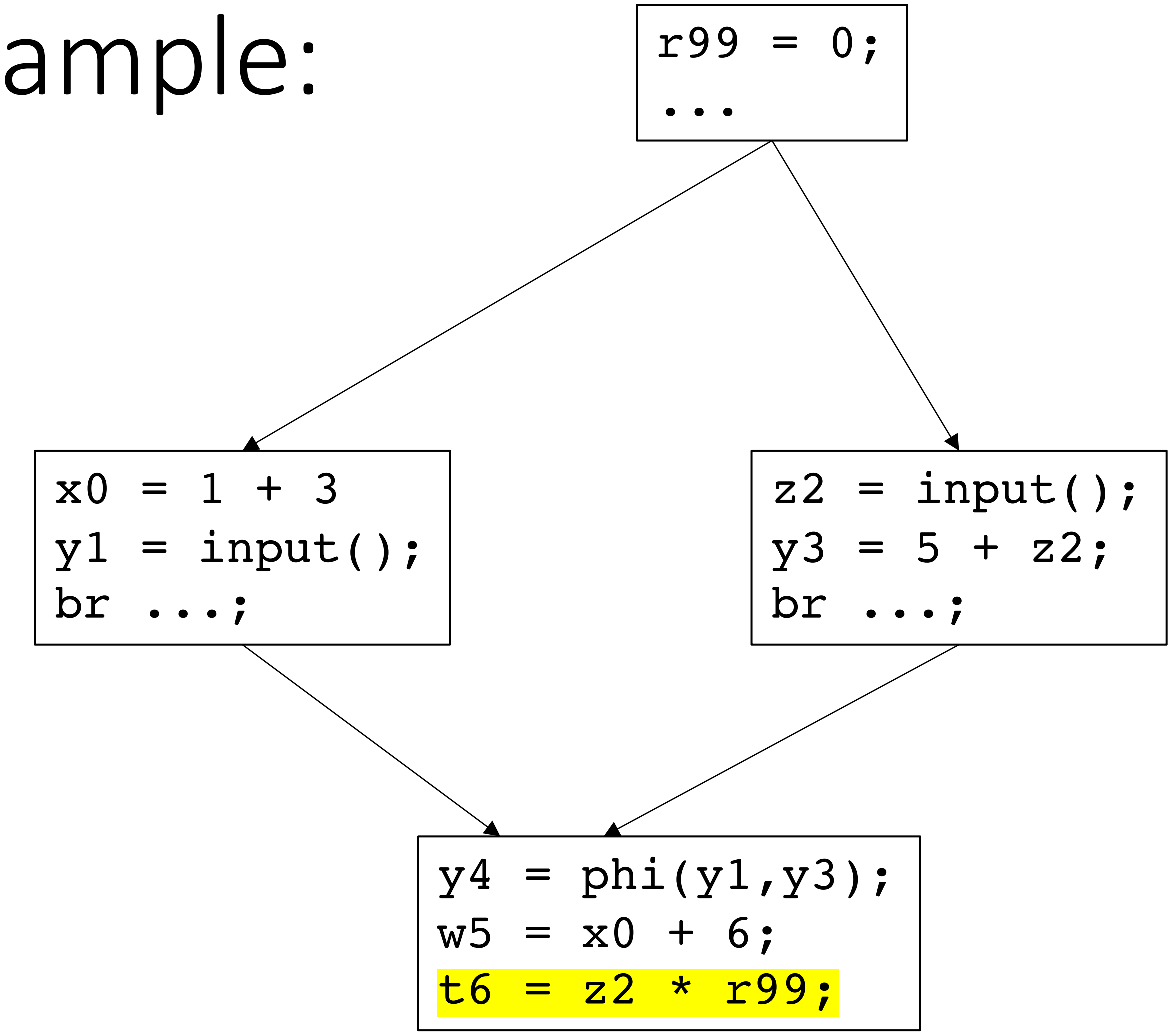
```
y4 = phi(y1,y3);
w5 = x0 + 6;
t6 = z2 * 0;
```

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

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

Can’t this be done
at the expression level?

Example:



```
r99 = 0;
...
```

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

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

```
y4 = phi(y1,y3);
w5 = x0 + 6;
t6 = z2 * r99;
```

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

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

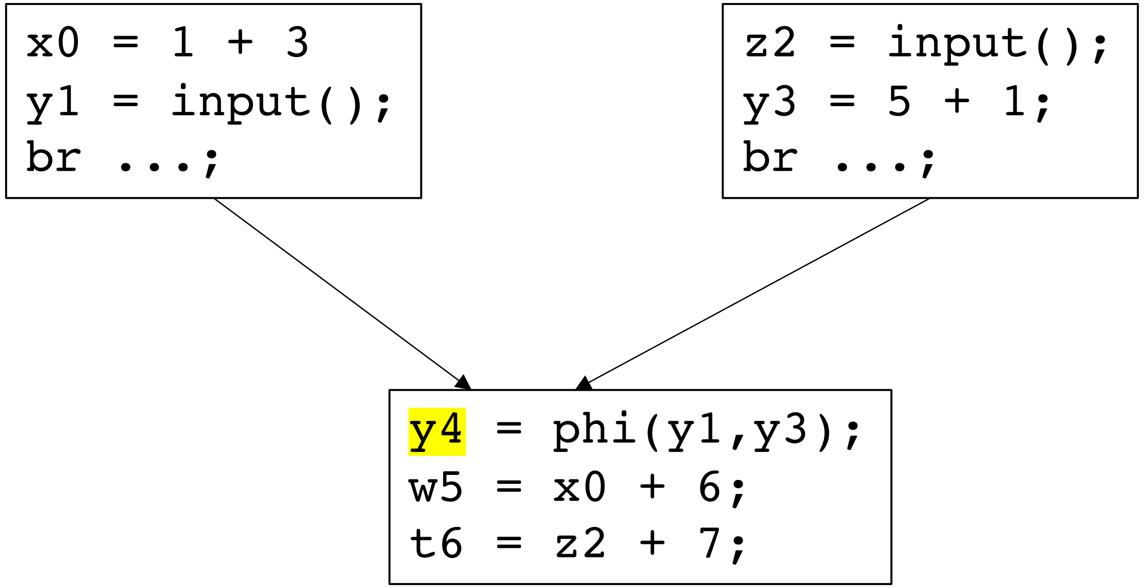
Worklist: `[x0,y1,z2,y3]`

Can’t this be done
at the expression level?

The elephant in the room

…

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 : T
   t6 : T
 }
```

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

Constant propagation algorithm

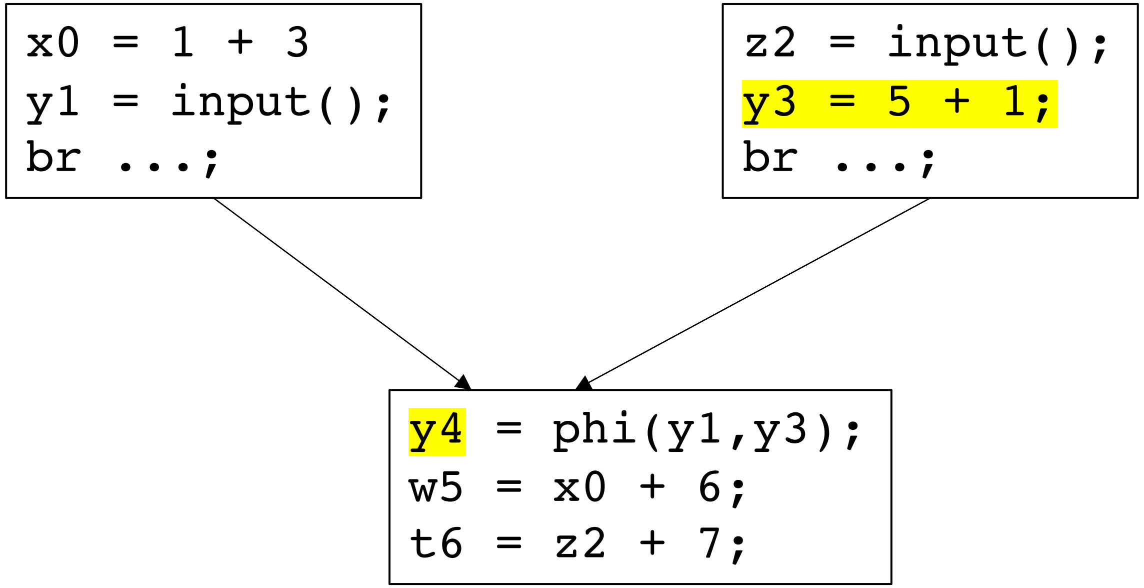
evaluate m over the lattice:

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

Example:



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

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

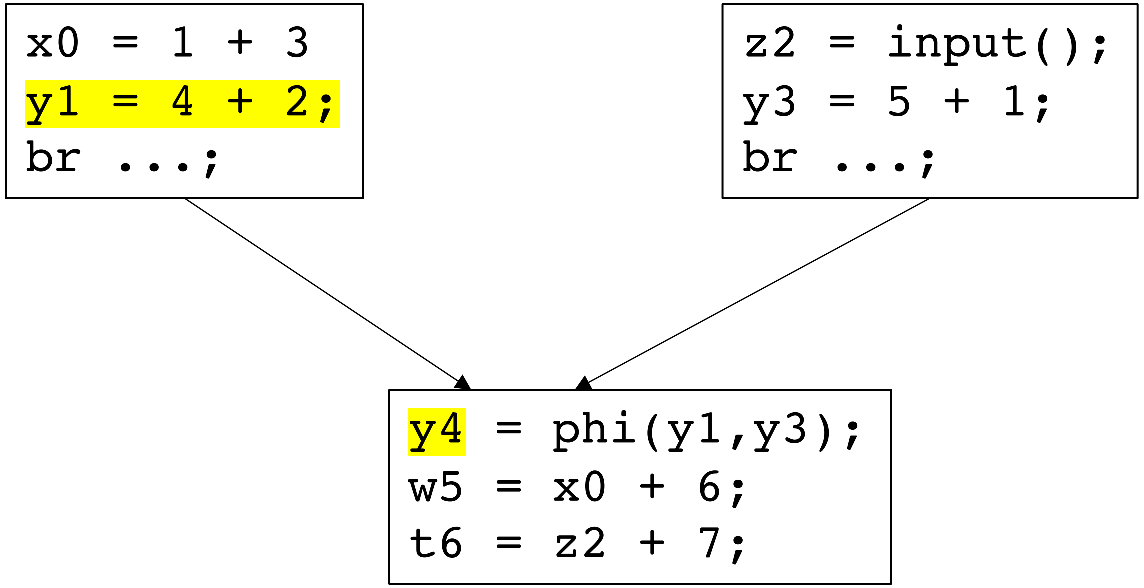
```
y4 = phi(y1,y3);
w5 = x0 + 6;
t6 = z2 + 7;
```

Worklist: `[x0,y1,y3]`

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

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

Example:



```
x0 = 1 + 3
y1 = 4 + 2;
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 : 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 Value(m) is not T and Value(m) has changed, then add m to the worklist

Constant propagation algorithm

evaluate m over the lattice:

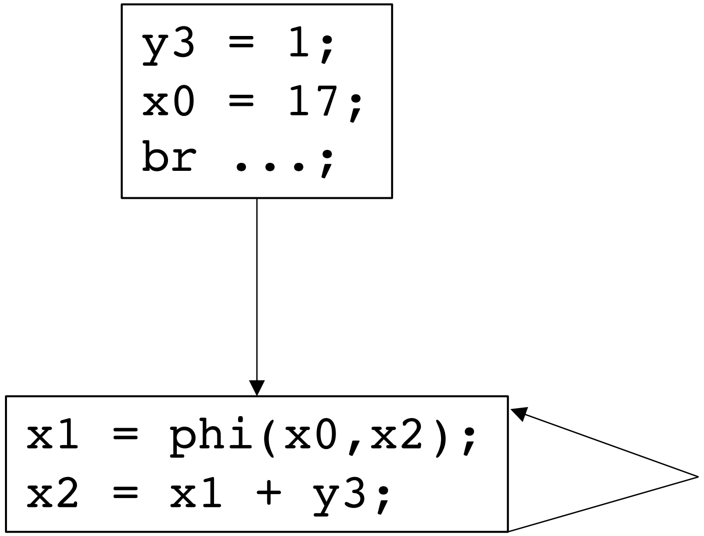
Issue here: potentially assigning a value that might not hold

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:



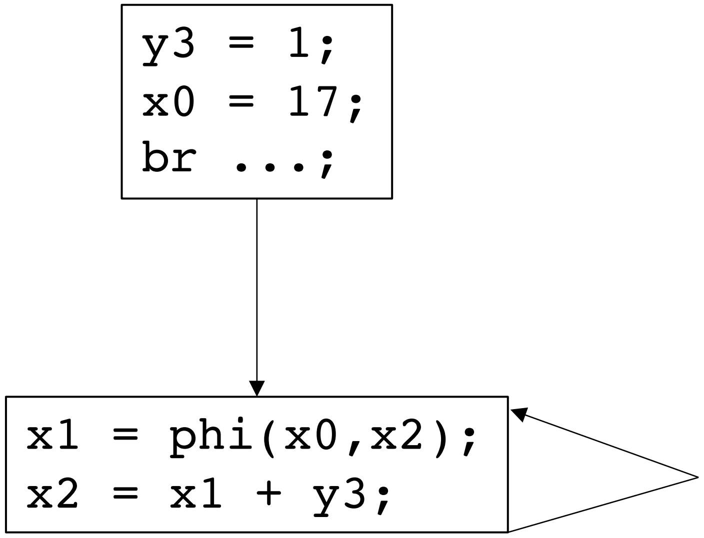
```
y3 = 1;
x0 = 17;
br ...;
```

```
x1 = phi(x0,x2);
x2 = x1 + y3;
```

Values
y3:
x0:
x1:
x2:

Uses
y3:
x0:
x1:
x2:

Example loop:



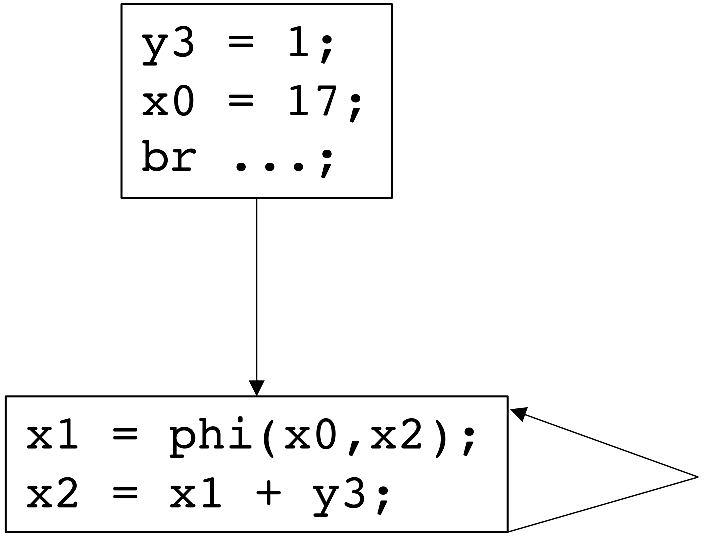
```
y3 = 1;
x0 = 17;
br ...;
```

```
x1 = phi(x0,x2);
x2 = x1 + y3;
```

Values
y3: 1
x0: 17
x1: T
x2: T

Uses
y3: [x2]
x0: [x1]
x1: [x2]
x2: [x1]

Example loop:



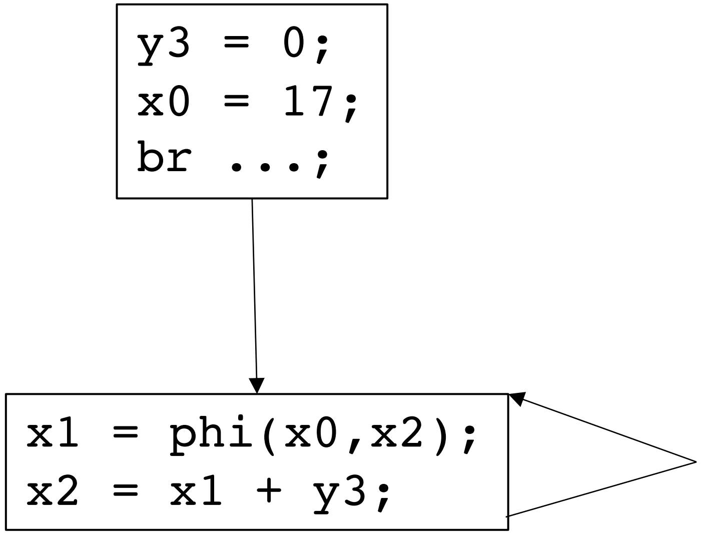
```
y3 = 1;
x0 = 17;
br ...;
```

```
x1 = phi(x0,x2);
x2 = x1 + y3;
```

Values
y3: 1
x0: 17
x1: B
x2: B

Uses
y3: [x2]
x0: [x1]
x1: [x2]
x2: [x1]

Example loop:



```
y3 = 0;
x0 = 17;
br ...;
```

```
x1 = phi(x0,x2);
x2 = x1 + y3;
```

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

Implementation: Assign unknowns to TOP

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

Implementation: Assign unknowns to BOTTOM

Pros/cons?

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:

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

For Loop unrolling

take the symbols to be integers

Simple meet operations for integers:

```
if $c_i$ != $c_j$ :
   $c_i \wedge c_j = \perp$

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: ∧

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

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

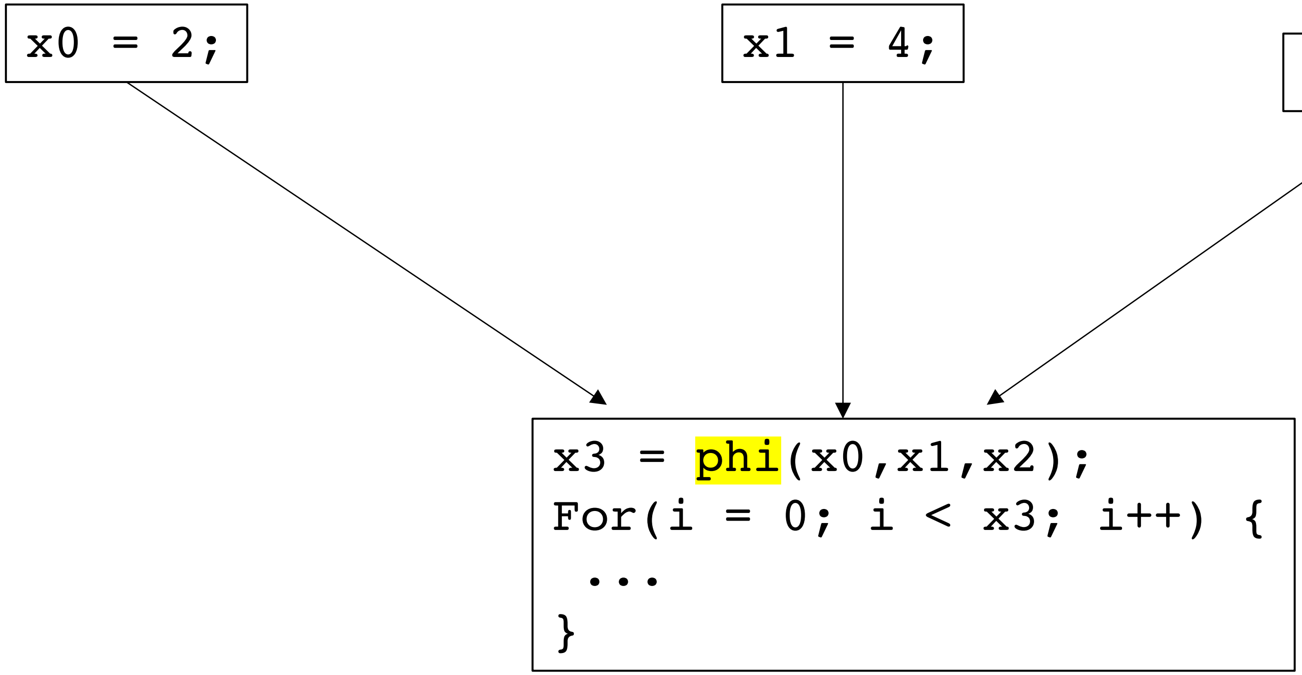
For Loop unrolling

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?



```
x0 = 2;
```

```
x1 = 4;
```

```
x2 = 6;
```

```
x3 = phi(x0,x1,x2);
For(i = 0; i < x3; i++) {
 ...
}
```

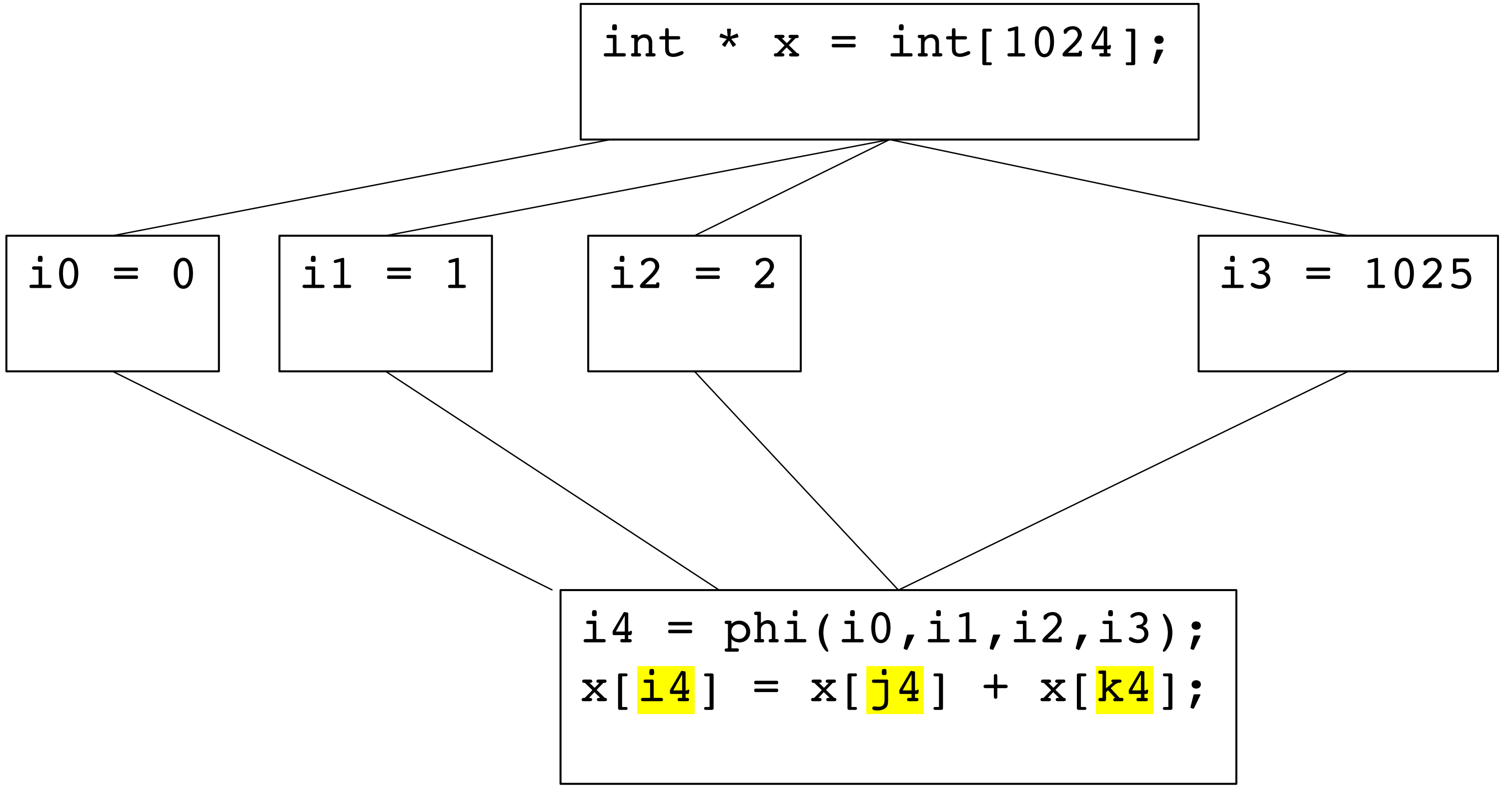
Another lattice

- Value ranges

Track if `i` are guaranteed to be between 0 and 1024.

Meet operator takes a union of possible values, ranges.

Values:
i0: [0]
i1: [1]
i2: [2]
i3: [1024]
i4: [[0-2], 1025]



```
int * x = int[1024];
```

```
i0 = 0
```

```
i1 = 1
```

```
i2 = 2
```

```
i3 = 1025
```

```
i4 = phi(i0,i1,i2,i3);
x[i4] = x[j4] + x[k4];
```

Converting out of SSA

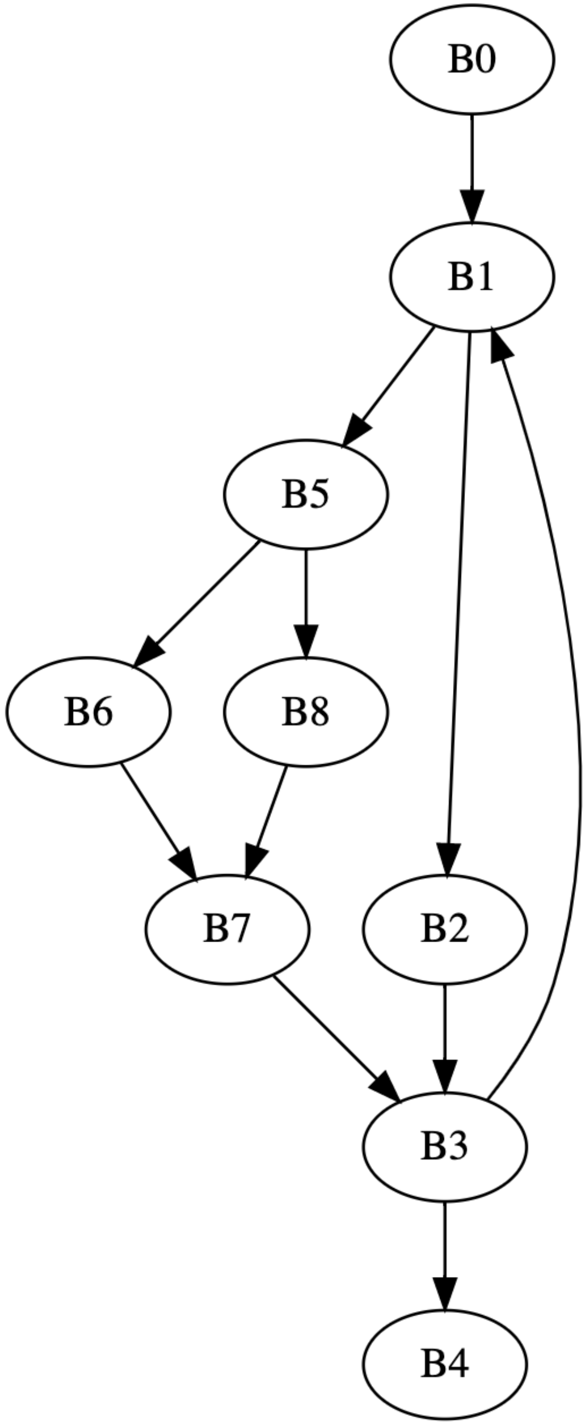
```
B0: i0 = ...;

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

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

B3: a10 = ϕ(...);
    b11 = ϕ(...);
    c12 = ϕ(...);
    d13 = ϕ(...);
    y14 = ...;
    z15 = ...;
    i16 = ...;
    br ... B1, B4;
```

```
B4: return

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

B6: d19 = ...;

B7: d20 = ϕ(...);
    c21 = ϕ(...);
    b22 = ...;

B8: c23 = ...;
    br B7;
```

Two approaches:

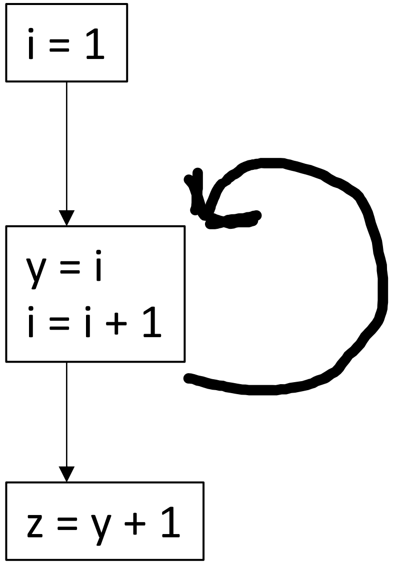
1. path tracking and conditionals
2. early assignment

Example using i in B1



Lost copy issue

An issue with early assignment algorithm



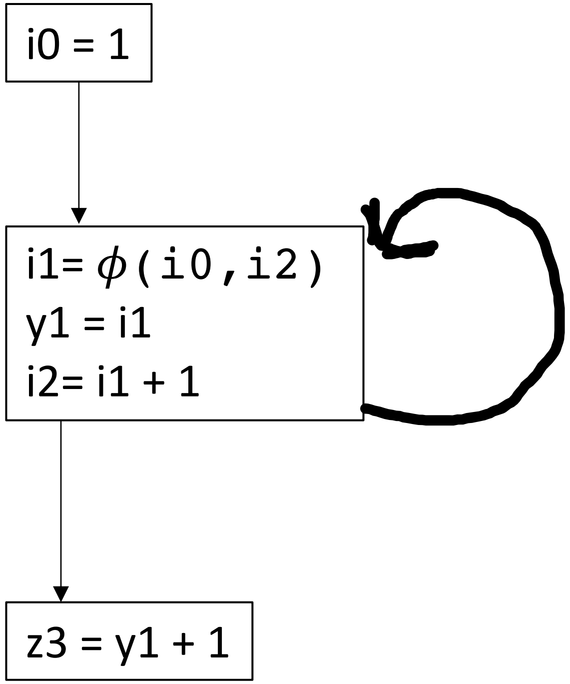
i = 1

y = i
i = i + 1

z = y + 1

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

Lost copy issue



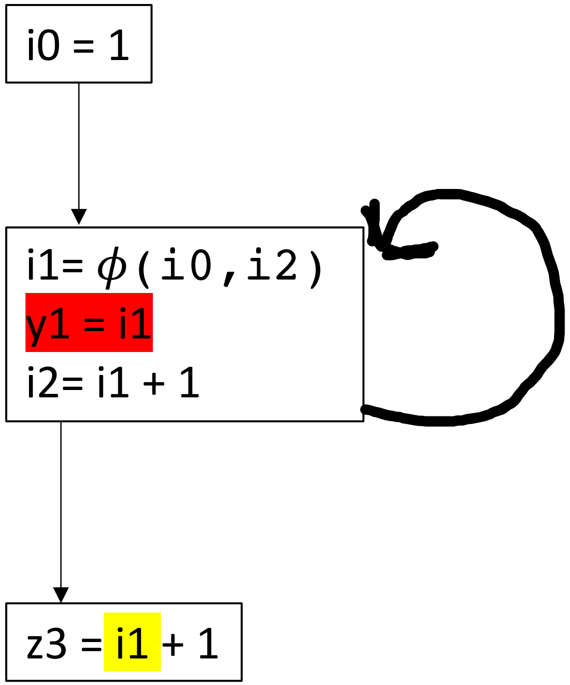
i0 = 1

i1= ϕ(i0,i2)
y1 = i1
i2= i1 + 1

z3 = y1 + 1

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

Lost copy issue



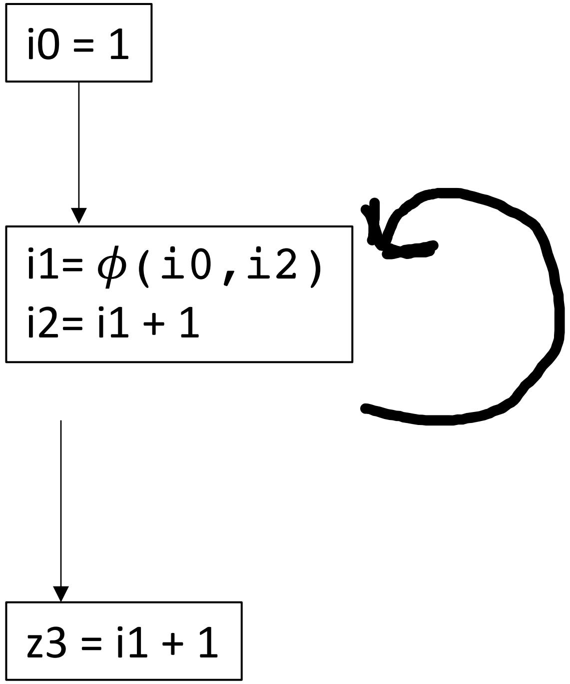
i0 = 1

i1= ϕ(i0,i2)
y1 = i1
i2= i1 + 1

z3 = i1 + 1

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

Lost copy issue



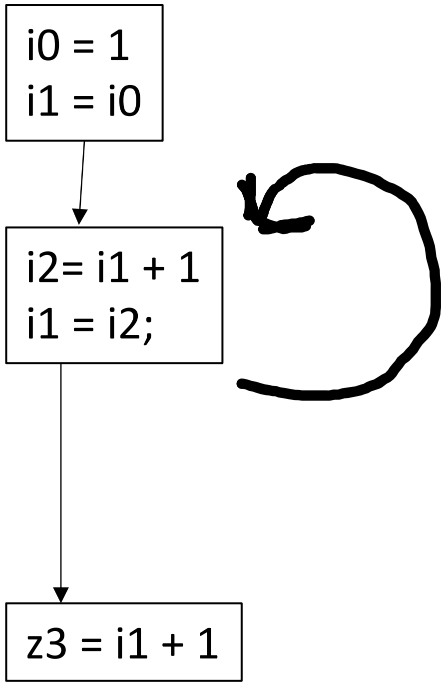
i0 = 1

i1= ϕ(i0,i2)
i2= i1 + 1

z3 = i1 + 1

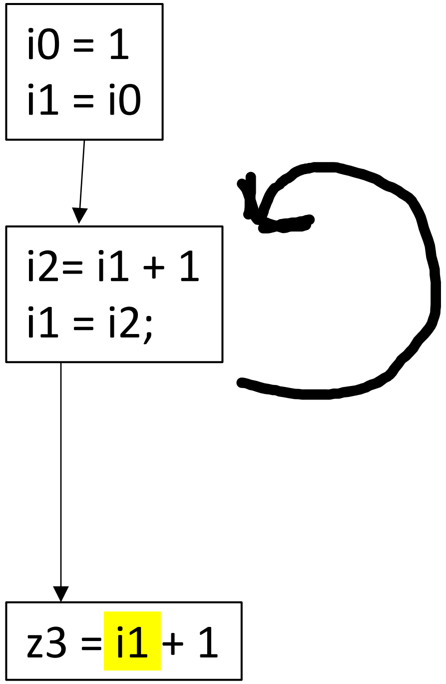
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue



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

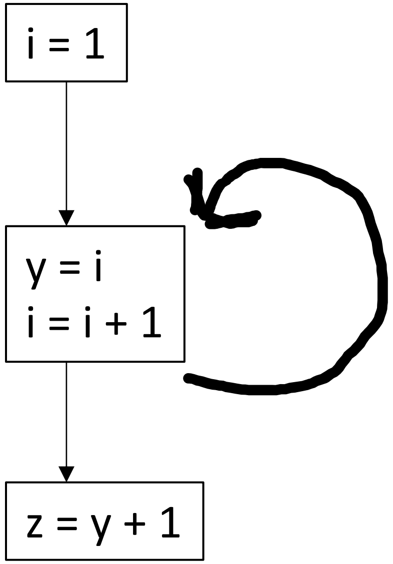
Lost copy issue



i0 = 1
i1 = i0

i2= i1 + 1
i1 = i2;

z3 = i1 + 1



i = 1

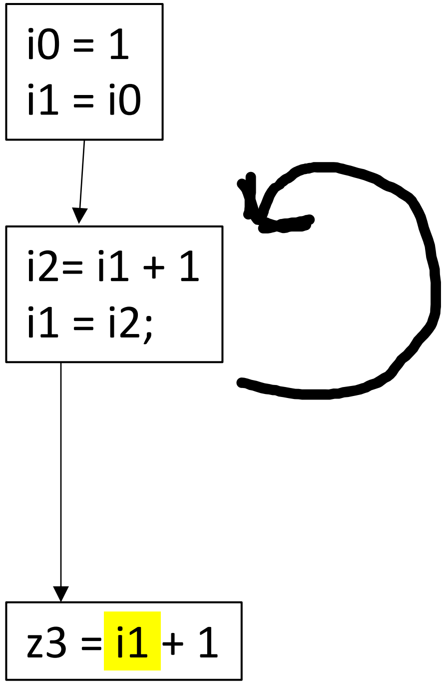
y = i
i = i + 1

z = y + 1

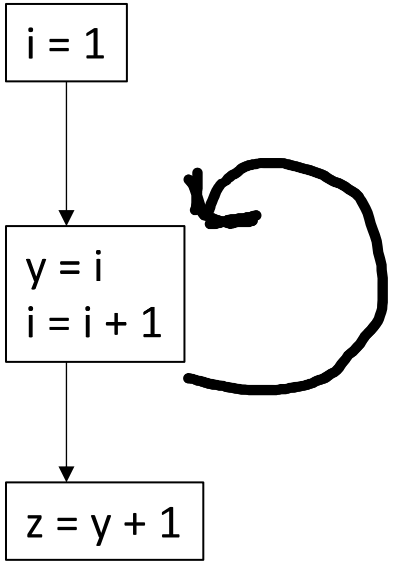
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue

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



```
i0 = 1
i1 = i0

i2= i1 + 1
i1 = i2;

z3 = i1 + 1
```



```
i = 1

y = i
i = i + 1

z = y + 1
```

Similar problem called the Swap problem

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

Hopefully see you in person on Thursday!

- Starting Module 3: DSLs and Parallelism
- Office hours tomorrow