CSE211: Compiler Design

Nov. 1, 2023

- **Topic**: More flow analysis applications and intro to SSA

- **Questions**:
  - *What is SSA form?*
  - *Has anyone heard of the phi instruction?*

```
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

- Paper assignment was due on Monday
  - Start thinking about your next paper assignment!
  - It is due on the day of the final! Don’t let things pile up!

- Homework 2 is out
  - Please have a partner by the end of day today (20% off and doing the assignment solo)
  - Due Nov. 13

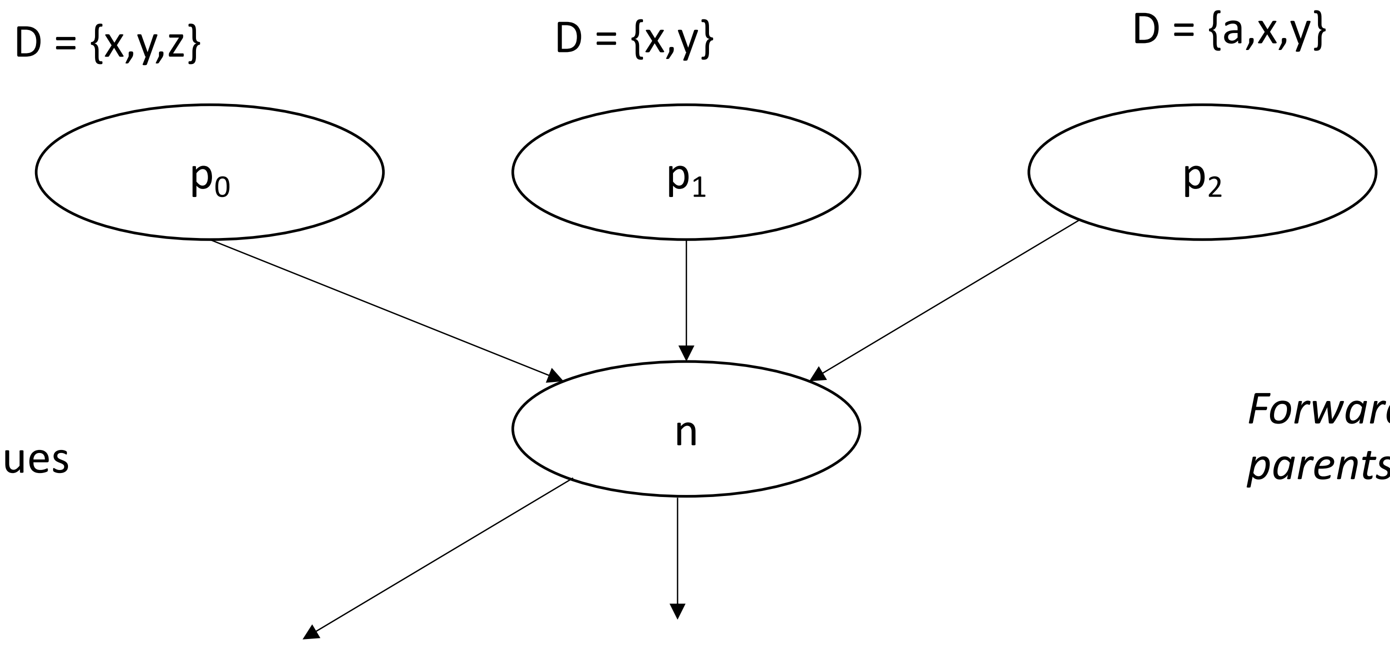
Announcements

- We are working on grading your assignments ASAP.
  Stay tuned!

Review global optimizations

Global optimizations review: Dominance

- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



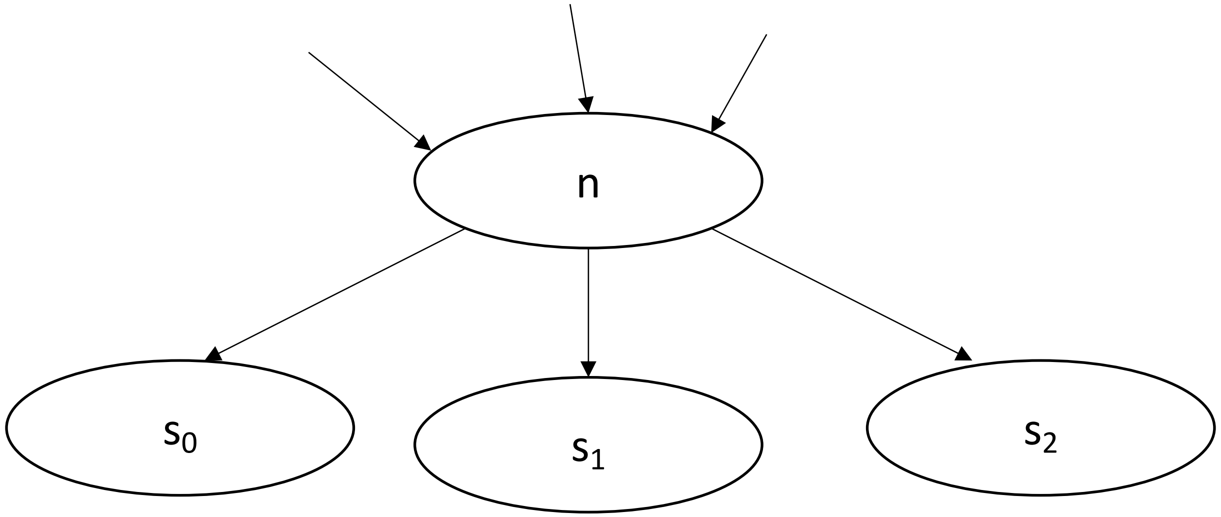
update:
intersection of parent values

Forward flow, as updates flow from parents to children.

$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in } preds(n)} Dom(p) \right)$$

Global optimizations review: Live variable analysis

$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}) \right)$$



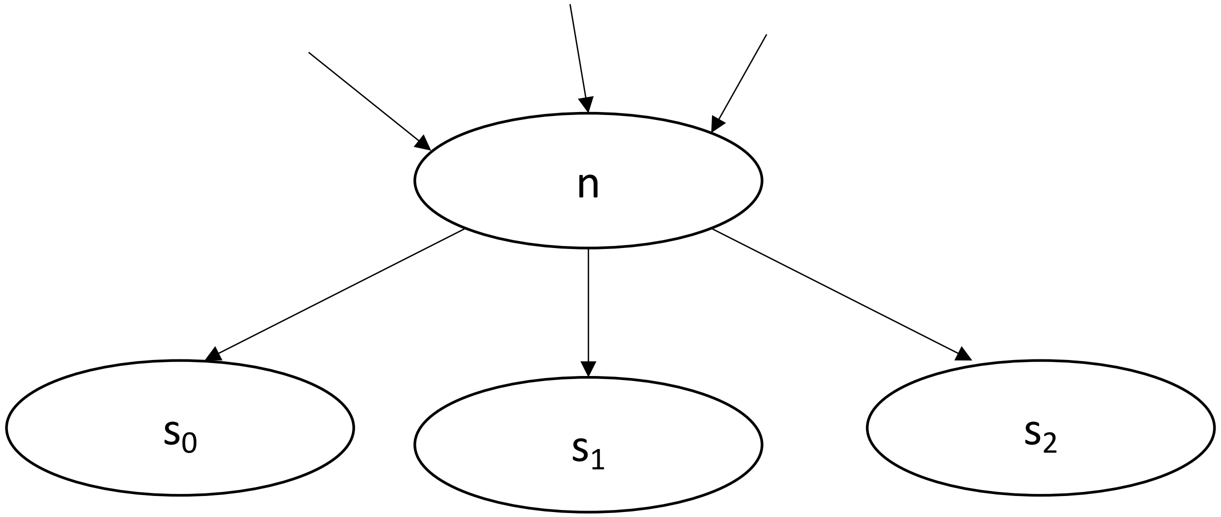
backwards flow

$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in } preds(n)} Dom(p) \right)$$

Global optimizations review: Live variable analysis

$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\text{LiveOut}(s) \cap \overline{VarKill(s)}\right)\right)$$

What are the sets?

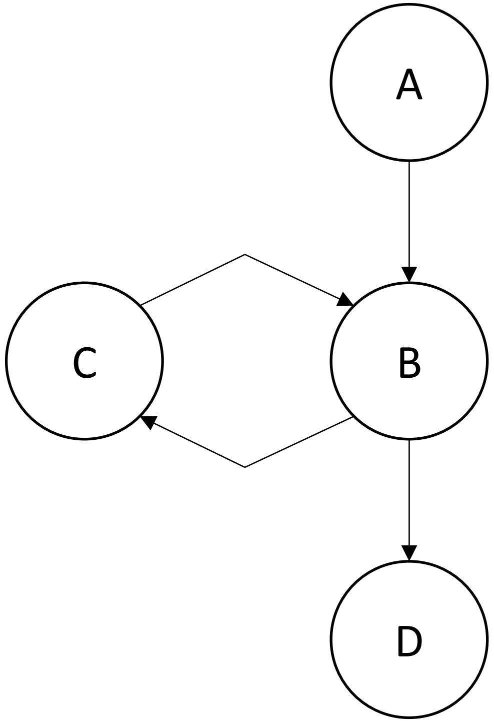


backwards flow

$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in } preds(n)} Dom(p) \right)$$

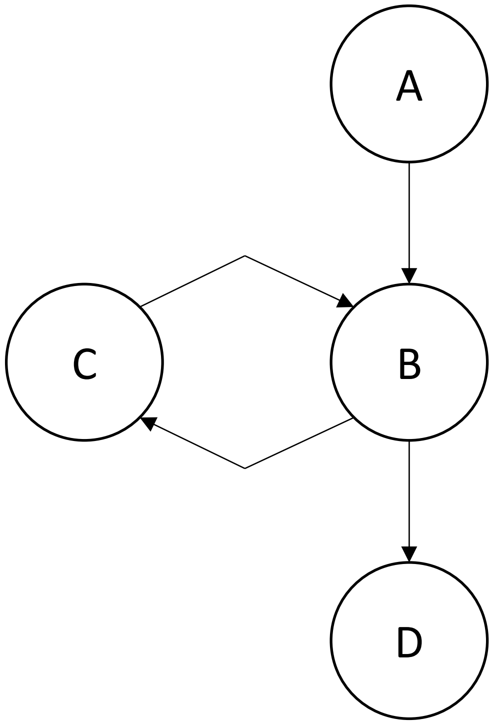
Example

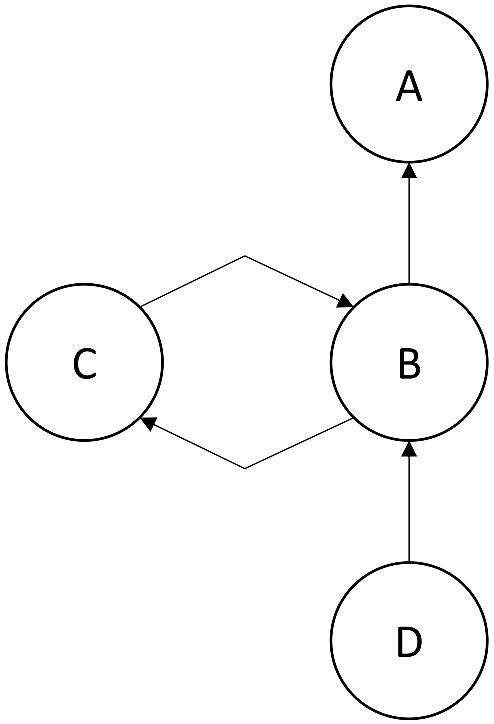
post order: D, C, B, A



acks: thanks to this blog post for the example!
https://eli.thegreenplace.net/2015/directed-graph-traversal-orderings-and-applications-to-data-flow-analysis/

Example



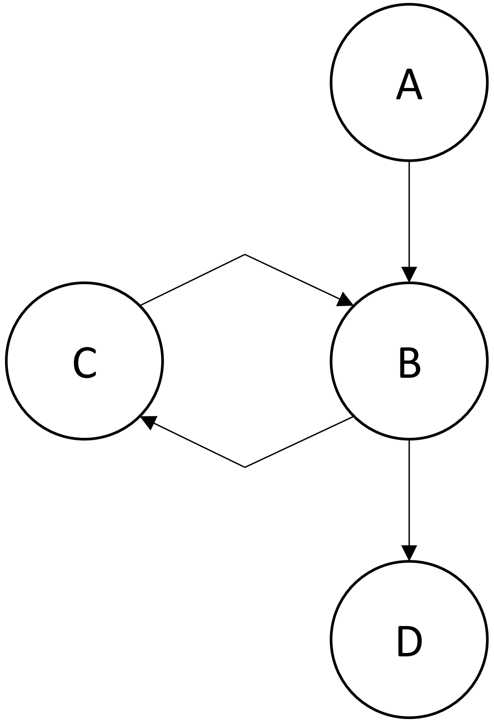


reverse CFG

post order: D, C, B, A

rpo on reverse CFG: D, B, C, A

Example

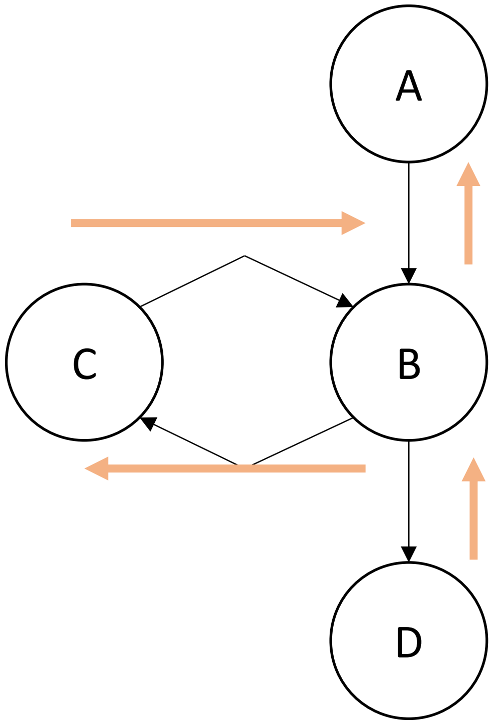


post order: D, C, B, A

rpo on reverse CFG: D, B, C, A

rpo on reverse CFG computes B before C, thus, C can see updated information from B

Example



updates in backwards flow

post order: D, C, B, A

rpo on reverse CFG: D, B, C, A

rpo on reverse CFG computes B before C, thus, C can see updated information from B

Live variable limitations

To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten.

Consider:

```
s = a[x] + 1;
```

Live variable limitations

To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten.

Consider:

```
s = a[x] + 1;
```

UEVar *needs to assume a[x] is any memory location that it cannot prove non-aliasing*

$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(LiveOut(s) \cap \overline{VarKill(s)}\right)\right)$$

Live variable limitations

To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten.

Consider:

```
a[x] = s + 1;
```

$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\text{LiveOut}(s) \cap \overline{VarKill(s)}\right)\right)$$

Live variable limitations

To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten.

Consider:

```
a[x] = s + 1;
```

VarKill *also needs to know about aliasing*

$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\text{LiveOut}(s) \cap \overline{VarKill(s)}\right)\right)$$

Live variable limitations

Imprecision can come from CFG construction:

consider:

```
br 1 < 0, dead_branch, alive_branch
```

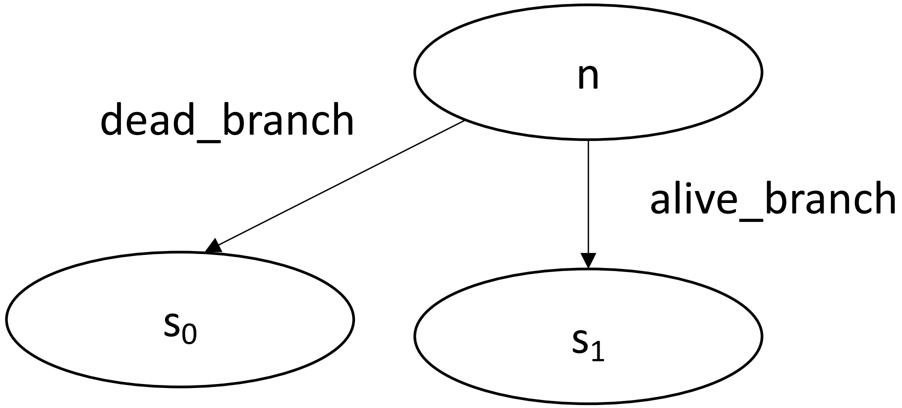
Live variable limitations

Imprecision can come from CFG construction:

consider:

```
br 1 < 0, dead_branch, alive_branch
```

could come from arguments, etc.



Live variable limitations

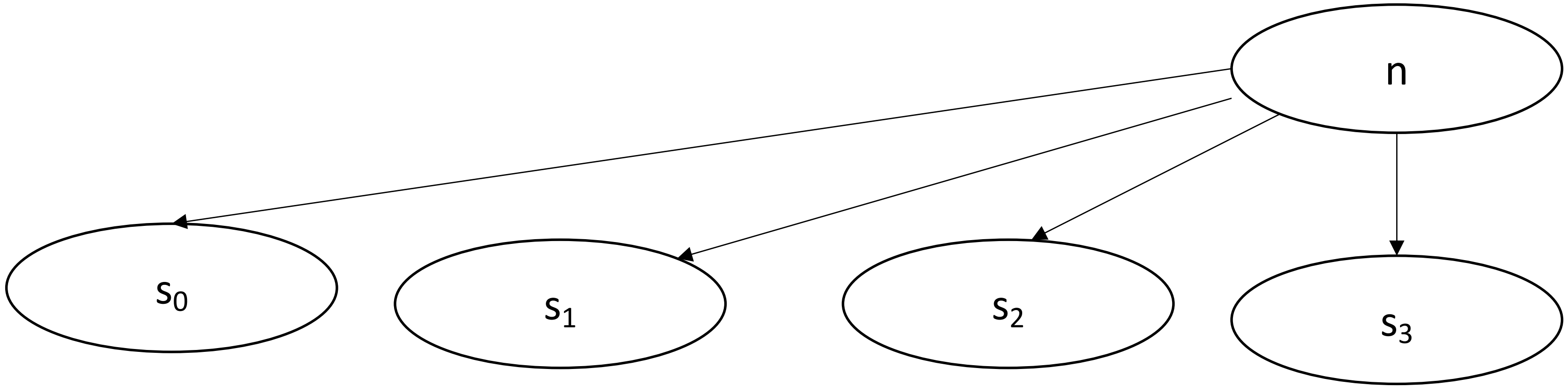
Imprecision can come from CFG construction:

consider first class labels (or functions):

```
br label_reg
```

where label_reg is a register that contains a register

need to branch to all possible basic blocks!



Finishing up global analysis

The Data Flow Framework

$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}\,) \right)$$

$$f(x) = Op_{v \text{ in } (succ \mid preds)}\, c_0(v)\ op_1\ (f(v)\ op_2\ c_2(v))$$

Available Expressions

$$AvailExpr(n) = \bigcap_{p \text{ in preds}} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

An expression e is "available" at the beginning of a basic block b_x if for all paths to b_x, e is evaluated and none of its arguments are overwritten

Available Expressions

$$AvailExpr(n) = \bigcap_{p \text{ in } preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

Forward Flow

Available Expressions

$$AvailExpr(n) = \bigcap_{p \text{ in preds}} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

intersection implies “must” analysis

Available Expressions

$$AvailExpr(n) = \bigcap_{p\ in\ preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

DEExpr(p) is all Downward Exposed Expressions in p. That is expressions that are evaluated AND operands are not redefined

Available Expressions

$$AvailExpr(n) = \bigcap_{p\ in\ preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

AvailExpr(p) is any expression that is available at p

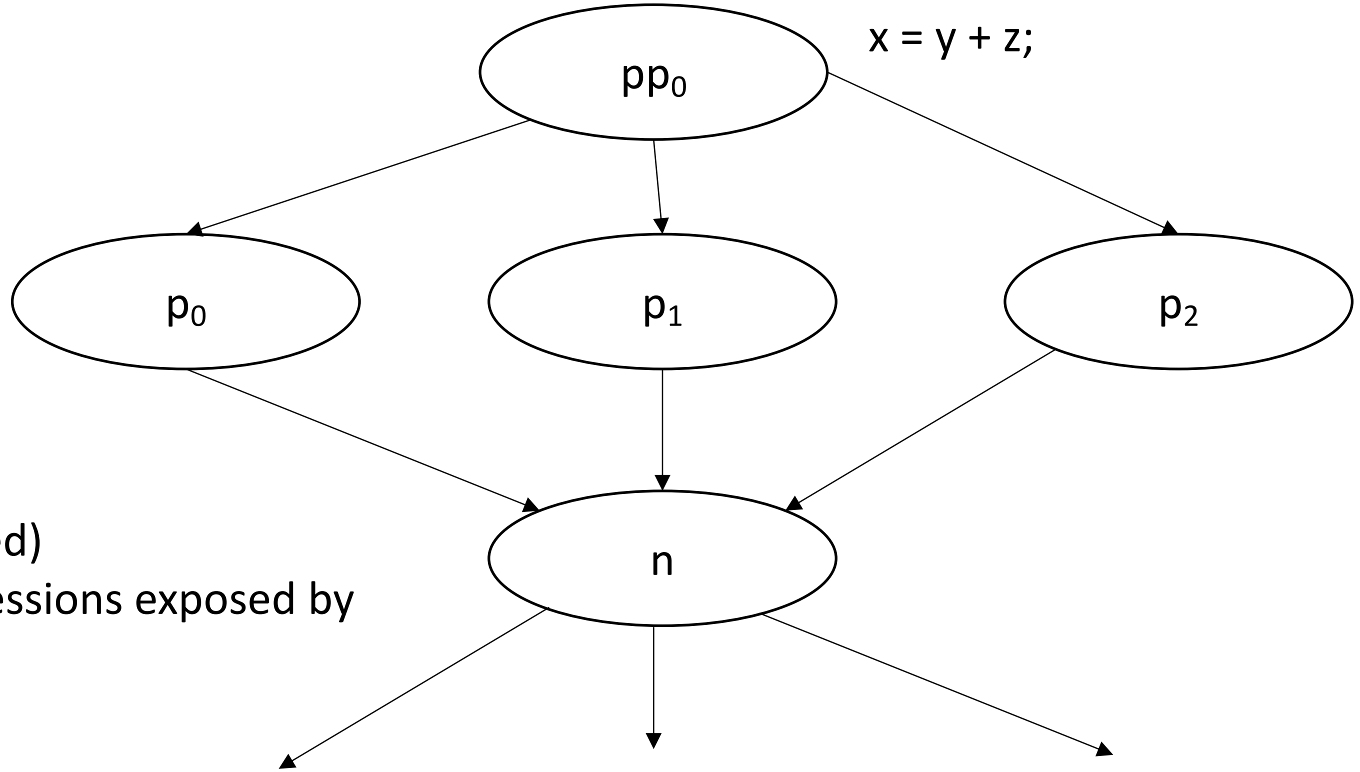
Available Expressions

$$AvailExpr(n) = \bigcap_{p \text{ in } preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

ExprKill(p) is any expression that p killed, i.e. if one or more of its operands is redefined in p

Available Expressions

$$AvailExpr(n)= \bigcap_{p \text{ in preds}} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$



Any expression
that is available (and not killed)
the parents, along with expressions exposed by
all the parents.

Available Expressions

$$AvailExpr(n) = \bigcap_{p\ in\ preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

Application*: you can add availExpr(n) to local optimizations in n, e.g. local value numbering*

Anticipable Expressions

$$AntOut(n) = \bigcap_{s\ in\ succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

An expression e is "anticipable" at a basic block b_x if for all paths that leave b_x, e is evaluated

Anticipable Expressions

$$AntOut(n) = \bigcap_{s\ in\ succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

Backwards flow

Anticipable Expressions

$$AntOut(n) = \bigcap_{s \text{ in } succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

"must" analysis

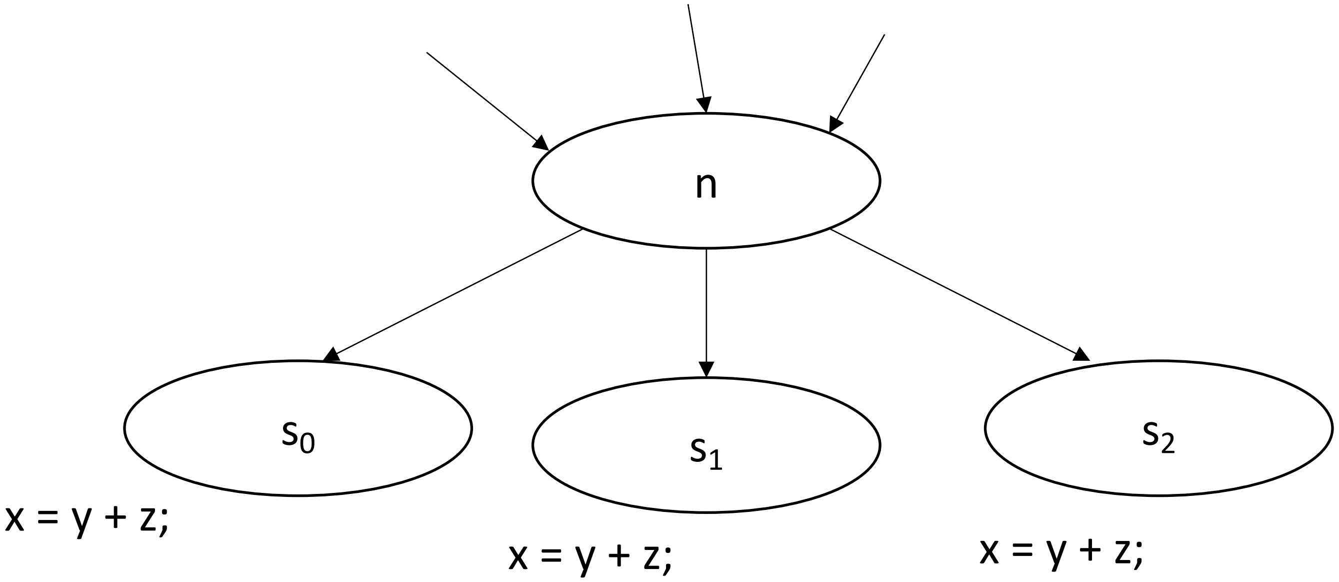
Anticipable Expressions

$$AntOut(n) = \bigcap_{s \in succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

UEExpr(p) is all Upward Exposed Expressions in p. That is expressions that are computed in p before operands are overwritten.

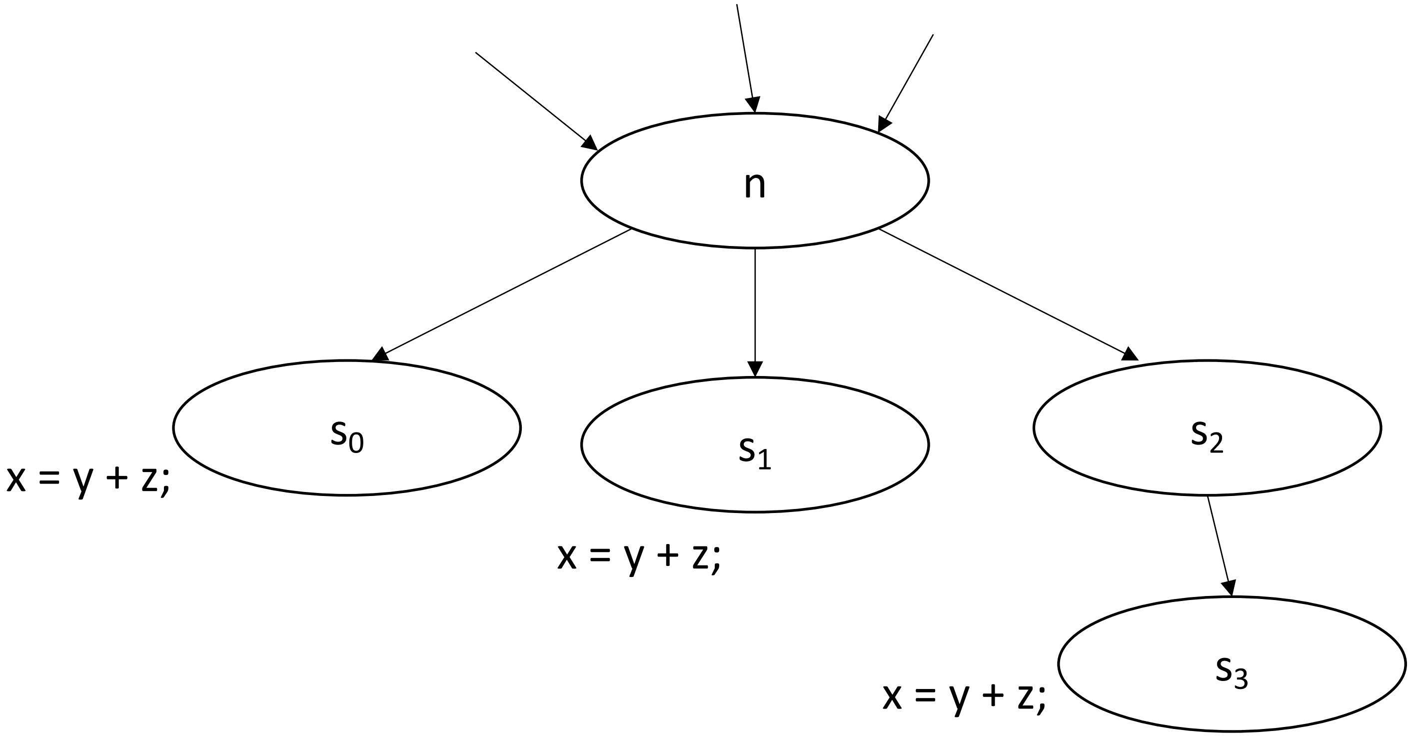
Anticipable Expressions

$$AntOut(n) = \bigcap_{s\ in\ succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$



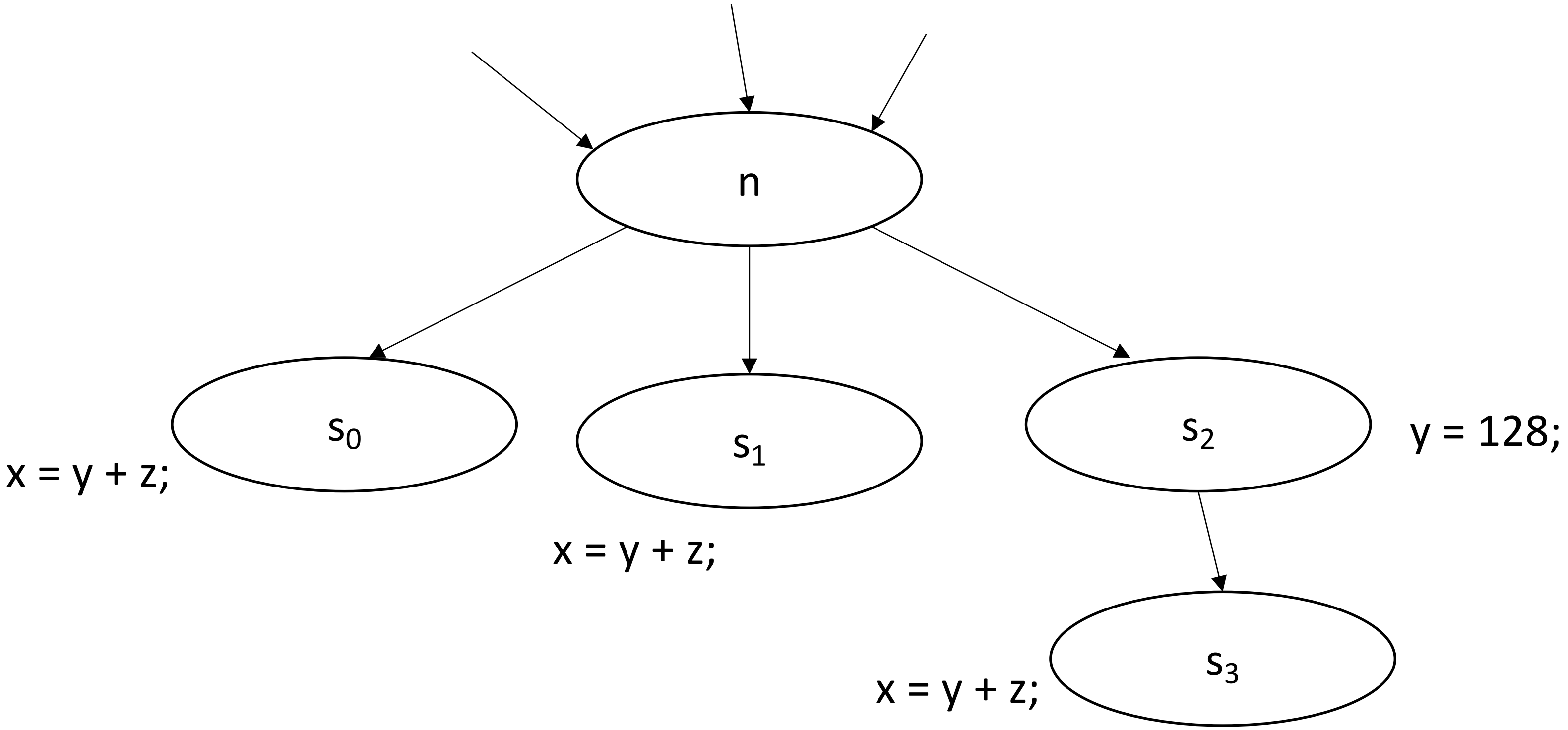
Anticipable Expressions

$$AntOut(n) = \bigcap_{s\ in\ succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$



Anticipable Expressions

$$AntOut(n) = \bigcap_{s \text{ in } succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$



Anticipable Expressions

$$AntOut(n) = \bigcap_{s \text{ in } succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

***Application**: you can hoist AntOut expressions to compute as early as possible*

potentially try to reduce code size: -Oz

More flow algorithms:

Check out chapter 9 in EAC: Several more algorithms.

“Reaching definitions” have applications in memory analysis

New material: 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
}
```

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

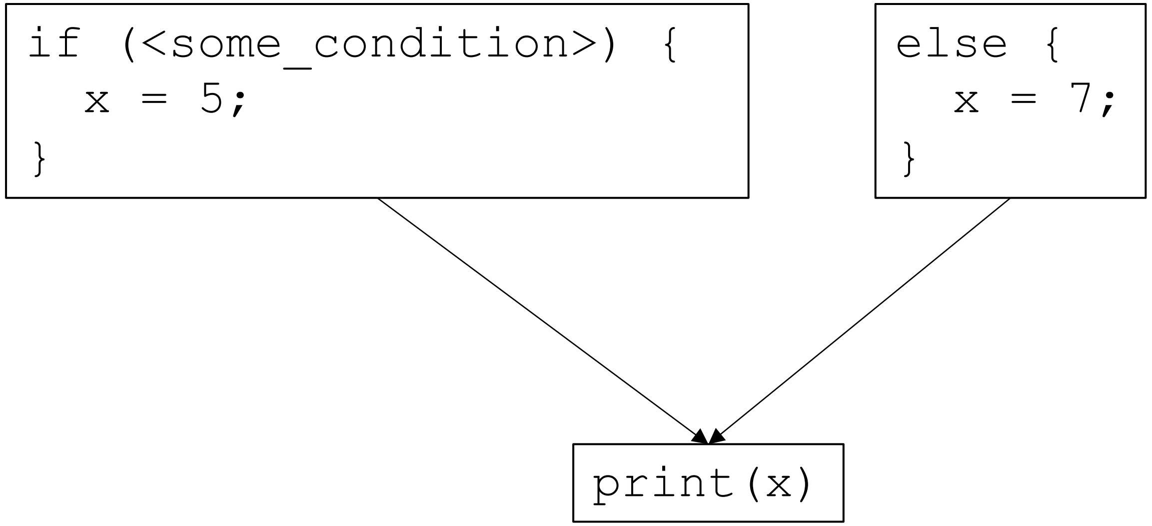
What here?

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

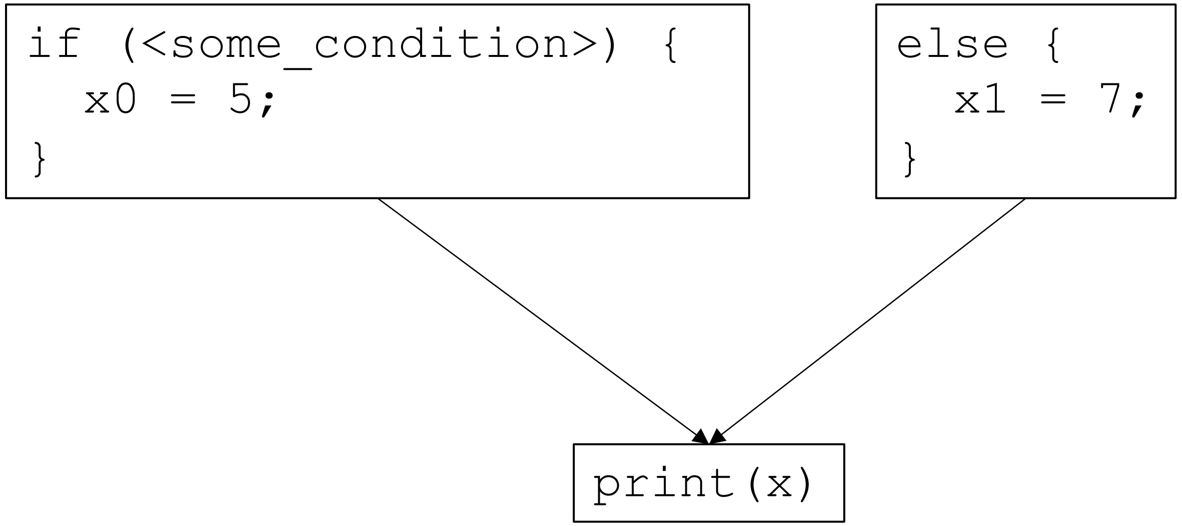


ϕ instructions

- Example: how to convert this code into SSA?

```
int x;

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

else {
  x1 = 7;
}

print(x)
```

number the variables

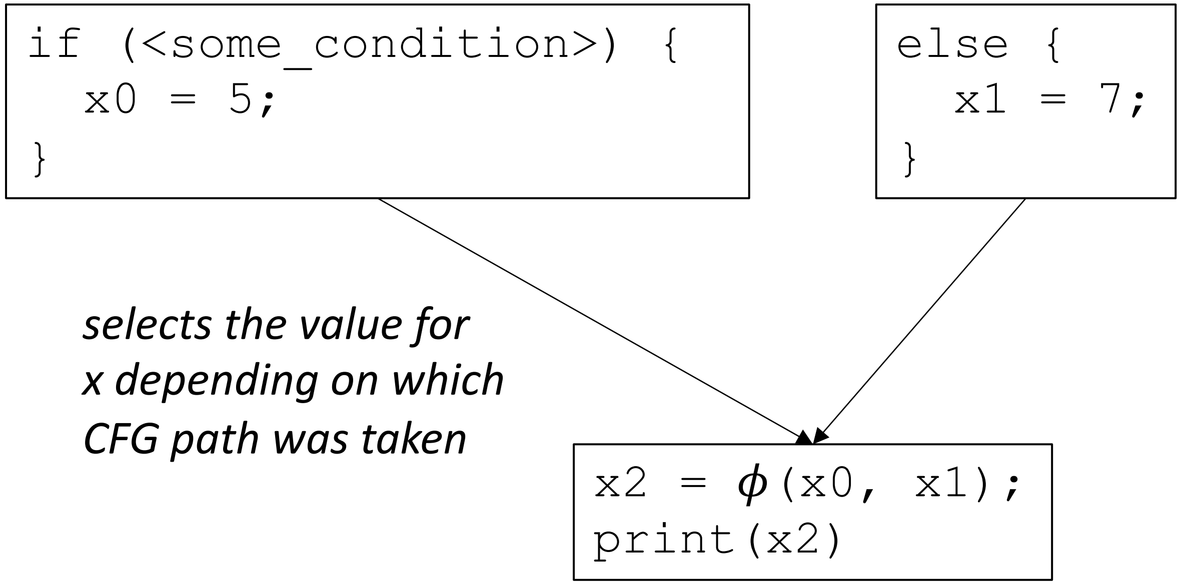


ϕ instructions

- 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

- $x_n = \phi(x_0, x_1, x_2, x_3...)$;

- 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

ϕ instructions

- $x_n = \phi(x_0, x_1, x_2, x_3 \ldots);$
- variables that haven't been assigned can appear (but they will not be evaluated)

```
        x₀ = 1;
        if (...) goto end_loop;
loop:
        x₁  = ϕ(x₀, x₂);
        x₂ = x₁ + 1;
        if (...) goto loop;
end_loop:
        x₃  = ϕ(x₀, x₂);
```

ϕ instructions

- $x_n = \phi(x_0, x_1, x_2, x_3 \ldots);$

- variables that haven't been assigned can appear (but they will not be evaluated)

```
        x0 = 1;
        if (...) goto end_loop;
loop:
        x1  = ϕ(x0, x2);
        x2 = x1 + 1;
        if (...) goto loop;
end_loop:
        x3  = ϕ(x0, 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

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)
```

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)
```

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)
```

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)
```

Optimized?

```
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)
```

More efficient translation?

Example

```
x = 1;
y = 2;

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

else {
  x = 6;
  y = 100;
}

print(x)
```

maximal SSA

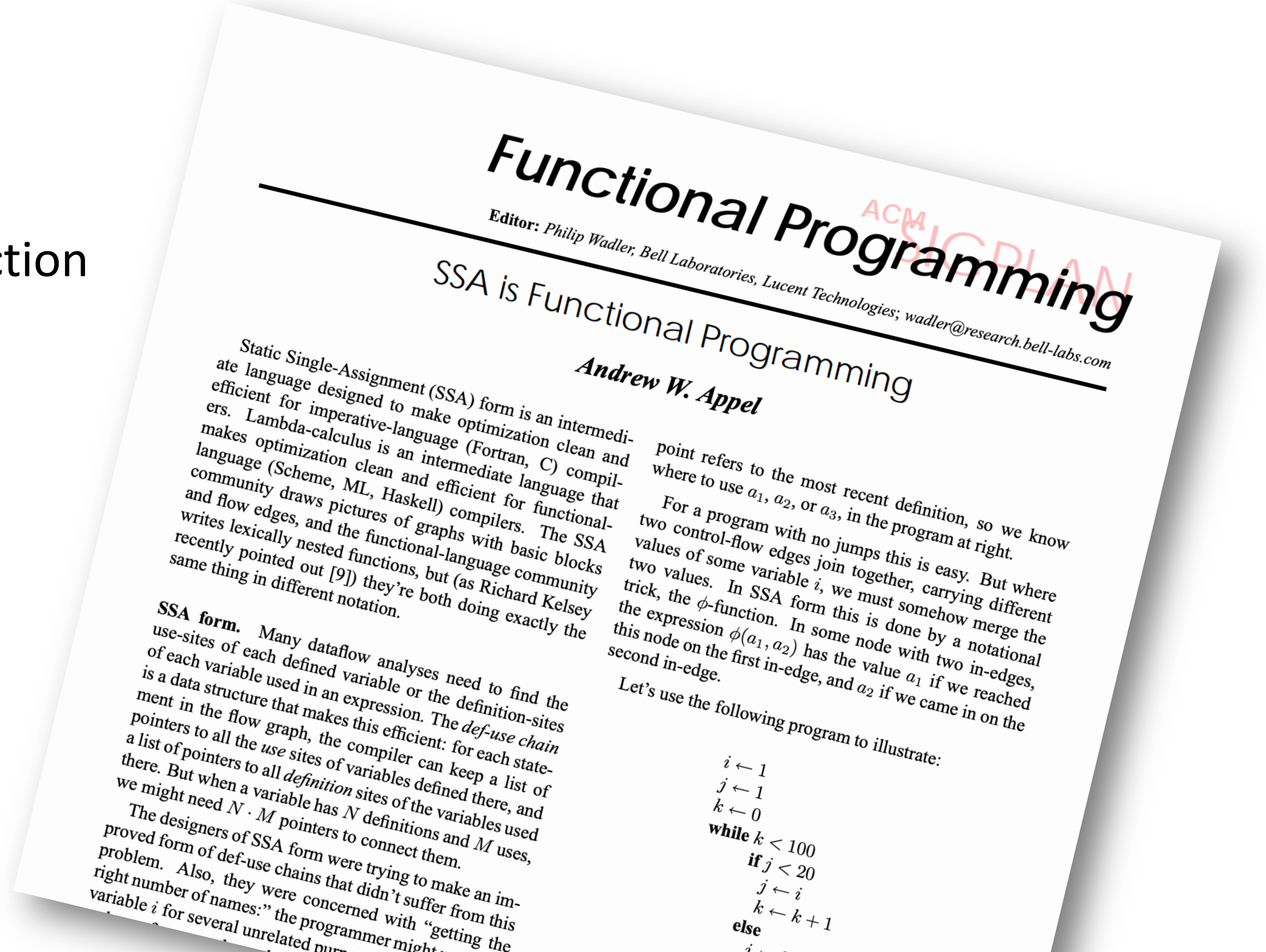
```
x0 = 1;
y1 = 2;

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:

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



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 pur...

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.

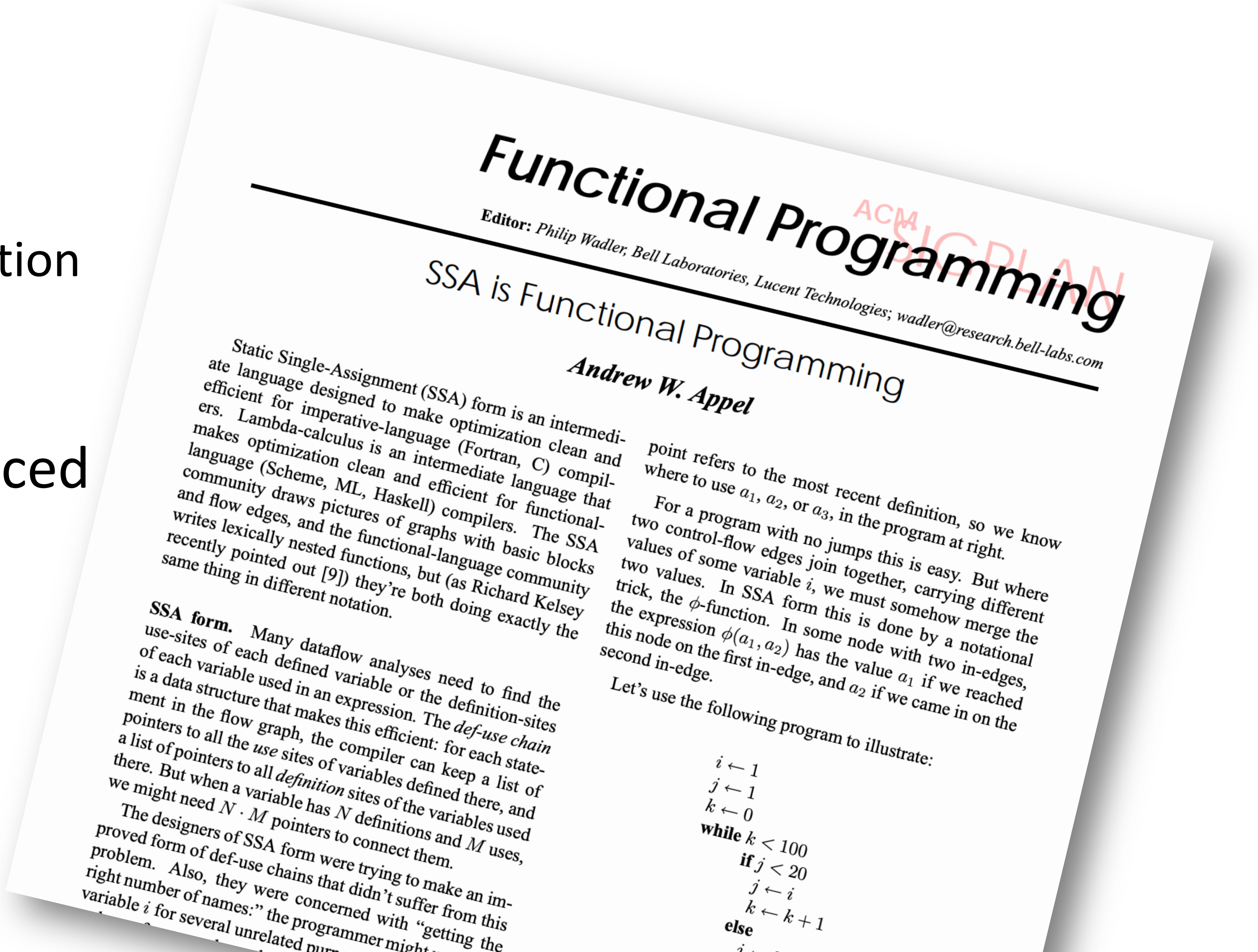
For a program with no jumps this is easy. But where two control-flow edges join together, carrying different values of some variable i, we must somehow merge the two values. In SSA form this is done by a notational trick, the ϕ-function. In some node with two in-edges, the expression $\phi(a_1, a_2)$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

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 use variable i for several unrelated pur

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

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)
```

A note on SSA variants:

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

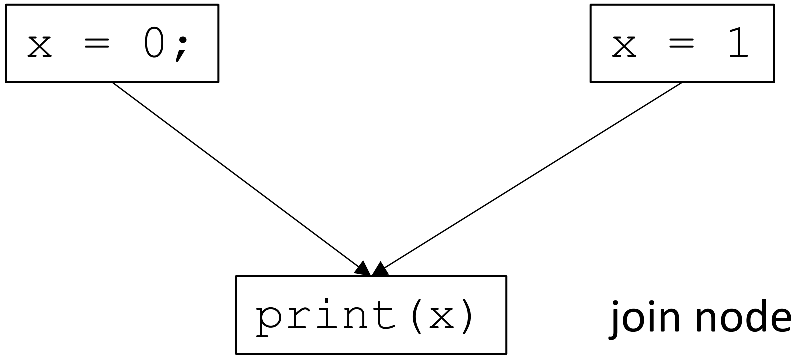
A more optimal approach for ϕ placements

- When is a ϕ needed?

A more optimal approach for ϕ placements

- When is a ϕ needed?

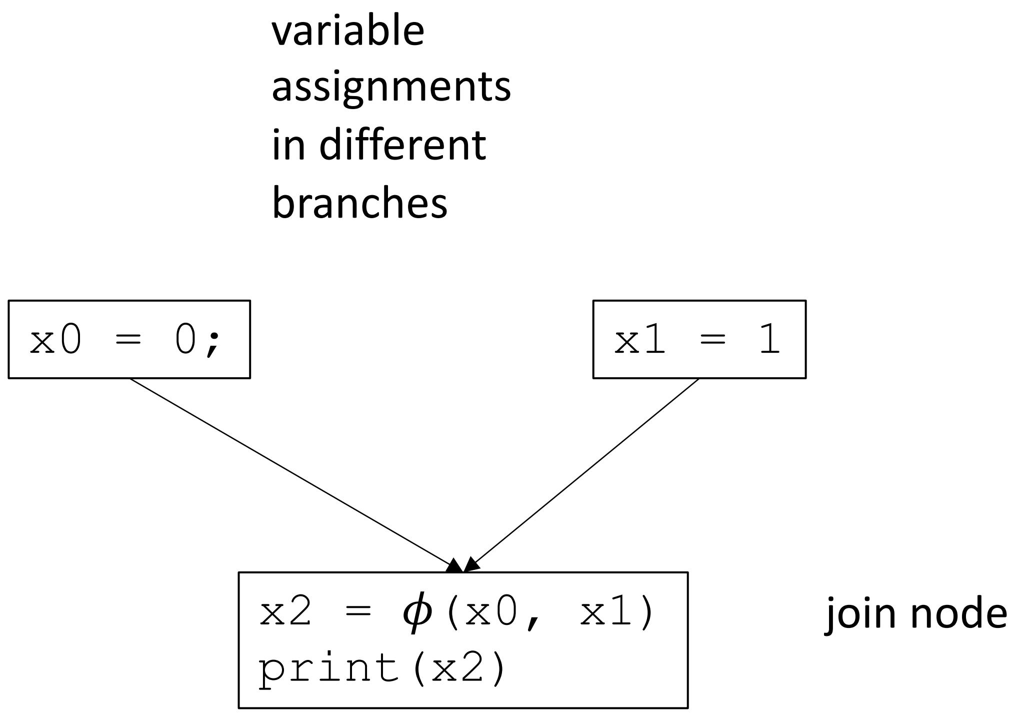
variable
assignments
in different
branches



join node

A more optimal approach for ϕ placements

- When is a ϕ needed?



A more optimal approach for ϕ placements

- When is a ϕ needed?

- More specific question: given a block i, find the set of blocks B which may need a ϕ instruction for a definition in block i.

```
x = 0;
```

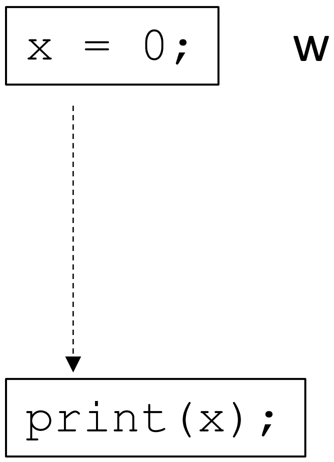
what set of blocks need a ϕ node to resolve conflicts on this assignment to x?

A more optimal approach for ϕ placements

- When is a ϕ needed?

- More specific question: given a block i, find the set of blocks B which may need a ϕ instruction for a definition in block i.

block i



what set of blocks need a ϕ node to resolve conflicts on this assignment to x?

block j

Does block j need a ϕ to resolve the assignment to x in block i?

A more optimal approach for ϕ placements

- When is a ϕ needed?

- More specific question: given a block i, find the set of blocks B which may need a ϕ instruction for a definition in block i.

block i

```
x = 0;
```

what set of blocks need a ϕ node to resolve conflicts on this assignment to x?

some path

is block j dominated by block i?
If so, then no ϕ node is needed

block j

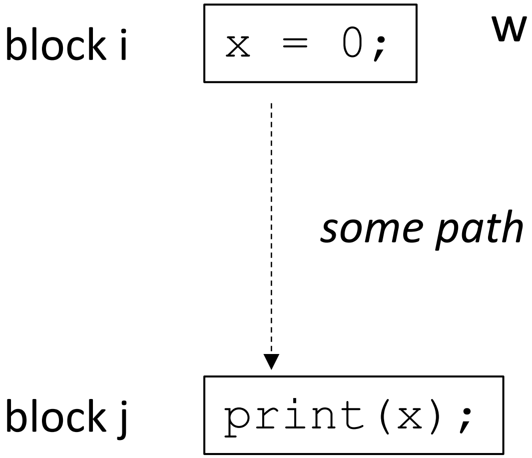
```
print(x);
```

Does block j need a ϕ to resolve the assignment to x in block i?

A more optimal approach for ϕ placements

- say j is dominated by i. Thus, no ϕ node is needed in block j

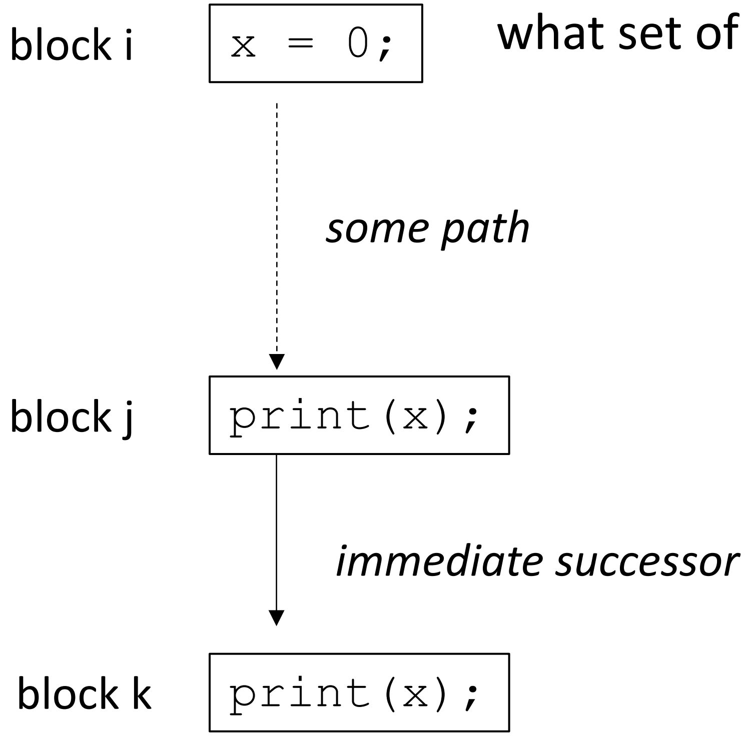
what set of blocks need a ϕ node to resolve conflicts on this assignment to x?



A more optimal approach for ϕ placements

- say j is dominated by i. Thus, no ϕ node is needed in block j

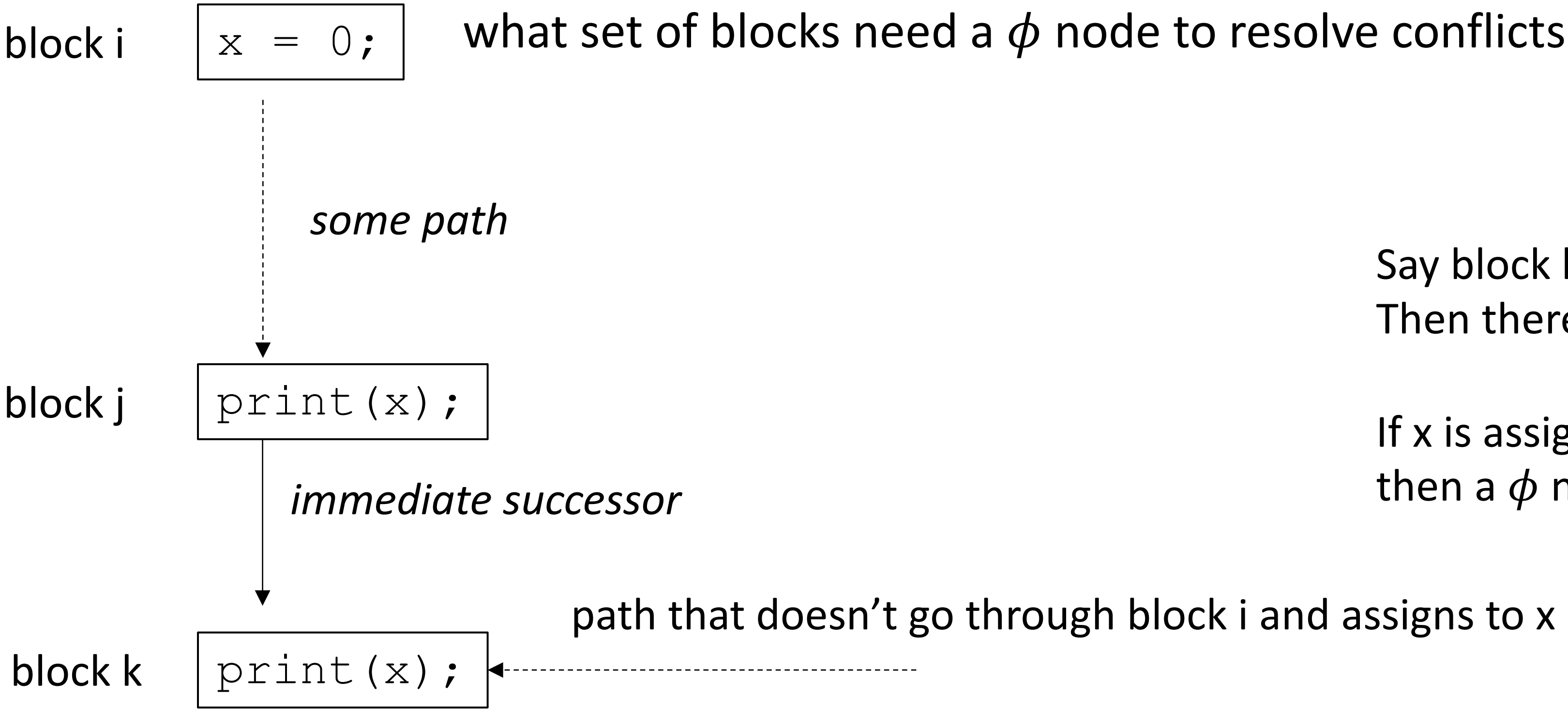
what set of blocks need a ϕ node to resolve conflicts on this assignment to x?



A more optimal approach for ϕ placements

- say j is dominated by i. Thus, no ϕ node is needed in block j

what set of blocks need a ϕ node to resolve conflicts on this assignment to x?



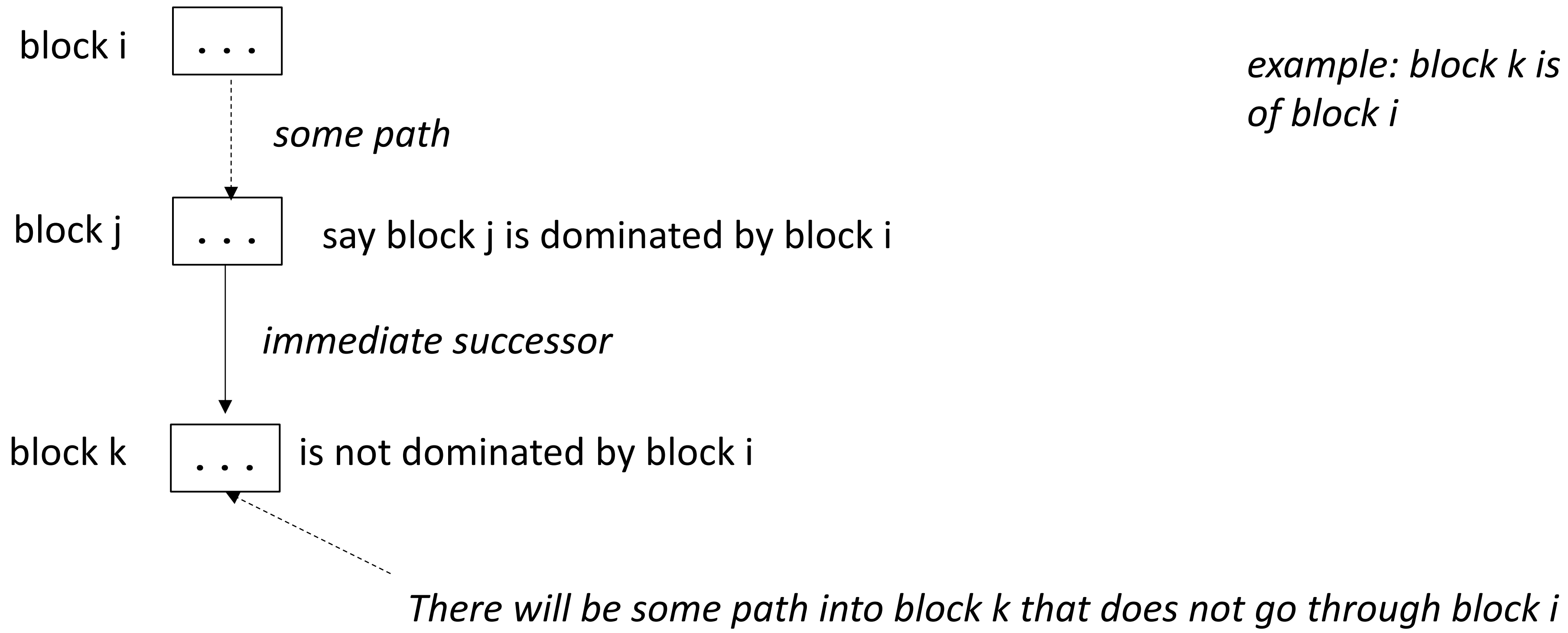
Say block k is not dominated by block i.
Then there exists another in-edge to block k.

If x is assigned along a path not through block i, then a ϕ 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

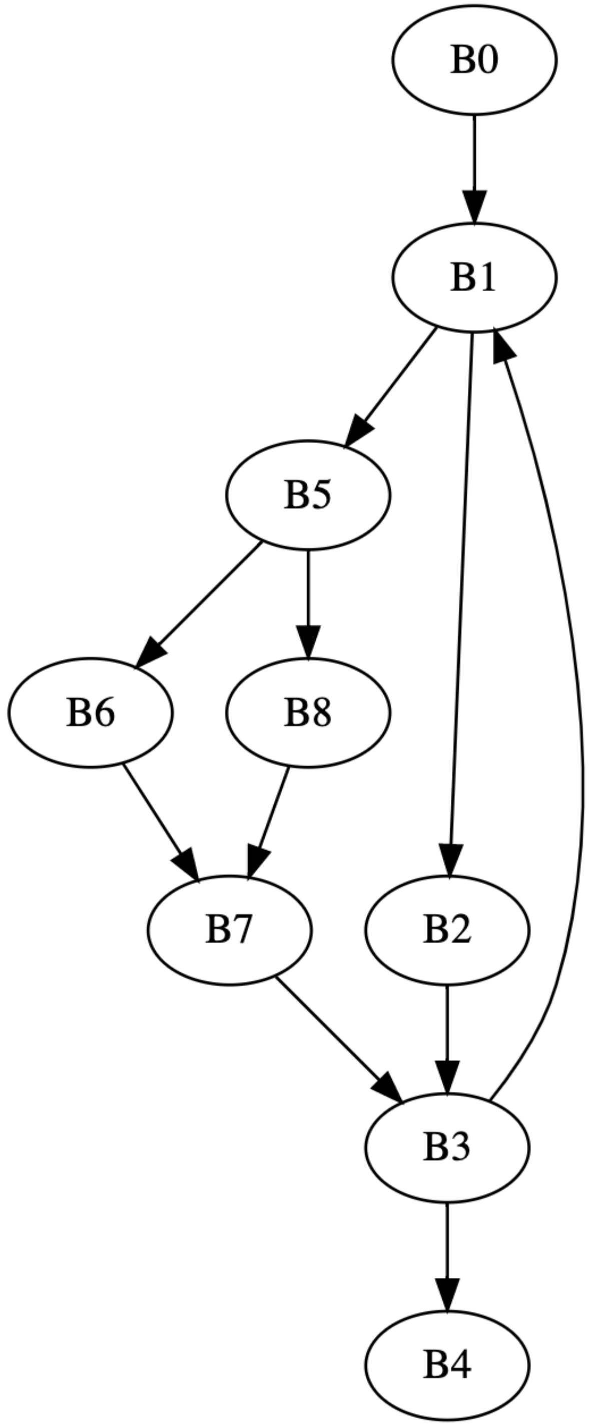
Dominance frontier

- a viz using coloring (thanks to Chris Liu!)

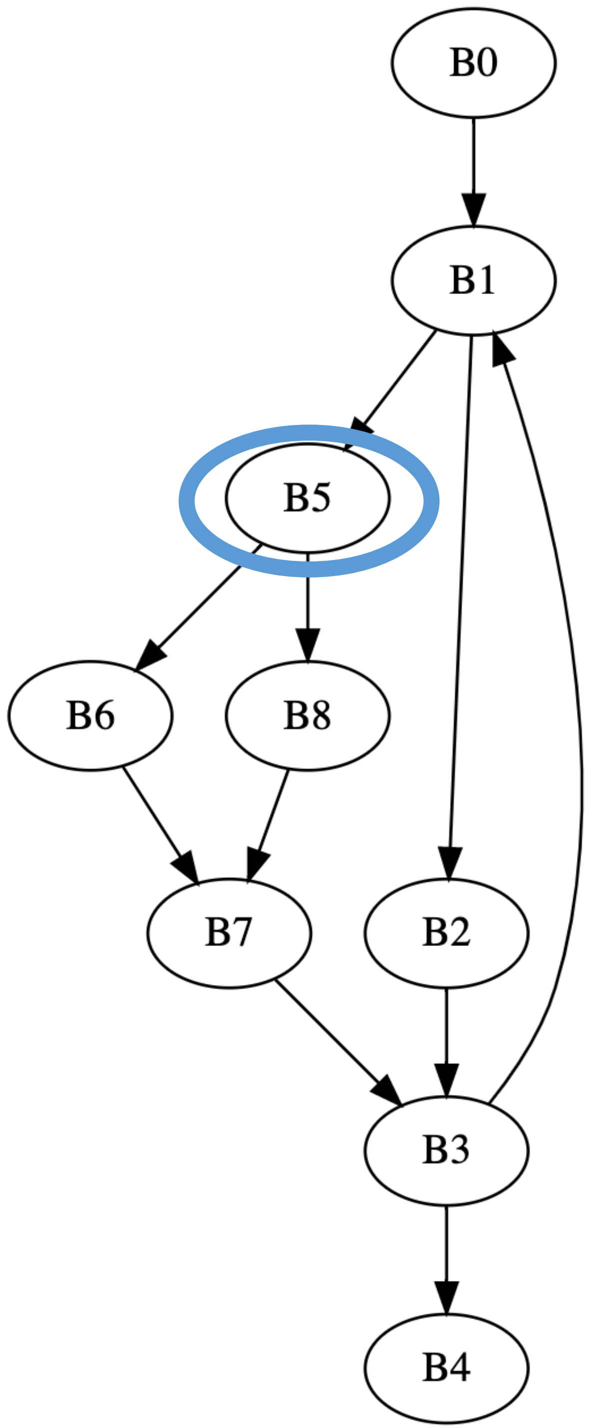
- Efficient algorithm for computing in EAC section 9.3.2 using a dominator tree. Please read when you get the chance!

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

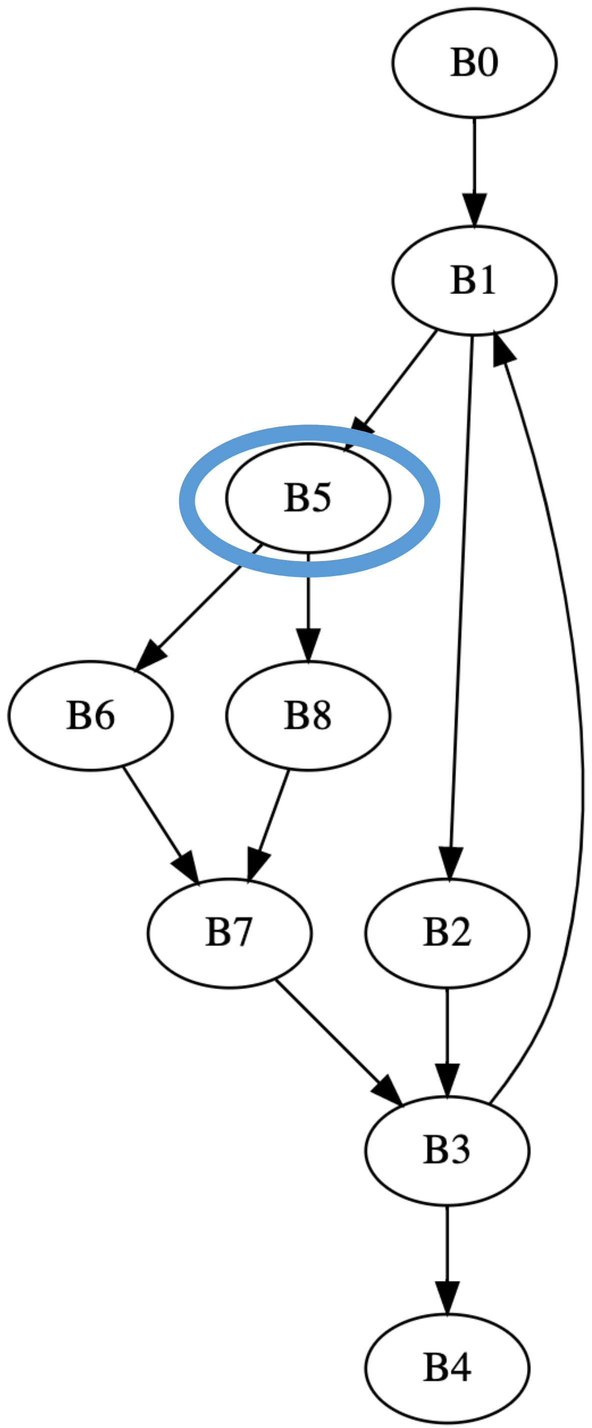
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



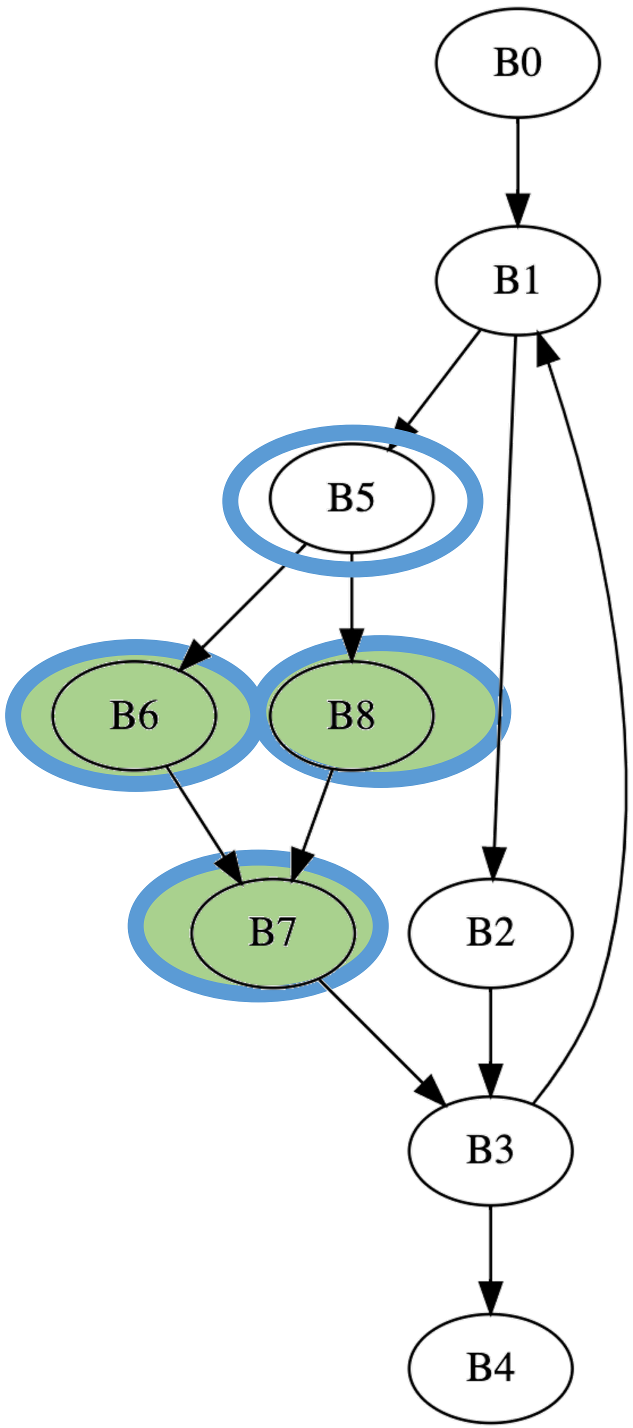
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



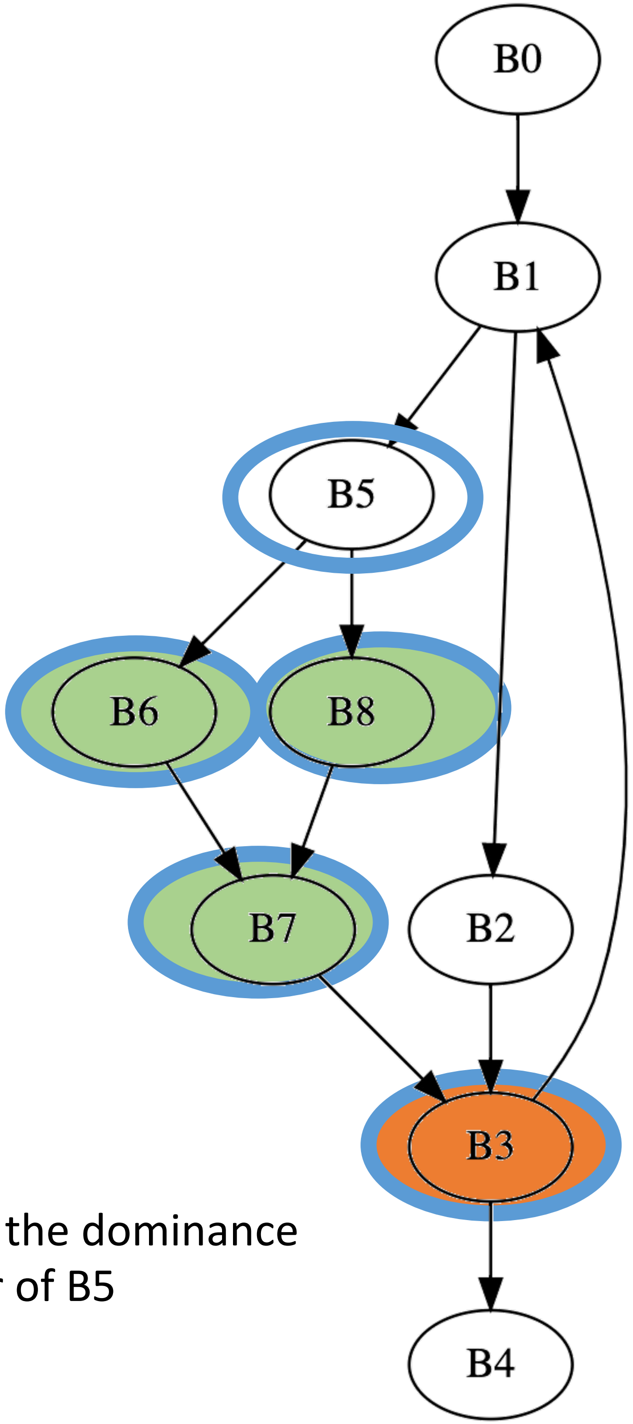
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



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,

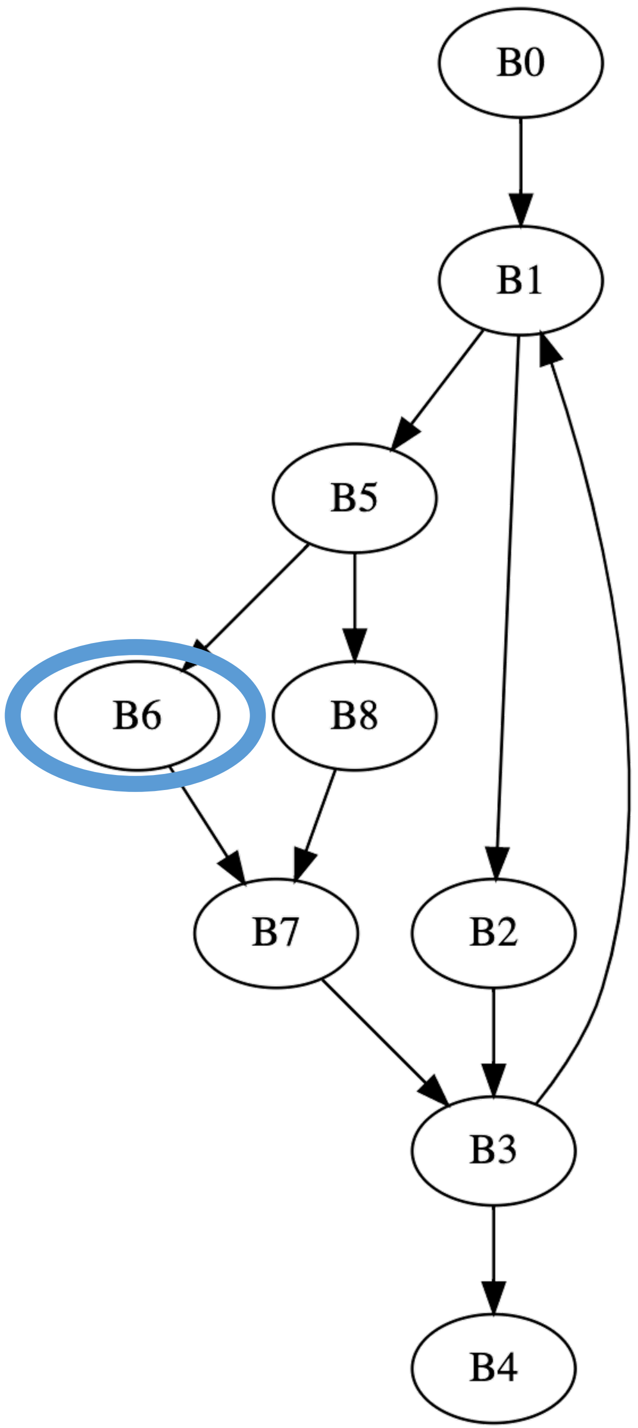


Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

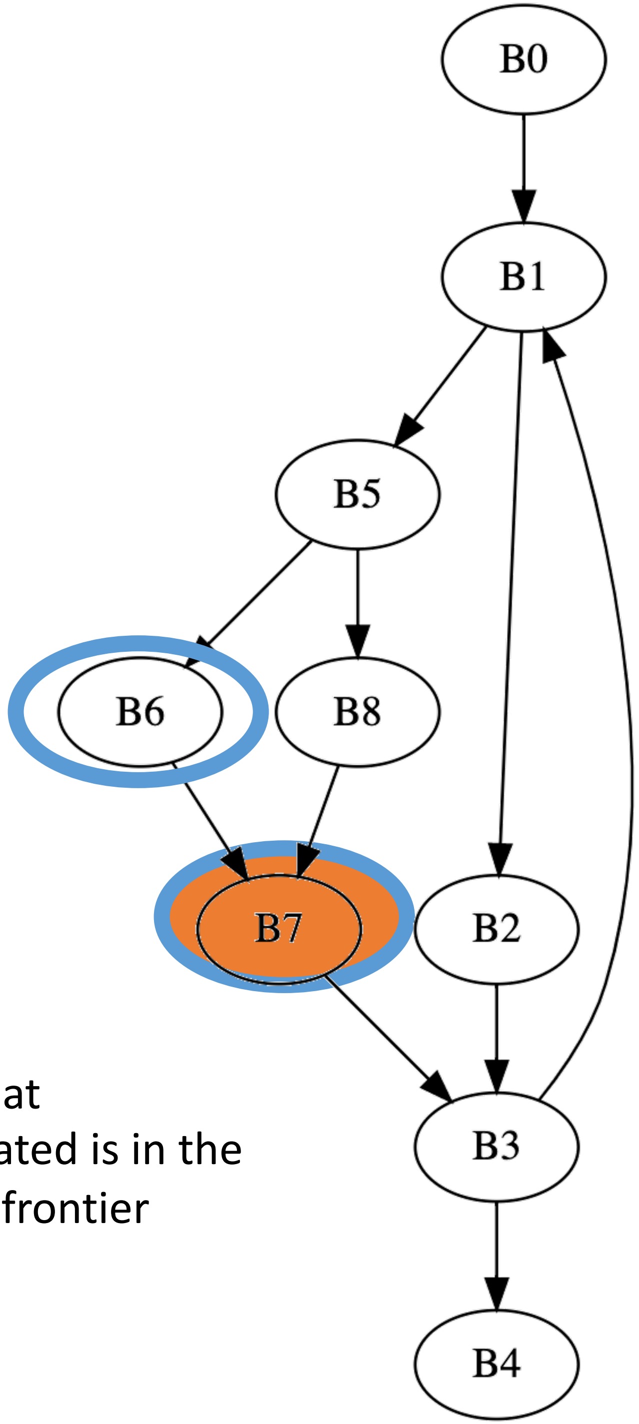


B3 is in the dominance frontier of B5

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,

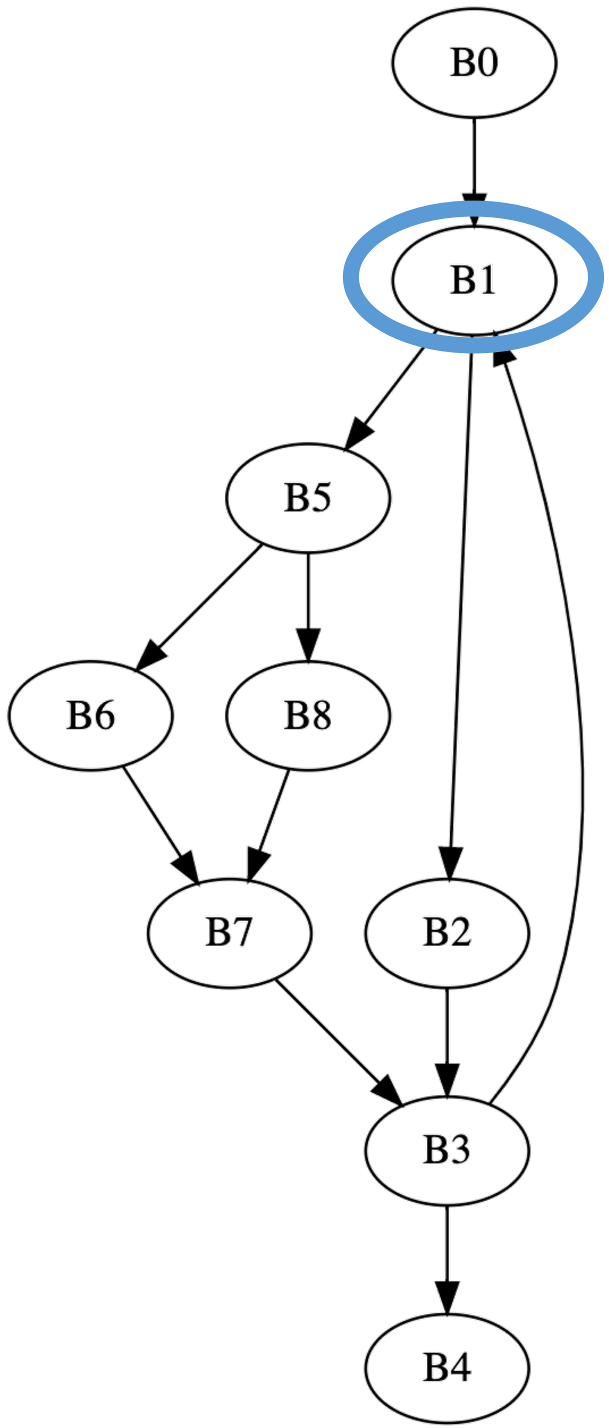


Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

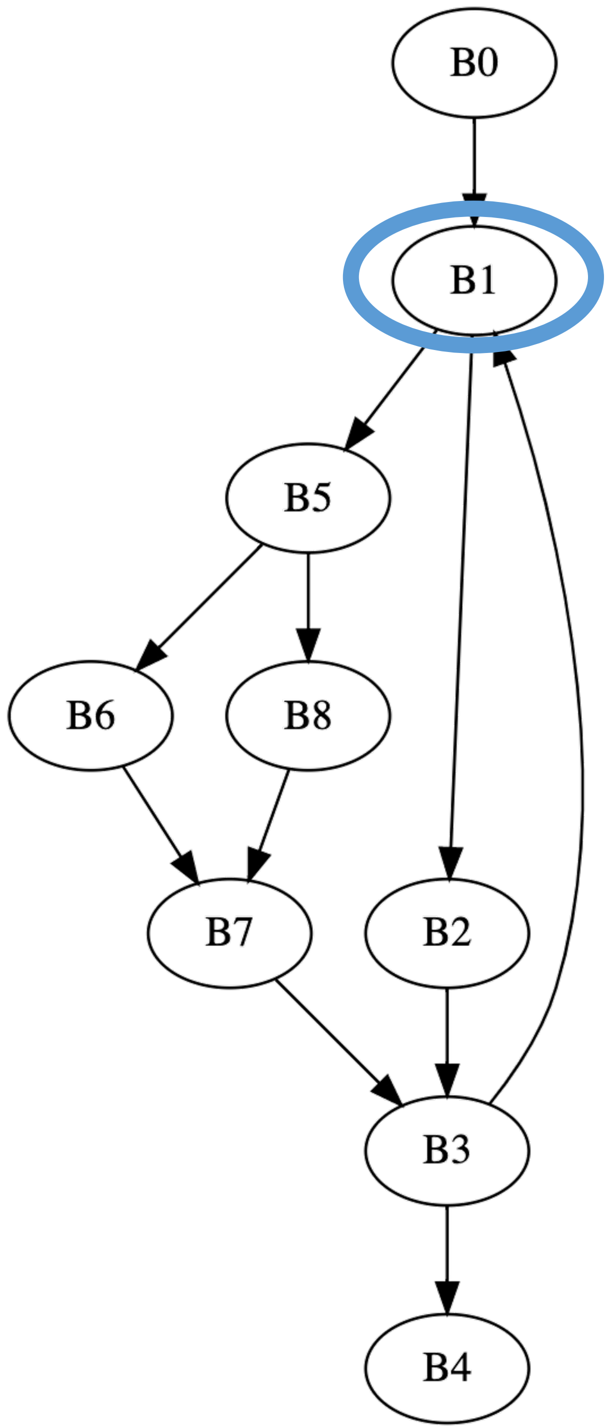


Any child that isn't dominated is in the dominance frontier

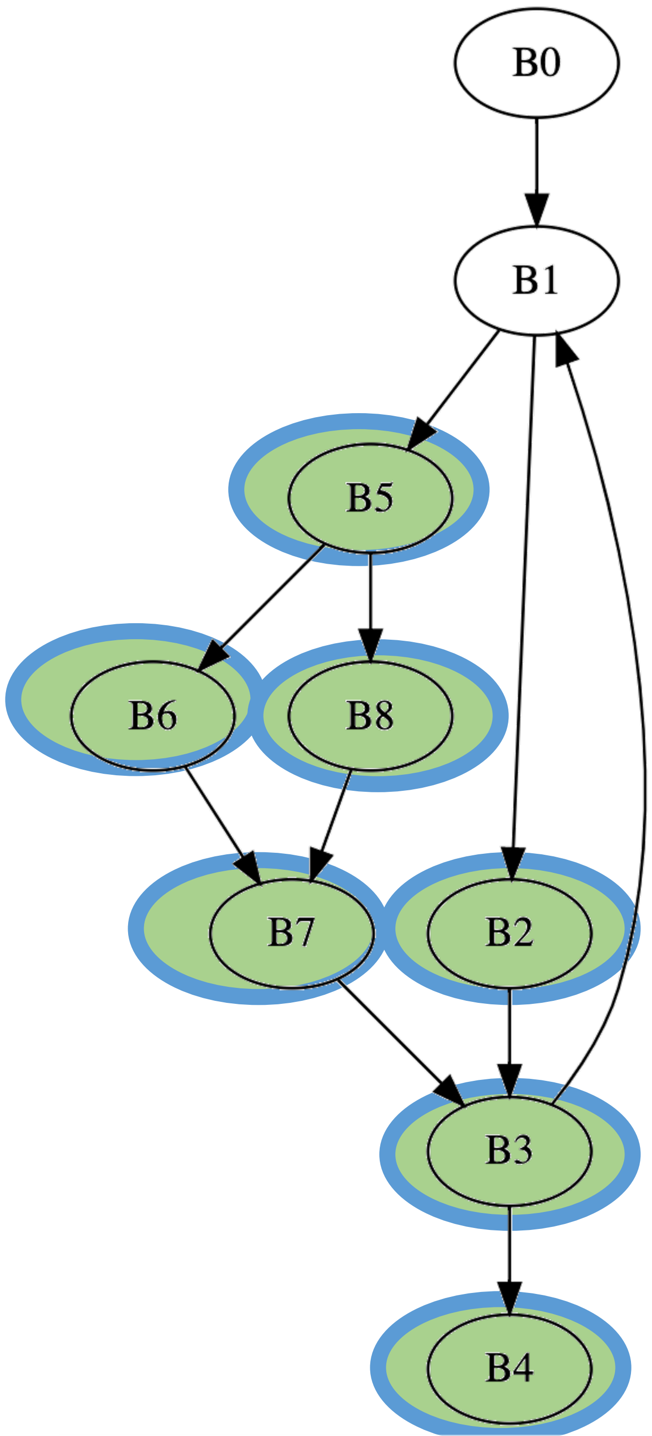
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



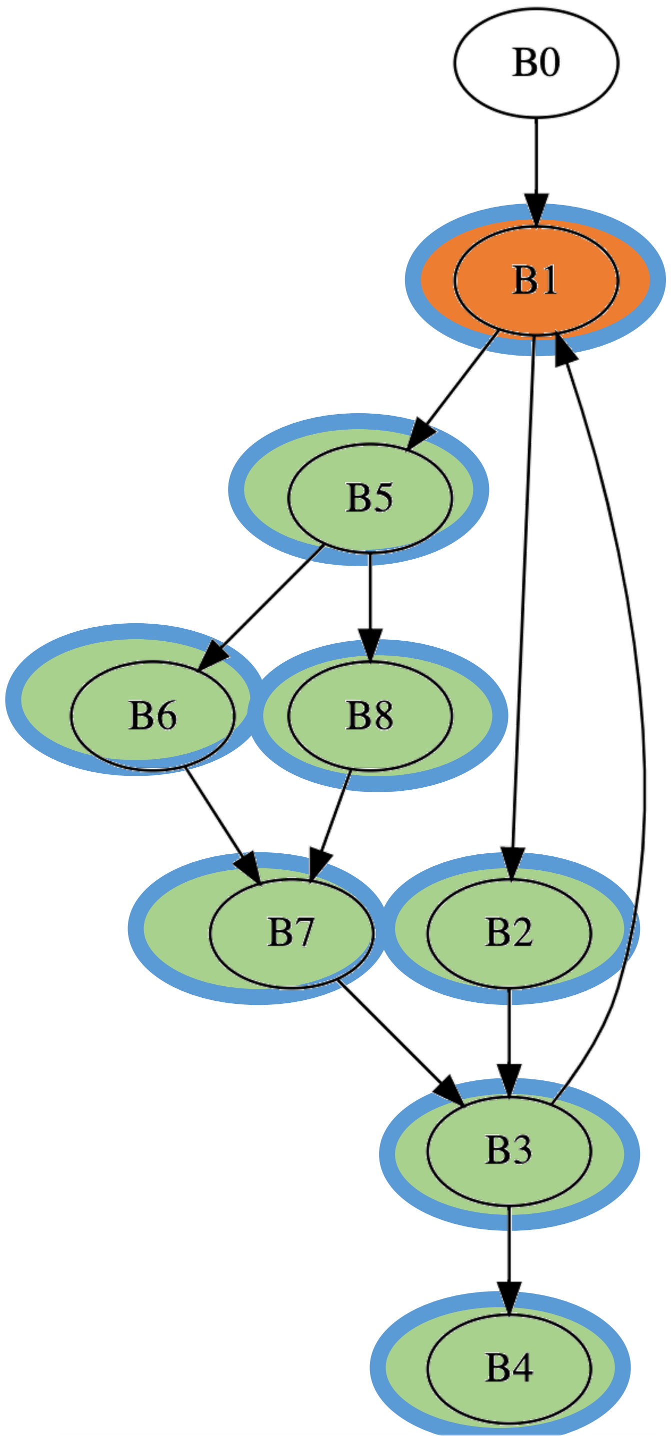
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



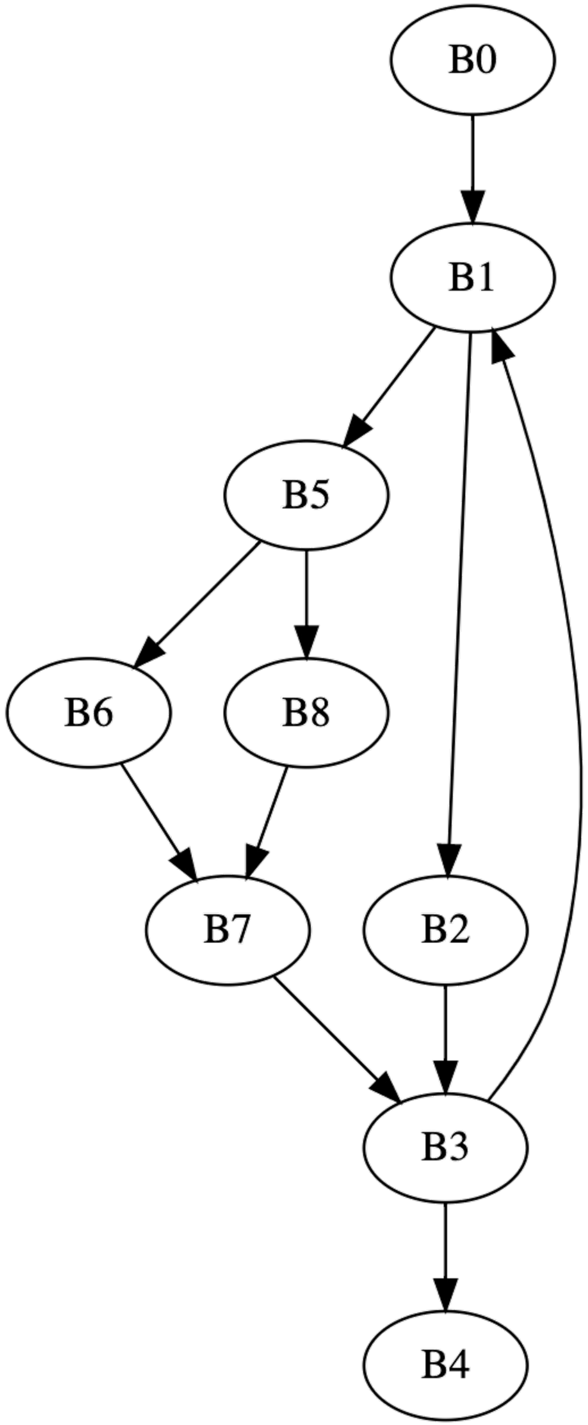
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



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

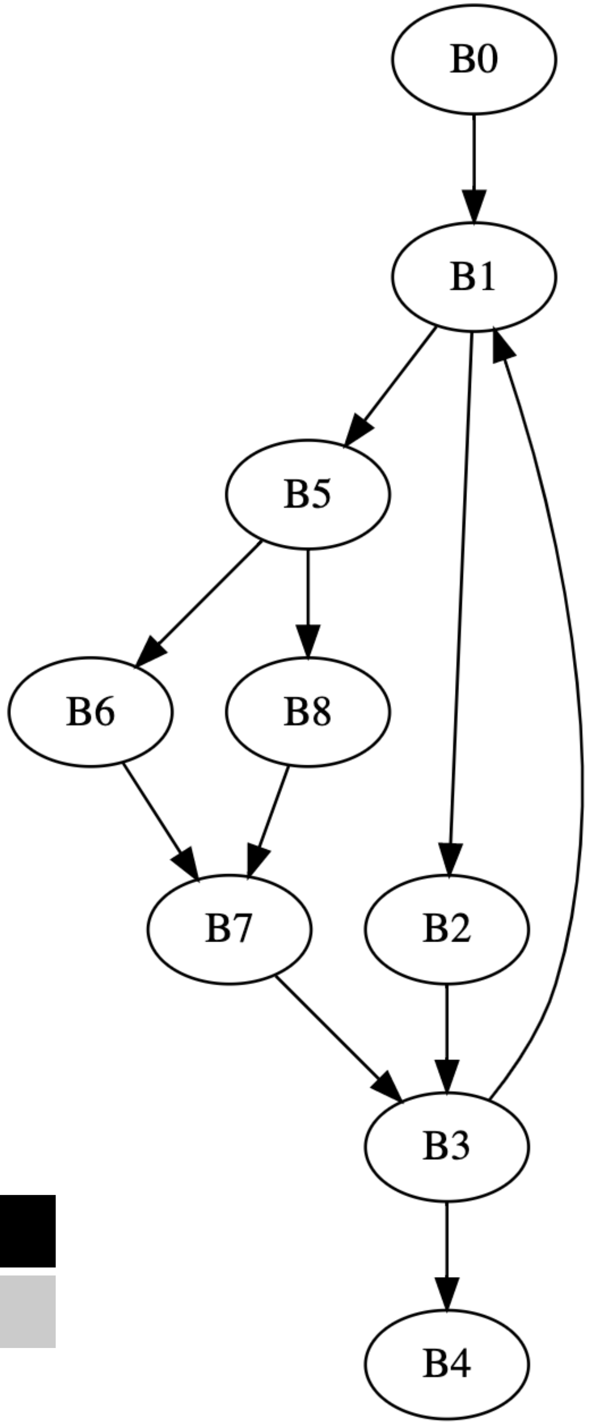


Dominance Frontier

- 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;
```

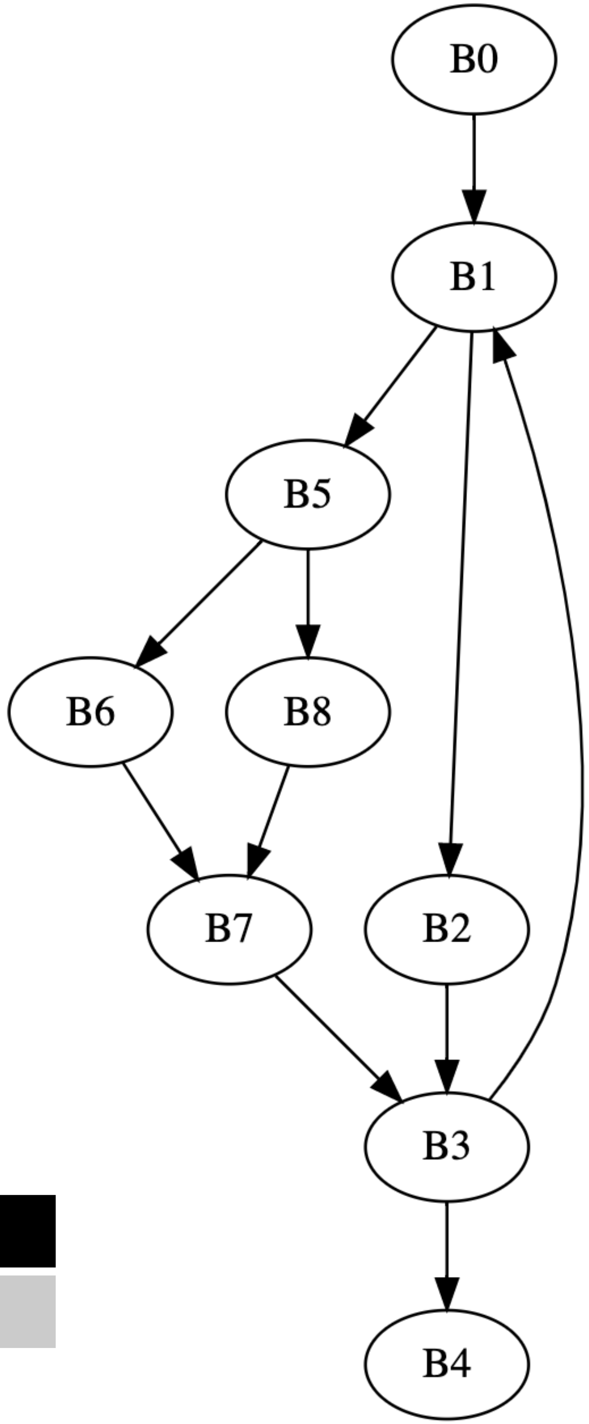
```
B5: a = ...;
    d = ...;
    br ... B6, B8;

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;
```

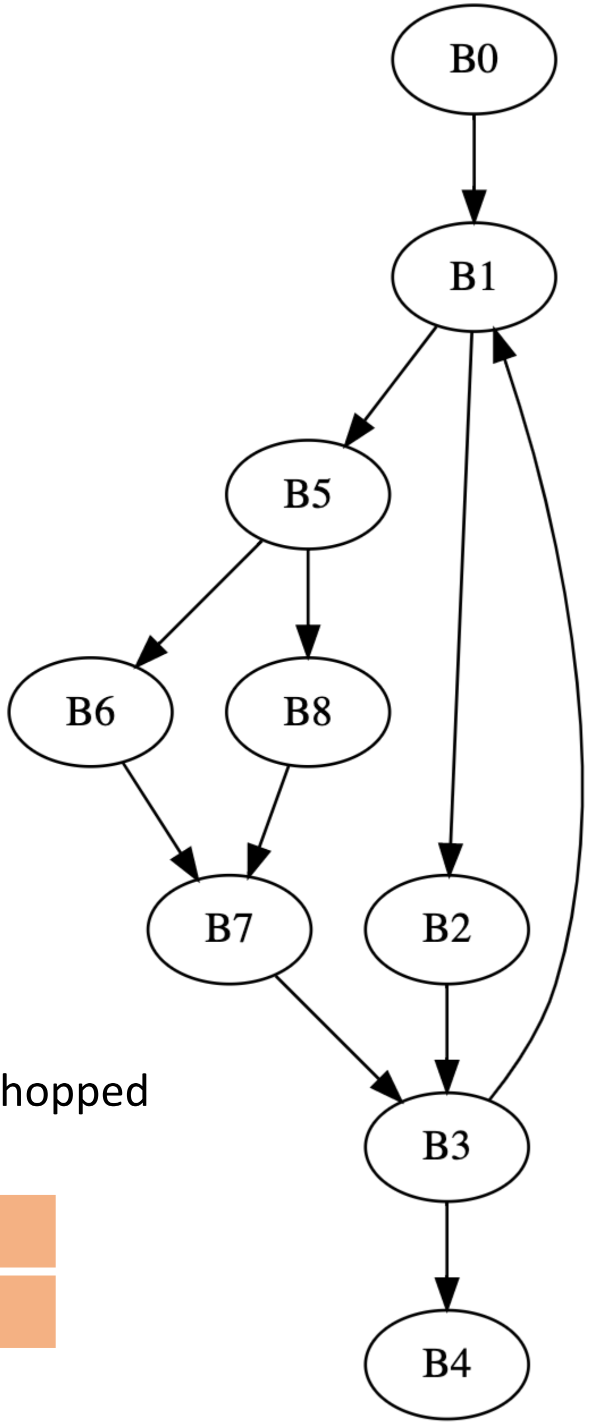
```
B5: a = ...;
    d = ...;
    br ... B6, B8;

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:
    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'!

```
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	b
Blocks	B1,B5,B3	B2,B7

```
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	b
Blocks	B1,B5,B3	B2,B7

```
B0: i = ...;

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

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

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

B4: return;
```

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

```
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

```
B0: i = ...;

B1: a = 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

```
B0: i = ...;

B1: a = 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;
```

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

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

```
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;
```

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

```
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

```
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;
```

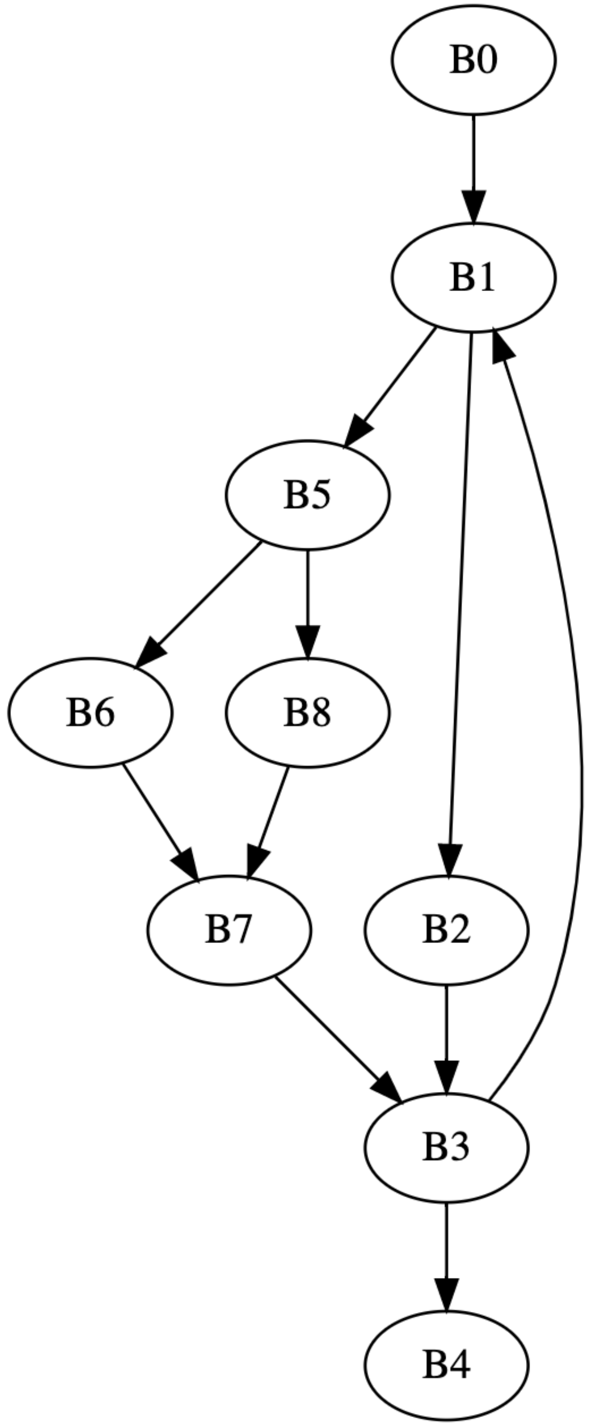
Var	a	b
Blocks	B1,B5,B3	B2,B7,B3.B1

```
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

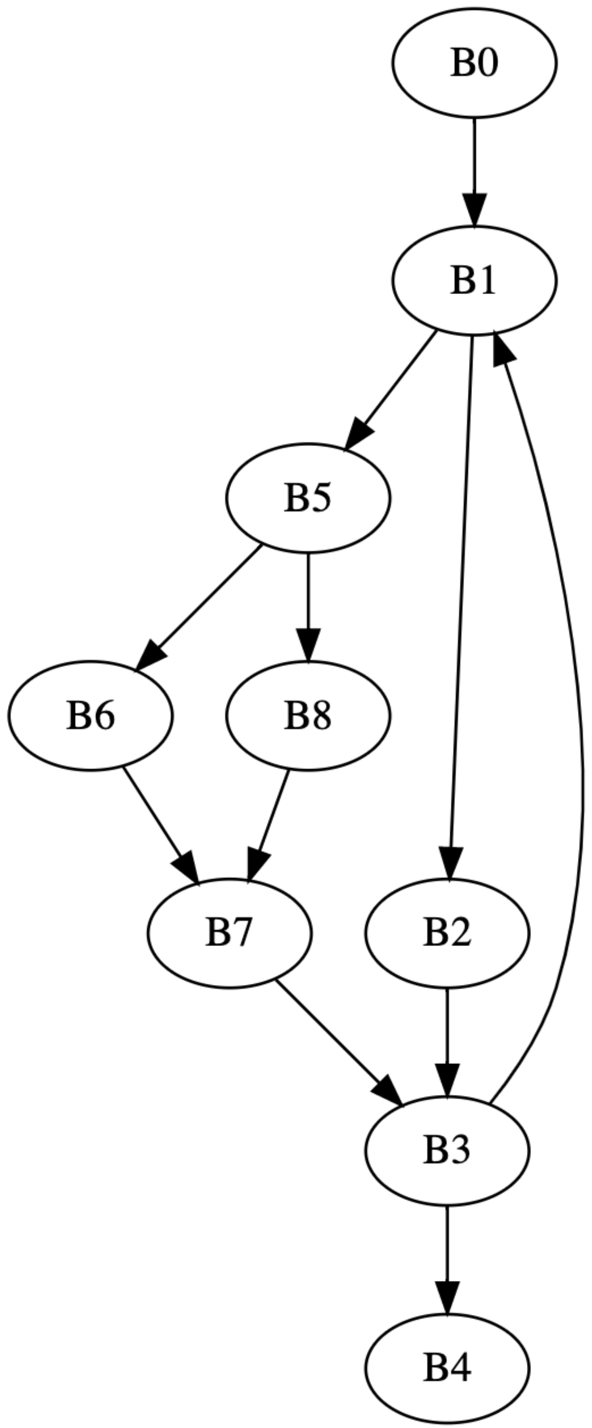
```
B0: i0 = ...;

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;
```



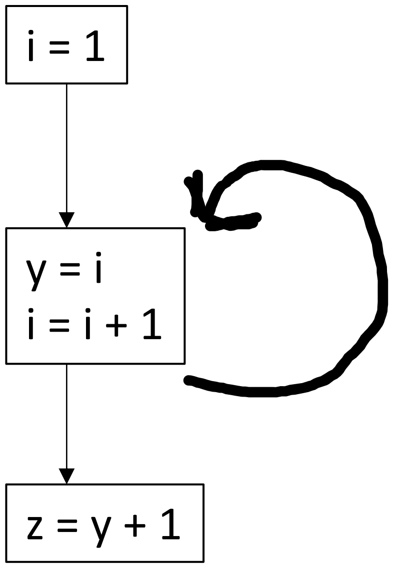
How to convert back to 3 address code from SSA?

- Can a processor execute phi instructions?

How to convert back to 3 address code from SSA?

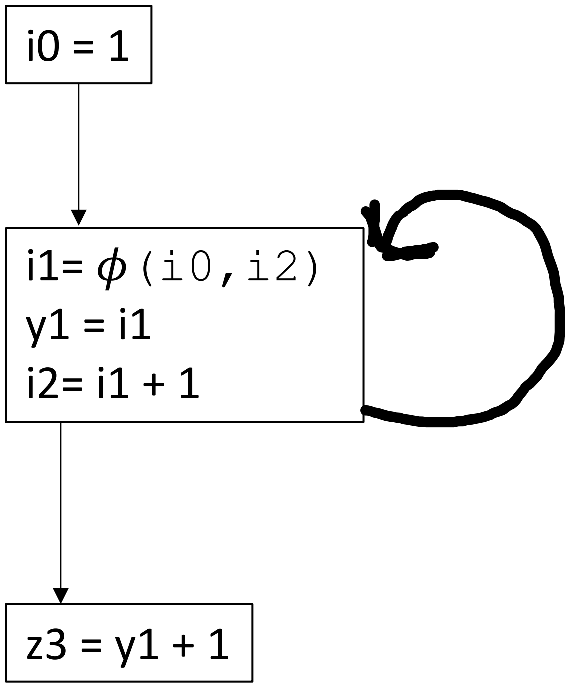
- Can a processor execute phi instructions?
- Just assign to the new variable in the parent?

Lost copy issue



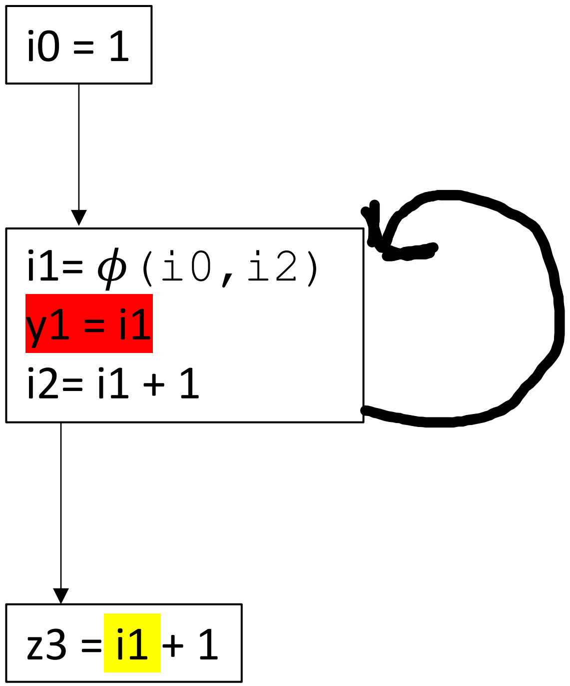
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue



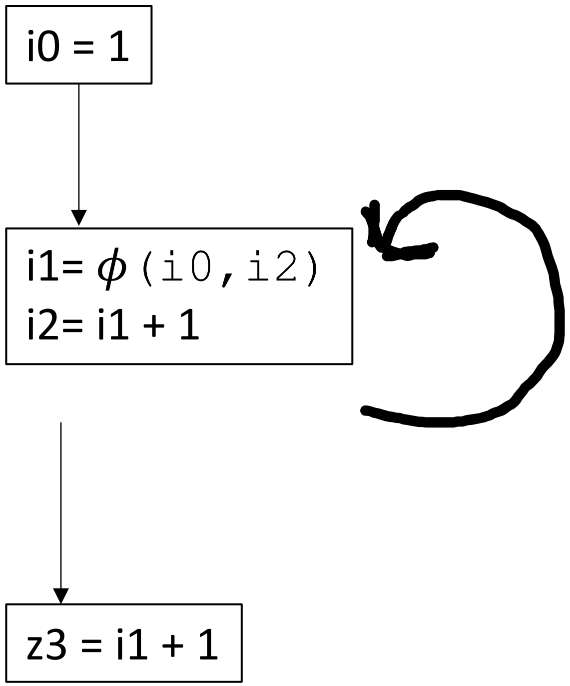
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue



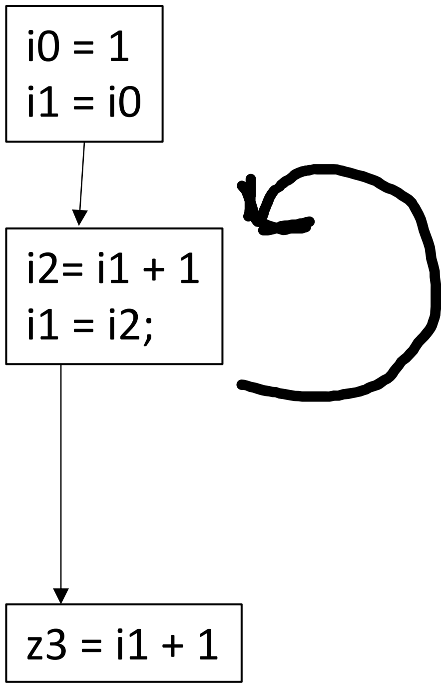
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

Lost copy issue



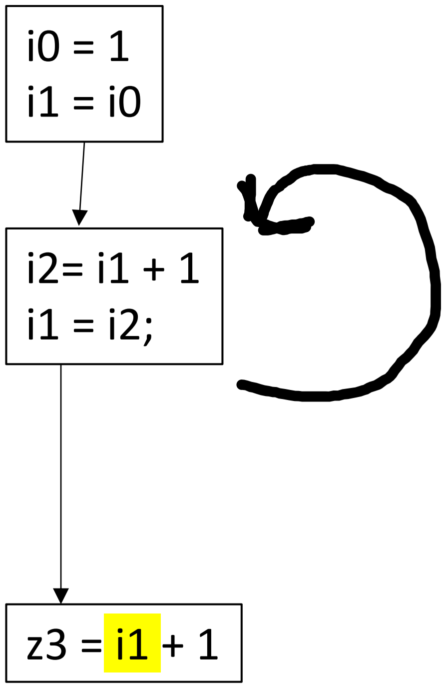
Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

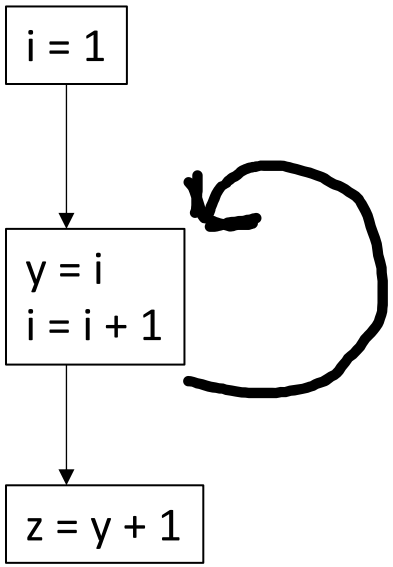
Lost copy issue



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

Lost copy issue

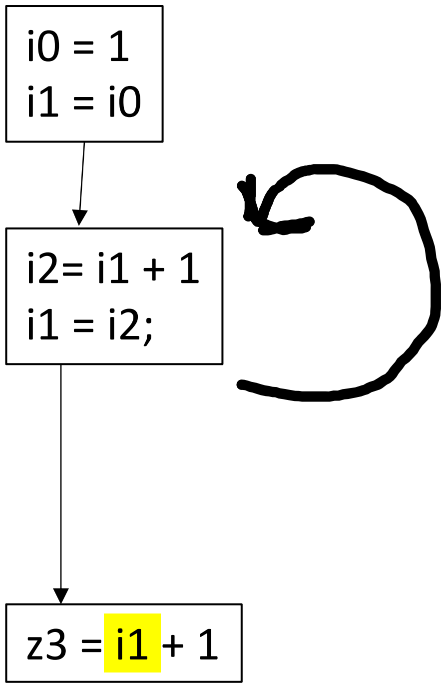


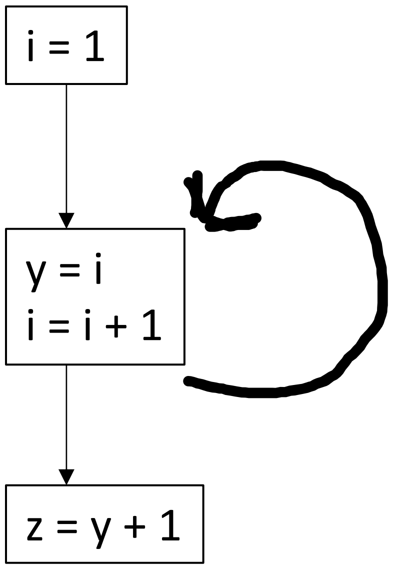


Example from https://www.clear.rice.edu/comp512/Lectures/13SSA-2.pdf

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

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: $\{c_1, c_2, c_3 \ldots\}$
- Special symbols:
  - Top : T
  - Bottom : ⊥

- Meet operator: ∧

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

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

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:

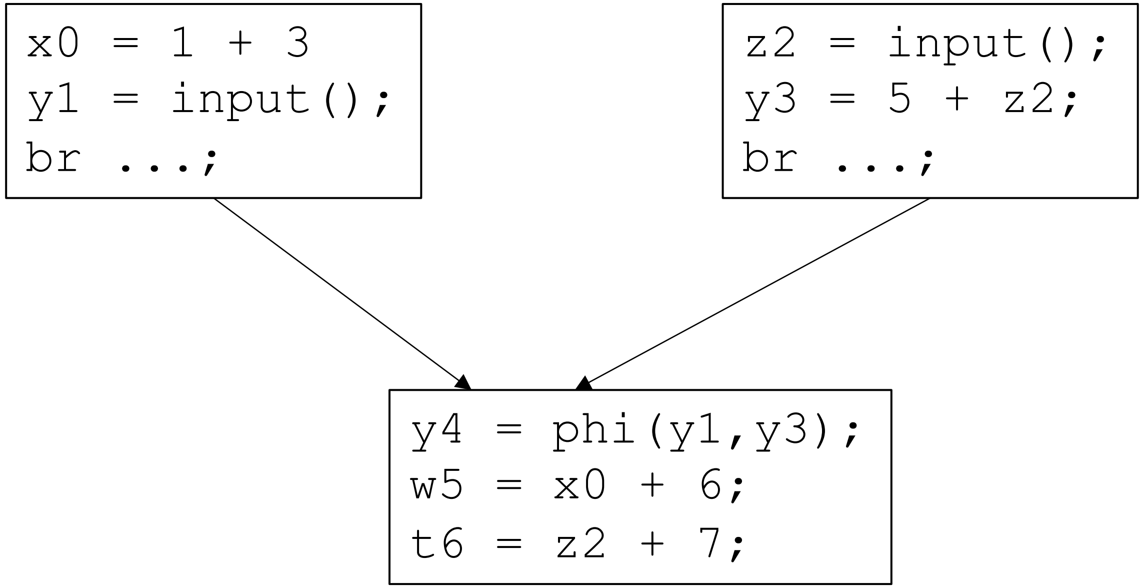
- a constant c_i if the value can be known
- $\bot$ if the value comes from an argument or input
- $\top$ otherwise, e.g. if the value comes from a ϕ node

Then, create a “uses” map

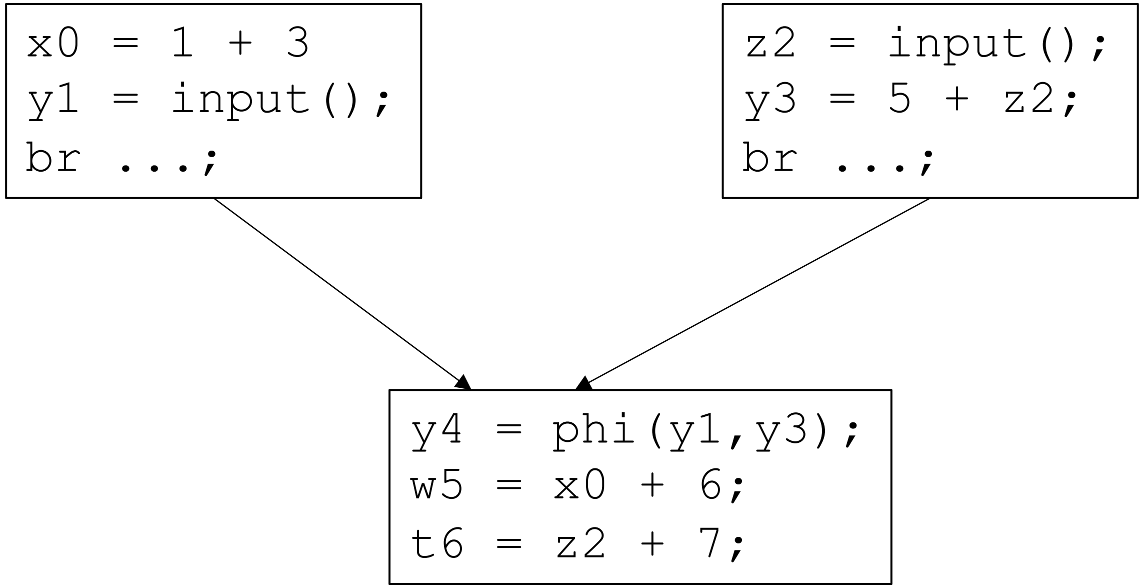
This can be done in a single pass

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:

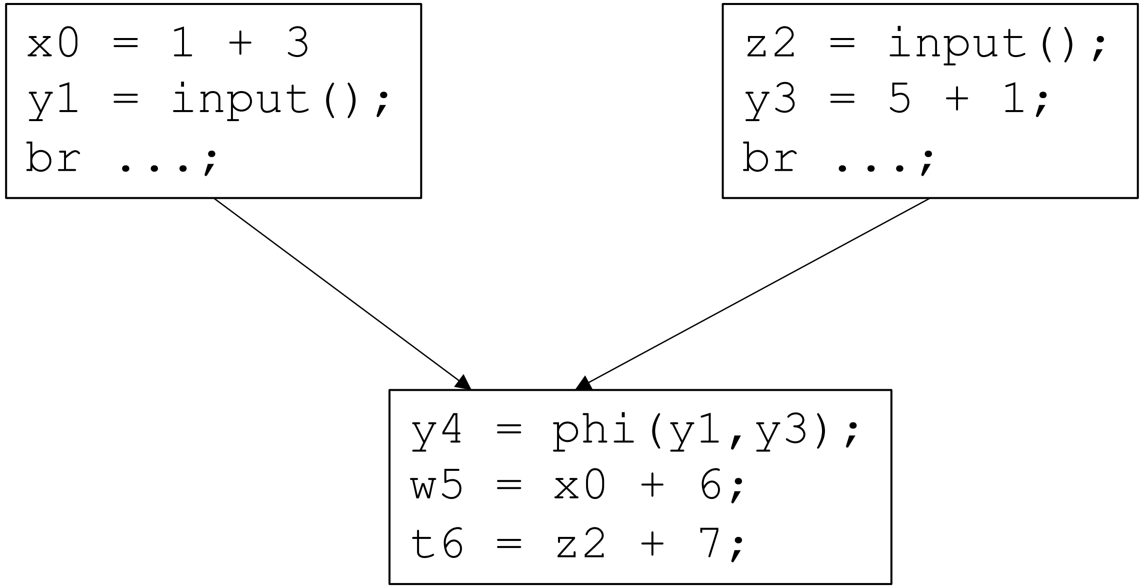
All variables **NOT** assigned to T get put on a worklist

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

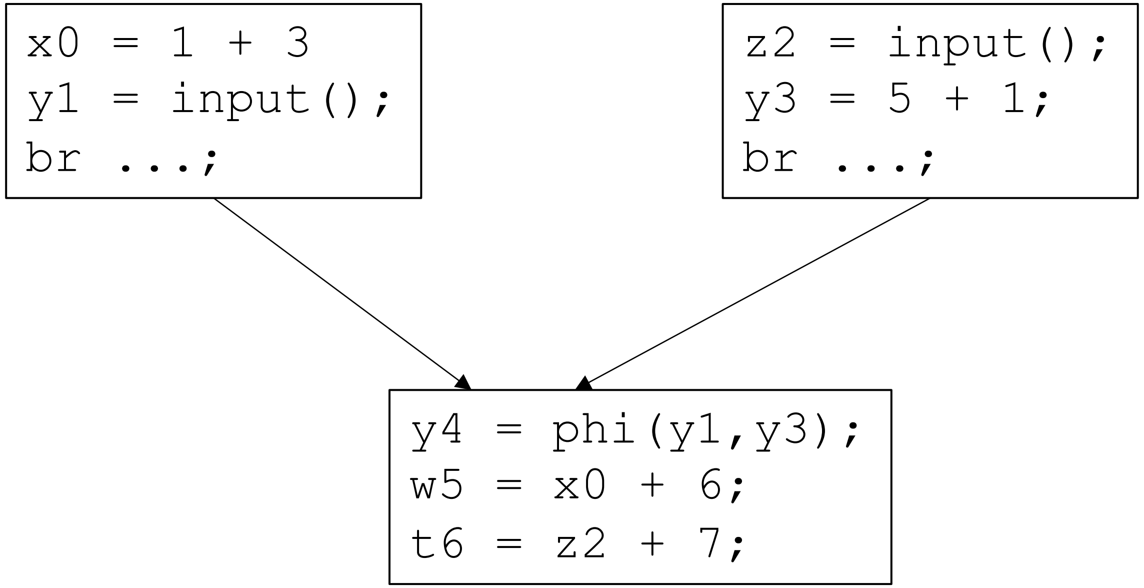
for each item in the worklist, evaluate all of it's uses *m* over the lattice (unique to each optimization)

Example: *m = n*x*

if (Value(n) is ⊥ or Value(x) is ⊥)

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

Example:



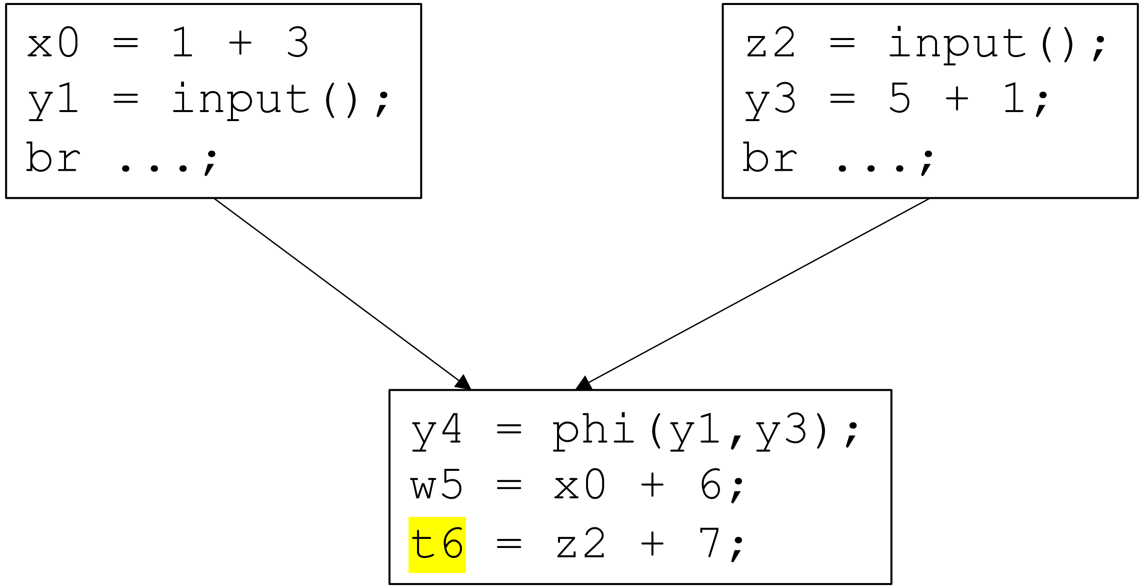

```
Value {
   x0 : 4
   y1 : B
   z2 : B
   y3 : 6
   y4 : T
   w5 : T
   t6 : T
}
```

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

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

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)

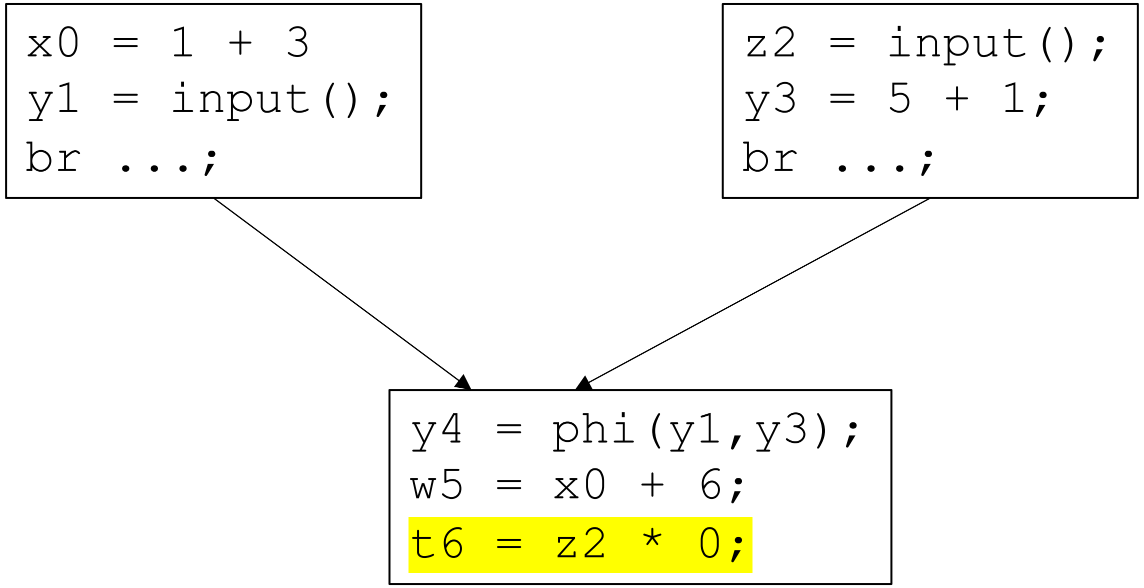
Example: *m = n*x*

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:

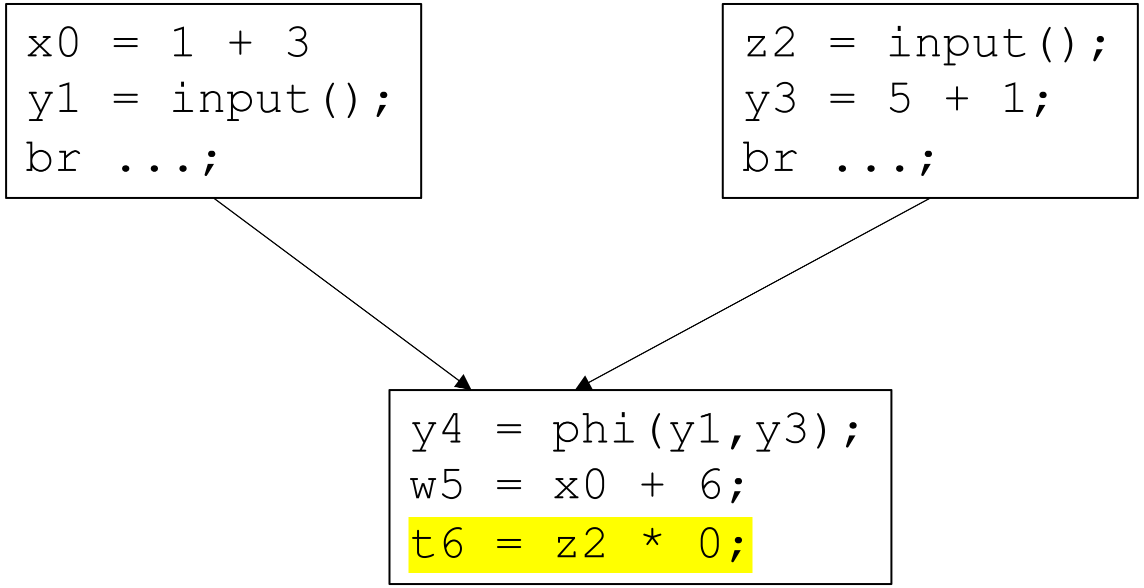



```
Value {
  x0 : 4
  y1 : B
  z2 : B
  y3 : 6
  y4 : T
  w5 : T
  t6 : T
}
```

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

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

Example:



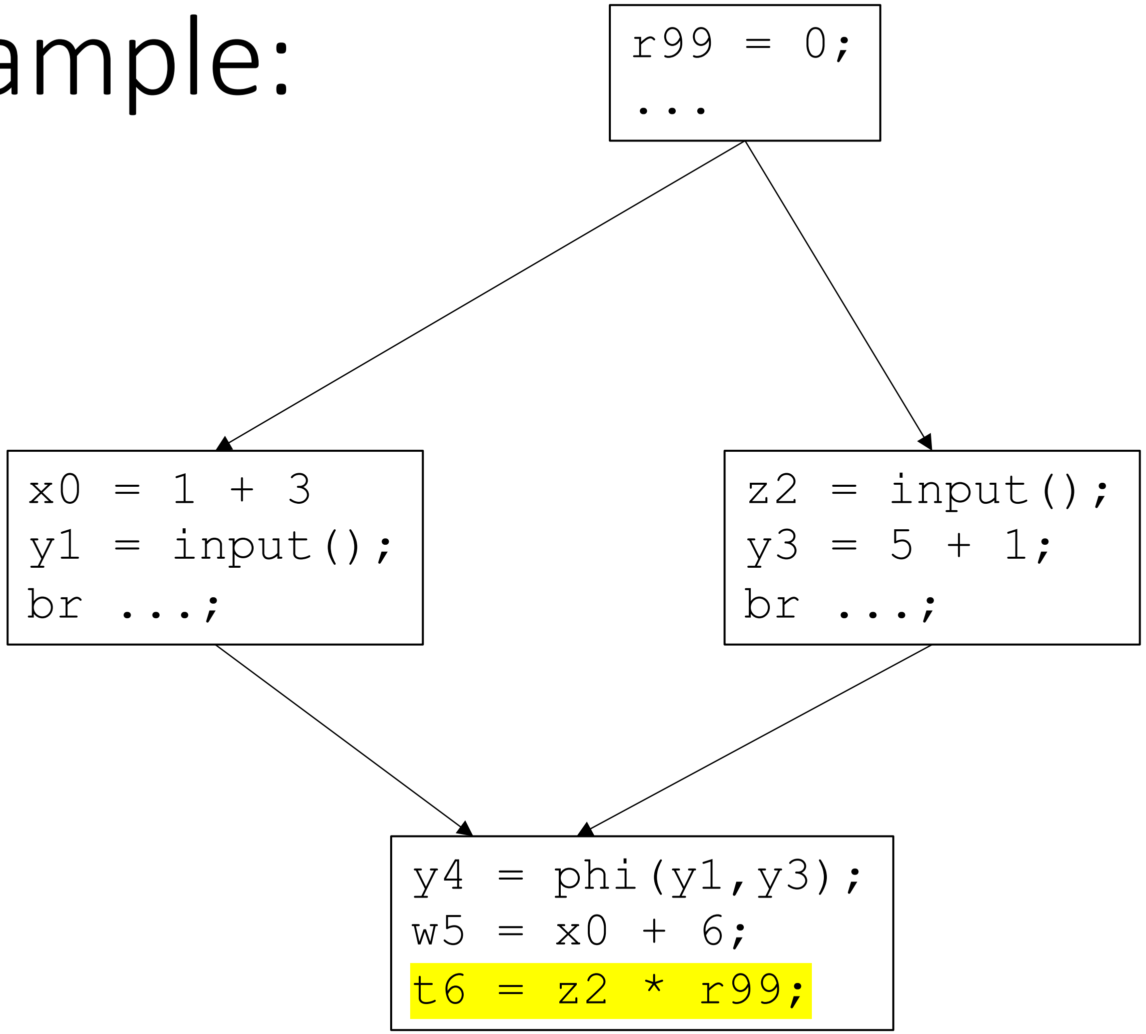

```
Value {
   x0 : 4
   y1 : B
   z2 : B
   y3 : 6
   y4 : T
   w5 : T
   t6 : T
}
```

```
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:

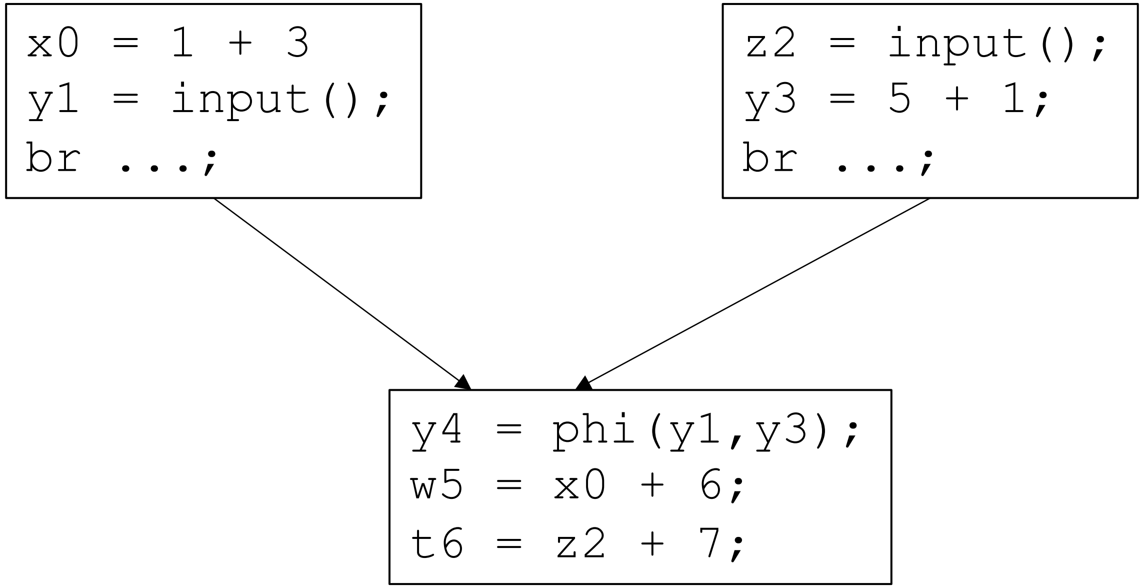



```
Value {
   x0  : 4
   y1  : B
   z2  : B
   y3  : 6
   y4  : T
   w5  : T
   t6  : T
   r99  : 0
 }

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:



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)

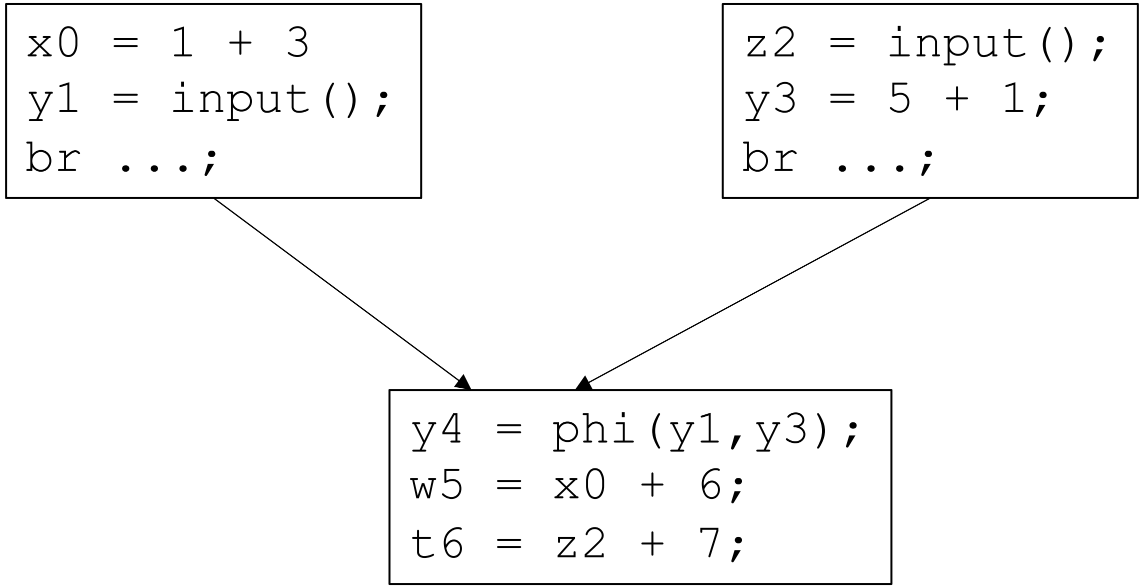
Example: *m = n*x*

// 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:

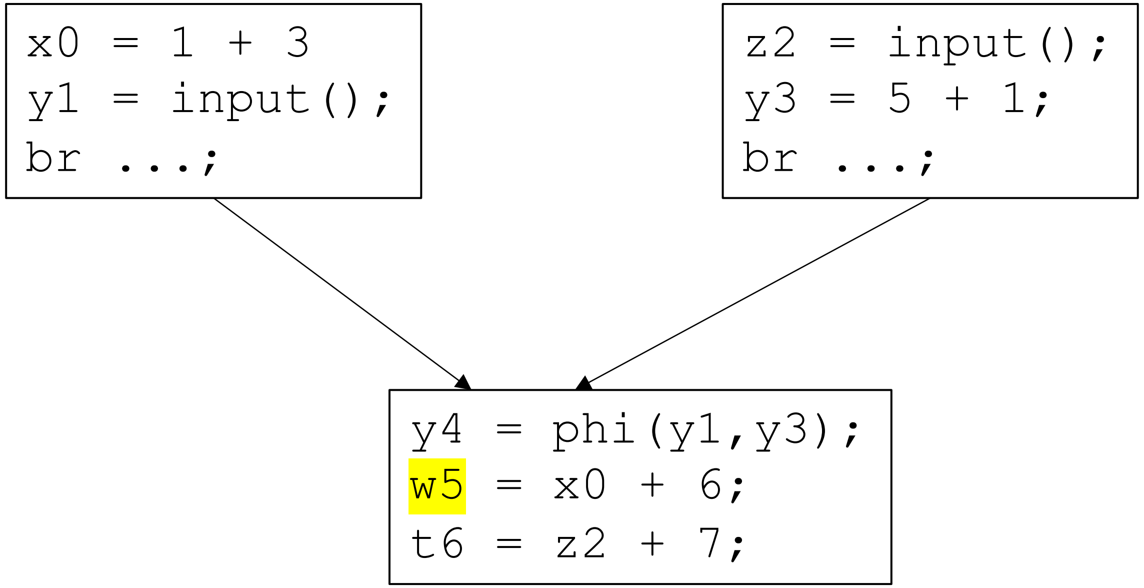



```
Value {
   x0 : 4
   y1 : B
   z2 : B
   y3 : 6
   y4 : T
   w5 : T
   t6 : T
 }
```

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

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

Example:

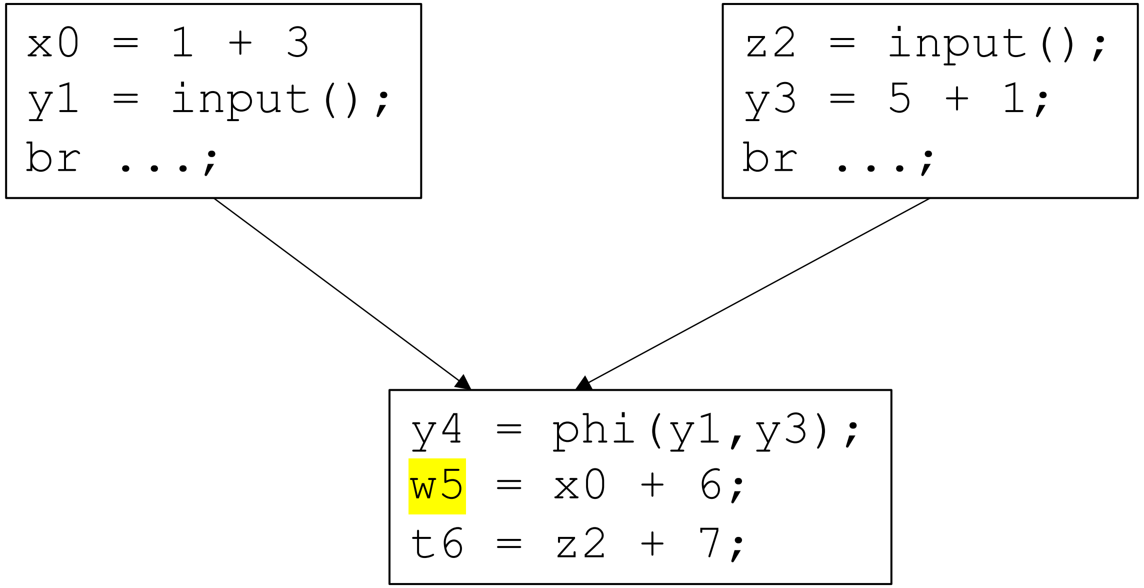



```
Value {
  x0 : 4
  y1 : B
  z2 : B
  y3 : 6
  y4 : T
  w5 : T
  t6 : T
}
```

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

Worklist: [x0,y1,y3]

Example:




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

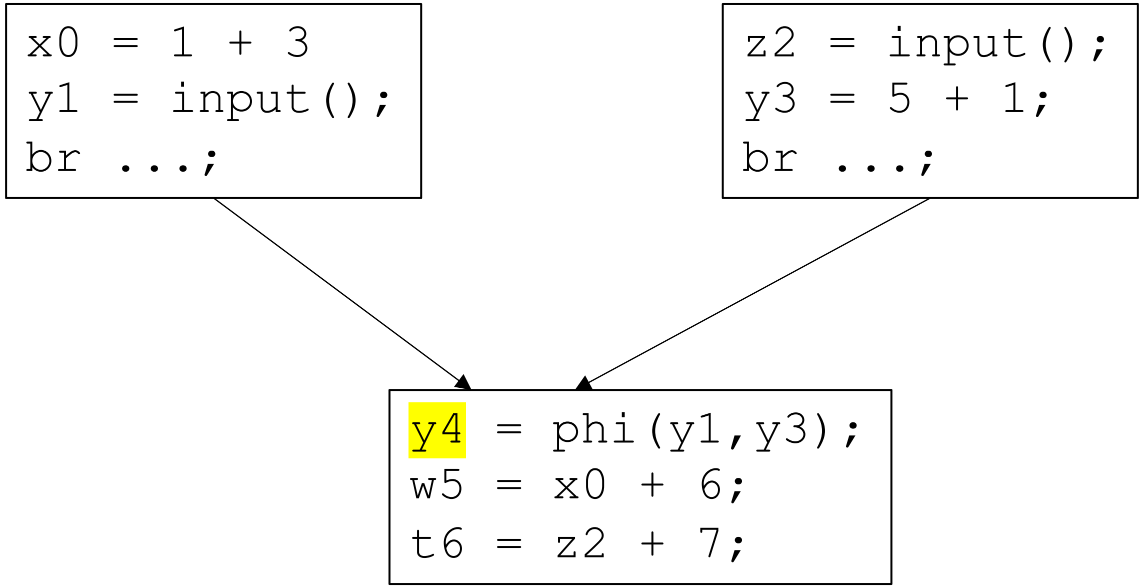
```
Uses {
  x0 : [w5]
  y1 : [y4]
  z2 : [t6]
  y3 : [y4]
  y4 : []
  w5 : []
  t6 : []
}
```

Worklist: [x0,y1,y3]

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

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 ...;
```

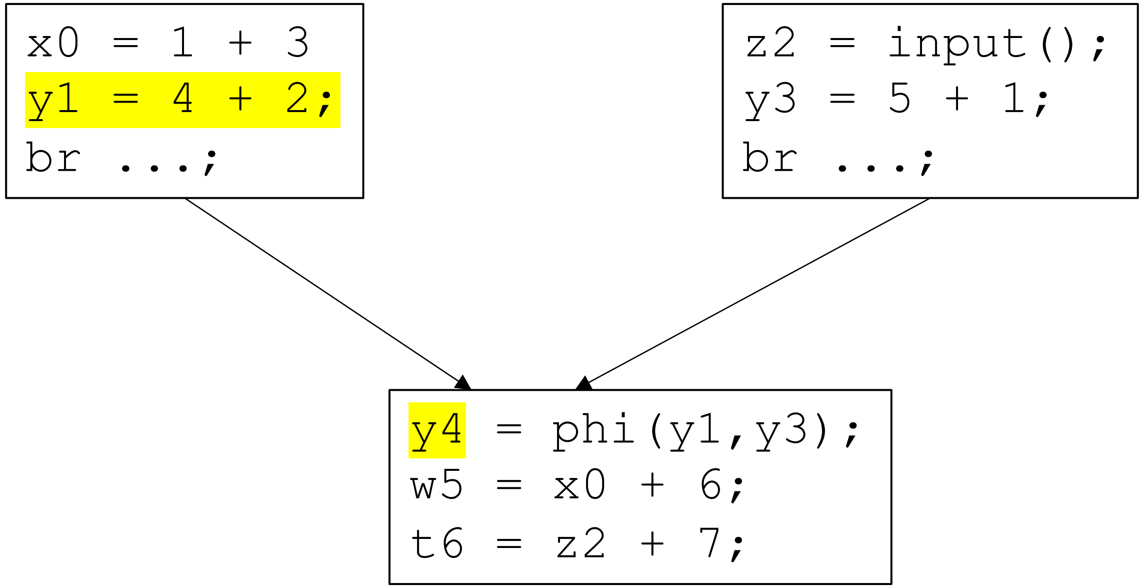
```
y4 = phi(y1,y3);
w5 = x0 + 6;
t6 = z2 + 7;
```

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:




```
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

Constant propagation algorithm

evaluate m over the lattice:

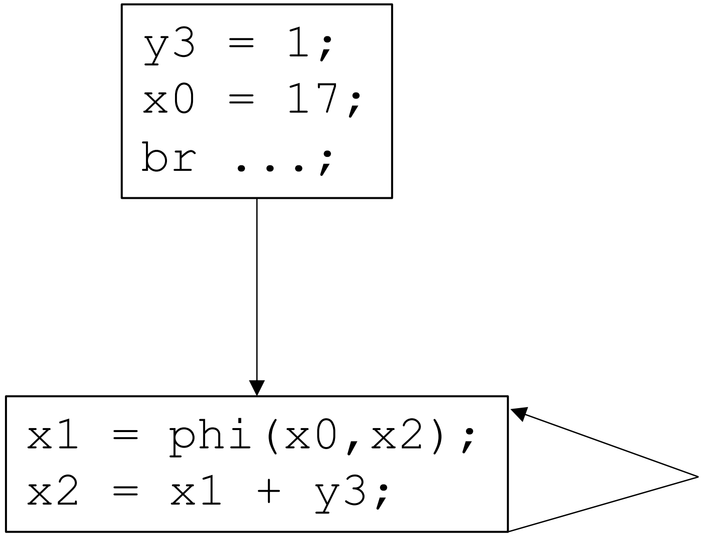
Example: $m = \phi(x_1 , x_2)$

Value(m) = $x_1 \wedge x_2$

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

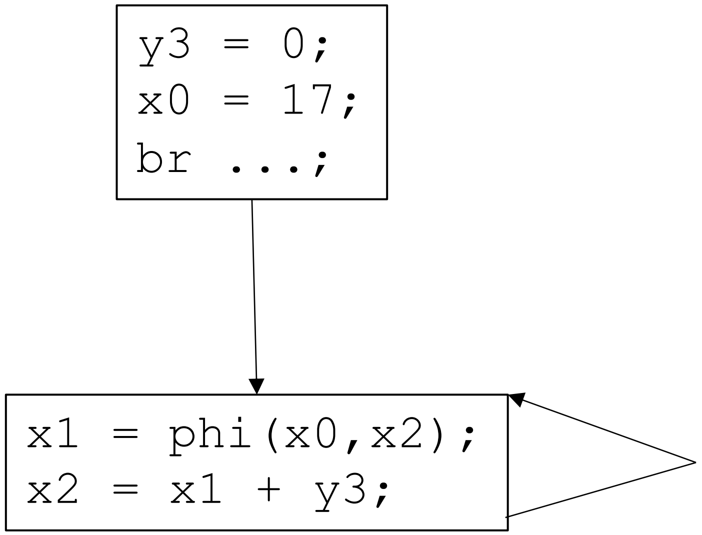
Issue here: potentially assigning a value that might not hold

Example loop:



x1:17

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: $\{c_1, c_2, c_3 \ldots\}$
- 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

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: ∧

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

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

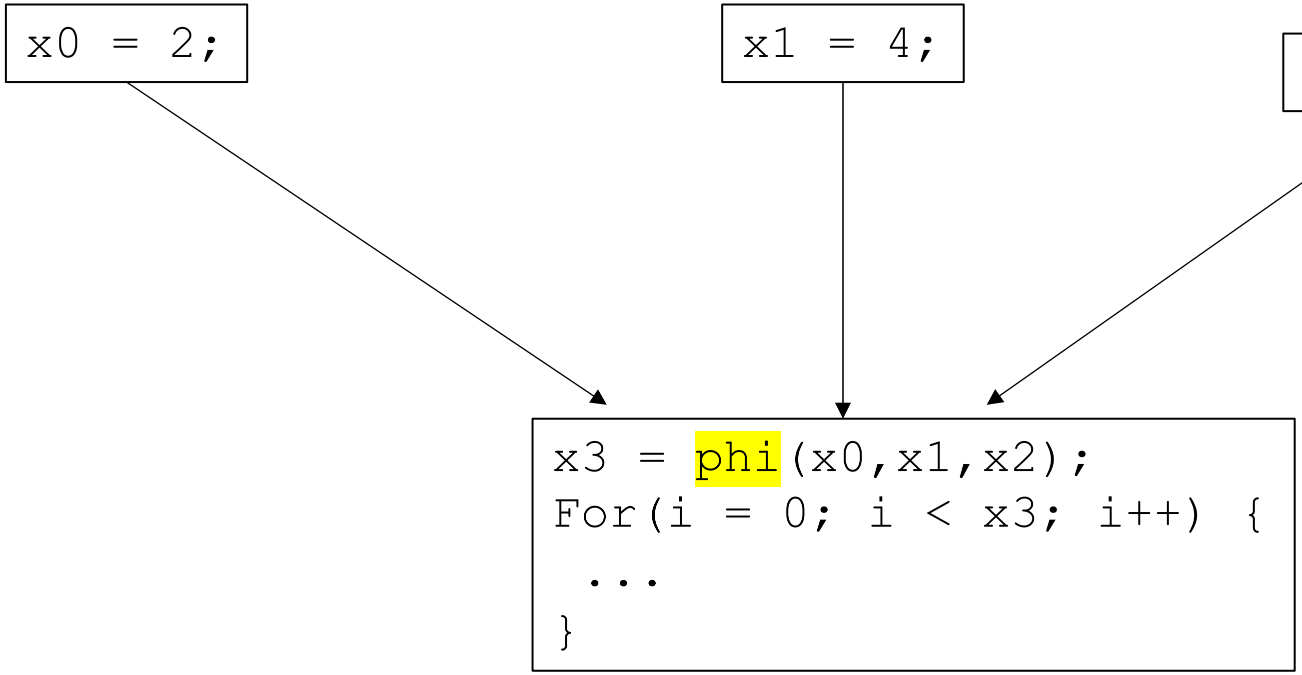
For Loop unrolling

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];
```

Have a nice weekend!

- See you in office hours or in a week!