CSE211: Compiler Design

Oct. 29, 2020

- **Topic**: Finish flow analysis.

  SSA form, producing SSA and optimization examples using SSA

- **Questions**:

  What did you think of using PLY in the homework? Pros, cons?

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

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

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

Announcements

- Homework 1 is due today
  - I will be copying from your submission folder first thing tomorrow morning
  - I will try to grade within 2 weeks
- Module 3 is pushed back 1 week
- Midterm will be released in 1 week: given on Nov. 5, due in 1 week: Nov. 12.

CSE211: Compiler Design

Oct. 29, 2020

- **Topic**: Finish flow analysis.

  SSA form, producing SSA and optimization examples using 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
}
```

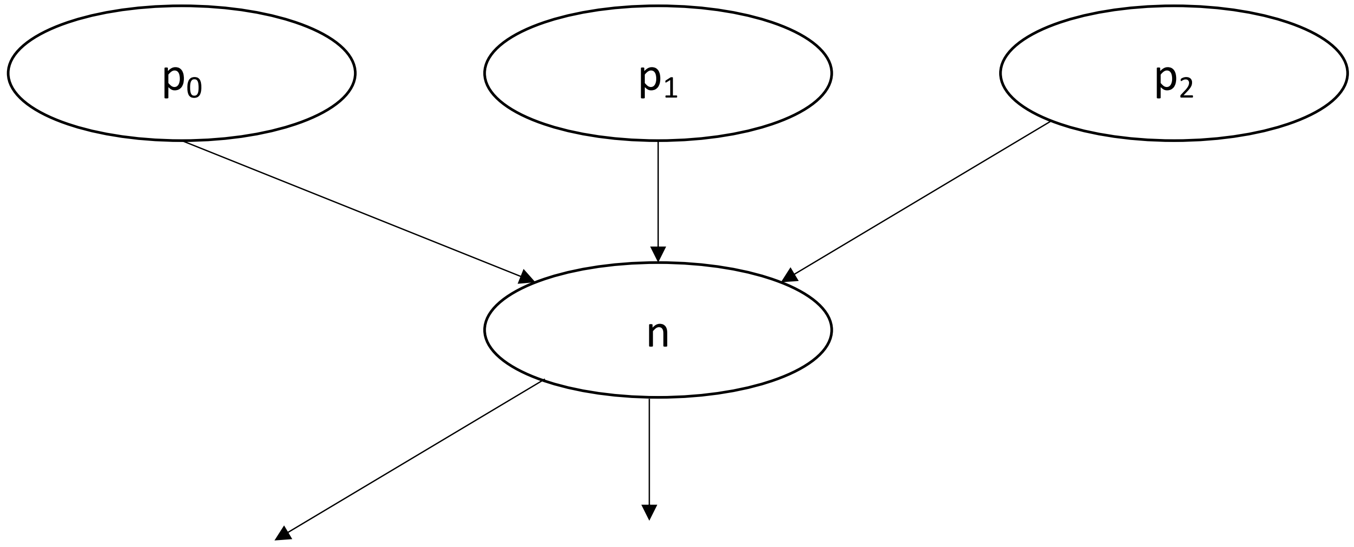
- **Questions**:

  What did you think of using PLY in the homework? Pros, cons?

Dominance

- dominators of node n are nodes for which every path from the start state, must be visited before reaching n

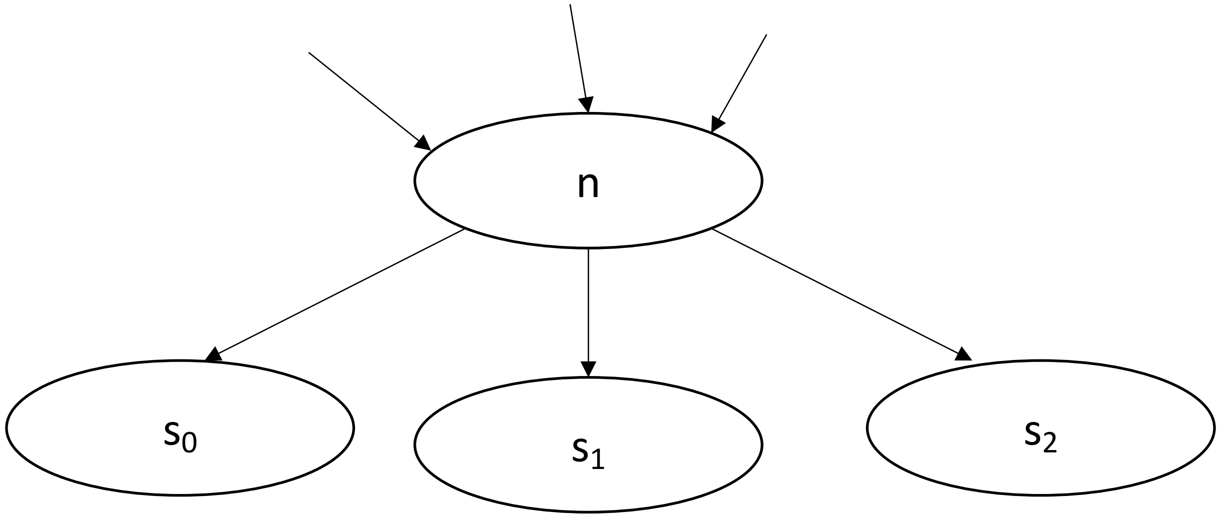
$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in } preds(n)} Dom(p) \right)$$

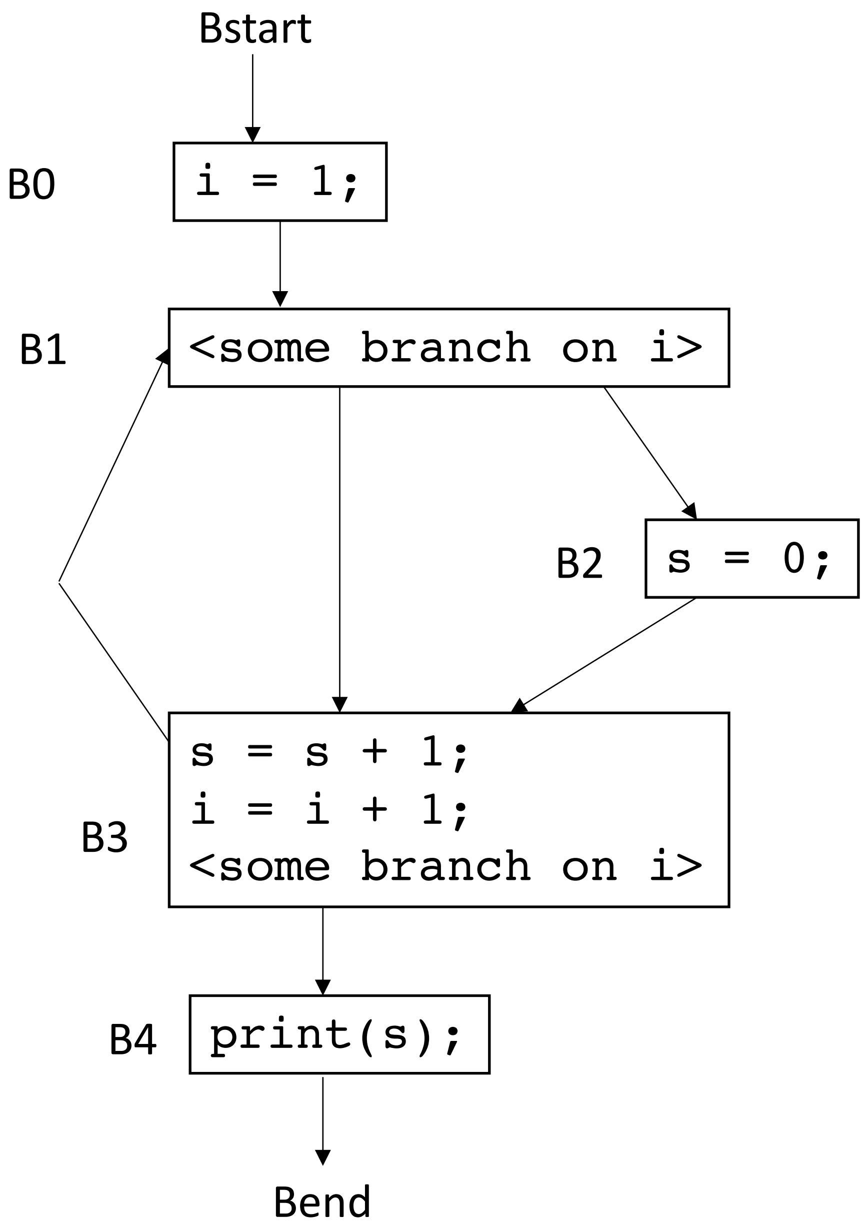


Live variable analysis in the CFG:

- A variable v is live in a node n if there exists some path in which the v is accessed (without being overwritten in the meantime)

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





Now we can perform the iterative fixed point computation:

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

Block	VarKill	UEVar	LiveOut I_0	LiveOut I_1	LiveOut I_2	LiveOut I_3
Bstart	{}	{}	{}	{}	{}	s
B0	i	{}	{}	i	s,i	s,i
B1	{}	i	{}	s,i	s,i	s,i
B2	s	{}	{}	s,i	s,i	s,i
B3	s,i	s,i	{}	s,i	s,i	s,i
B4	{}	s	{}	{}	{}	{}
Bend	{}	{}	{}	{}	{}	{}

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 potentially 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*

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

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;
```

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

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*

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

Sound vs. Complete

- **Sound**: results might be false, but facts are never missed. i.e. if variable x is found to be live, it might not be. But there will never exist a variable y that is live, but not claimed to be.

- **Complete**: claims are always true, but true facts may be missed. i.e. if variable x is found to be live, then it definitely is. If variable y is NOT claimed to be live, then it still may be.

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

How to instantiate the UEVar and VarKill for sound/complete analysis w.r.t. memory?

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

Live variable limitations

Imprecision can come from CFG construction:

consider:

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

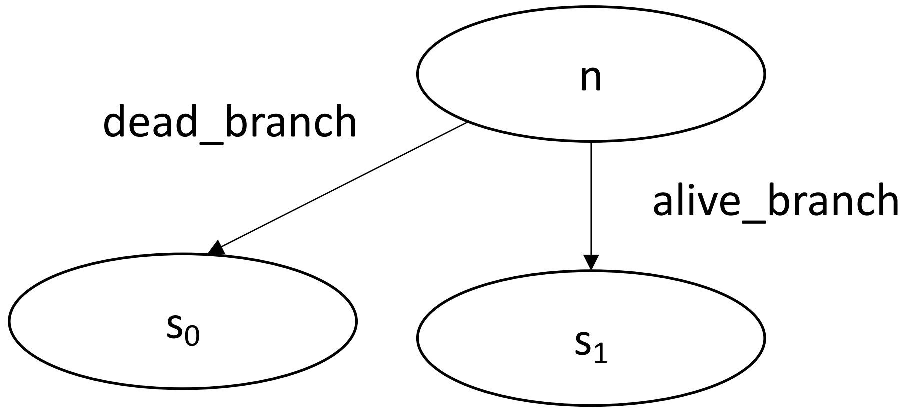
Live variable limitations

Imprecision can come from CFG construction:

consider:

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

could come from arguments, etc.



Live variable limitations

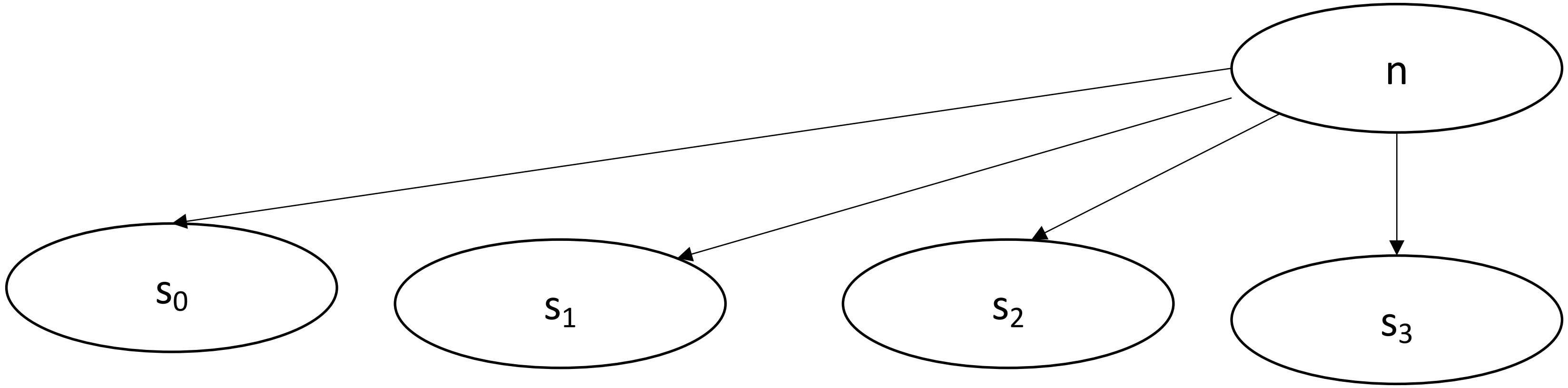
Imprecision can come from CFG construction:

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!



The Data Flow Framework

The Data Flow Framework

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

The Data Flow Framework

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

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

Available Expressions

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

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

Available Expressions

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

Forward Flow

Available Expressions

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

intersection implies “must” analysis

Available Expressions

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

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

Available Expressions

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

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

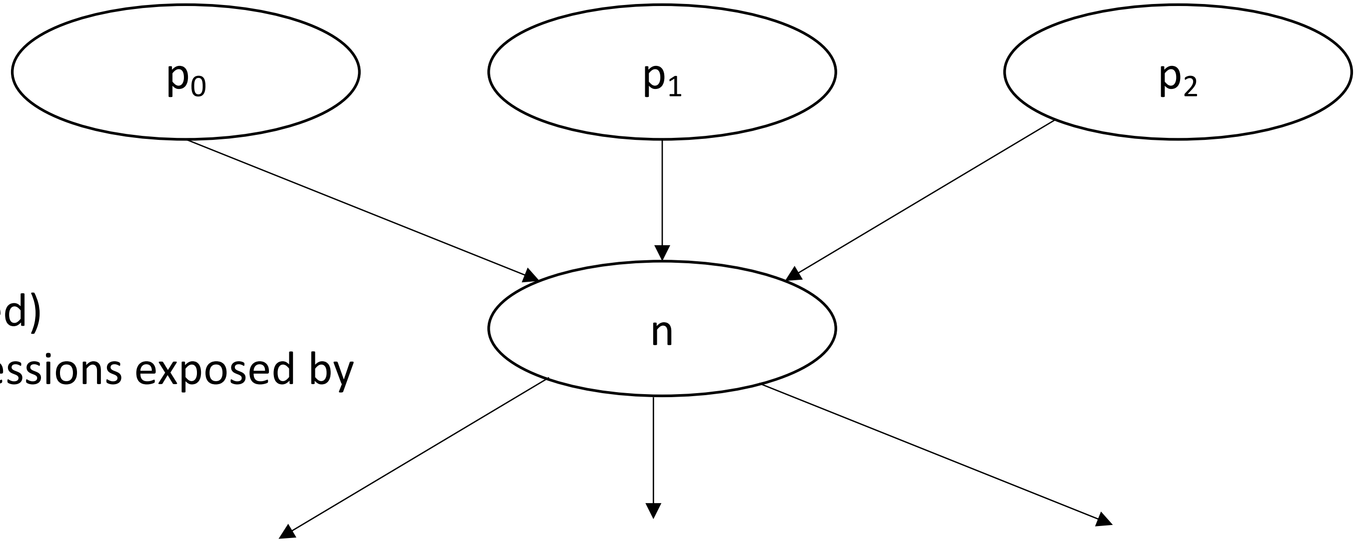
Available Expressions

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

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

Available Expressions

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



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

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

"must" analysis

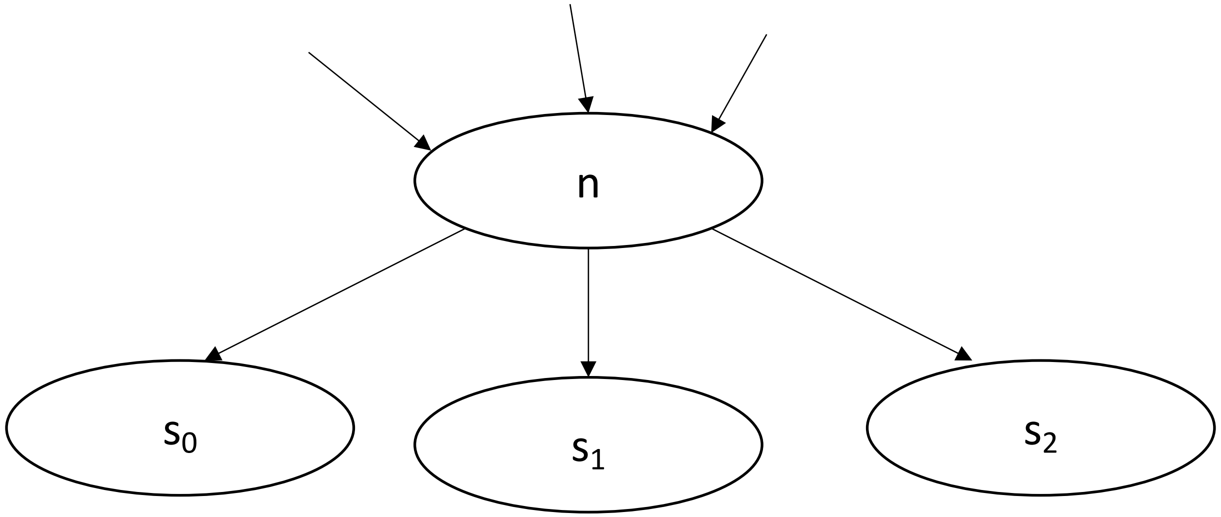
Anticipable Expressions

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

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

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

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

Reaching Definitions

- Read about this in 9.2.4
- trace variable usages in block b to possible definitions
- can be used in alias analysis

Static Single-Assignment Form (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

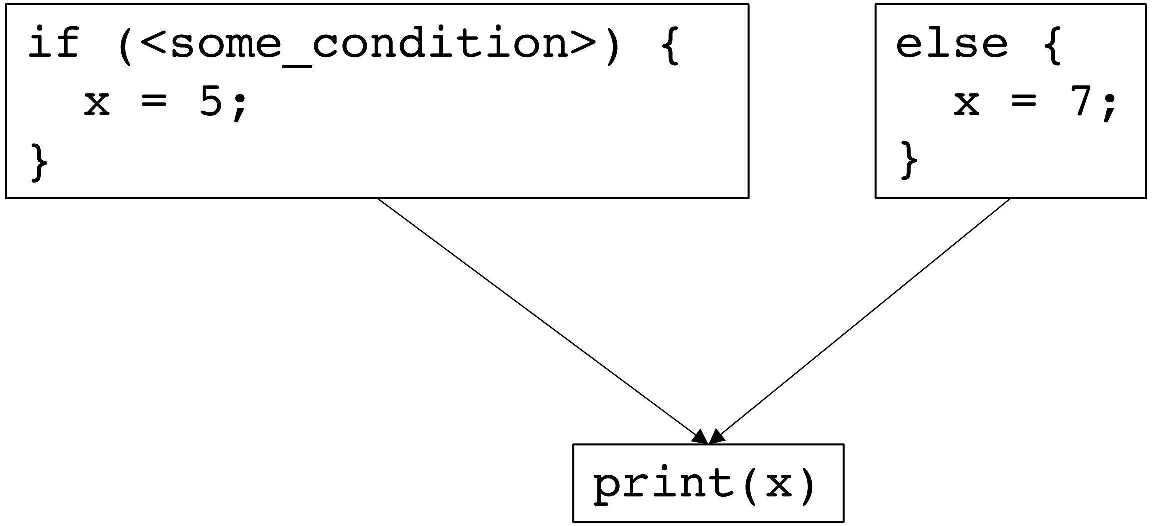
What here?

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

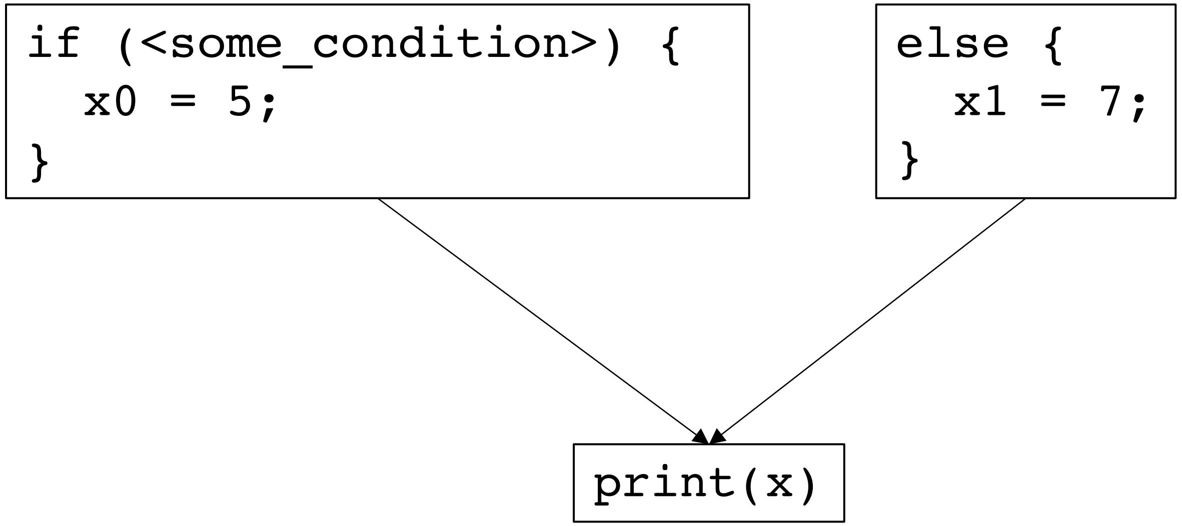


ϕ instructions

- Example: how to convert this code?

```
int x;

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

else {
  x1 = 7;
}

print(x)
```

number the variables

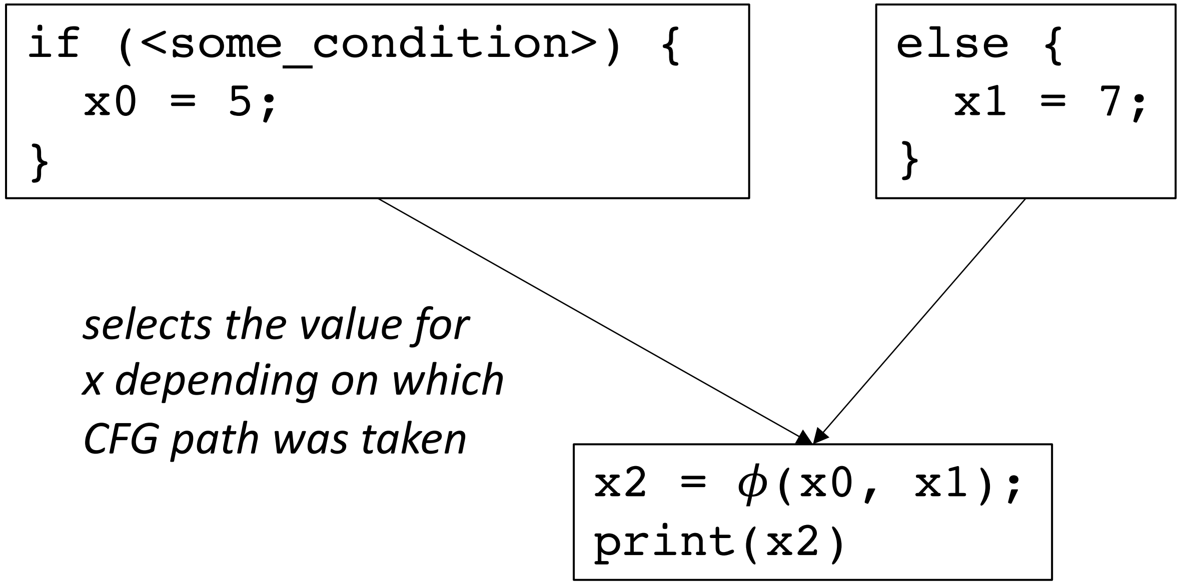


ϕ instructions

- Example: how to convert this code?

```
int x;

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

else {
  x1 = 7;
}

print(x)
```

number the variables



ϕ instructions

- LLVM example

ϕ instructions

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

- 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

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

Insert ϕ with argument placeholders

Example

```
x = 1;
y = 2;

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

else {
  x = 6;
  y = 100;
}

print(x)
```

```
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 (...) {
   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

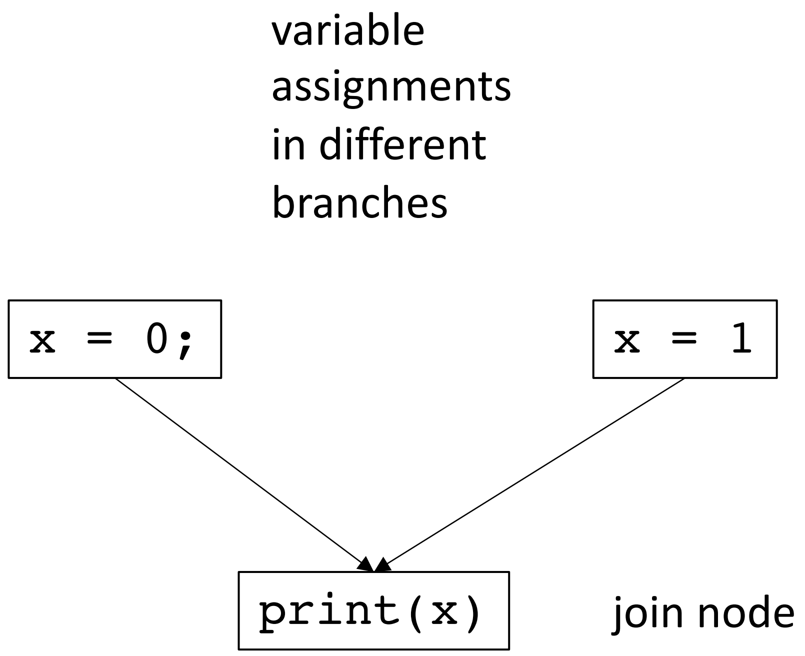
```
x0 = 1;
y1 = 2;

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

else {
  x8 = 6;
  y9 = 100;
}

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

A more optimal approach for ϕ placements

- When is a ϕ needed?

A more optimal approach for ϕ placements

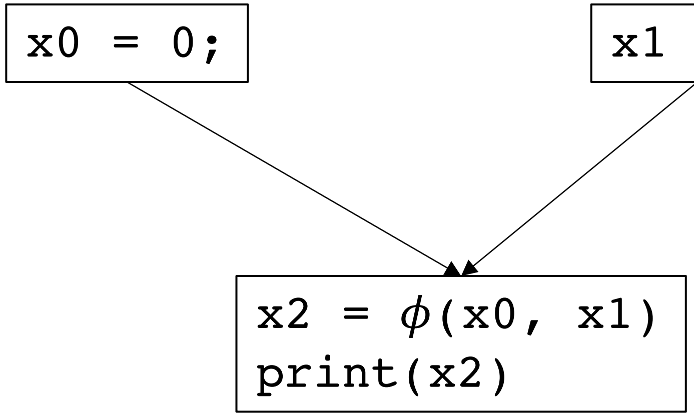
- When is a ϕ needed?



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?

- 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 for variable 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 `x = 0;` what set of blocks need a ϕ node for variable x?

some path

block j `print(x);` Does block j need a ϕ for variable 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 (of variable v) in block i.

block i `x = 0;` what set of blocks need a ϕ node for variable x?

some path

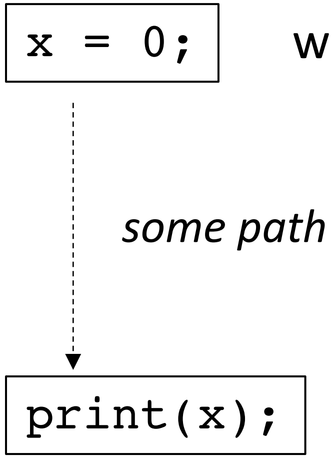
block j `print(x);` Does block j need a ϕ for variable x?

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

A more optimal approach for ϕ placements

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

block i



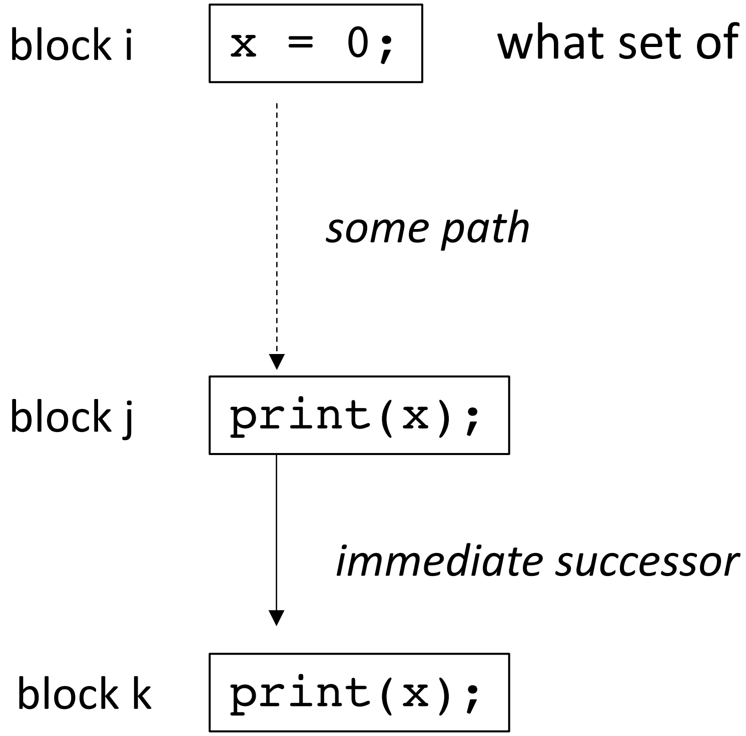
block j

what set of blocks need a ϕ node for variable 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 for variable x?



A more optimal approach for ϕ placements

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

block i `x = 0;`

what set of blocks need a ϕ node for variable x?

some path

block j `print(x);`

immediate successor

block k `print(x);`

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

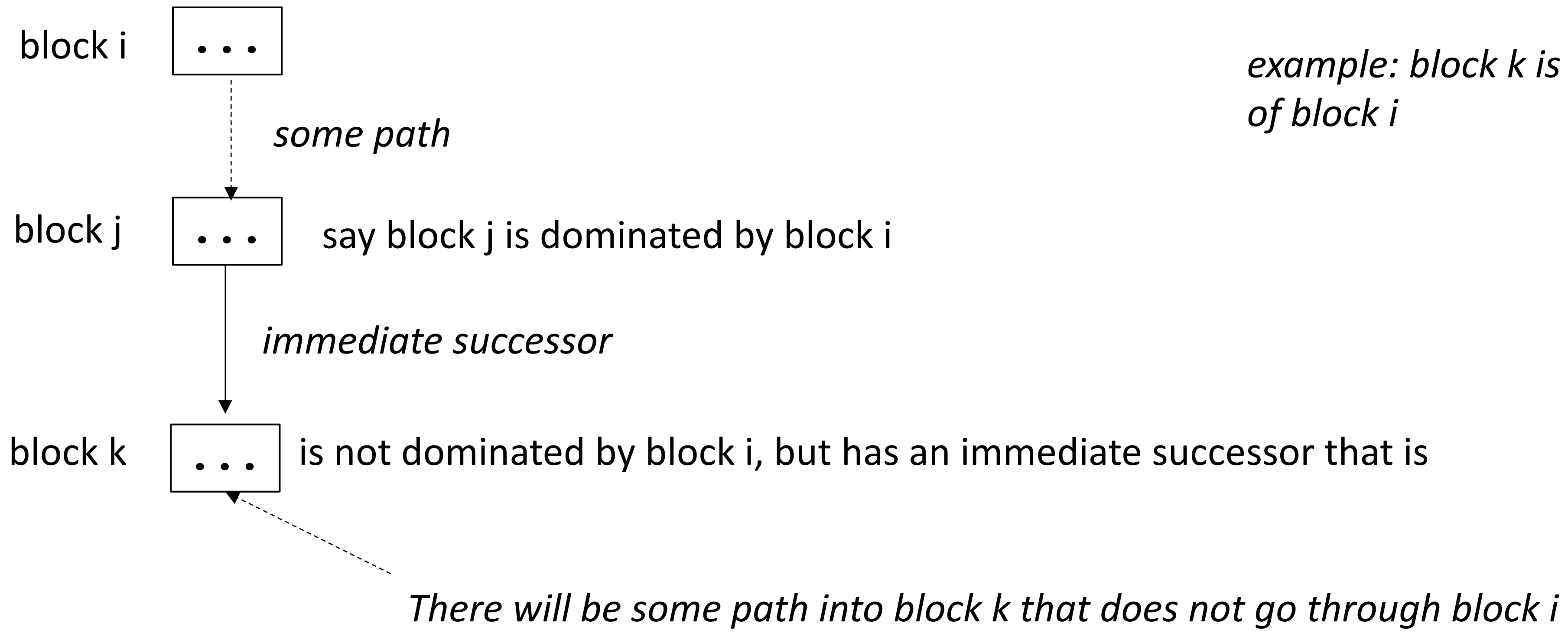
Say block k is not dominated by node i.
Then there exists another in-edge to block k.

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

Dominance Frontier

- 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



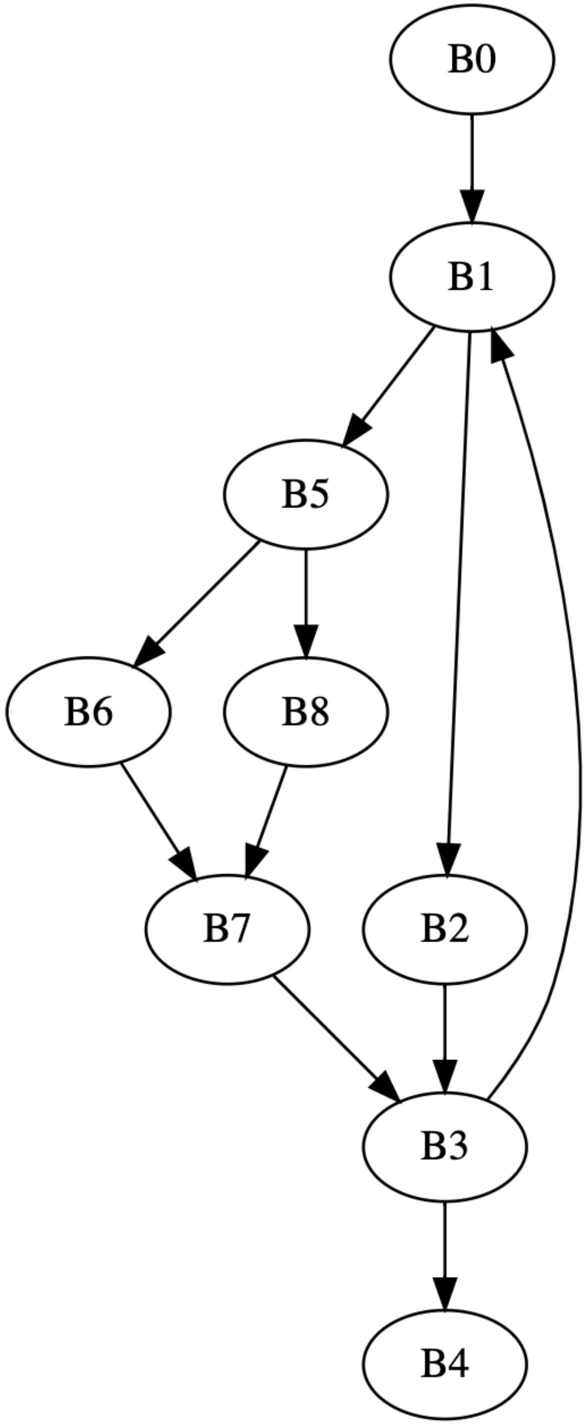
Dominance Frontier

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

Dominance Frontier

Candidates are join points: B1, B7, B3

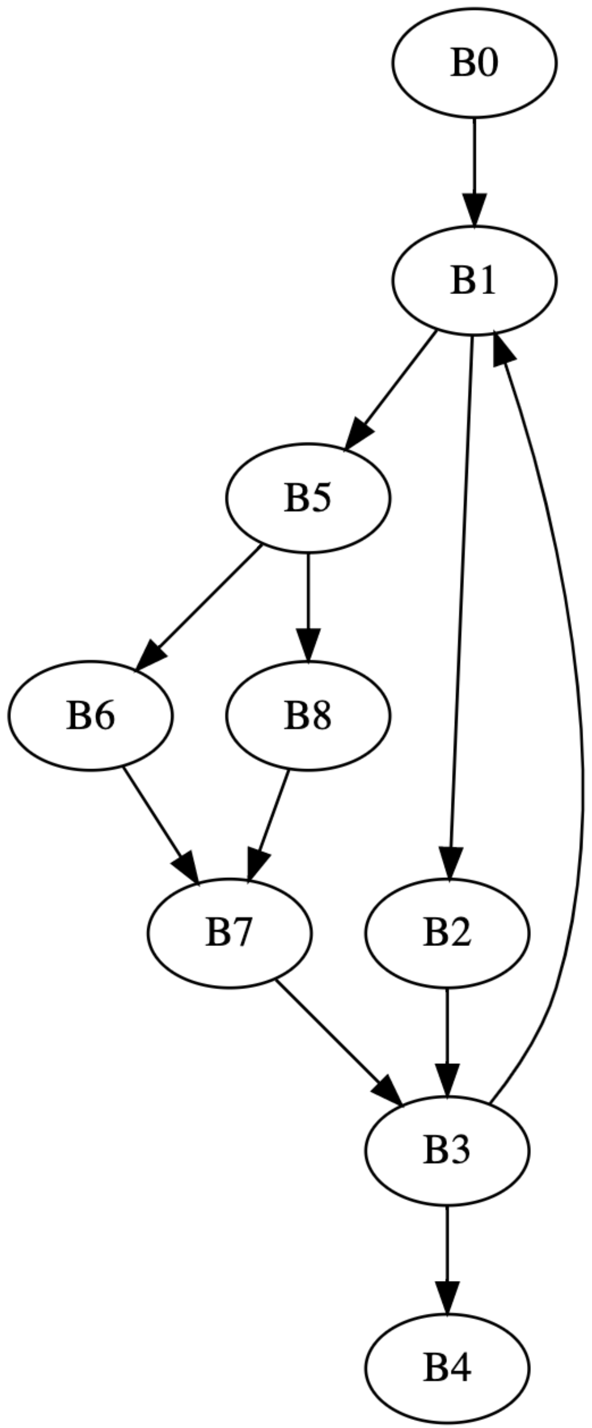
Node	Dominator Frontier	
B0	{}	first
B1	B1	fourth
B2	B3	third
B3		
B4		
B5	B3	second
B6	B7	
B7		
B8		



Dominance Frontier

Candidates are join points: B1, B7, B3

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

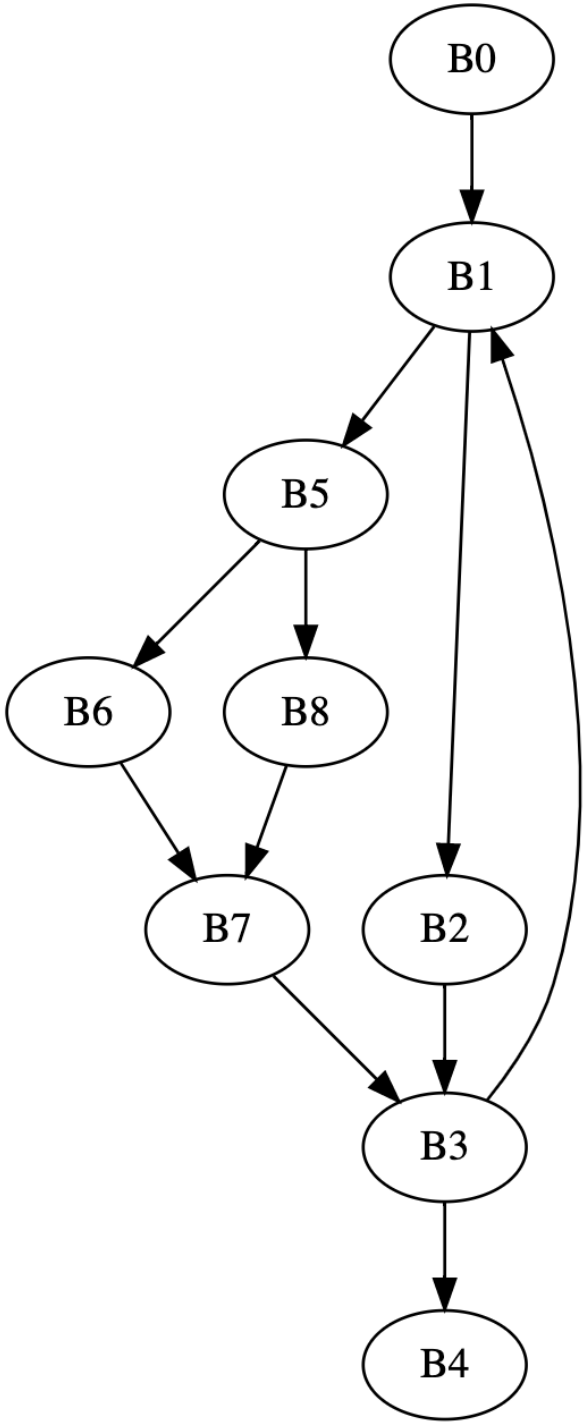


Dominance Frontier

Candidates are join points: B1, B7, B3

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

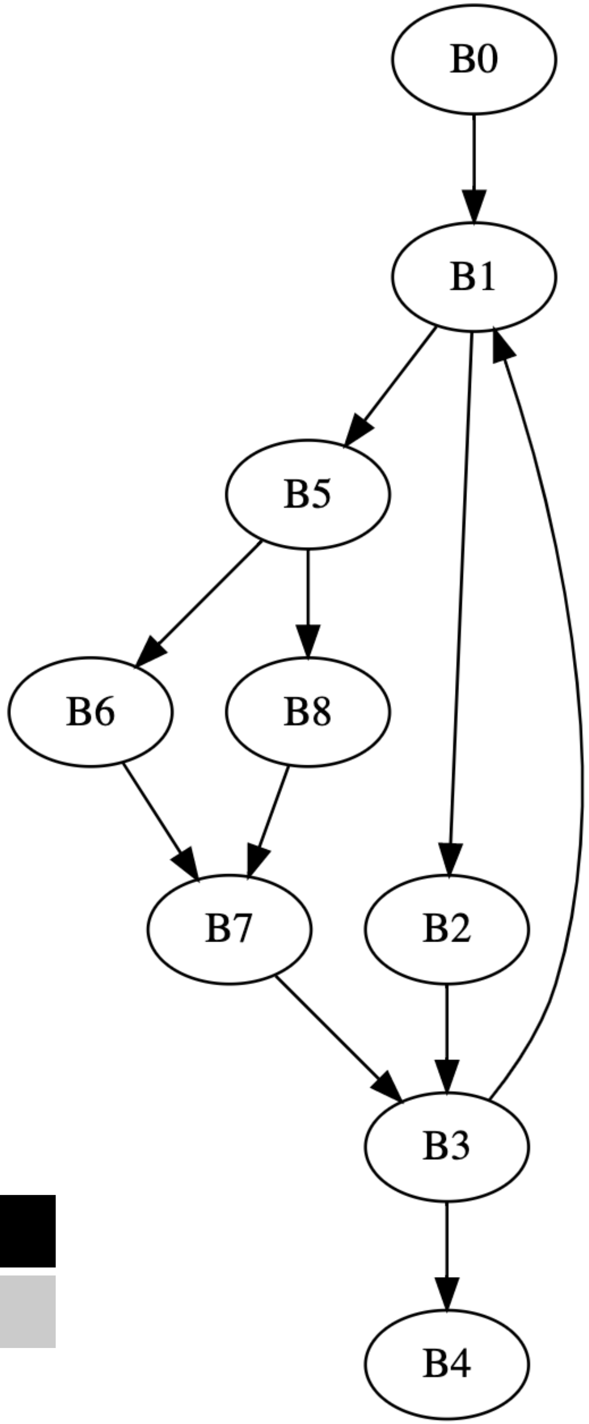
Use strict dominance (nodes don't dominate themselves)



Variable Assignment-to-Block Map

```
B0: i = ...;

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

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

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

B4: return;
```

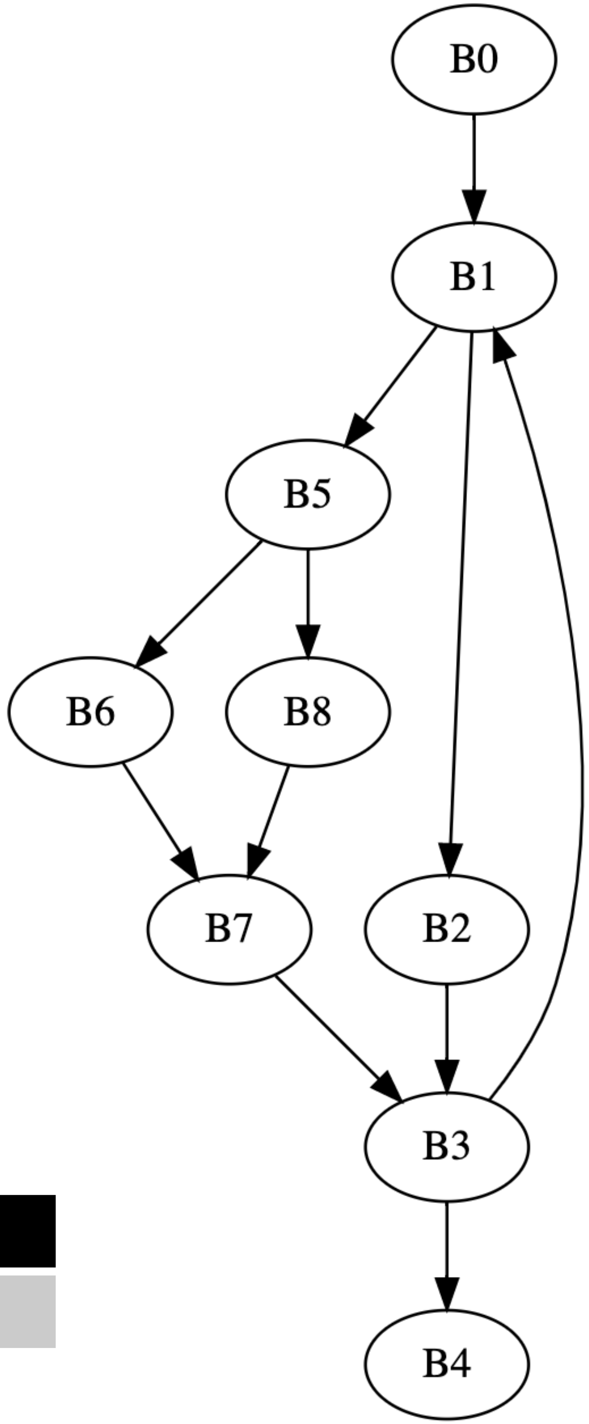
```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

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



Var	a	b	c	d	i	y	z
Blocks	B1, B5						

```
B0: i = ...;

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

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

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

B4: return;
```

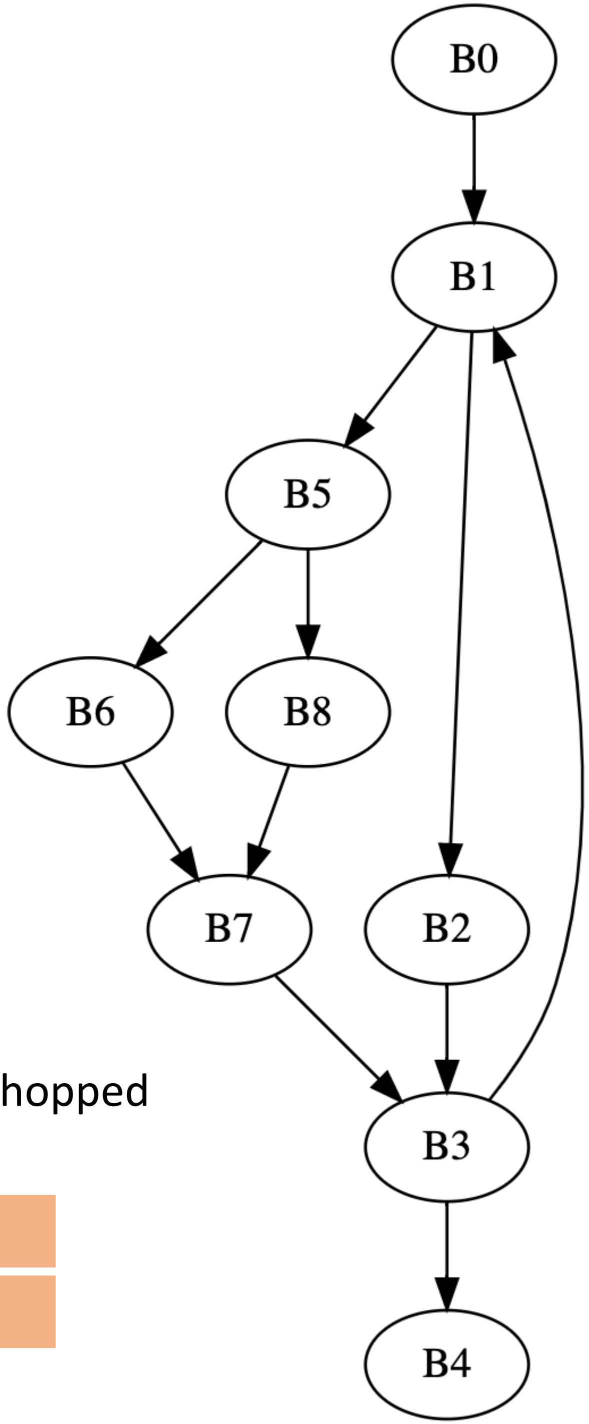
```
B5: a = ...;
    d = ...;
    br ... B6, B8;

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 block b:
    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 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: 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: 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

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;
```

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;
```

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

```
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

```
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

```
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

Next lecture

- Variable renaming with pruned ϕ's
- Global Constant Propagation using SSA