CSE211: Compiler Design

Oct. 25, 2021

- **Topic**: SSA intermediate representation

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

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

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

Announcements

- Homework 2:
  - Due Nov. 1
  - Great questions on slack!
  - I'll have office hours this Thursday
- Midterm assigned on Wednesday by midnight!
  - 1 week to do the midterm
  - Do not ask questions on slack, instead message me directly! I will create a canvas discussion with FAQs. Only I can post!
  - Do not discuss with classmates until after the due date
  - Plan on about 2.5 hours (not including studying!)
    - Students have reported anywhere from 2 to 7 hours

CSE211: Compiler Design

Oct. 25, 2021

- **Topic**: SSA intermediate representation

```
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
}
```

Review: Flow analysis

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

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

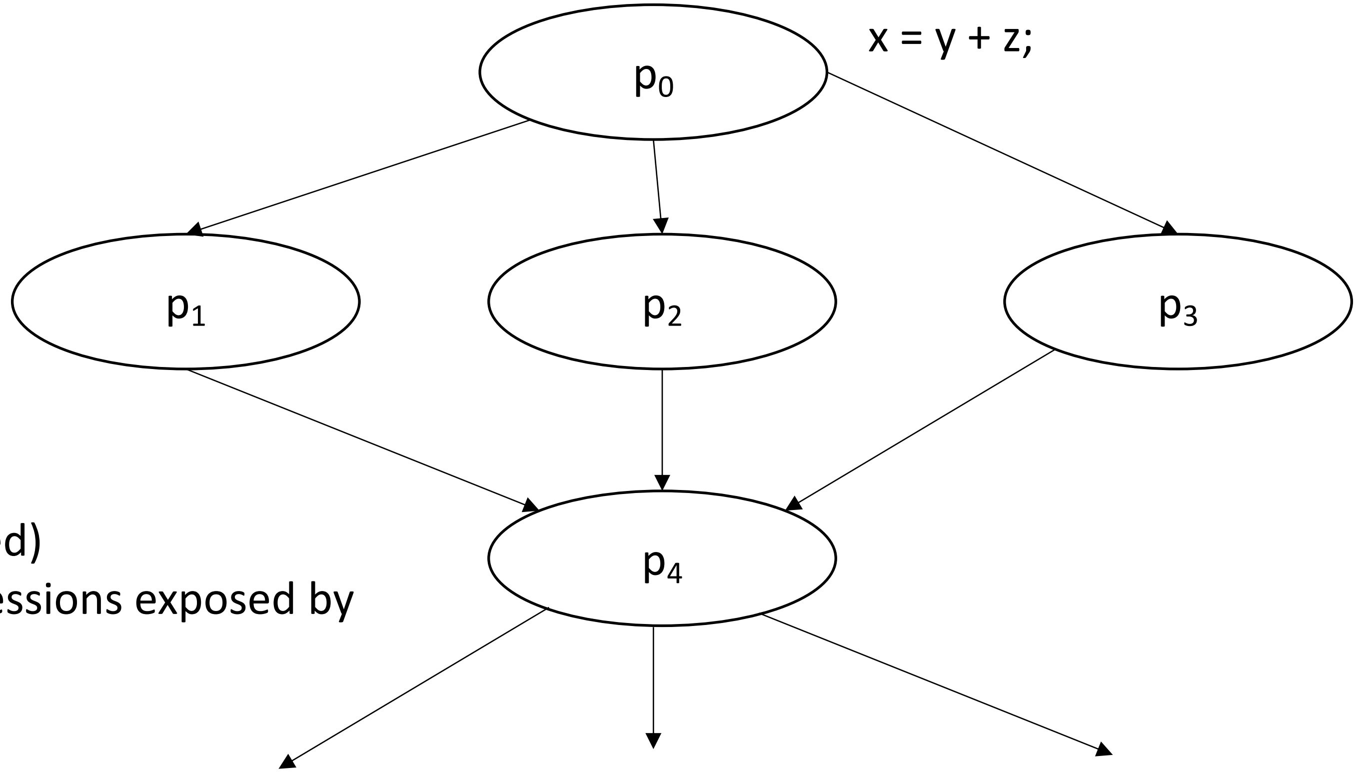
Review: Flow analysis

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

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

Review: 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.

Review: Flow analysis

- Sound analysis: potential false positives

- Complete analysis: no false positives, but might miss some bugs

- *In practice: usually somewhere in between*

Godbolt example

- Clang is pretty good at warning for uninitialized variables
  - But doesn’t do much with memory

- GCC is better at warning for memory, but not so good for variables

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

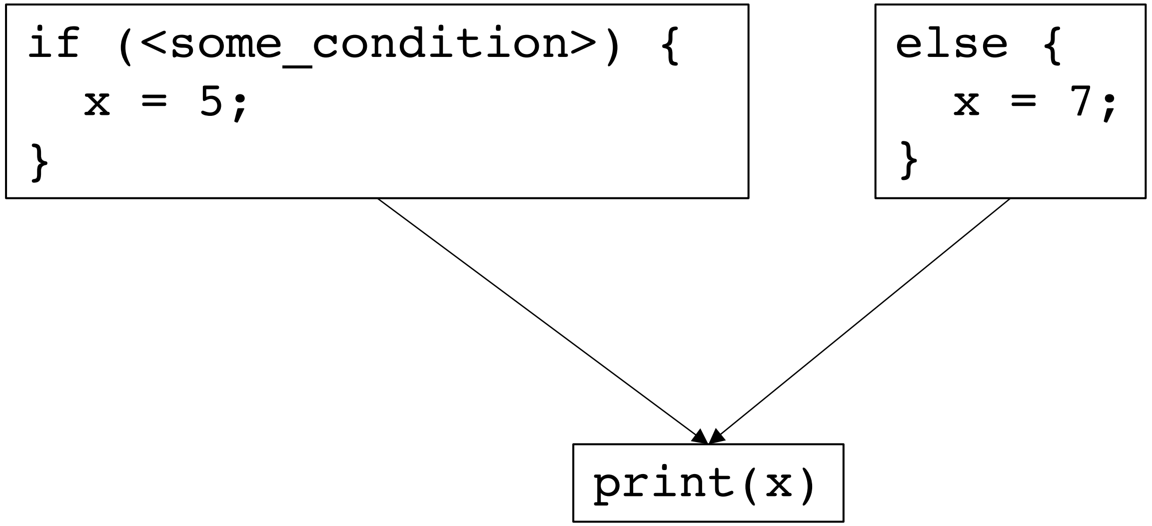
What here?

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

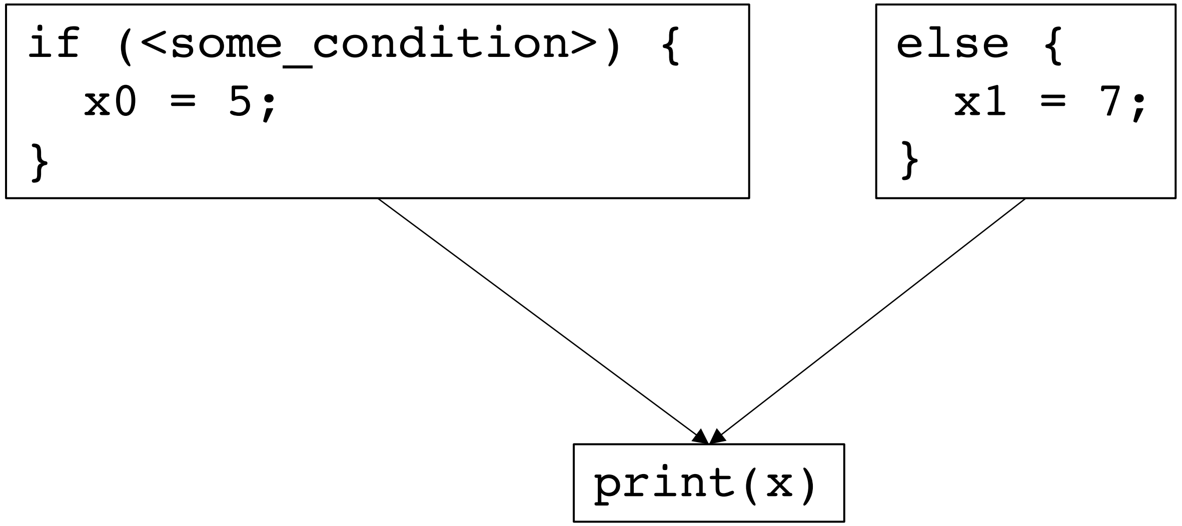


ϕ instructions

- Example: how to convert this code into SSA?

```
int x;

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

else {
  x1 = 7;
}

print(x)
```

number the variables

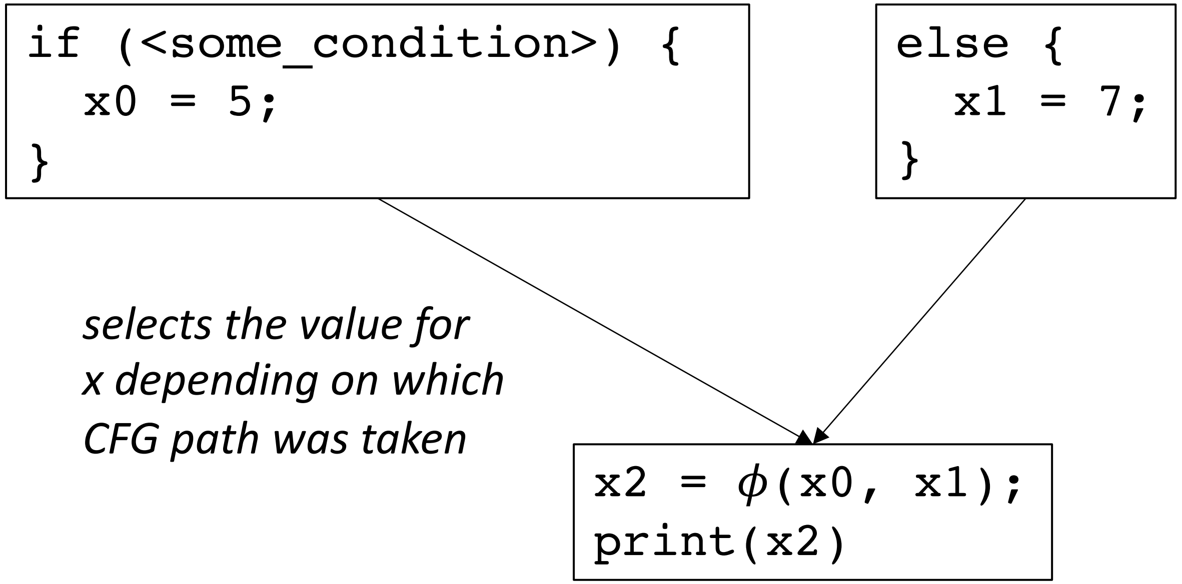


ϕ instructions

- 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

- LLVM example
  - Need ”opt” program to run -mem2reg

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

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 (...) {
  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

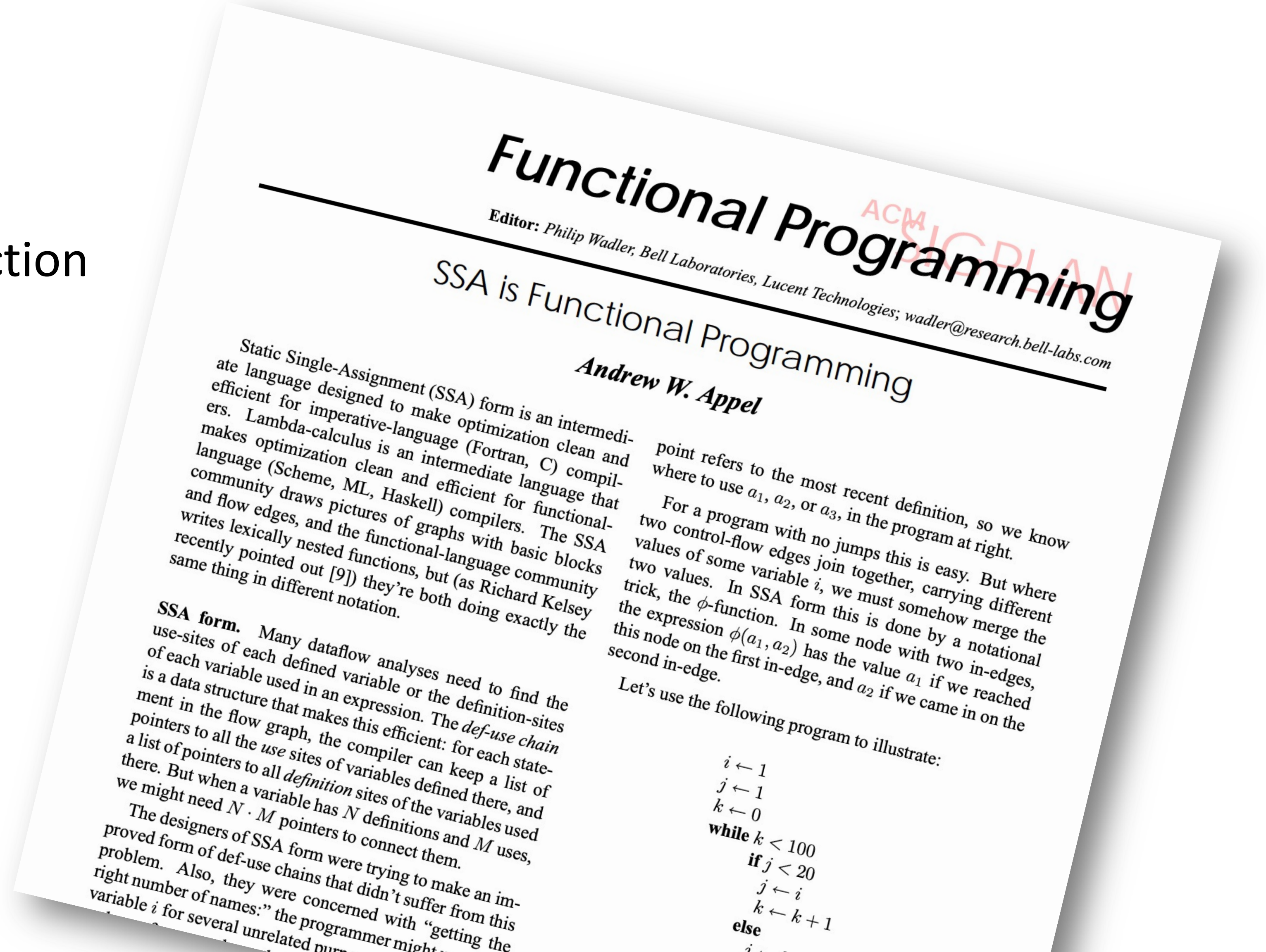
```
x0 = 1;
y1 = 2;

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 use the variable i for several unrelated purposes ...

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.

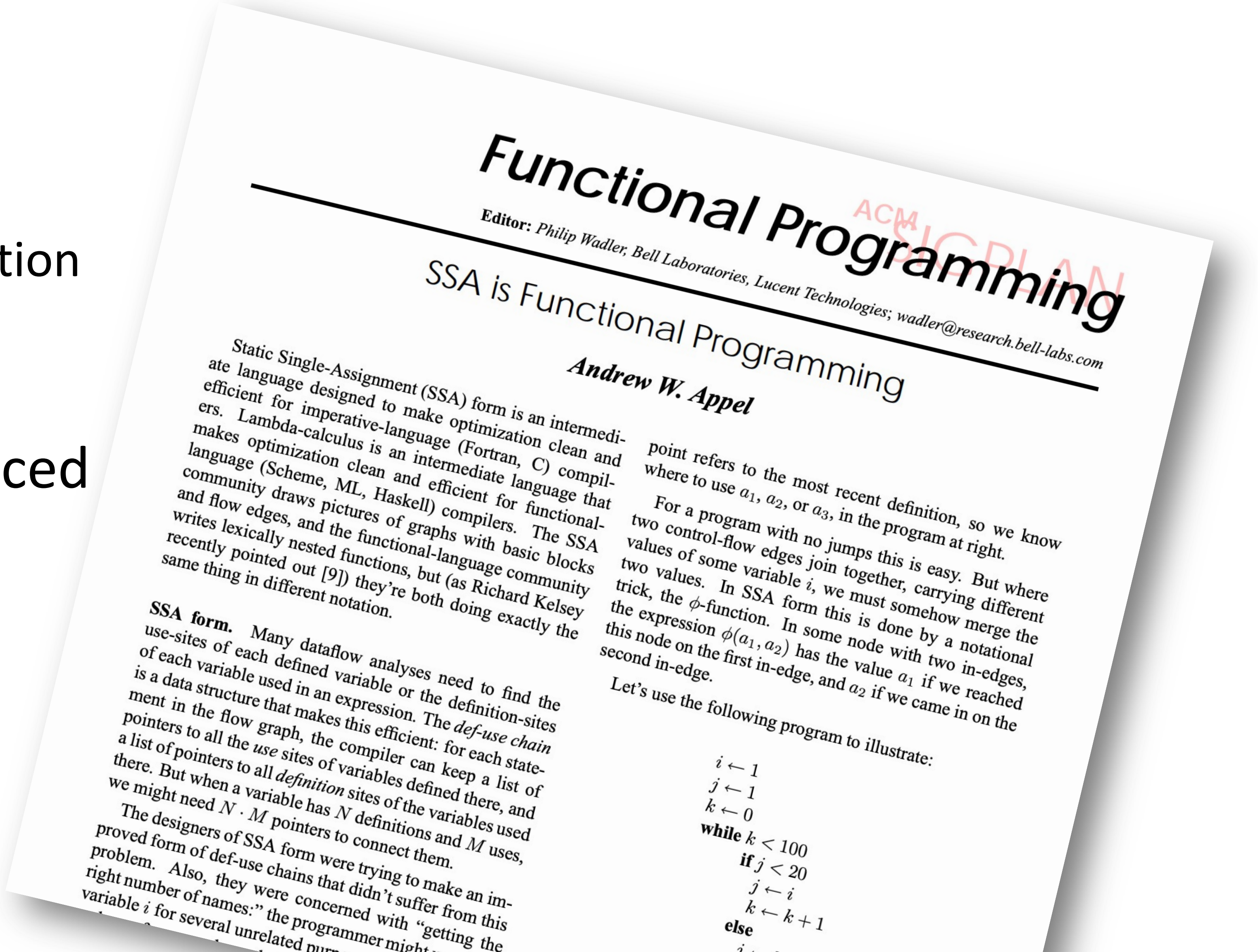
For a program with no jumps this is easy. But where two control-flow edges join together, carrying different values of some variable i, we must somehow merge the two values. In SSA form this is done by a notational trick, the ϕ-function. In some node with two in-edges, the expression $\phi(a_1, a_2)$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

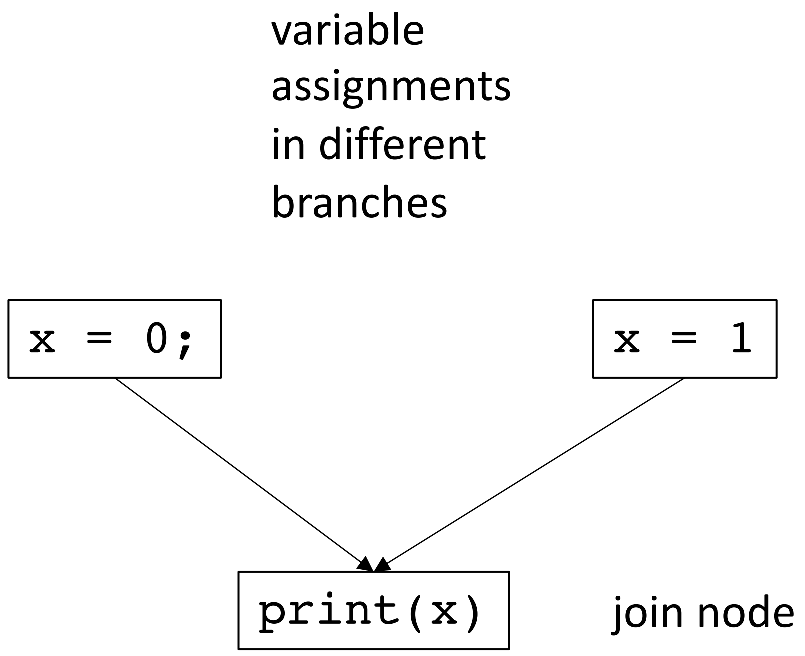
- EAC book describes:
  - Minimal SSA
  - Pruned SSA
  - **Semipruned SSA: We will discuss this one**

A more optimal approach for ϕ placements

- When is a ϕ needed?

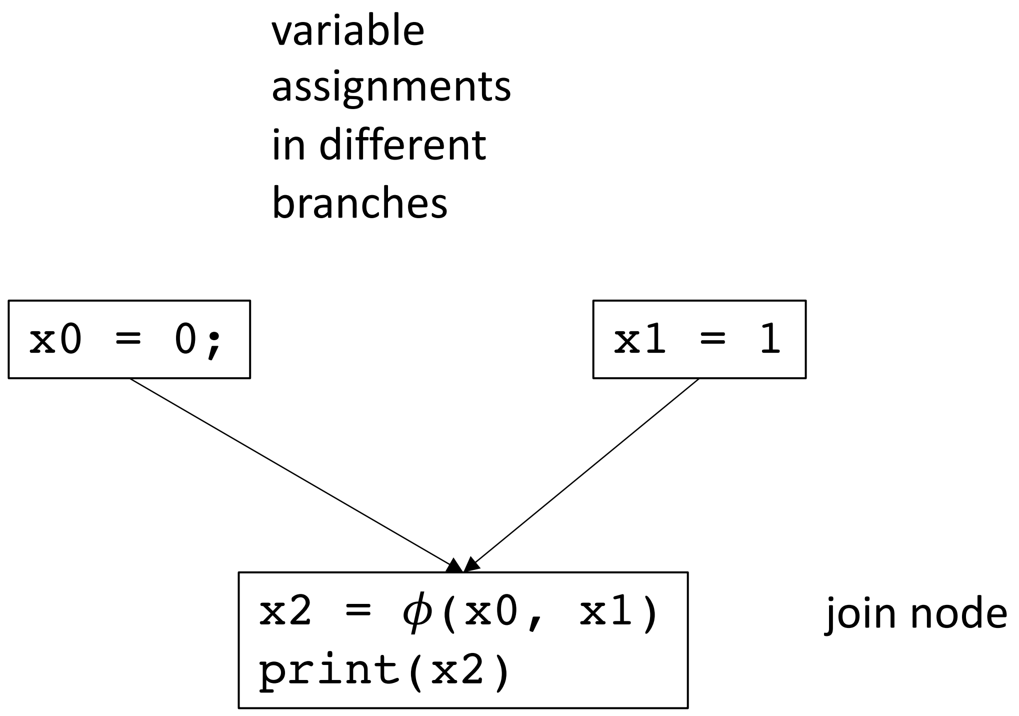
A more optimal approach for ϕ placements

- When is a ϕ needed?



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.

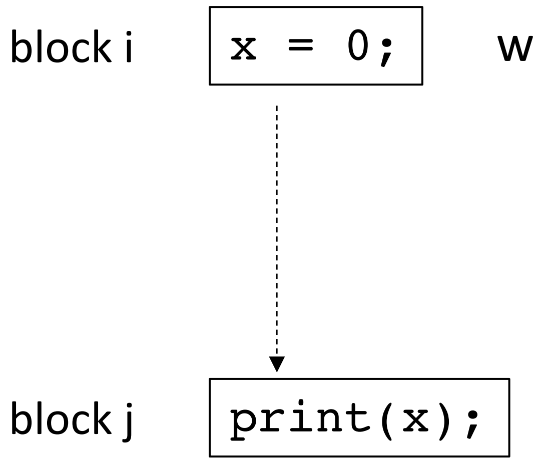
```
x = 0;
```

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.



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

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

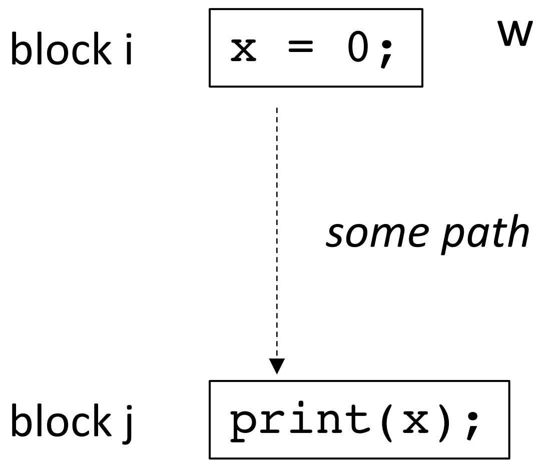
```
print(x);
```

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

A more optimal approach for ϕ placements

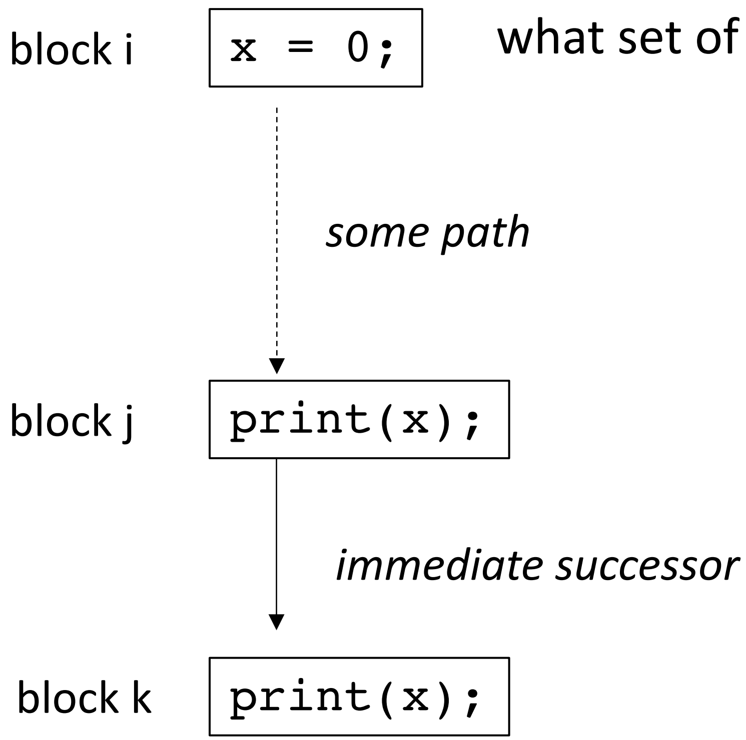
- say j is dominated by i. Thus, no ϕ node is needed in block j

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

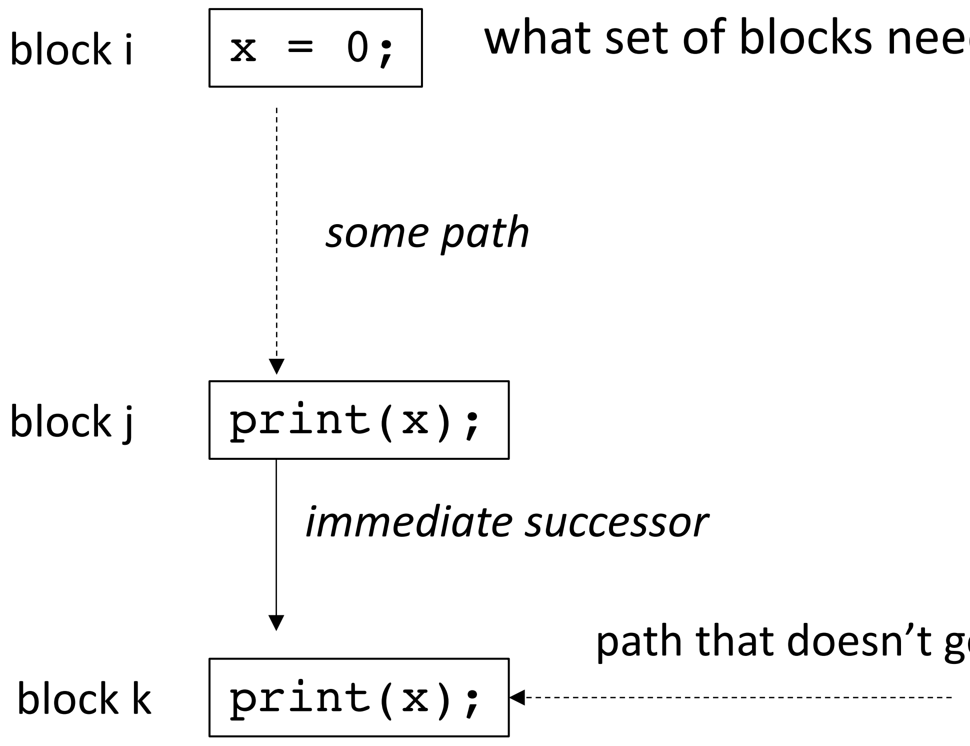


what set of blocks need a ϕ node to resolve conflicts on this assignment to x?

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?



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

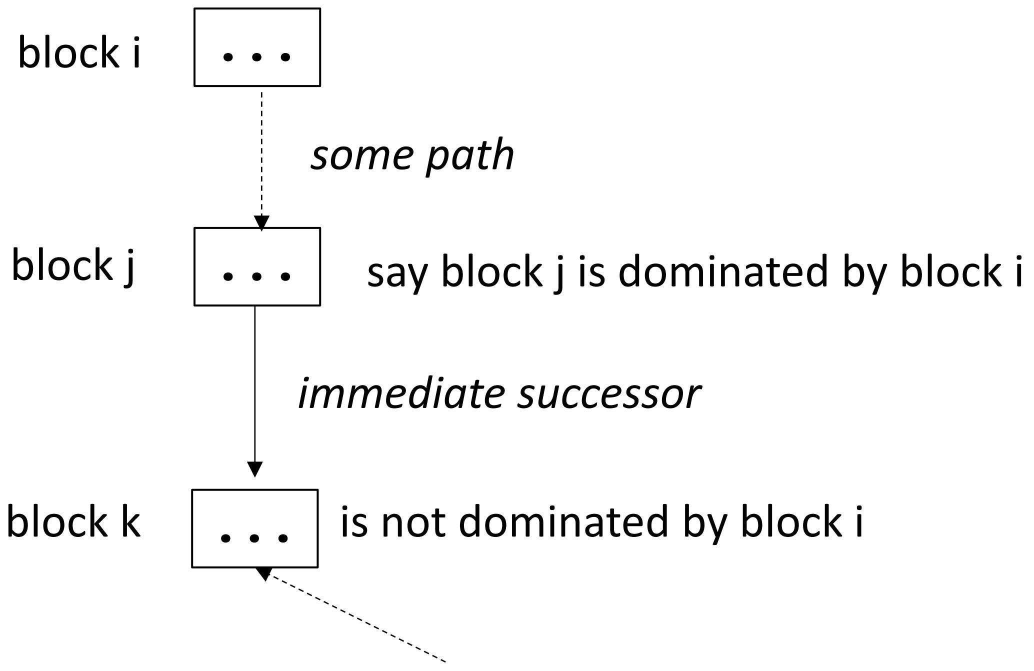
Say block k is not dominated by block i.
Then there exists another in-edge to block k.

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

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

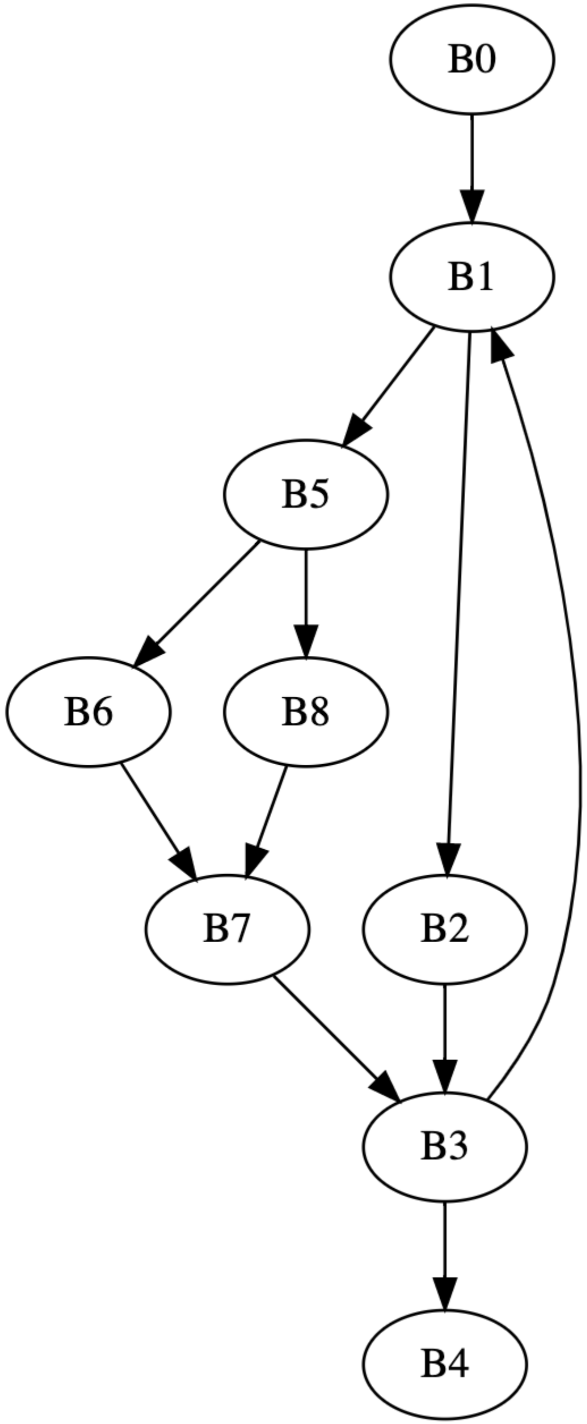
Dominance frontier

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

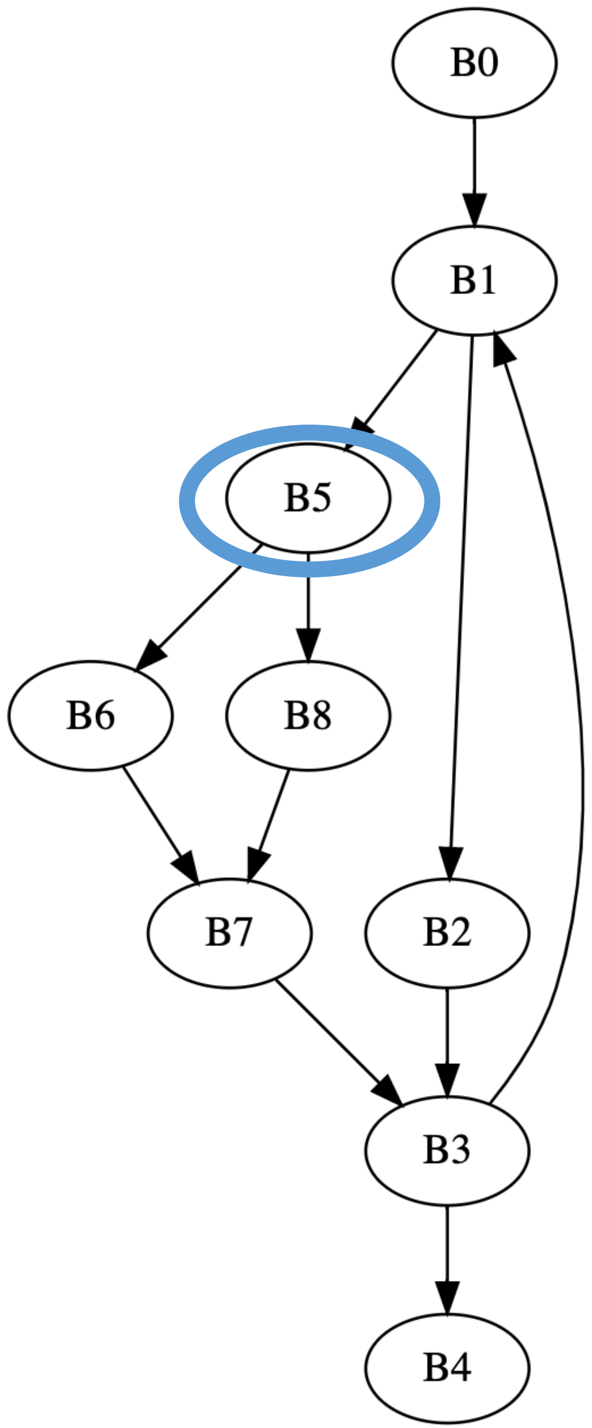
- Efficient algorithm for computing in EAC section 9.3.2 using a dominator tree. Please read when you get the chance!

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

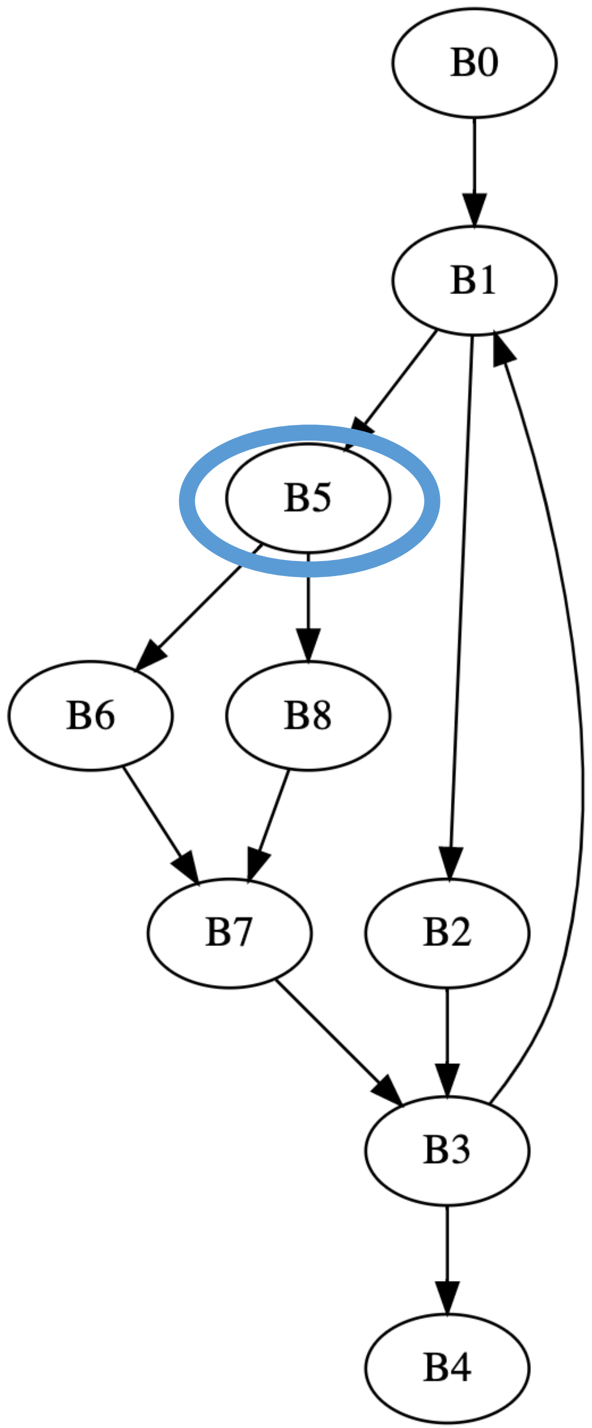
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



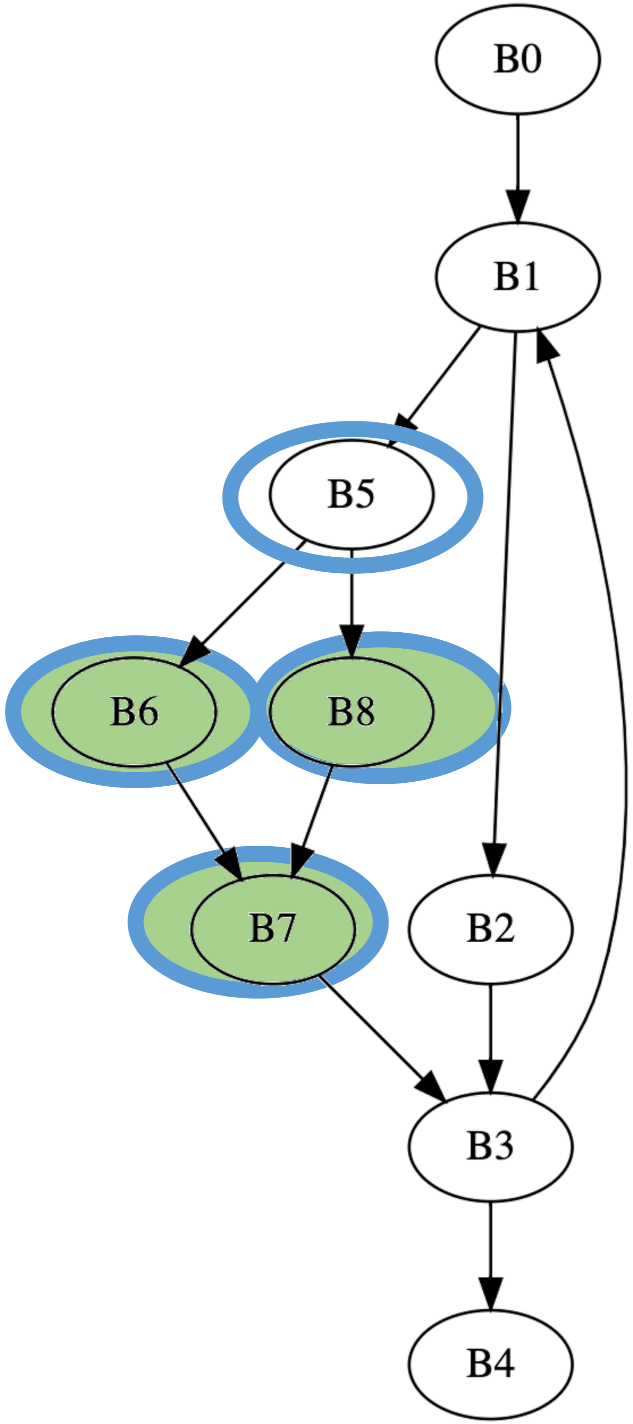
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



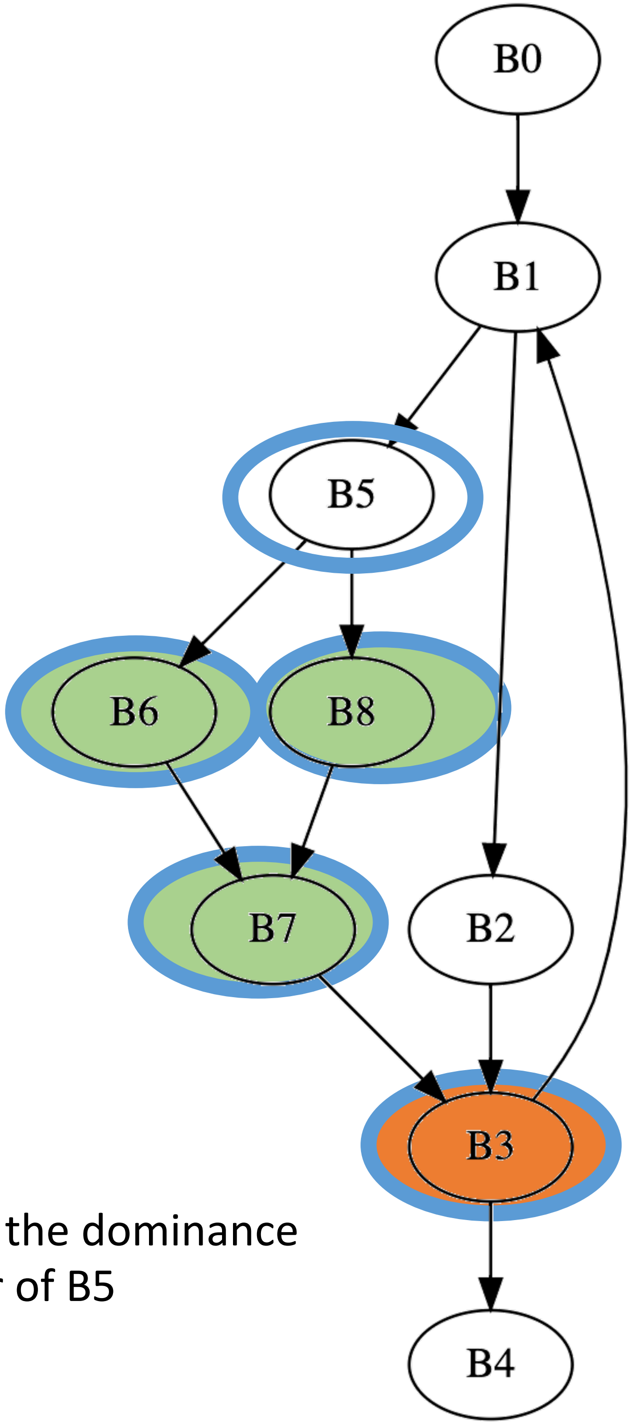
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



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,

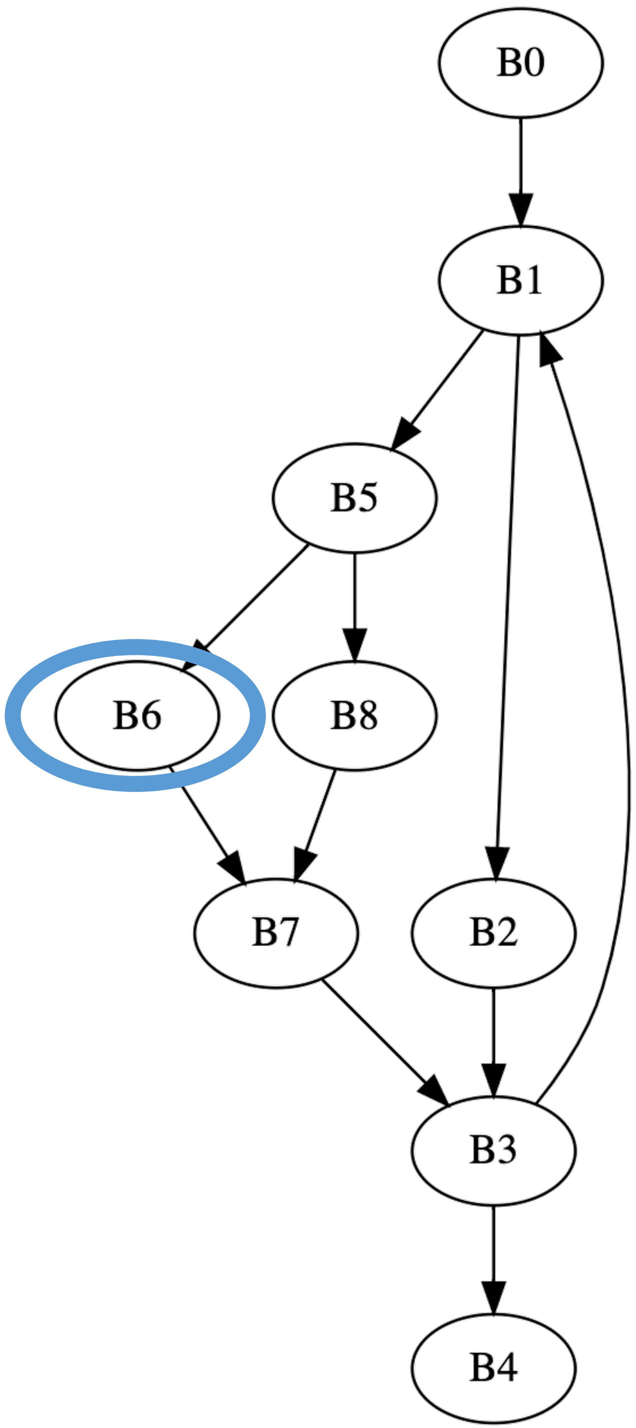


Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

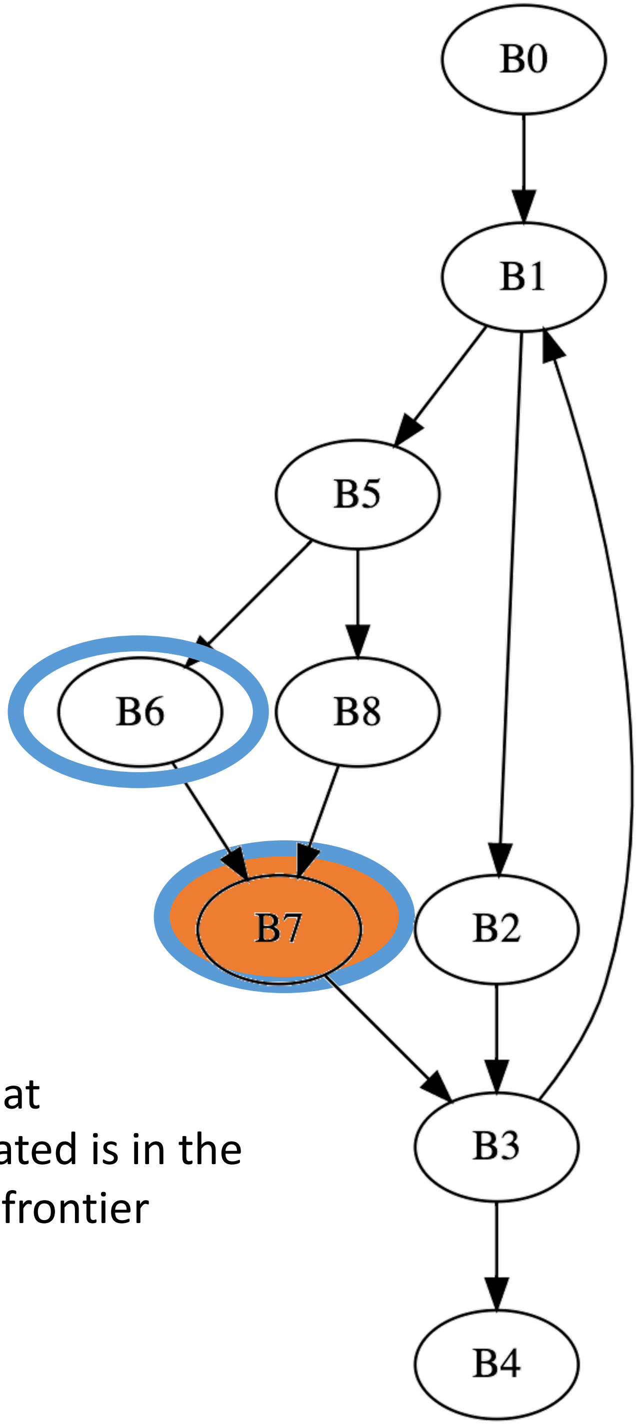


B3 is in the dominance frontier of B5

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,

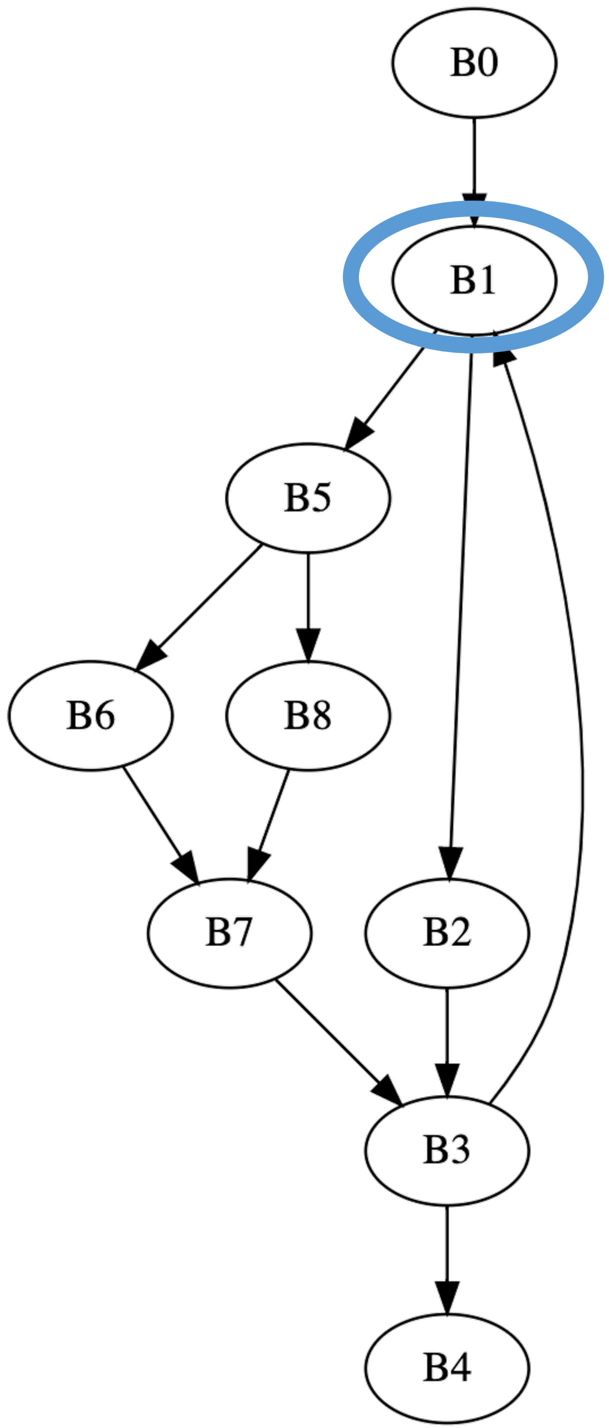


Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

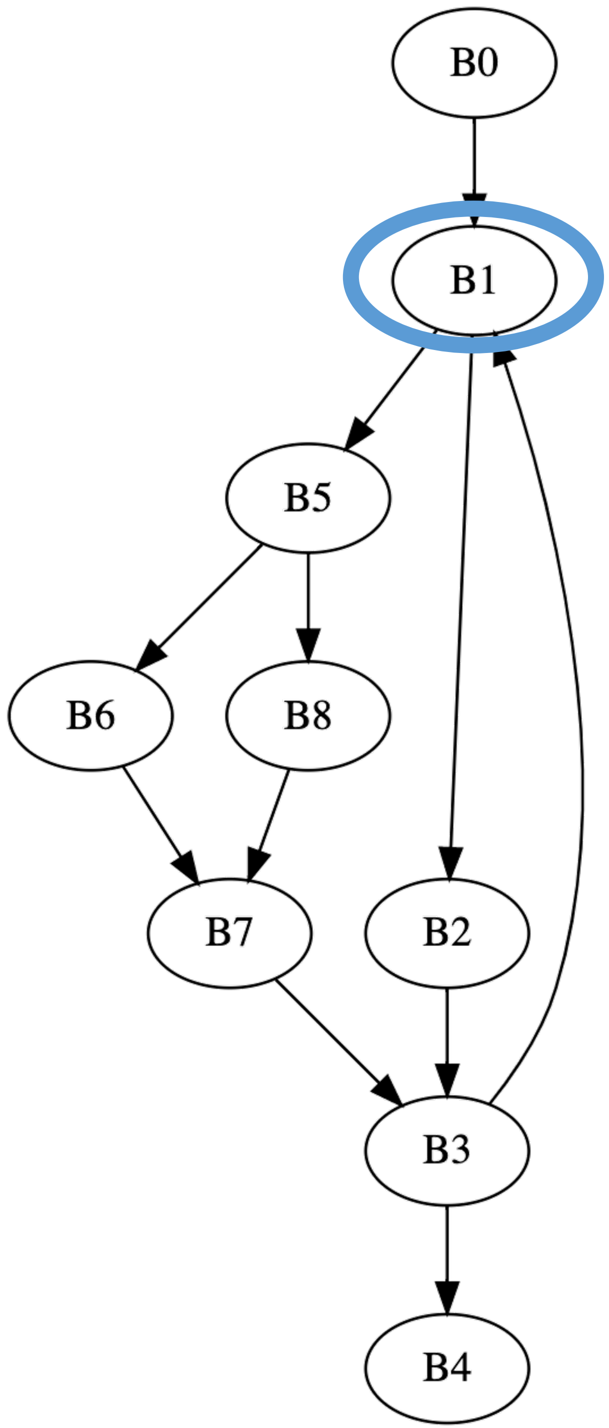


Any child that isn't dominated is in the dominance frontier

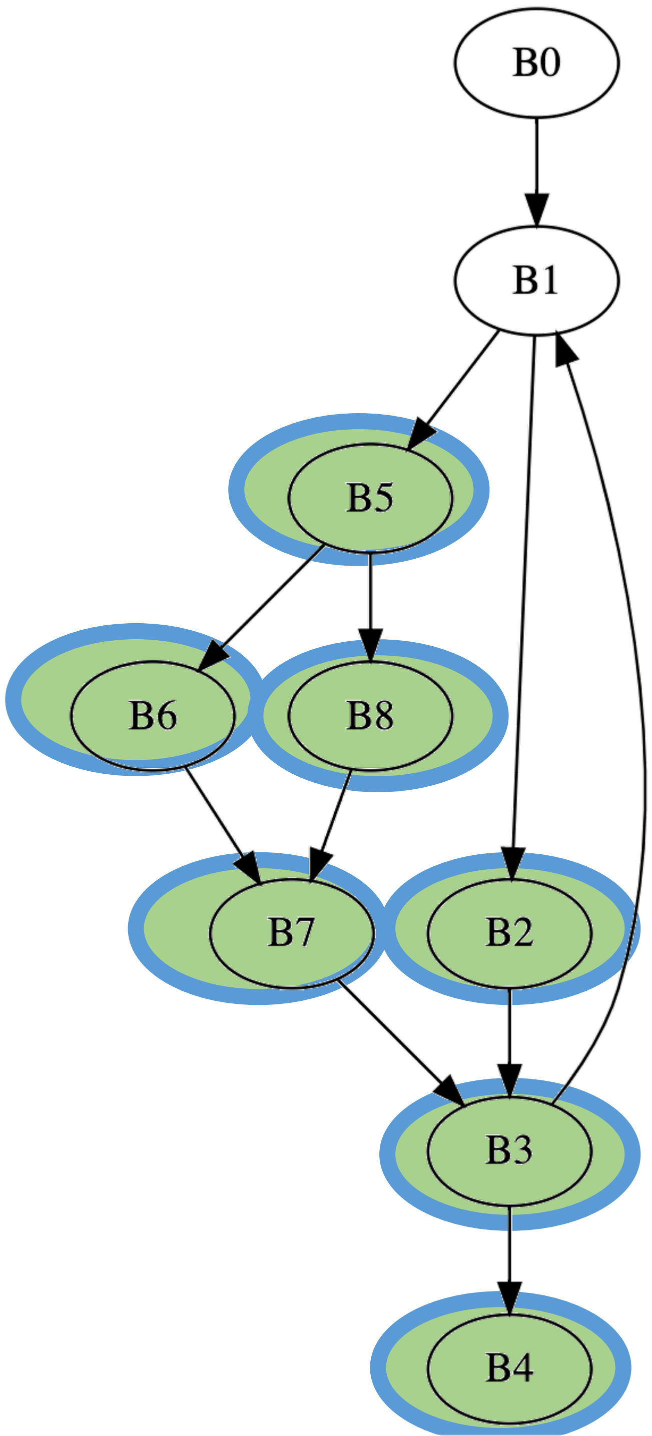
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



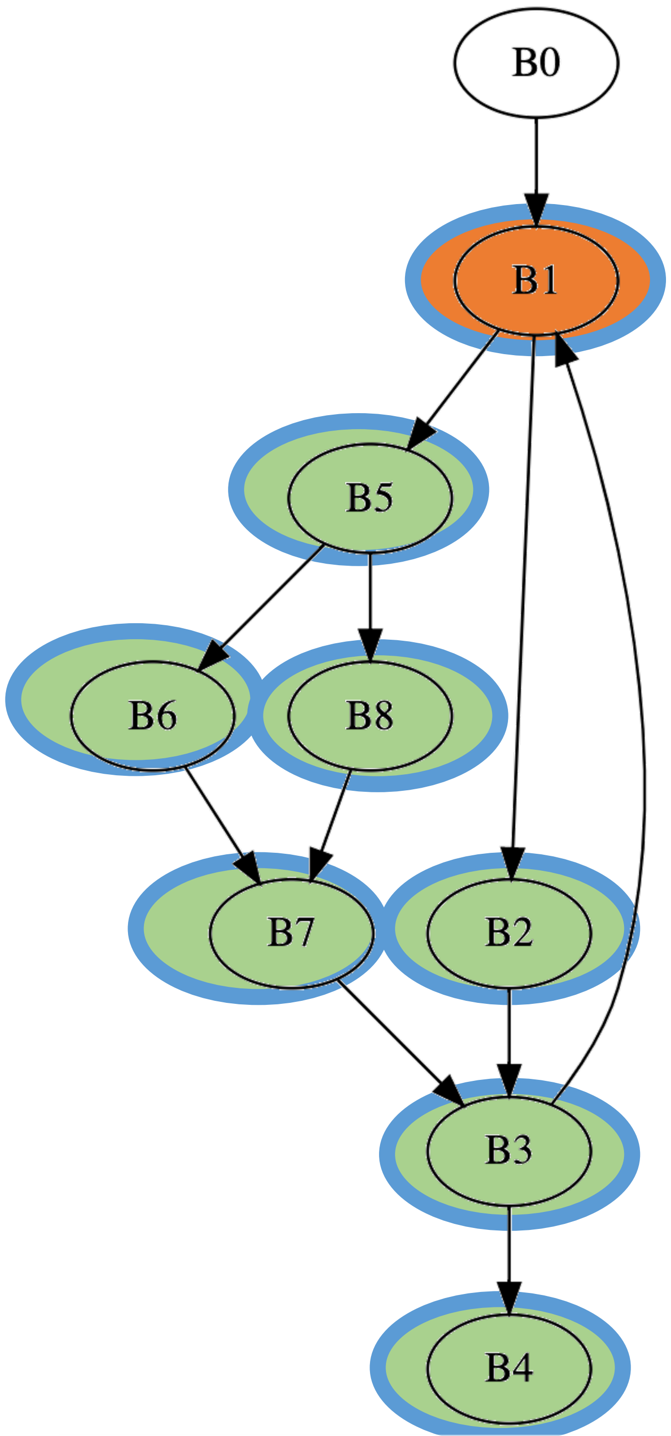
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



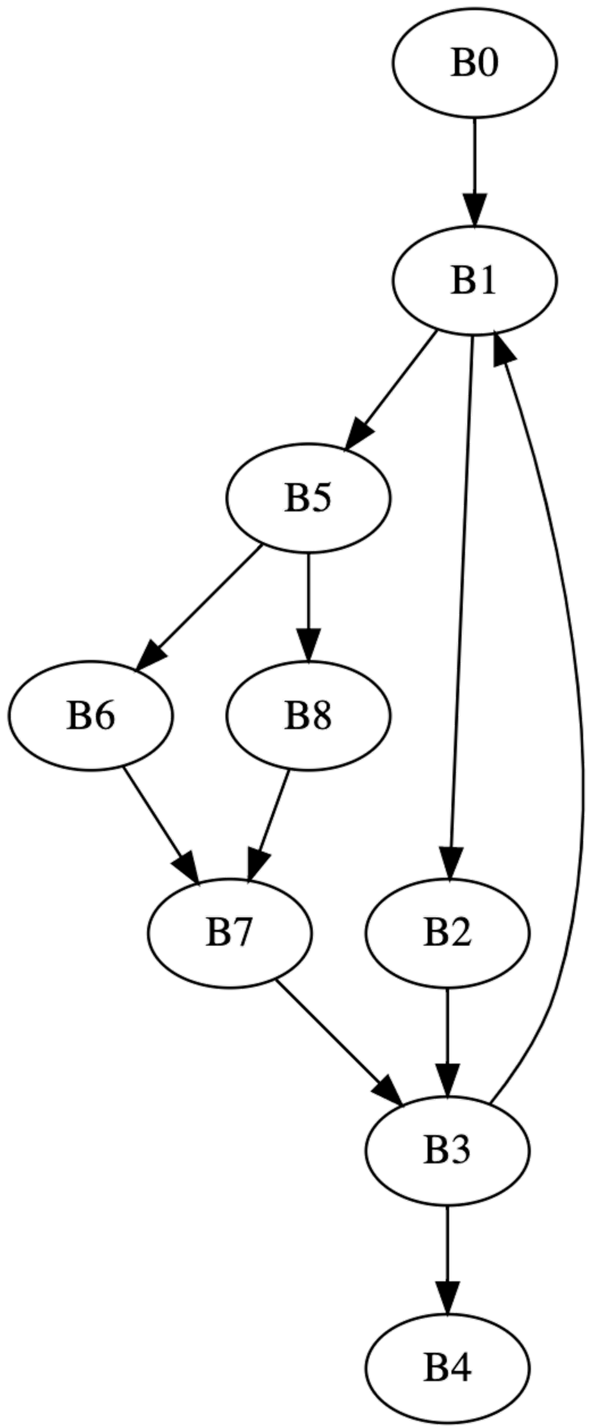
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



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

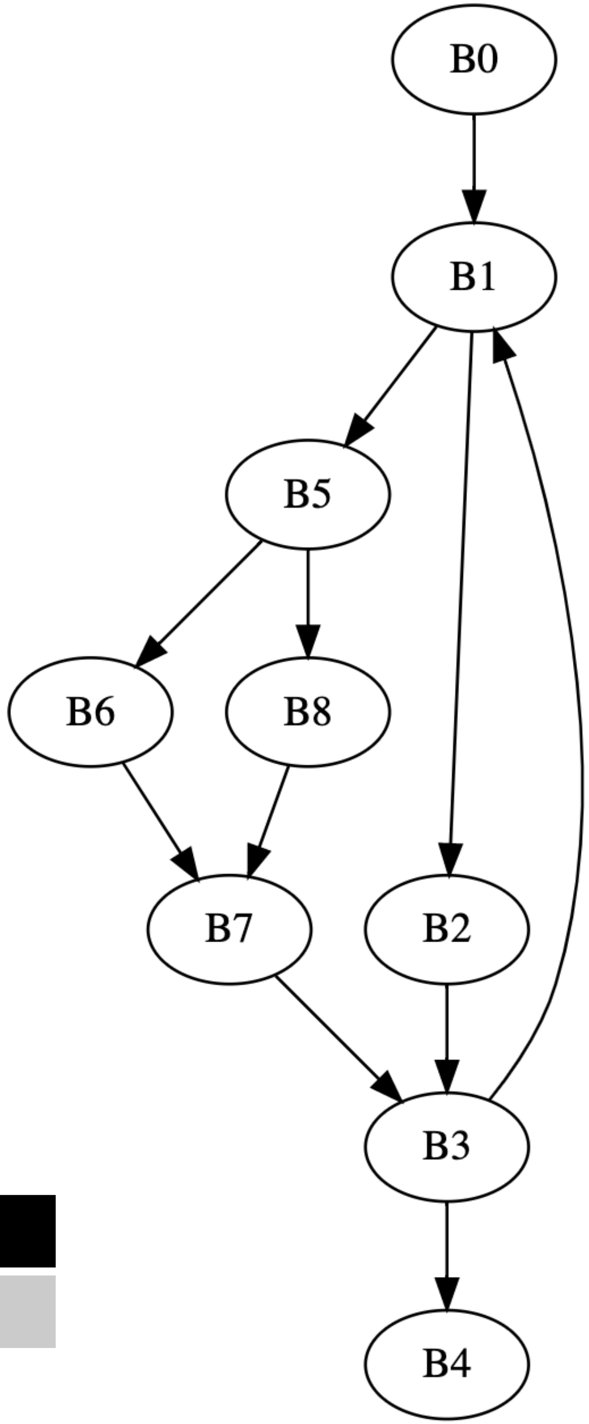


Dominance Frontier

- 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;
```

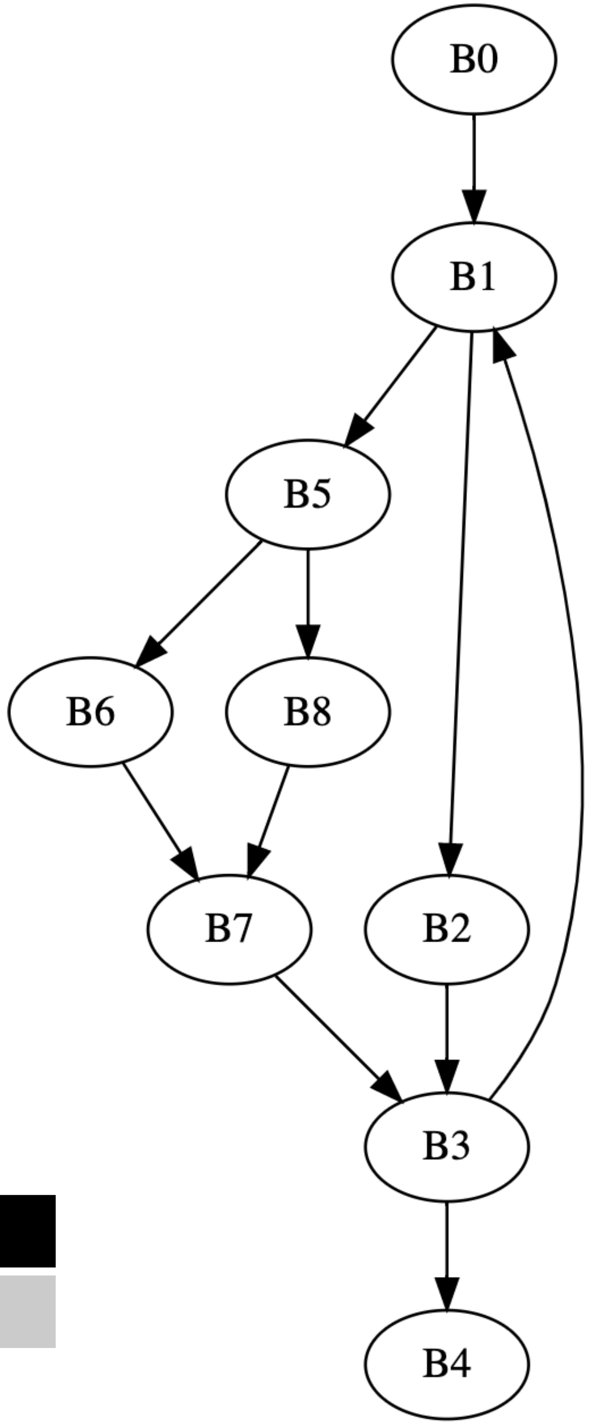
```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

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



Var	a	b	c	d	i	y	z
Blocks	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;
```

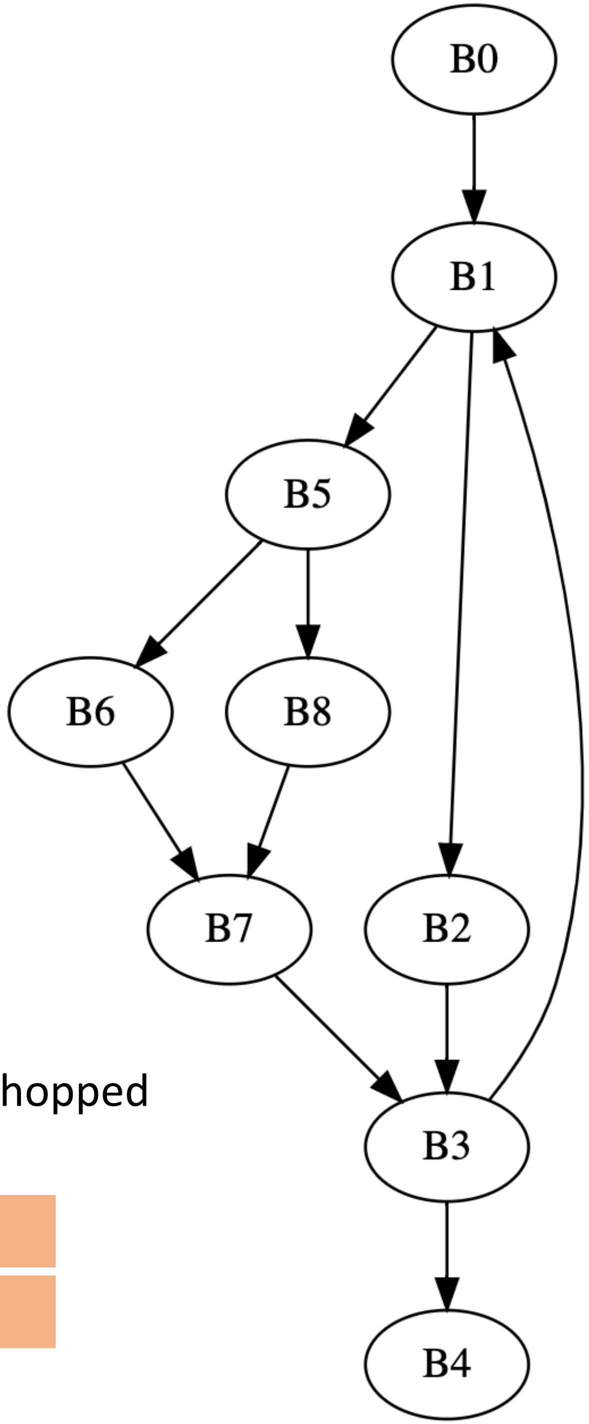
```
B5: a = ...;
    d = ...;
    br ... B6, B8;

B6: d = ...;

B7: b = ...;

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



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;
```

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,B3

```
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

Renaming

- Details are in the book:
  - iteratively do a reverse post-order traversal until all variables are named and every ϕ has arguments.

See you on Wednesday

- Optimizations for IRs in SSA form!

- Midterm assigned on Wednesday before midnight!