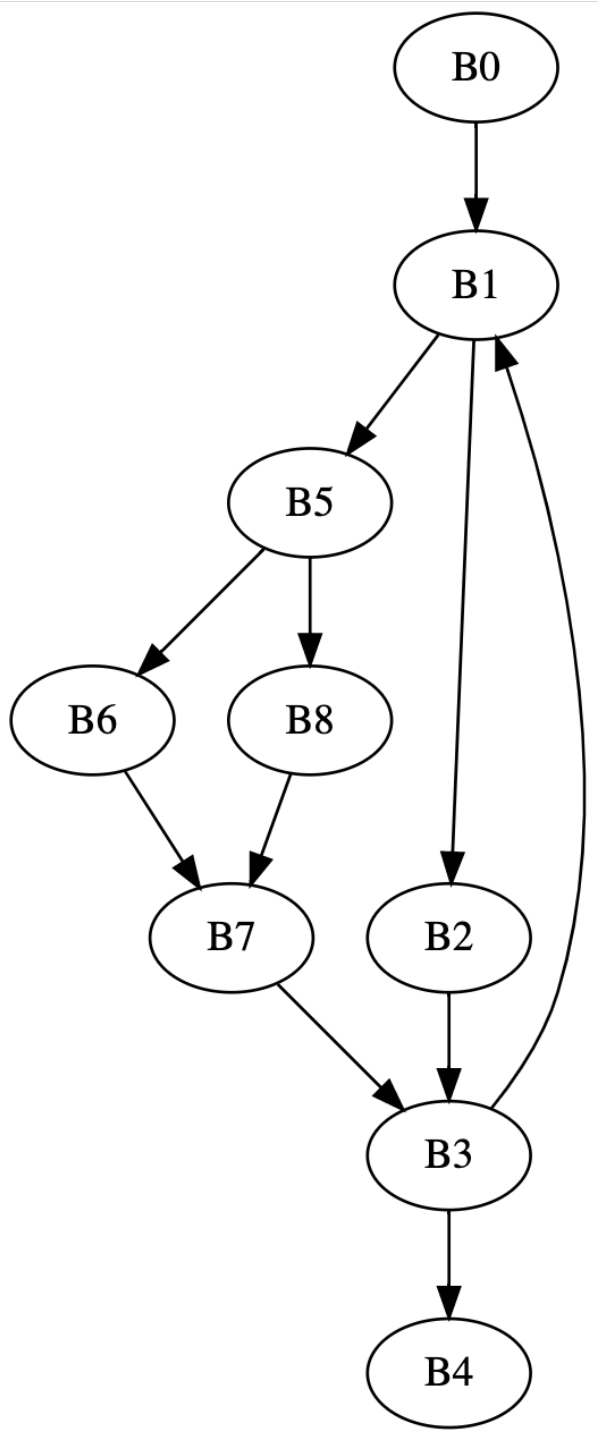
CSE211: Compiler Design

Nov. 2, 2020

- **Topic**: SSA form:
  - dominance frontiers
  - constant propagation



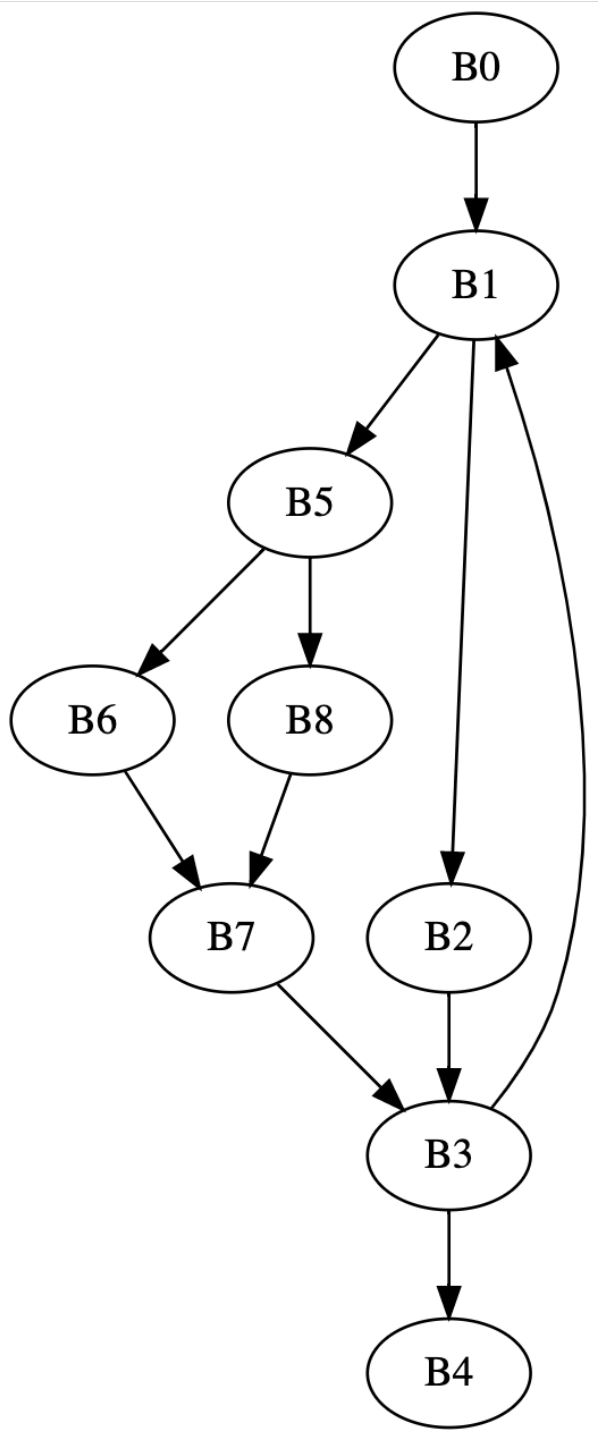
Announcements

- Homework 2 is due Nov. 12
- I will work on grading homework 1 in the next week or two
  - There should not be any big surprises if you past the testing scripts!
- API for pycfg can be a little confusing. Please discuss on Canvas
- Midterm will be released next Thursday, along with homework 3
- VOTE!!!!

CSE211: Compiler Design

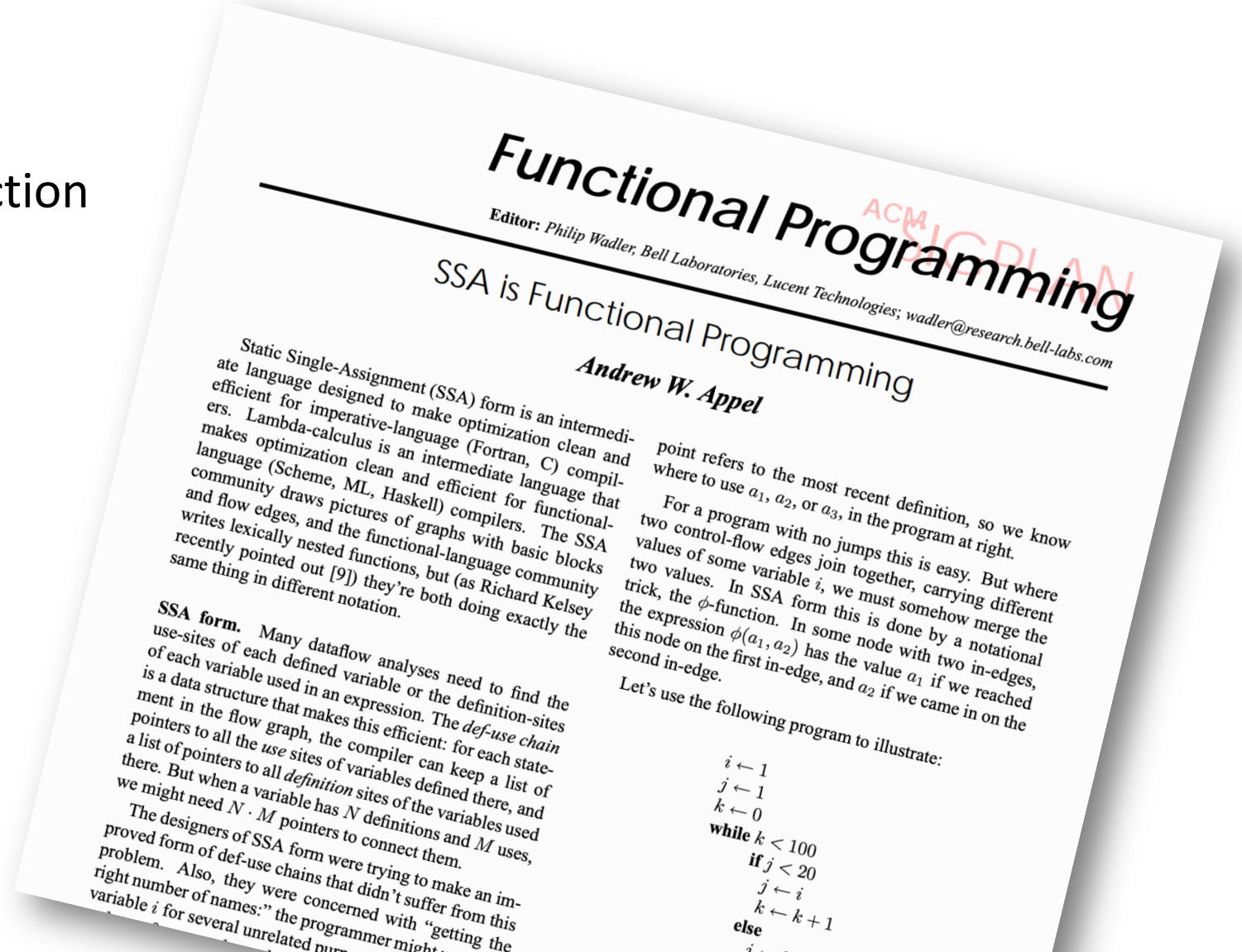
Nov. 2, 2020

- **Topic**: SSA form:
  - dominance frontiers
  - constant propagation



A note on SSA variants:

- Really Crude Approach:
  - Every block has a ϕ instruction for every variable



Functional Programming

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 [illegible] variable i for several unrelated [illegible]

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.

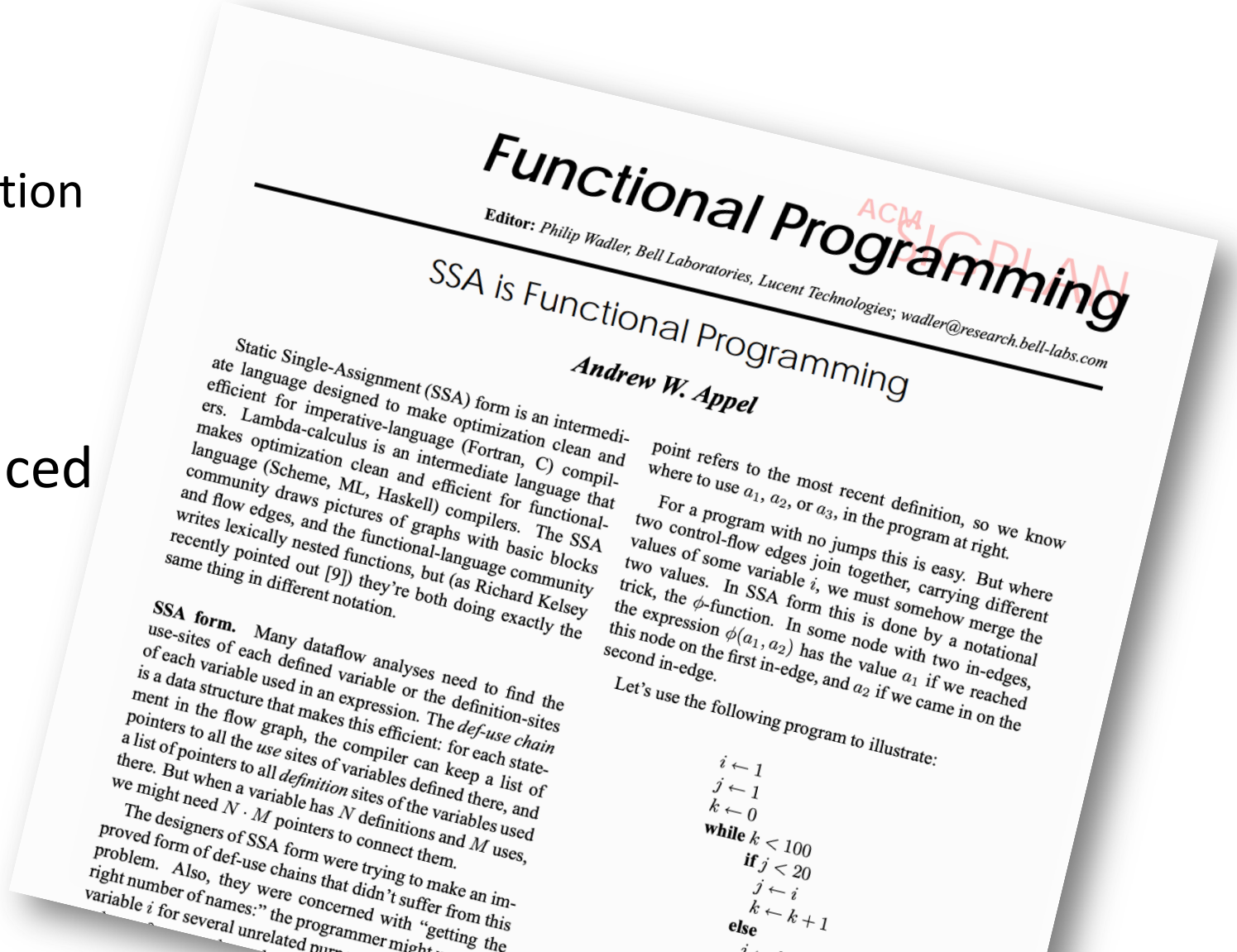
For a program with no jumps this is easy. But where two control-flow edges join together, carrying different values of some variable i, we must somehow merge the two values. In SSA form this is done by a notational trick, the ϕ-function. In some node with two in-edges, the expression $\phi(a_1, a_2)$ 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:
  - Every block has a ϕ instruction for every variable
- This approach was referenced in a later paper as "Maximal SSA"



Functional Programming

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

SSA is Functional Programming

Andrew W. Appel

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

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

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

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

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

Let's use the following program to illustrate:

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

Maximal SSA

Example

```
x = 1;
y = 2;

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

else {
  x = 6;
  y = 100;
}

print(x)
```

Insert ϕ with argument placeholders

```
x = 1;
y = 2;

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

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

x = ϕ(...);
y = ϕ(...);
print(x)
```

Rename variables iterate through basic blocks with a global counter

```
x0 = 1;
y1 = 2;

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

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

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

fill in ϕ arguments by considering CFG

```
x0 = 1;
y1 = 2;

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

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

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

A note on SSA variants:

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

```
x0 = 1;
y1 = 2;

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

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

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

A note on SSA variants:

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

```
x0 = 1;
y1 = 2;

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

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

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

A note on SSA variants:

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

```
x0 = 1;
y1 = 2;

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

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

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

A note on SSA variants:

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

```
x0 = 1;
y1 = 2;

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

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

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

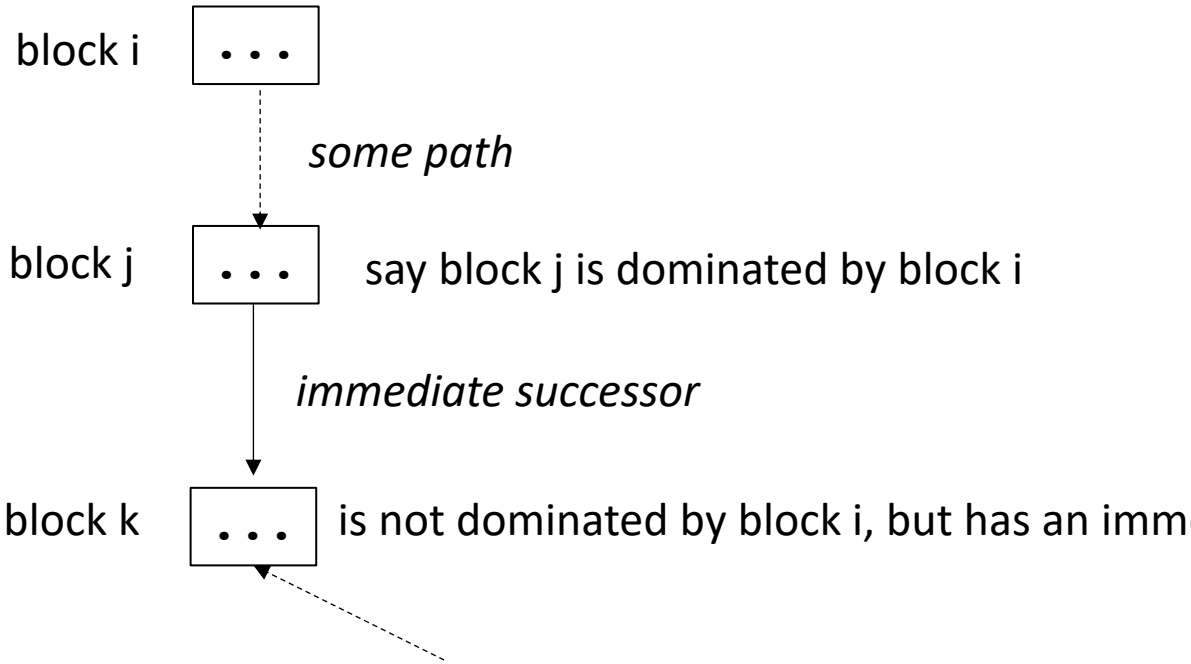
A note on SSA variants:

- EAC book describes:
  - Minimal SSA
  - Pruned SSA
  - **Semipruned SSA**

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

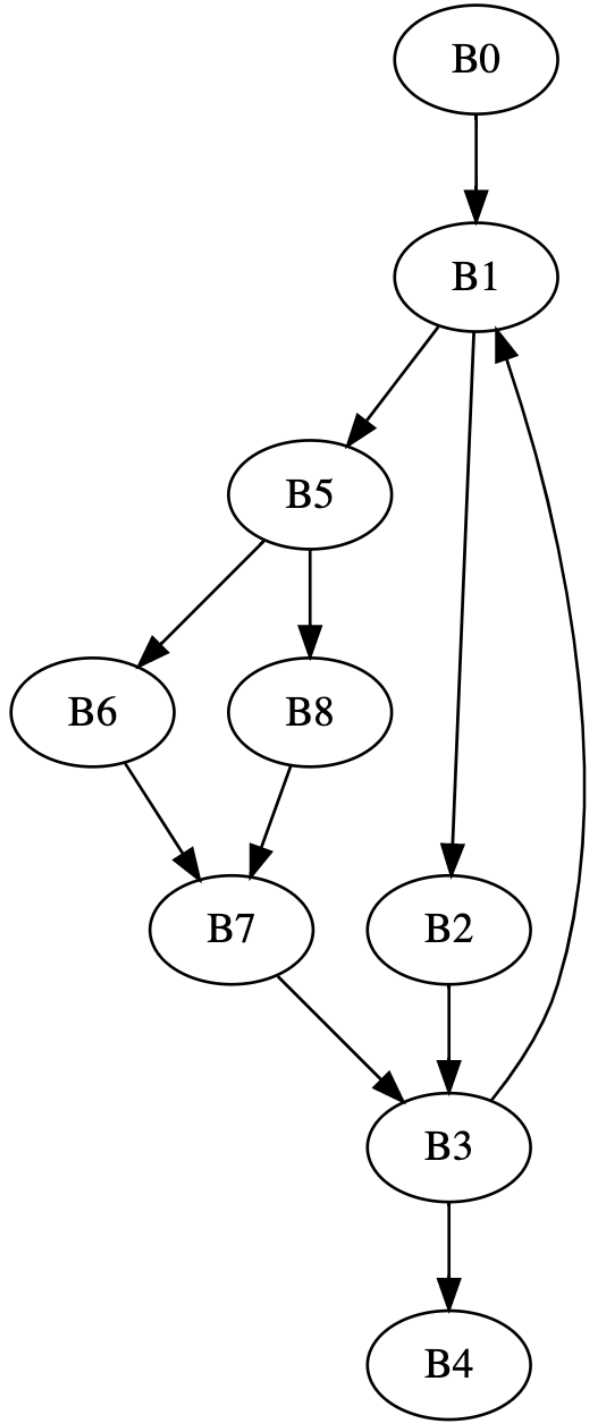
Dominance Frontier

- a viz using coloring (thanks Chris!)

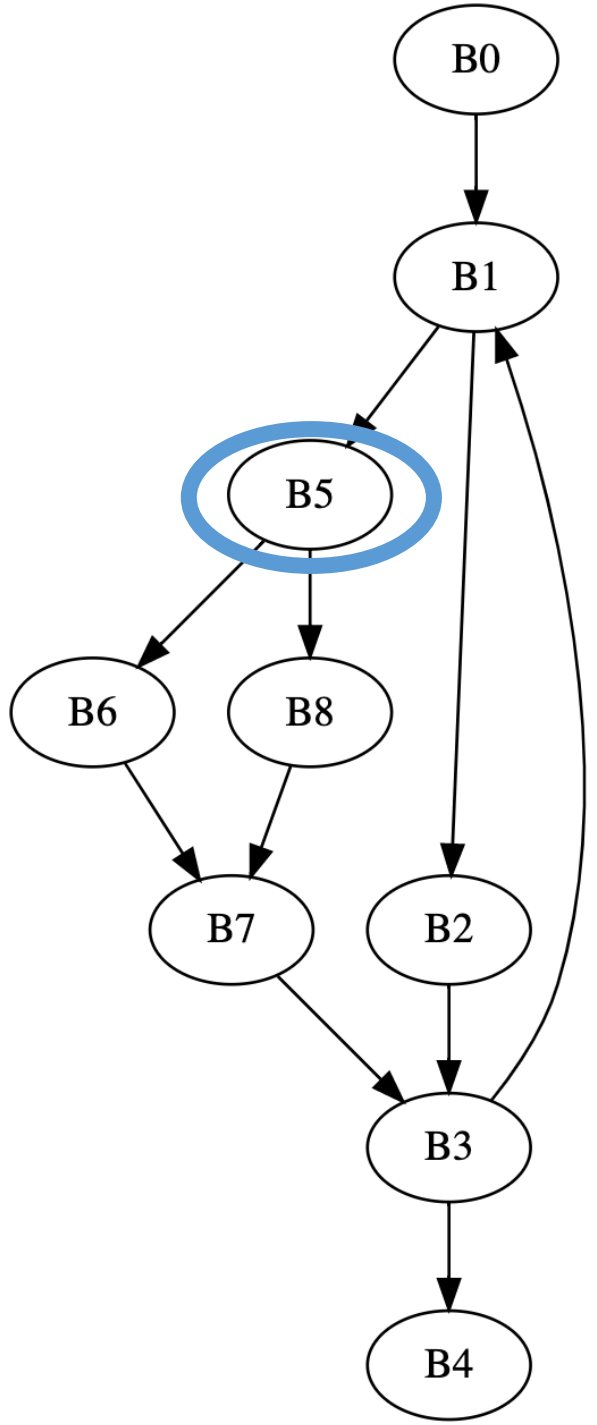
- Efficient algorithm for computing in EAC section 9.3.2 using a dominator tree. Please read when you get the chance!

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

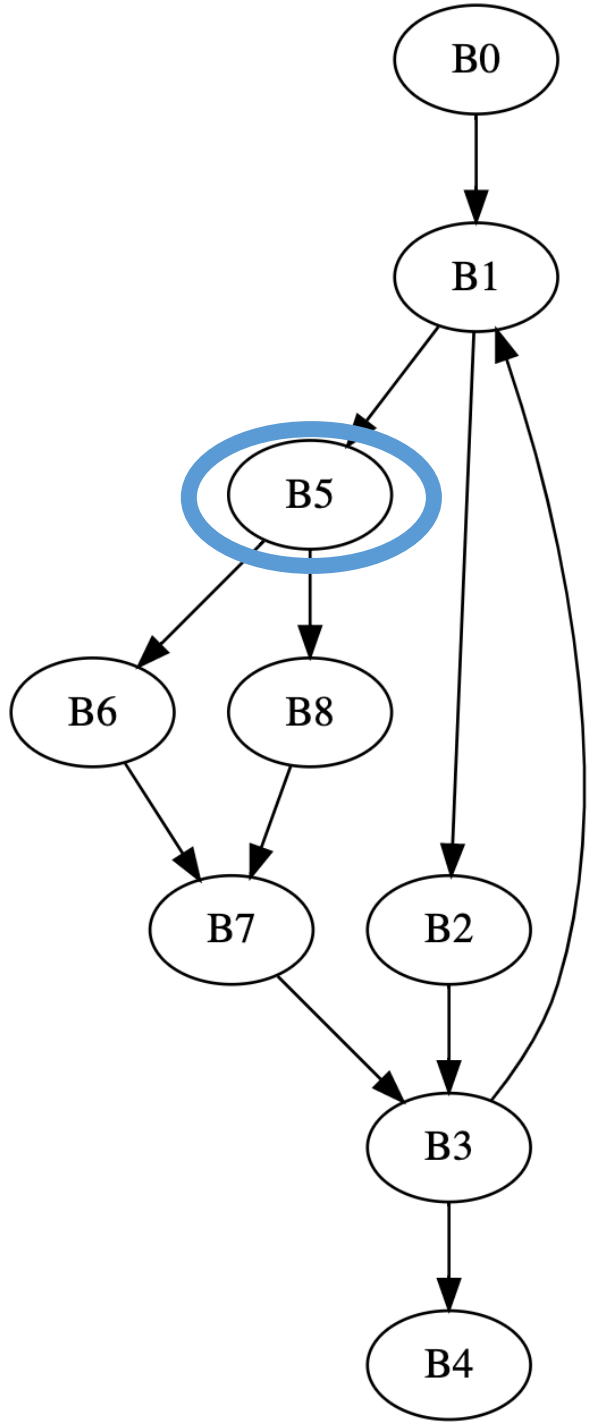
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



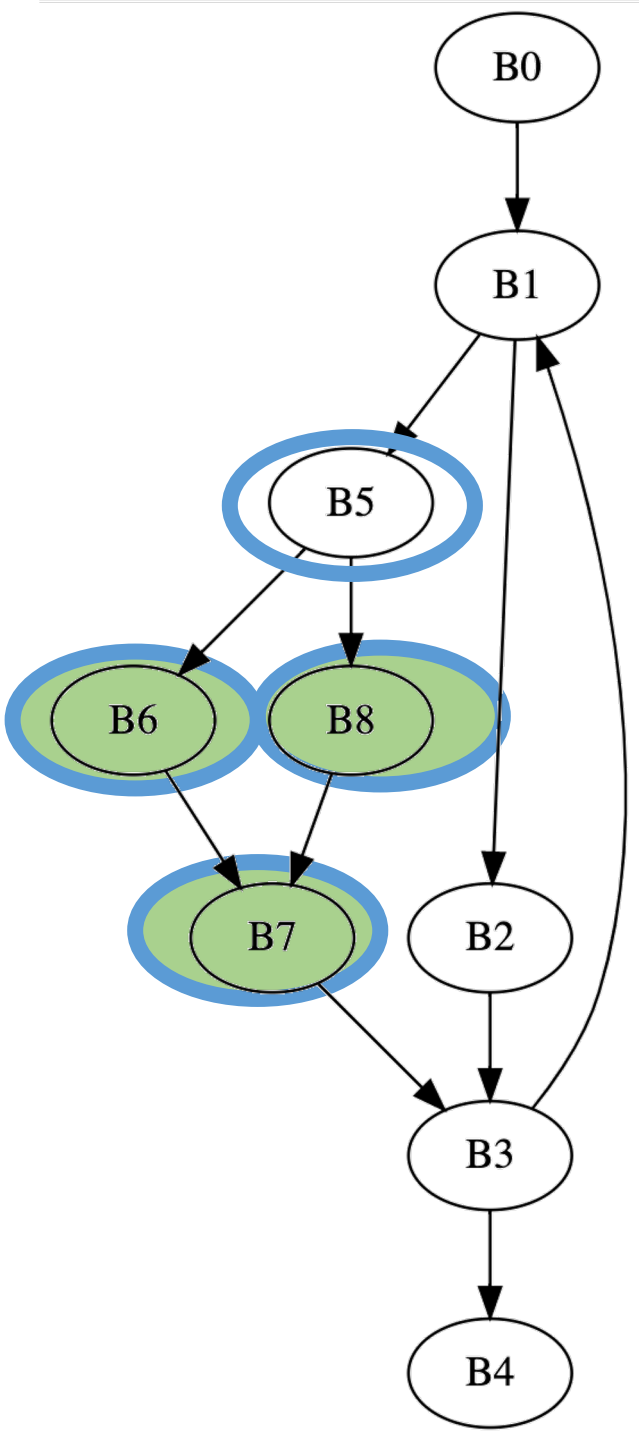
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



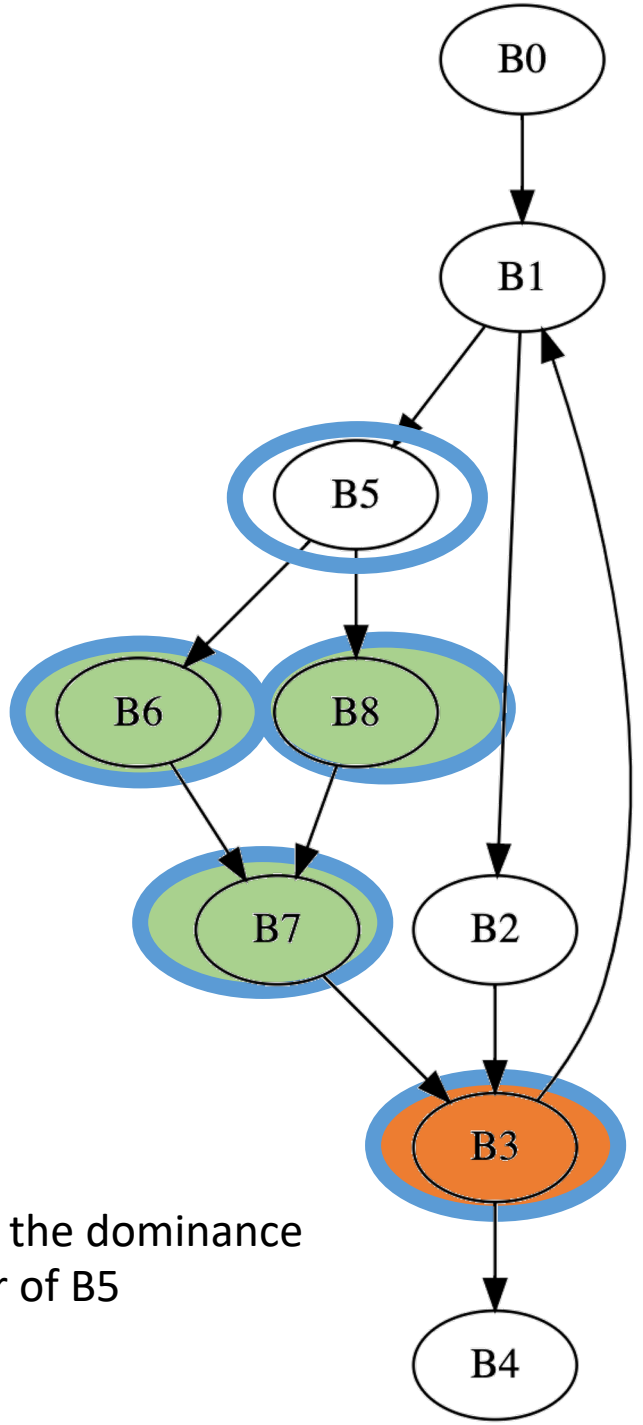
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



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,

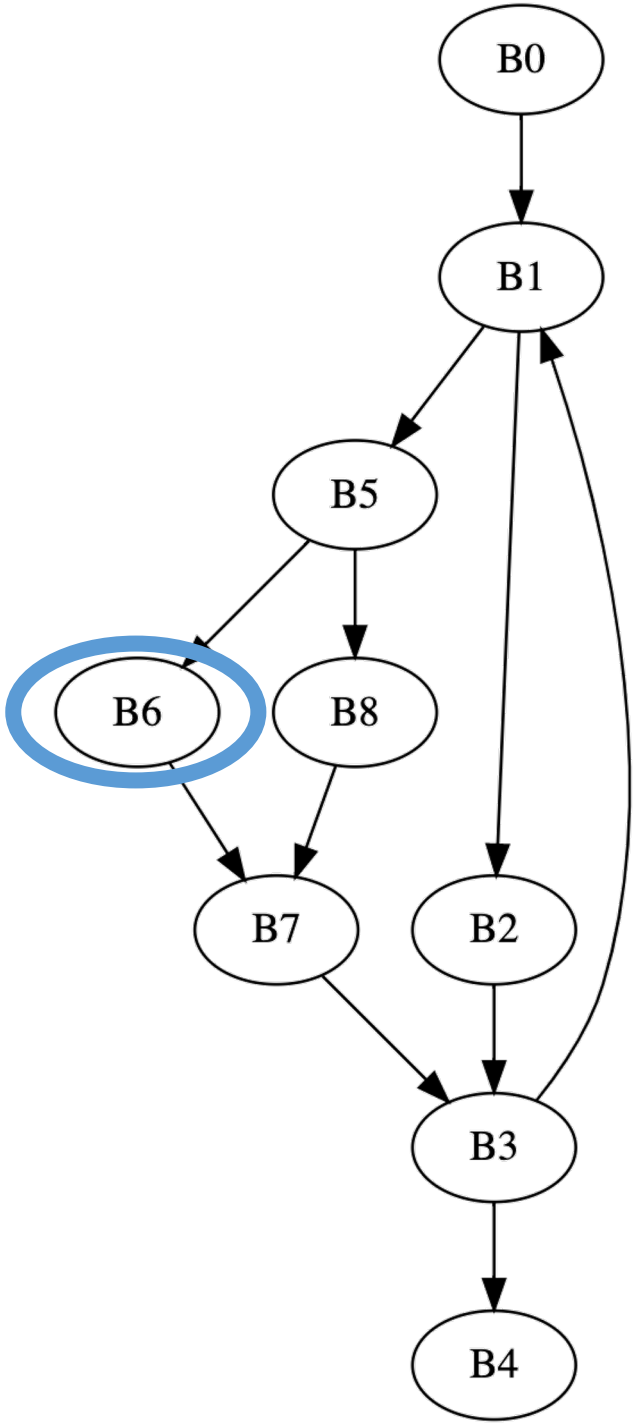


Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

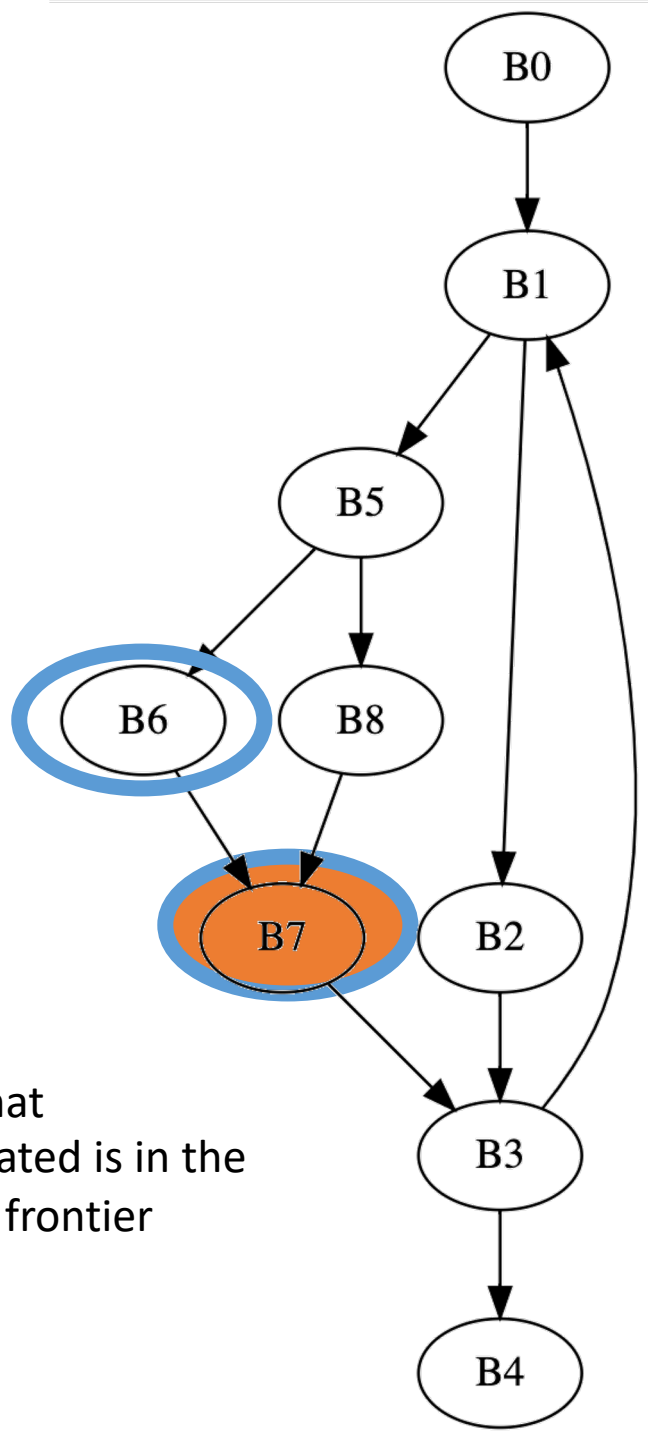


B3 is in the dominance frontier of B5

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,

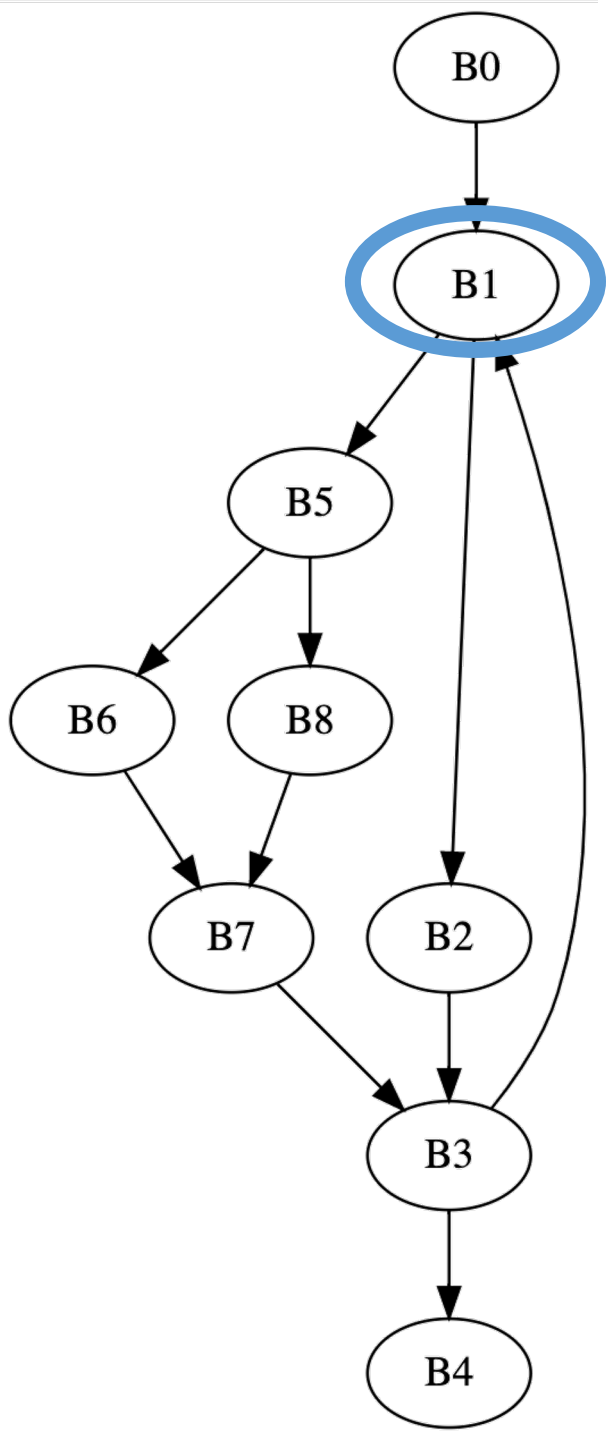


Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,

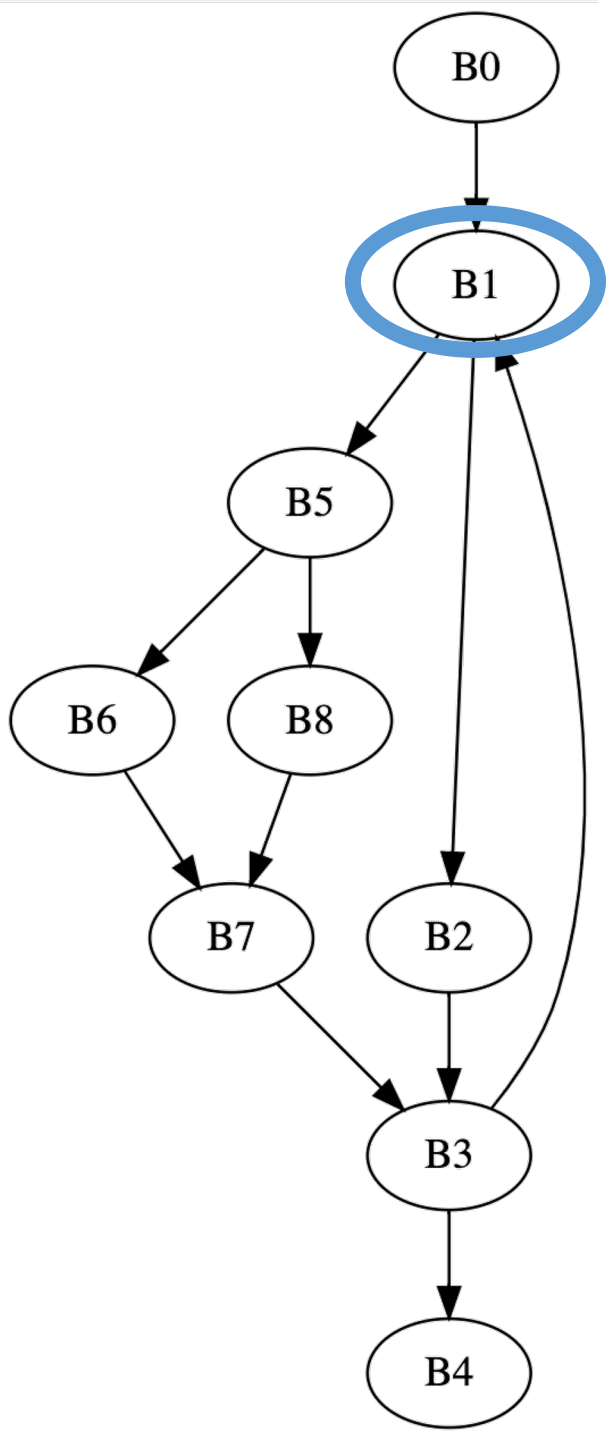


Any child that isn't dominated is in the dominance frontier

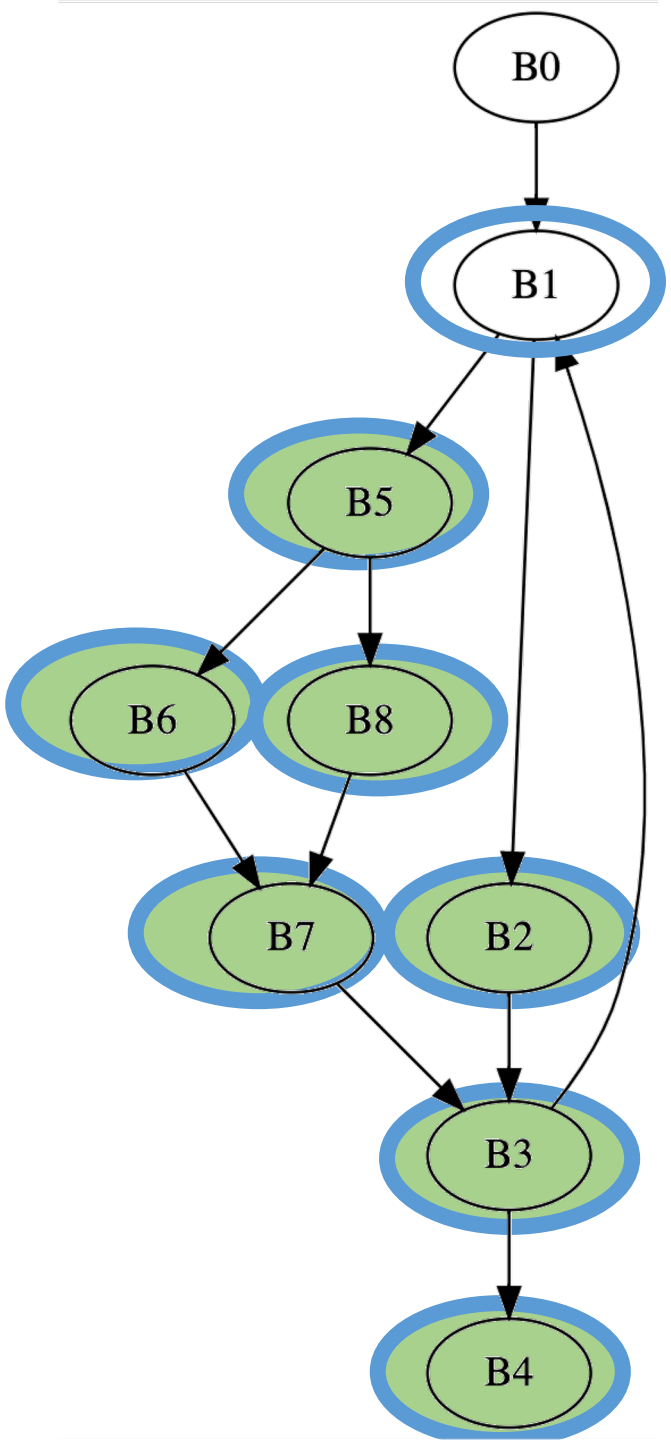
Node	Dominators
B0	
B1	B0,
B2	B0, B1,
B3	B0, B1,
B4	B0, B1, B3,
B5	B0, B1,
B6	B0, B1, B5,
B7	B0, B1, B5,
B8	B0, B1, B5,



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,

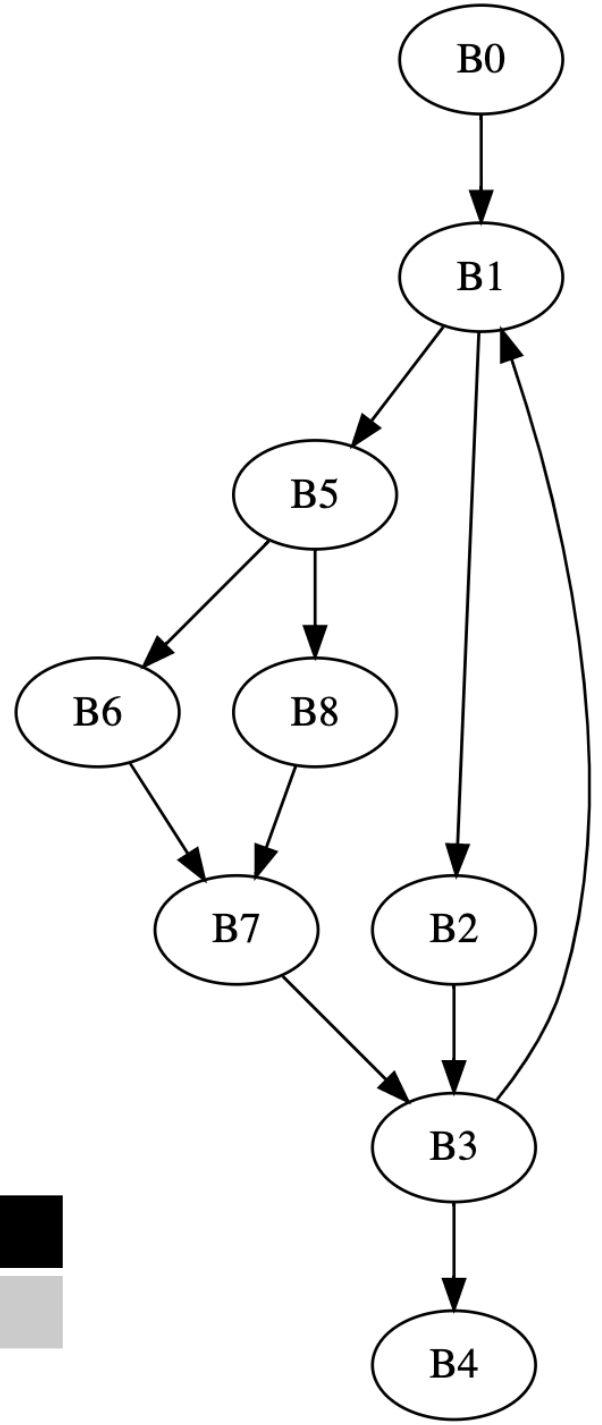


Dominance Frontier

- 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;
```

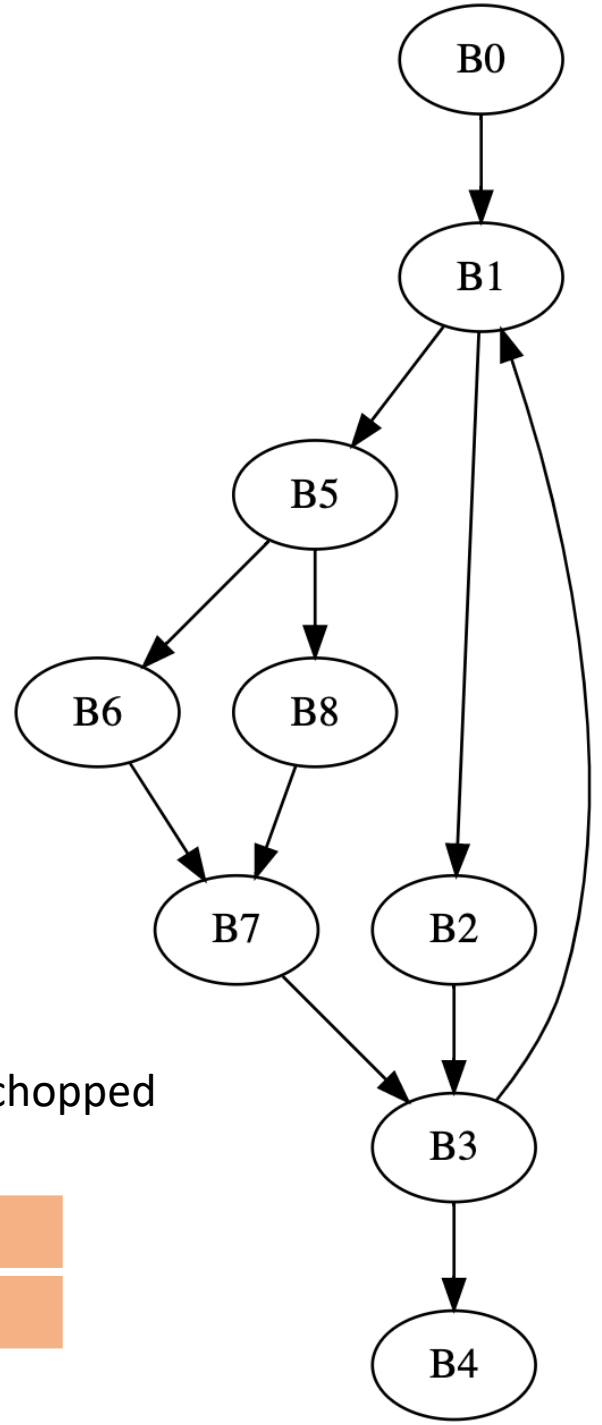
```
B5: a = ...;
    d = ...;
    br ... B6, B8;

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;
```



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;
```

```
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;
```

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;
```

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;
```

```
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,B1

```
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;
```

```
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.B1

Renaming

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

An optimization 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> {
   y0 = 5;
}
else {
   y1 = z;
}
y2 = phi(y0,y1);
w = y2;
```

A simple lattice

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

A simple lattice

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

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

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

$c_i = c_j$
if i == j

$c_i \wedge c_j = \bot$
if i != j 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 x based on its expression:

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

This can be done in a single pass

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 track the “uses” of *n,* e.g.
m = n * x;

evaluate m over the lattice:

Constant propagation algorithm

evaluate m over the lattice:

m = n*x

if (m in [⊥, c_x])

  break;

if (Value(n) not in [⊥, T] and Value(x) not in [⊥, T])

  c_m evaluate n*x using their constants

  Value(m) = c_m

  Add m to the worklist

Constant propagation algorithm

evaluate m over the lattice:

$m = \phi(x_1 , x_2)$

$c_m = x_1 \wedge x_2$

if c_m is not T then add m to the worklist

Next week

- Start on automatic parallelization (finally!)