Compiling Techniques

Lecture 14: Building SSA Form

Reminder: Static Single-Assignment (SSA) Form

*"A program is defined to be in **Static Single-Assignment (SSA)** form if each variable is a target of exactly one assignment statement in the program text."*

- Each assignment statement defines a unique name.
- Each use refers to a single name.

Representing Control Flow

```
x = 0

if (a == 42)

    x = x + 1

else

    x = 3


y = x + 5
```

Representing Control Flow

```
x = 0
if (a == 42)
    x = x + 1
else
    x = 3


y = x + 5
```

$\Longrightarrow$

```
x₁ = 0
if (a == 42)
    x₂ = x₁ + 1
else
    x₃ = 3


y = x₍?₎ + 5
```

$x_1 = 0$

if (a == 42)

$x_2 = x_1 + 1$

else

$x_3 = 3$

$y = x_? + 5$

Representing Control Flow

```
x = 0

if (a == 42)

    x = x + 1

else

    x = 3


y = x + 5
```

⟹

$x_1 = 0$

if (a == 42)

    $x_2 = x_1 + 1$

else

    $x_3 = 3$

$y = x_? + 5$

Control Flow Graph (CFG)

$x_1 = 0$

$x_2 = x_1 + 1$ $x_3 = 3$

$y = x_? + 5$

basic block

Representing Control Flow

```
x = 0

if (a == 42)

    x = x + 1

else

    x = 3



y = x + 5
```

$\Longrightarrow$

$x_1 = 0$
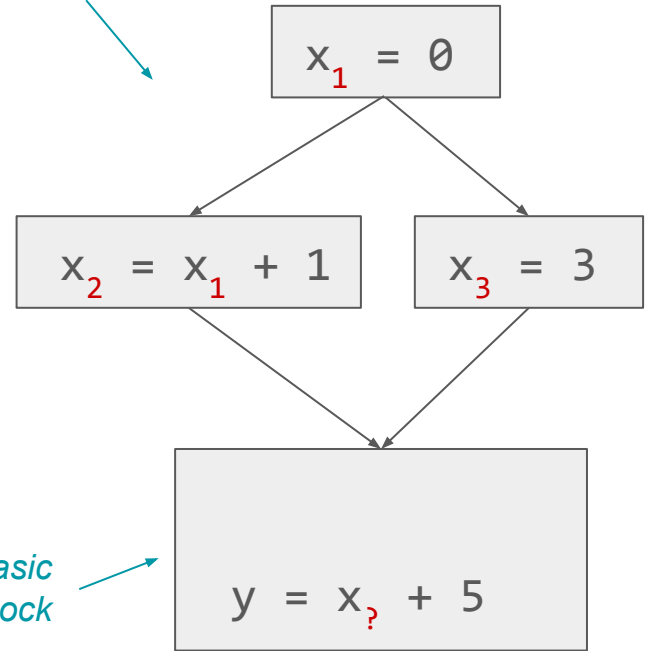
if (a == 42)

$\quad x_2 = x_1 + 1$

else

$\quad x_3 = 3$
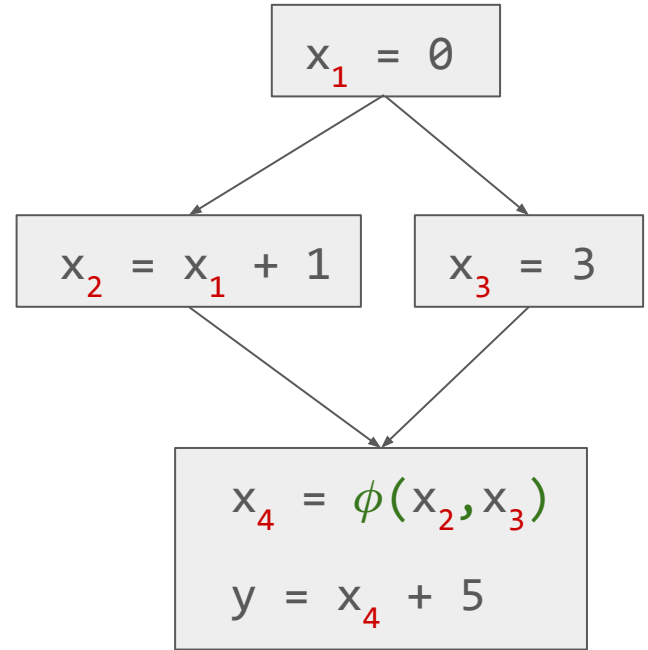
$x_4 = \phi(x_2, x_3)$

$y = x_4 + 5$

$x_1 = 0$

$x_2 = x_1 + 1$ | $x_3 = 3$

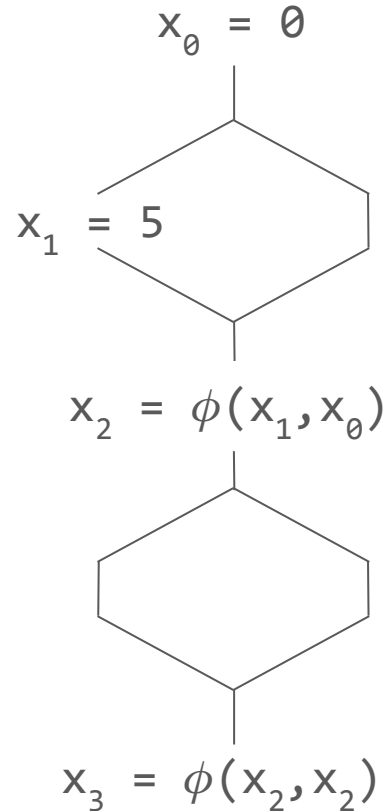$x_4 = \phi(x_2, x_3)$

$y = x_4 + 5$

6

ϕ-function placement

Naive approach:

1. At each join point insert a ϕ-function for every variable name

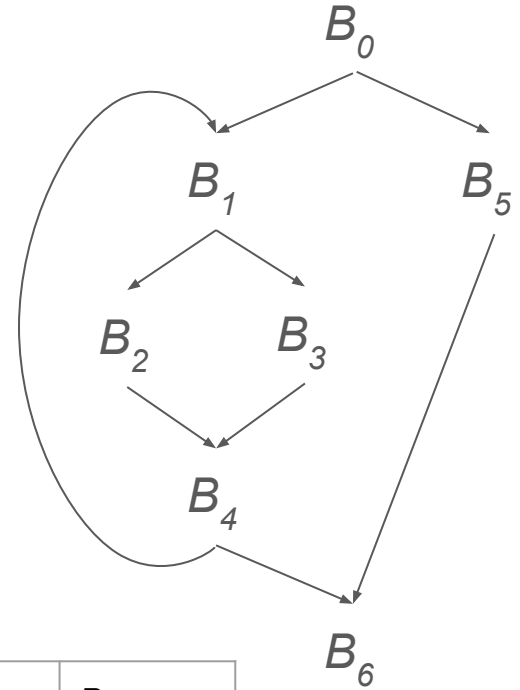2. Rename adding unique subscripts

Computes *maximal SSA form*

$$x_0 = 0$$

$$x_1 = 5$$

$$x_2 = \phi(x_1, x_0)$$

$$x_3 = \phi(x_2, x_2)$$

redundant ϕ-function

Dominators

p dominates q (*p >> q, p dom q*) iff
every path from the entry node b_0 to q
also visits *p*.

Dom(q) – set of nodes that dominate *q*.

B_0

B_1 B_5

B_2 B_3

B_4

B_6

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DOM(B)	B_0						

8

Dominators

p dominates q (*p >> q, p dom q*) iff
every path from the entry node b_0 to q
also visits *p*.

Dom(q) – set of nodes that dominate *q*.

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DOM(B)	B_0	B_0, B_1					

Dominators

p dominates q (*p >> q, p dom q*) iff
  every path from the entry node b_0 to *q*
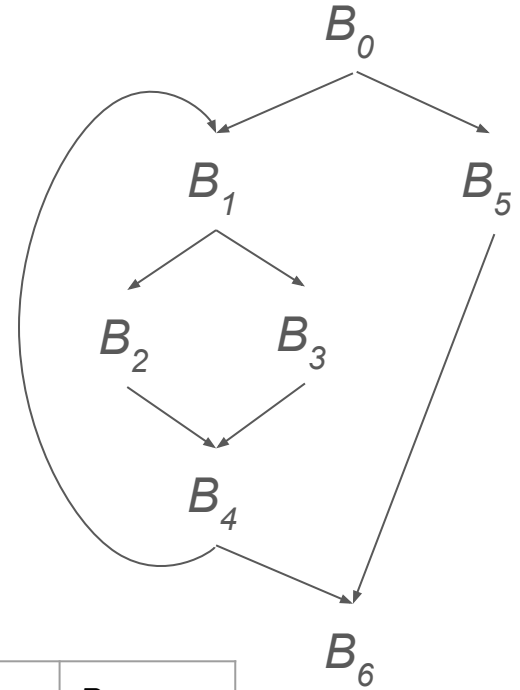 also visits *p*.

Dom(q) – set of nodes that dominate *q*.

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DOM(B)	B_0	B_0, B_1	B_0, B_1, B_2	B_0, B_1, B_3			

Dominators

p dominates q (*p >> q, p dom q*) iff
    every path from the entry node b_0 to q
    also visits *p*.

Dom(q) – set of nodes that dominate q.
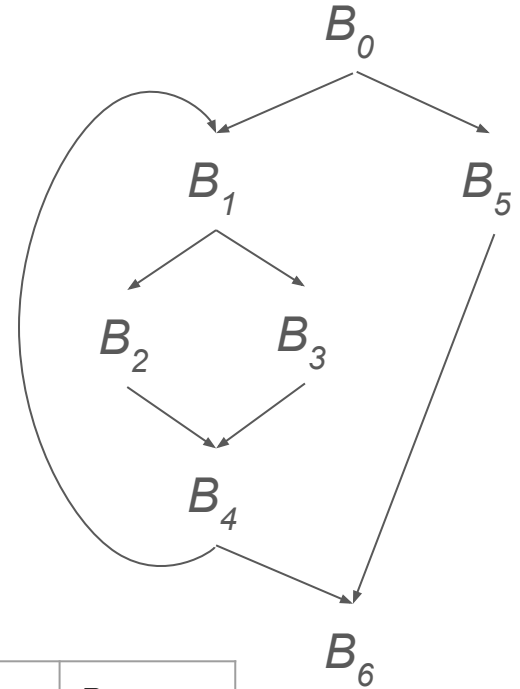

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DOM(B)	B_0	B_0, B_1	B_0, B_1, B_2	B_0, B_1, B_3	B_0, B_1, B_4		

Dominators

p dominates *q* (*p >> q, p dom q*) iff
every path from the entry node b_0 to *q*
also visits *p*.

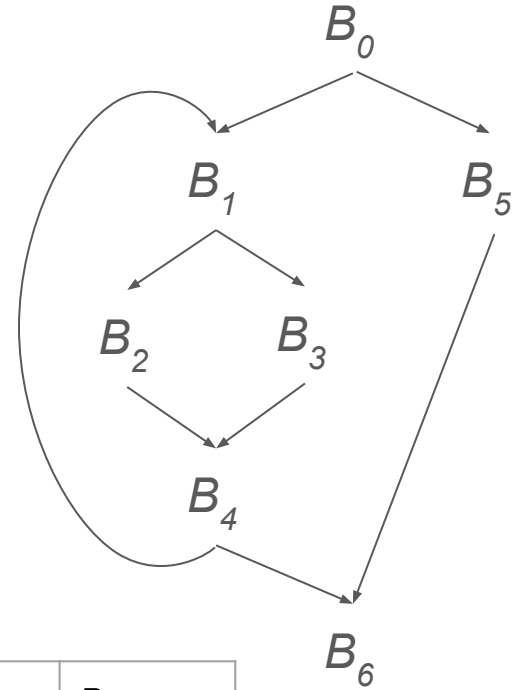*Dom(q)* – set of nodes that dominate *q*.

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DOM(B)	B_0	B_0, B_1	B_0, B_1, B_2	B_0, B_1, B_3	B_0, B_1, B_4	B_0, B_5	

Dominators

p dominates q ($p \gg q$, *p dom* q) iff
every path from the entry node b_0 to q
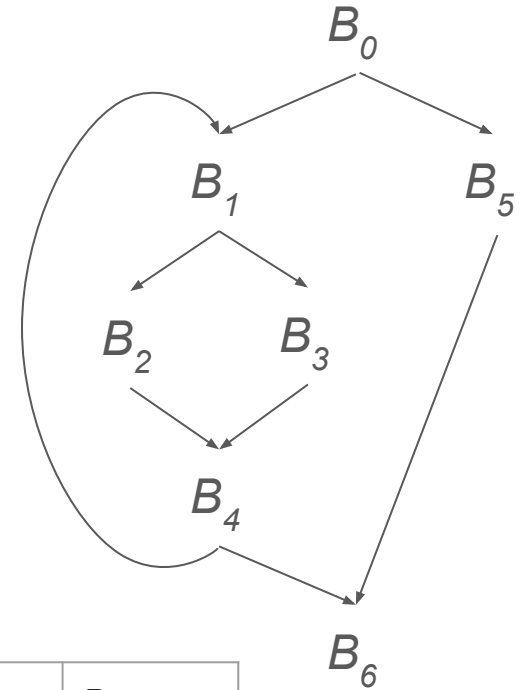also visits *p*.
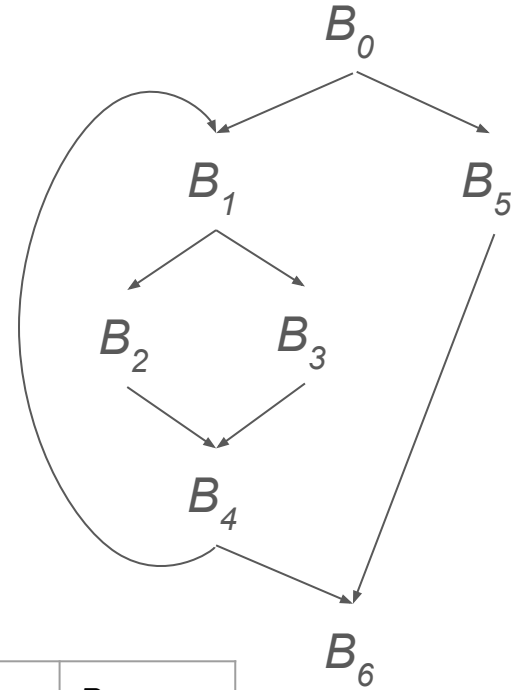
Dom(q) – set of nodes that dominate q.

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DOM(B)	B_0	B_0, B_1	B_0, B_1, B_2	B_0, B_1, B_3	B_0, B_1, B_4	B_0, B_5	B_0, B_6

dom Relation

- reflexive

  a dom a

- antisymmetric

  a dom b $\land$ b dom a $\Rightarrow$ a = b

- transitive

  a dom b $\land$ b dom c $\Rightarrow$ a dom c

i.e. it's a *partial order*

Dominator Tree

IDom

B_0

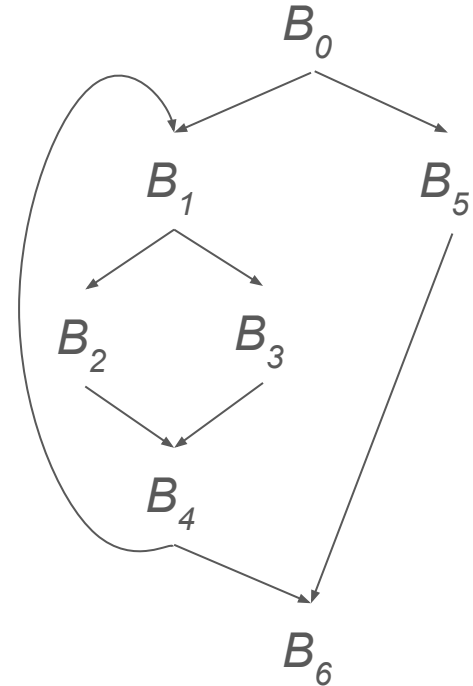B_1 B_5 B_6

B_2 B_3 B_4

Dominance frontier

p *strictly dominates* q iff
    p dominates q and p ≠ q.

q is in *dominance frontier* of p iff
- p domitates a predecessor of q.
- p does not strictly dominate q.

DF(p) – dominance frontier of p.

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DF(B)							

Dominance frontier

p strictly dominates q iff
> *p* dominates *q* and $p \neq q$.

q is in *dominance frontier* of *p* iff
- *p* domitates a predecessor of *q*.
- *p* does not strictly dominate *q*.

DF(p) – dominance frontier of *p*.

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DF(B)	$\varnothing$						

Dominance frontier

p strictly dominates q iff
  p dominates *q* and *p* ≠ *q*.

q is in *dominance frontier* of *p* iff
- *p* domitates a predecessor of *q*.
- *p* does not strictly dominate *q*.

DF(p) – dominance frontier of *p*.

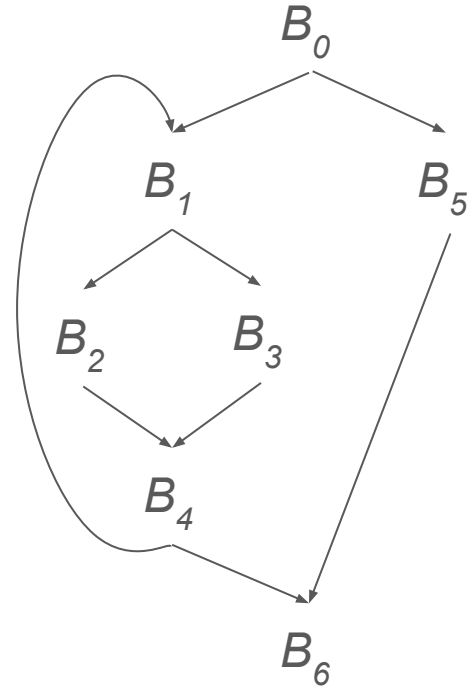| B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DF(B)	∅	B_1, B_6					

Dominance frontier

p *strictly dominates* q iff
 p dominates q and $p \neq q$.

q is in *dominance frontier* of p iff
- p domitates a predecessor of q.
- p does not strictly dominate q.

DF(p) – dominance frontier of p.

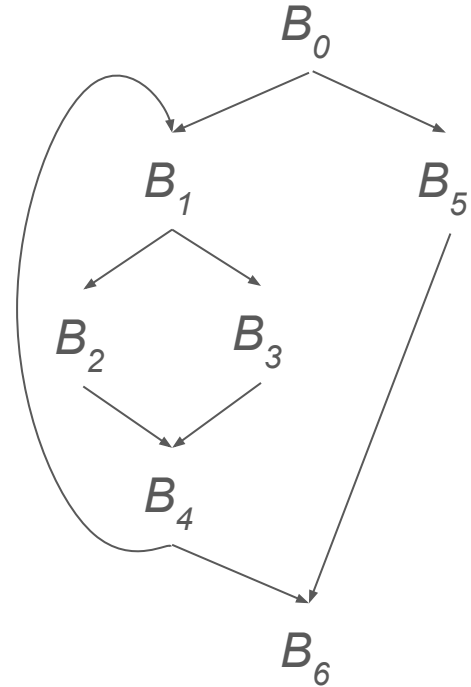| **B**	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DF(B)	$\varnothing$	B_1, B_6	B_4	B_4			

Dominance frontier

p *strictly dominates* q iff
  p dominates q and $p \neq q$.

q is in *dominance frontier* of p iff
- p domitates a predecessor of q.
- p does not strictly dominate q.

$DF(p)$ – dominance frontier of p.

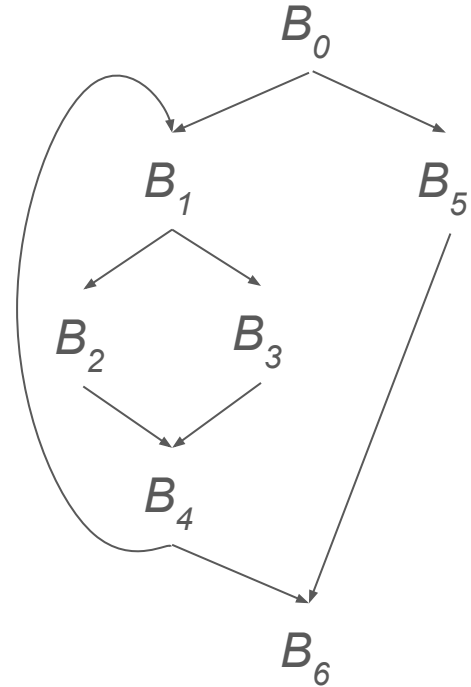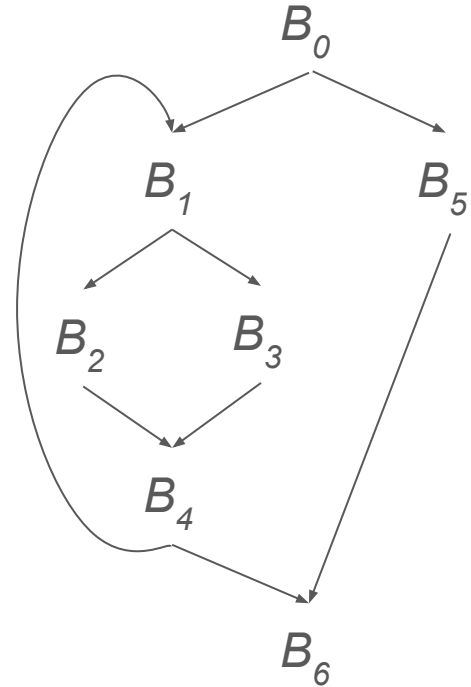| B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
$DF(B)$	$\varnothing$	B_1, B_6	B_4	B_4	B_1, B_6		

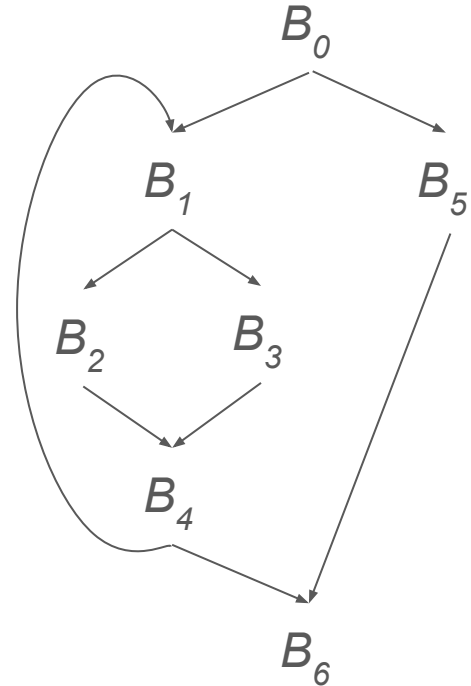# Dominance frontier

p *strictly dominates* q iff
  p dominates q and $p \neq q$.

q is in *dominance frontier* of p iff
- p domitates a predecessor of q.
- p does not strictly dominate q.

DF(p) – dominance frontier of p.

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DF(B)	$\varnothing$	B_1, B_6	B_4	B_4	B_1, B_6	B_6	

Dominance frontier

p *strictly dominates* q iff
  p dominates q and $p \neq q$.

q is in *dominance frontier* of p iff
- p domitates a predecessor of q.
- p does not strictly dominate q.

DF(p) – dominance frontier of p.

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DF(B)	$\varnothing$	B_1, B_6	B_4	B_4	B_1, B_6	B_6	$\varnothing$

Minimal SSA

Idea

an assignment to x in the node B introduces a ϕ-function in every node from DF(B)

1. **ϕ-function placement**
2. renaming

$B_0:$ x = 0
y = 1

$B_1:$ x = x + 1

$B_5:$ x = 2

B_2

$B_3:$ y = y * 2

$B_4:$

$B_6:$

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DF(B)	∅	B_1, B_6	B_4	B_4	B_1, B_6	B_6	∅

Minimal SSA

Idea

an assignment to x in the node B introduces a ϕ-function in every node from DF(B)

1. **ϕ-function placement**
2. renaming

$B_0:$ x = 0
        y = 1

$B_1:$ x = ϕ(x, x)
        x = x + 1

$B_5:$ x = 2

B_2

$B_3:$ y = y * 2

$B_4:$

$B_6:$ x = ϕ(x, x)

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
$DF(B)$	$\varnothing$	B_1, B_6	B_4	B_4	B_1, B_6	B_6	$\varnothing$

23

Minimal SSA

Idea

an assignment to x in the node *B* introduces a ϕ-function in every node from DF(B)
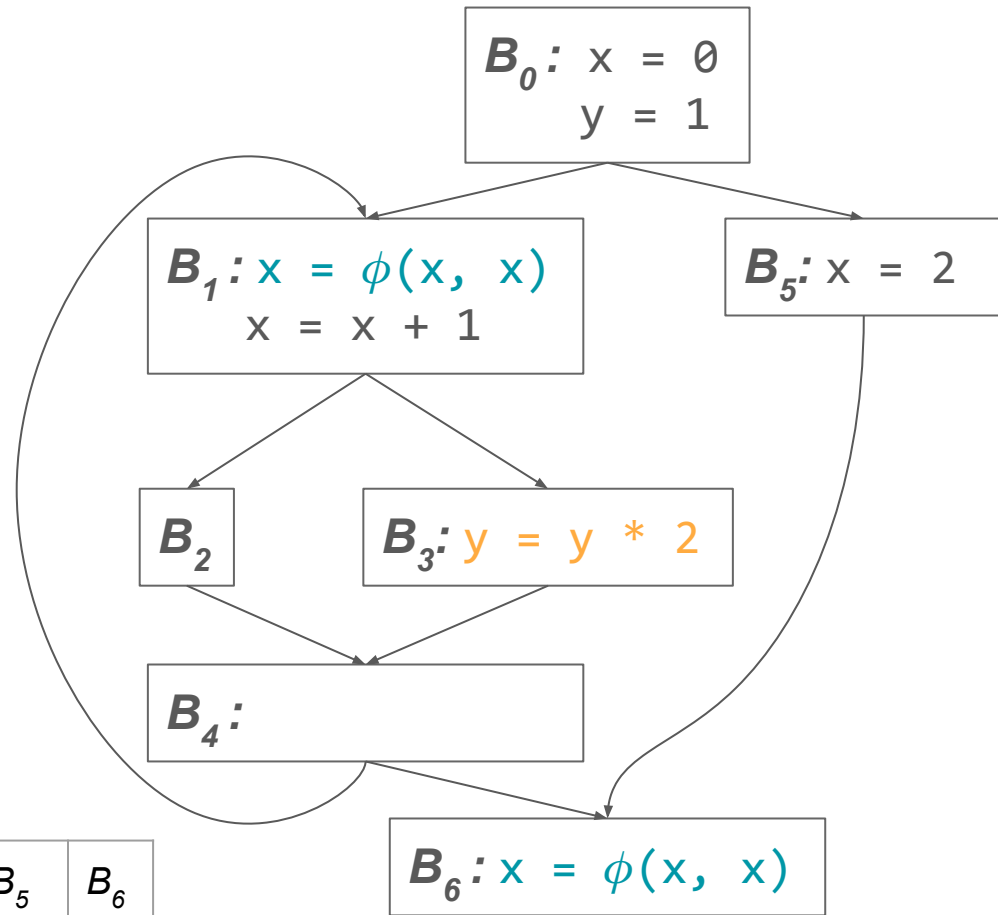
1. **ϕ-function placement**
2. renaming

B_0: x = 0
y = 1

B_1: x = ϕ(x, x)
x = x + 1

B_5: x = 2

B_2

B_3: y = y * 2

B_4: y = ϕ(y, y)

B_6: x = ϕ(x, x)

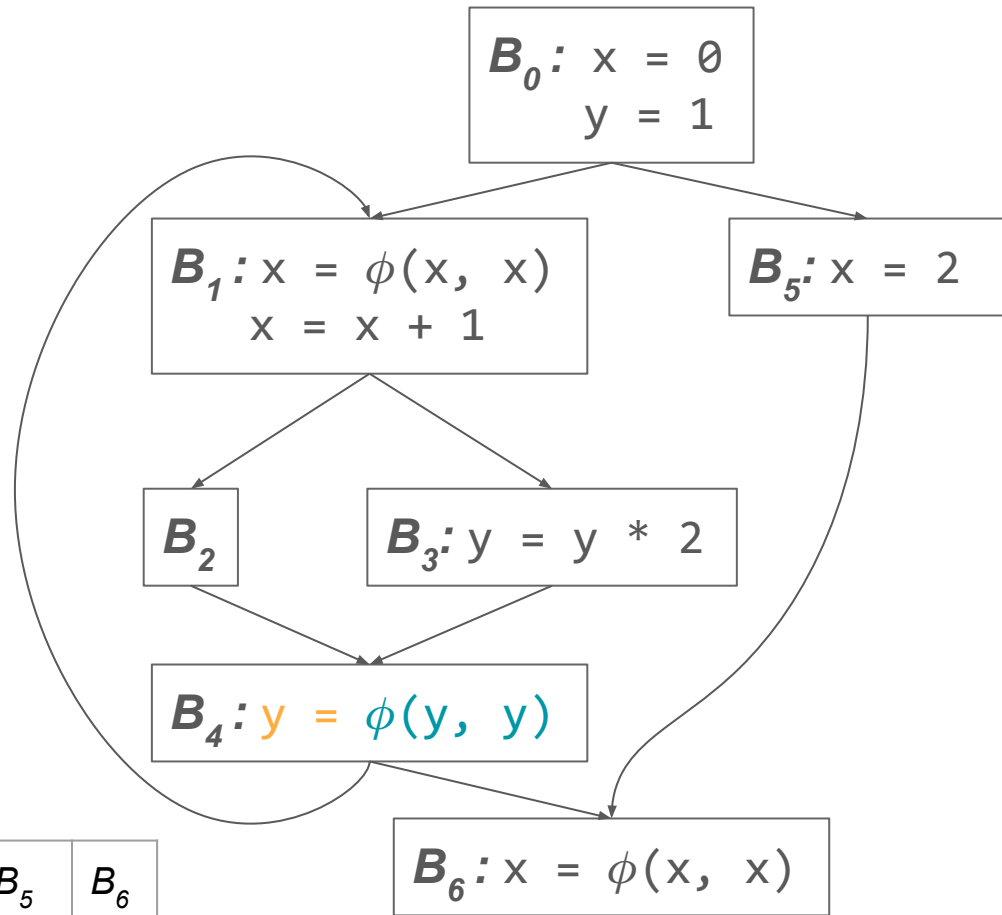| *B*	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DF(B)	$\varnothing$	B_1, B_6	B_4	B_4	B_1, B_6	B_6	$\varnothing$

Minimal SSA

Idea
an assignment to x in the node B introduces a ϕ-function in every node from DF(B)

1. **ϕ-function placement**
2. renaming

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
$DF(B)$	$\varnothing$	B_1, B_6	B_4	B_4	B_1, B_6	B_6	$\varnothing$

B_0: x = 0
   y = 1

B_1: x = ϕ(x, x)
   y = ϕ(y, y)
   x = x + 1

B_5: x = 2

B_2

B_3: y = y * 2

B_4: y = ϕ(y, y)

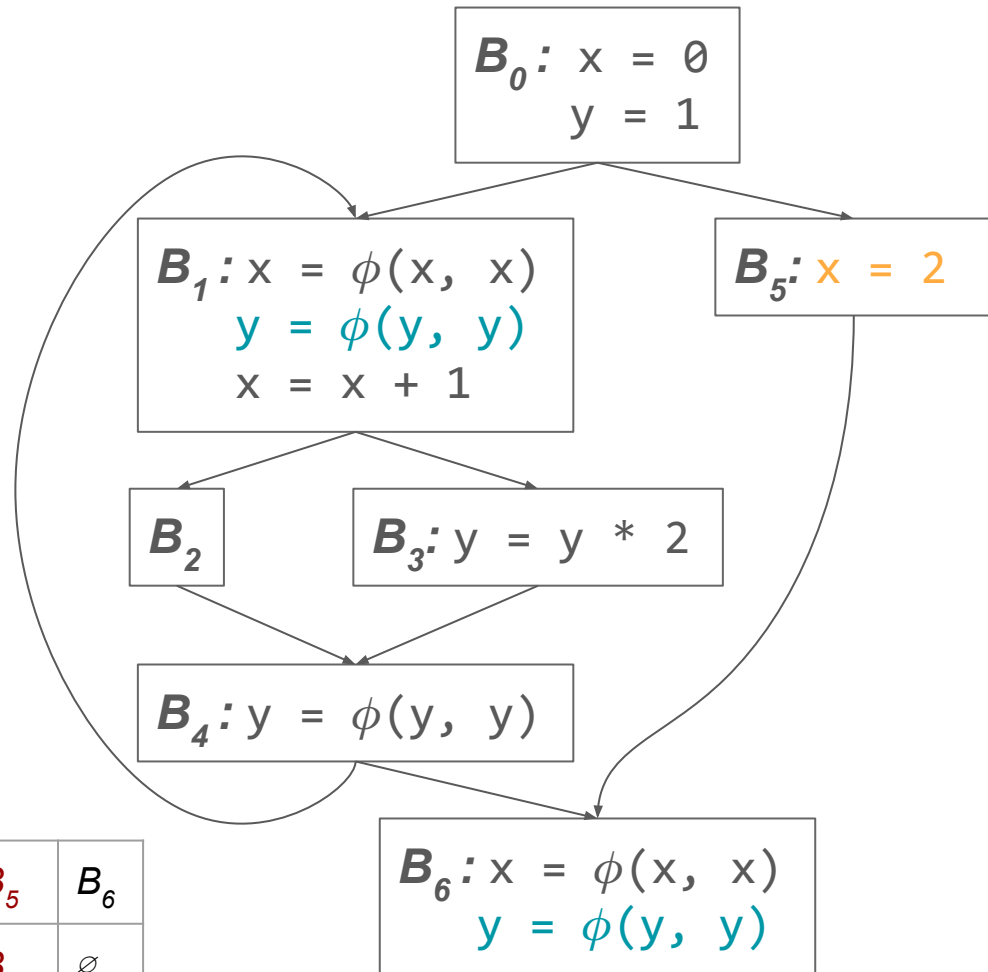B_6: x = ϕ(x, x)
   y = ϕ(y, y)

Minimal SSA

Idea
an assignment to x in the node B introduces a ϕ-function in every node from DF(B)

1. **ϕ-function placement**
2. renaming

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
DF(B)	$\varnothing$	B_1, B_6	B_4	B_4	B_1, B_6	B_6	$\varnothing$

B_0: x = 0
   y = 1

B_1: x = ϕ(x, x)
   y = ϕ(y, y)
   x = x + 1

B_5: x = 2

B_2

B_3: y = y * 2

B_4: y = ϕ(y, y)
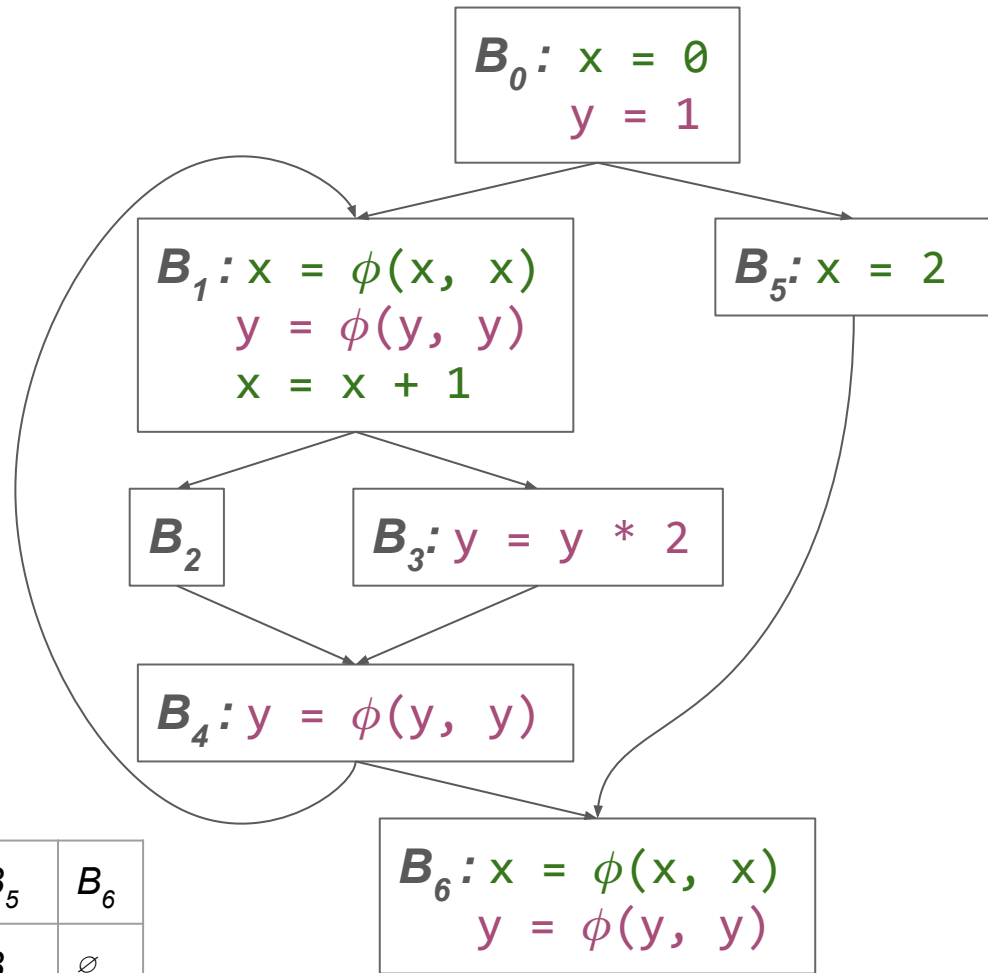
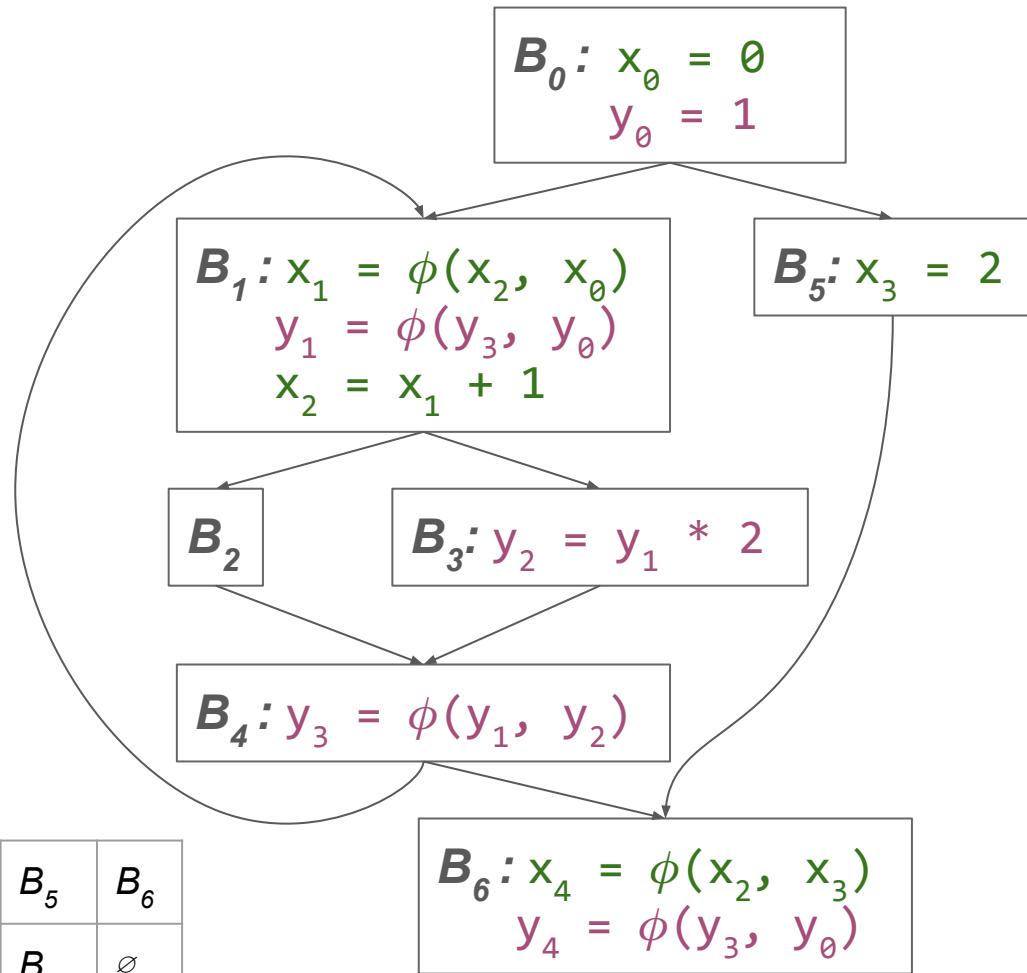B_6: x = ϕ(x, x)
   y = ϕ(y, y)

Minimal SSA

Idea

      an assignment to x in the node B introduces a ϕ-function in every node from DF(B)

1. ϕ-function placement
2. **renaming**

$$B_0 : x_0 = 0$$
$$y_0 = 1$$

$$B_1 : x_1 = \phi(x_2, x_0)$$
$$y_1 = \phi(y_3, y_0)$$
$$x_2 = x_1 + 1$$

$$B_5 : x_3 = 2$$

$$B_2$$

$$B_3 : y_2 = y_1 * 2$$

$$B_4 : y_3 = \phi(y_1, y_2)$$

$$B_6 : x_4 = \phi(x_2, x_3)$$
$$y_4 = \phi(y_3, y_0)$$

B	B_0	B_1	B_2	B_3	B_4	B_5	B_6
$DF(B)$	$\varnothing$	B_1, B_6	B_4	B_4	B_1, B_6	B_6	$\varnothing$

SSA forms

❖ **Maximal SSA**
Introduce a ϕ-function at every join node for every variable

❖ **Minimal SSA**
Introduce a ϕ-function at every join node for every variable where two distinct definitions of the same name meet

❖ **Pruned SSA**
Same as minimal SSA, but don't insert ϕ-functions if its result is not *live*.

❖ **Semipruned SSA**
Same as minimal SSA, but don't insert ϕ-functions for names that are not live across a block boundary

Block Arguments

Instead of using ϕ-nodes (like LLVM), xDSL and MLIR use block arguments to represent control flow – dependent values.

```
func.func @simple(i64, i1) -> i64 {
^bb0(%a: i64, %cond: i1): // Code dominated by ^bb0 may refer to %a
  cf.cond_br %cond, ^bb1, ^bb2


^bb1:
  cf.br ^bb3(%a: i64)    // Branch passes %a as the argument


^bb2:
  %b = arith.addi %a, %a : i64
  cf.br ^bb3(%b: i64)    // Branch passes %b as the argument

// ^bb3 receives an argument, named %c, from predecessors
// and passes it on to bb4 along with %a. %a is referenced
// directly from its defining operation and is not passed through
// an argument of ^bb3.
^bb3(%c: i64):
  cf.br ^bb4(%c, %a : i64, i64)


^bb4(%d : i64, %e : i64):
  %0 = arith.addi %d, %e : i64
  return %0 : i64   // Return is also a terminator.
}
```

29