Intermediate Representations
Part II

Quiz

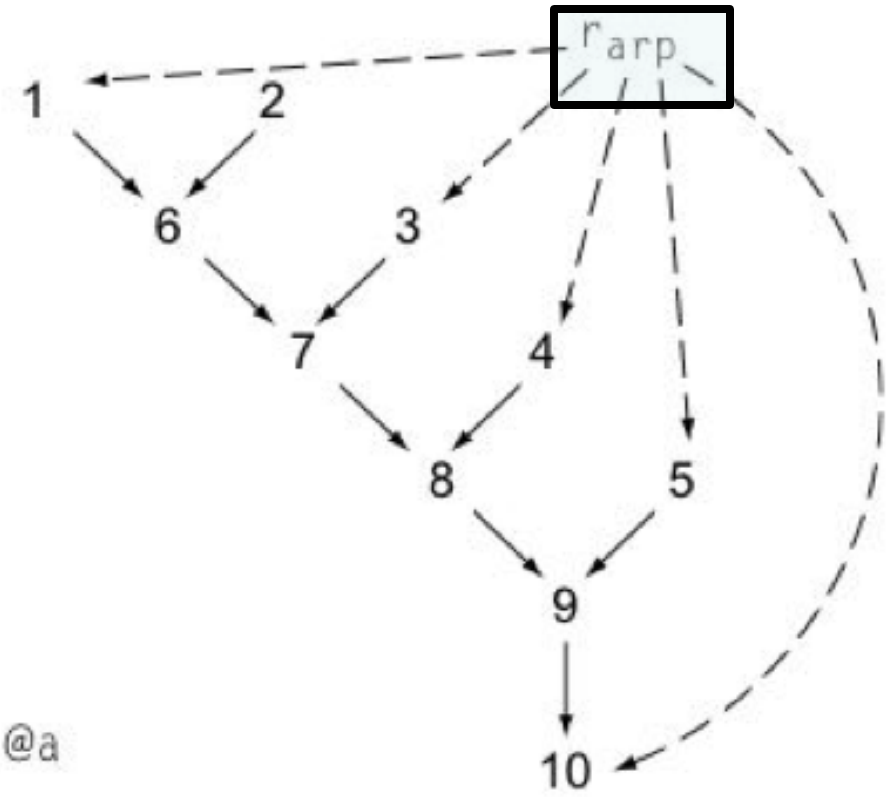

```
1        A = 4
2        t1 = A * B
3   L1:  t2 = t1/C
4        if t2 < W goto L2
5        M = t1 * k
6        t3 = M + I
7   L2:  H = I
8        M = t3 - H
9        if t3 >= 0 goto L4
10  L3:  goto L1
11  L4:  goto L3
```

What are the leaders?

Please use the line numbers!

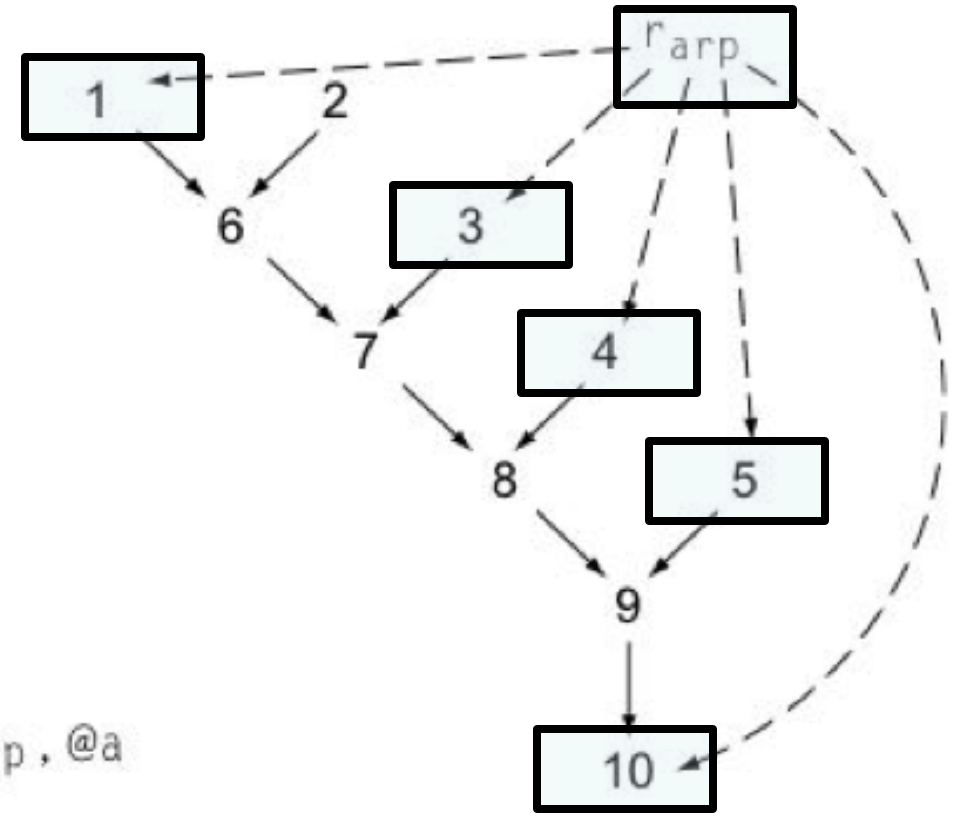
Dependence Graph

1	loadAI	r_{arp}, @a	⇒	r_a
2	loadI	2	⇒	r_2
3	loadAI	r_{arp}, @b	⇒	r_b
4	loadAI	r_{arp}, @c	⇒	r_c
5	loadAI	r_{arp}, @d	⇒	r_d
6	mult	r_a, r_2	⇒	r_a
7	mult	r_a, r_b	⇒	r_a
8	mult	r_a, r_c	⇒	r_a
9	mult	r_a, r_d	⇒	r_a
10	storeAI	r_a	⇒	r_{arp}, @a



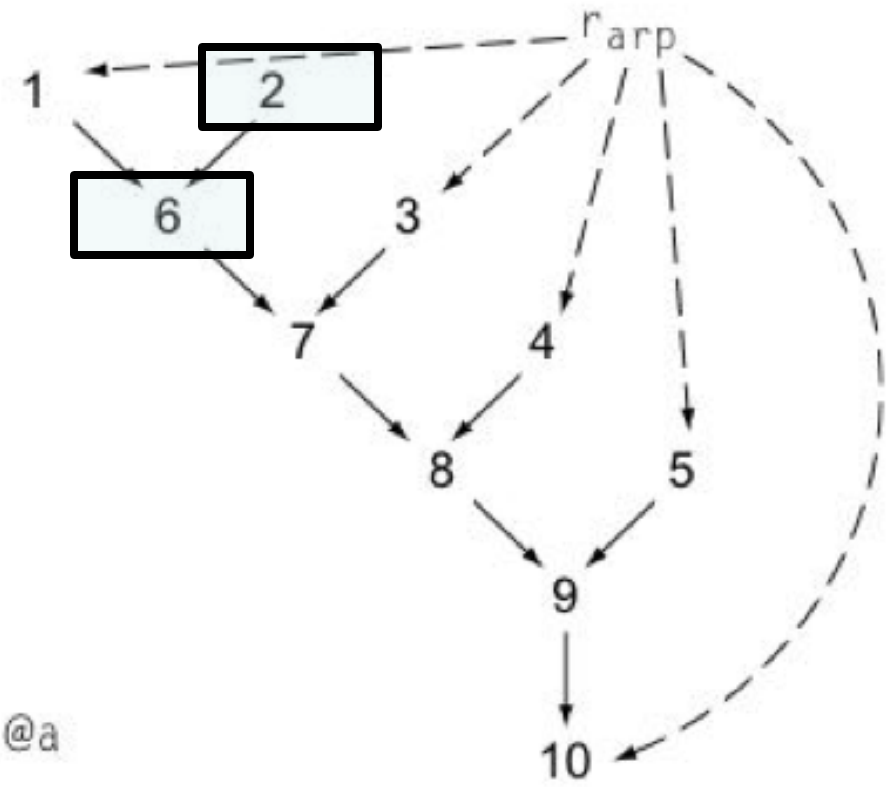
Dependence Graph

1	loadAI	r_{arp}, @a	⇒	r_a
2	loadI	2	⇒	r_2
3	loadAI	r_{arp}, @b	⇒	r_b
4	loadAI	r_{arp}, @c	⇒	r_c
5	loadAI	r_{arp}, @d	⇒	r_d
6	mult	r_a, r_2	⇒	r_a
7	mult	r_a, r_b	⇒	r_a
8	mult	r_a, r_c	⇒	r_a
9	mult	r_a, r_d	⇒	r_a
10	storeAI	r_a	⇒	r_{arp}, @a



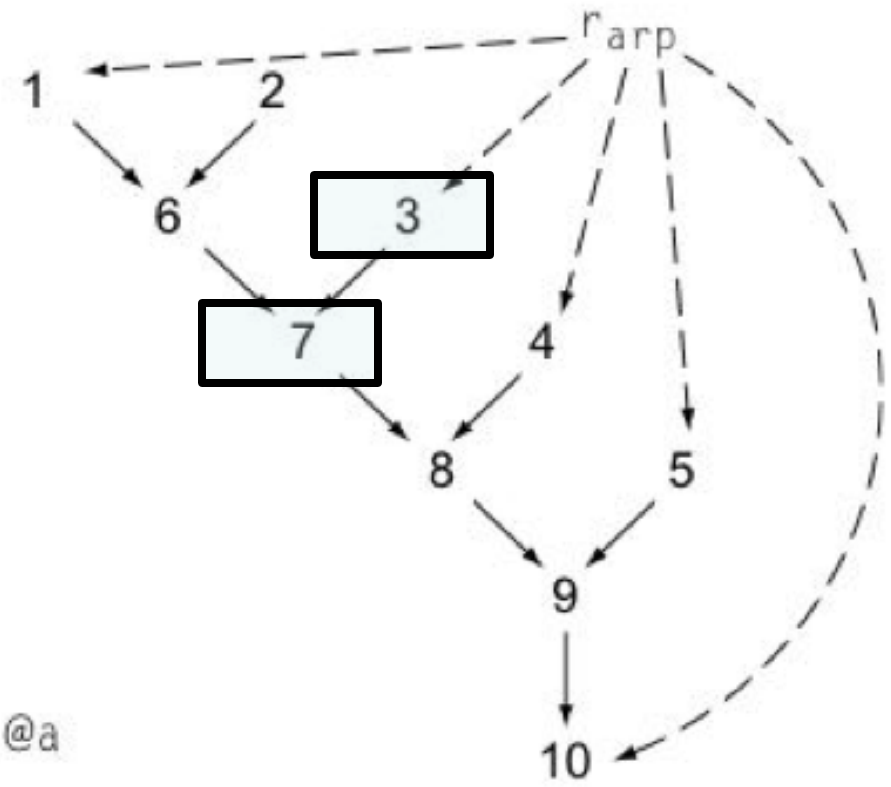
Dependence Graph

1	loadAI	r_{arp}, @a	⇒	r_a
2	loadI	2	⇒	r_2
3	loadAI	r_{arp}, @b	⇒	r_b
4	loadAI	r_{arp}, @c	⇒	r_c
5	loadAI	r_{arp}, @d	⇒	r_d
6	mult	r_a, r_2	⇒	r_a
7	mult	r_a, r_b	⇒	r_a
8	mult	r_a, r_c	⇒	r_a
9	mult	r_a, r_d	⇒	r_a
10	storeAI	r_a	⇒	r_{arp}, @a



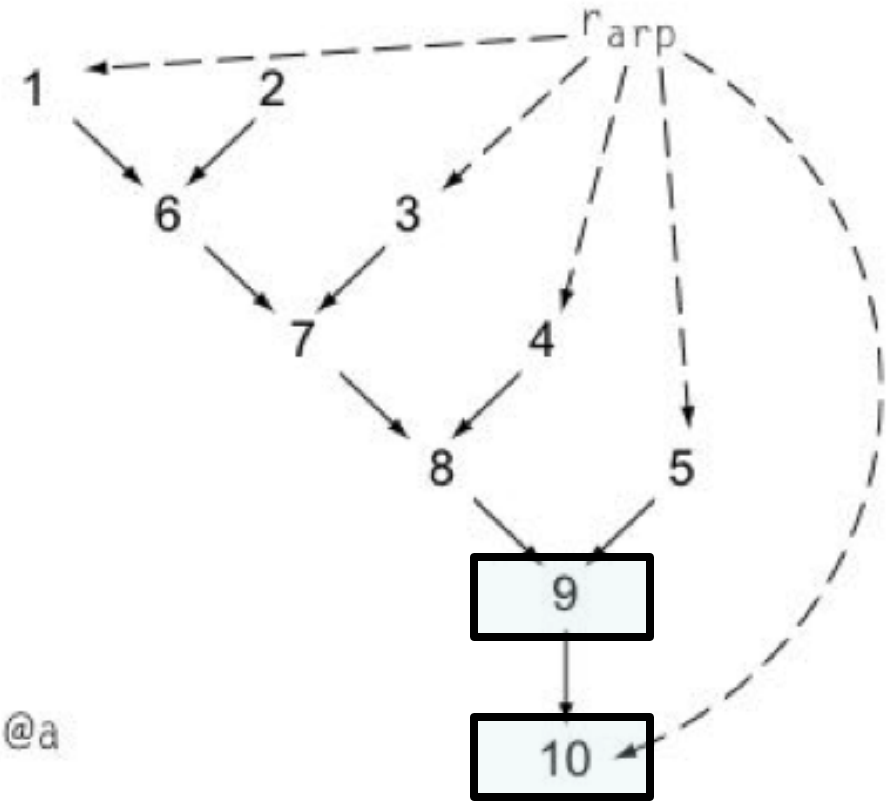
Dependence Graph

1	loadAI	r_{arp}, @a	⇒	r_a
2	loadI	2	⇒	r_2
3	loadAI	r_{arp}, @b	⇒	r_b
4	loadAI	r_{arp}, @c	⇒	r_c
5	loadAI	r_{arp}, @d	⇒	r_d
6	mult	r_a, r_2	⇒	r_a
7	mult	r_a, r_b	⇒	r_a
8	mult	r_a, r_c	⇒	r_a
9	mult	r_a, r_d	⇒	r_a
10	storeAI	r_a	⇒	r_{arp}, @a

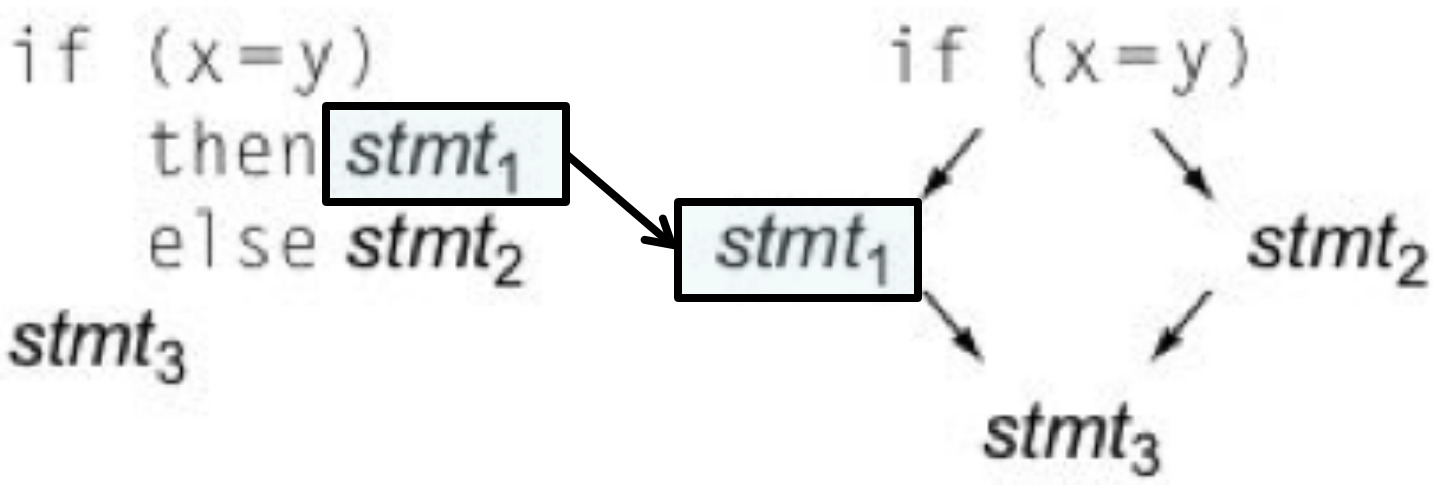


Dependence Graph

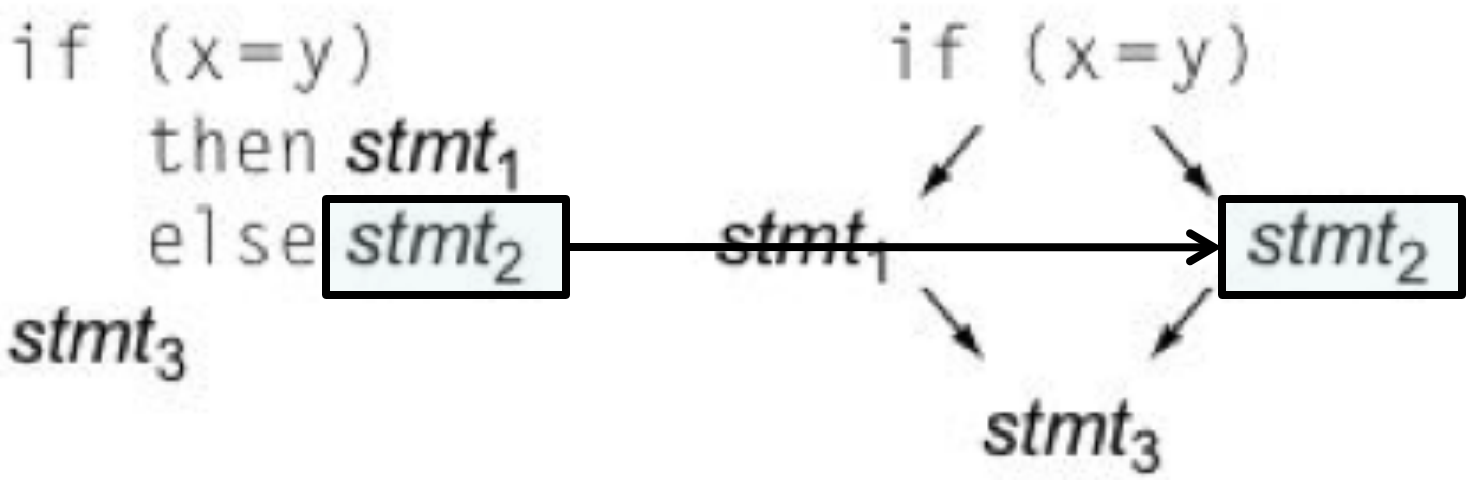
1	loadAI	r_{arp}, @a	⇒	r_a
2	loadI	2	⇒	r_2
3	loadAI	r_{arp}, @b	⇒	r_b
4	loadAI	r_{arp}, @c	⇒	r_c
5	loadAI	r_{arp}, @d	⇒	r_d
6	mult	r_a, r_2	⇒	r_a
7	mult	r_a, r_b	⇒	r_a
8	mult	r_a, r_c	⇒	r_a
9	mult	r_a, r_d	⇒	r_a
10	storeAI	r_a	⇒	r_{arp}, @a



Quick Review of CFG

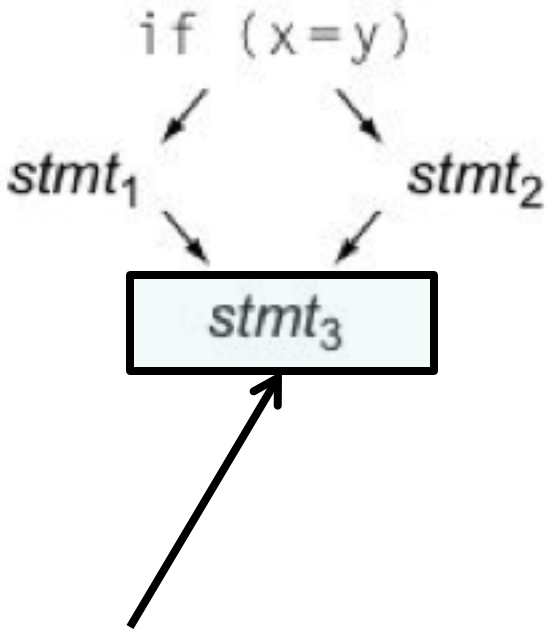


Quick Review of CFG



Quick Review of CFG

```
if (x=y)
    then stmt1
    else stmt2
stmt3
```



This is a join point.

Single Static Assignment: High-Level Definition

- Each assignment to variable given a unique name
- All uses reached by that assignment are renamed

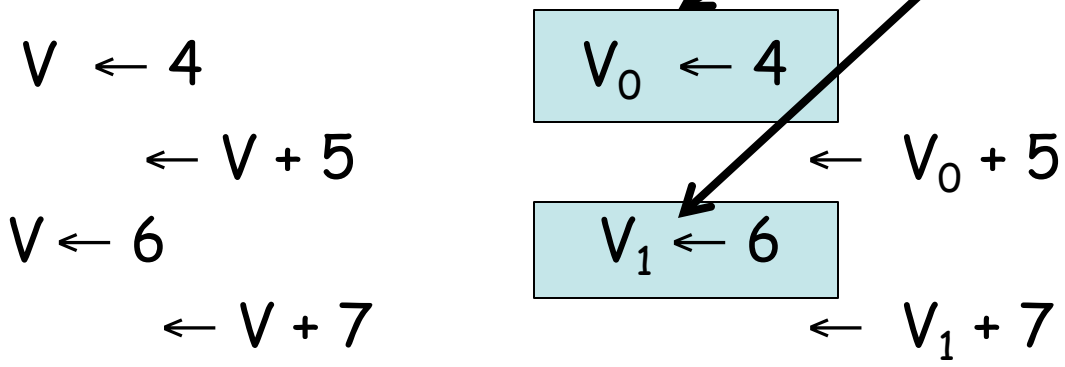
Single Static Assignment: High-Level Definition

- Each assignment to variable given a unique name
- All uses reached by that assignment are renamed
- Easy for straight-line code

Before SSA	After SSA
$V \leftarrow 4$	$V_0 \leftarrow 4$
$\leftarrow V + 5$	$\leftarrow V_0 + 5$
$V \leftarrow 6$	$V_1 \leftarrow 6$
$\leftarrow V + 7$	$\leftarrow V_1 + 7$

Single Static Assignment: High-Level Definition

- Each assignment to variable given a unique name
- All uses reached by that assignment are renamed
- Easy for straight-line code

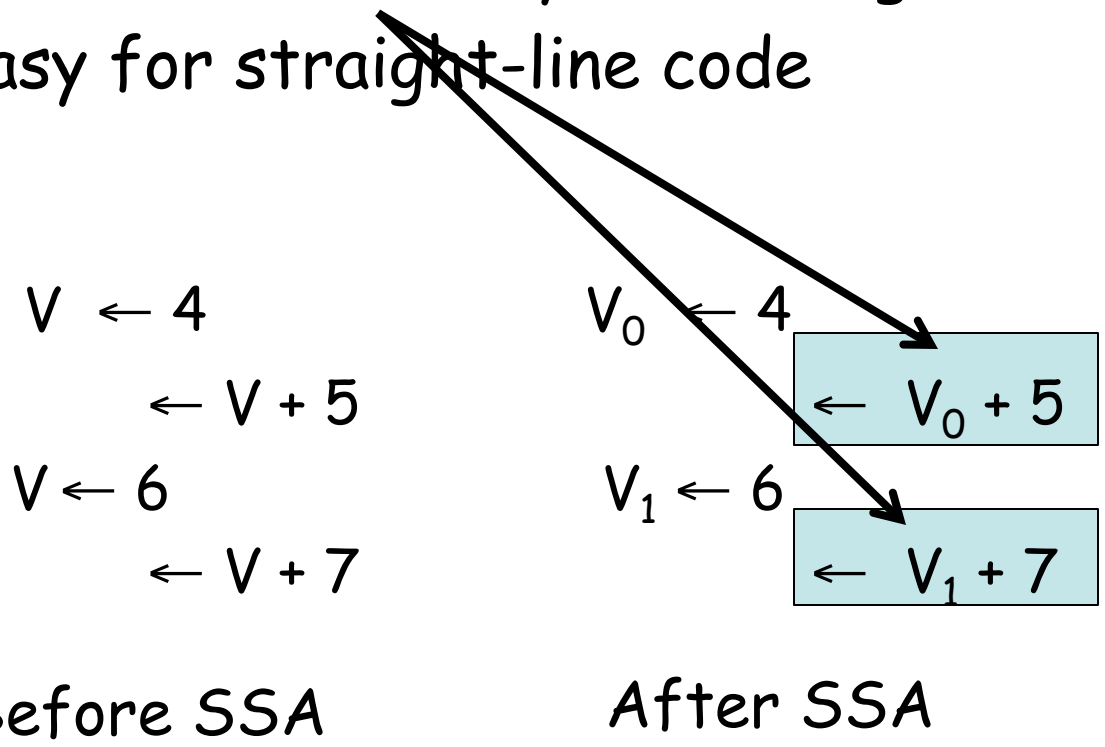


Before SSA

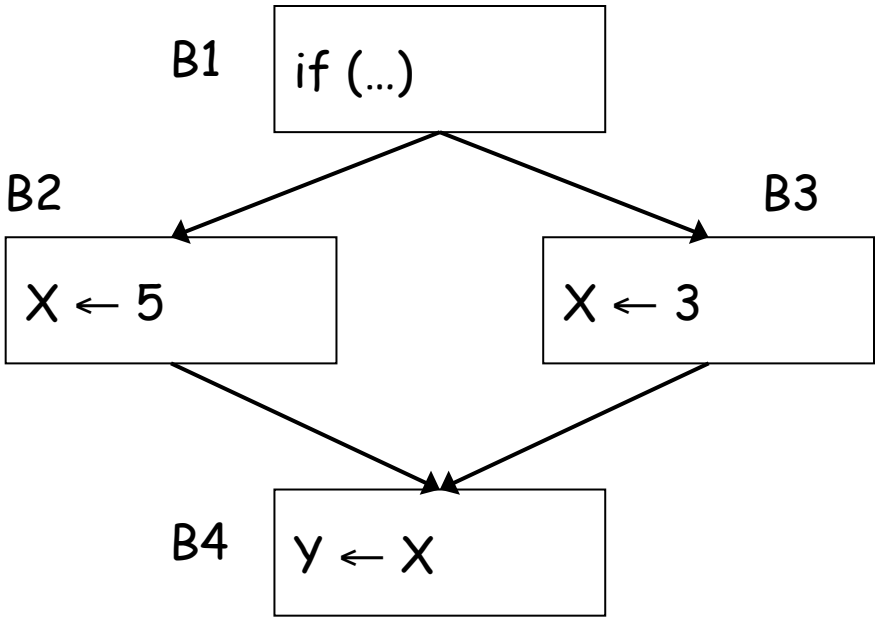
After SSA

Single Static Assignment: High-Level Definition

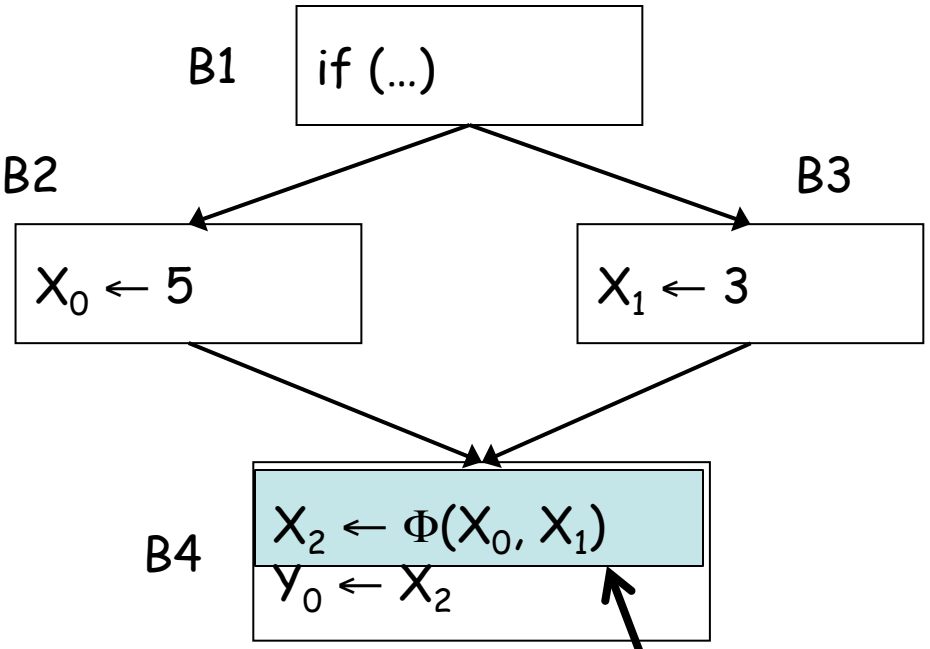
- Each assignment to a variable is given a unique name
- All uses reached by that assignment are renamed
- Easy for straight-line code



What About Control Flow?



Before SSA

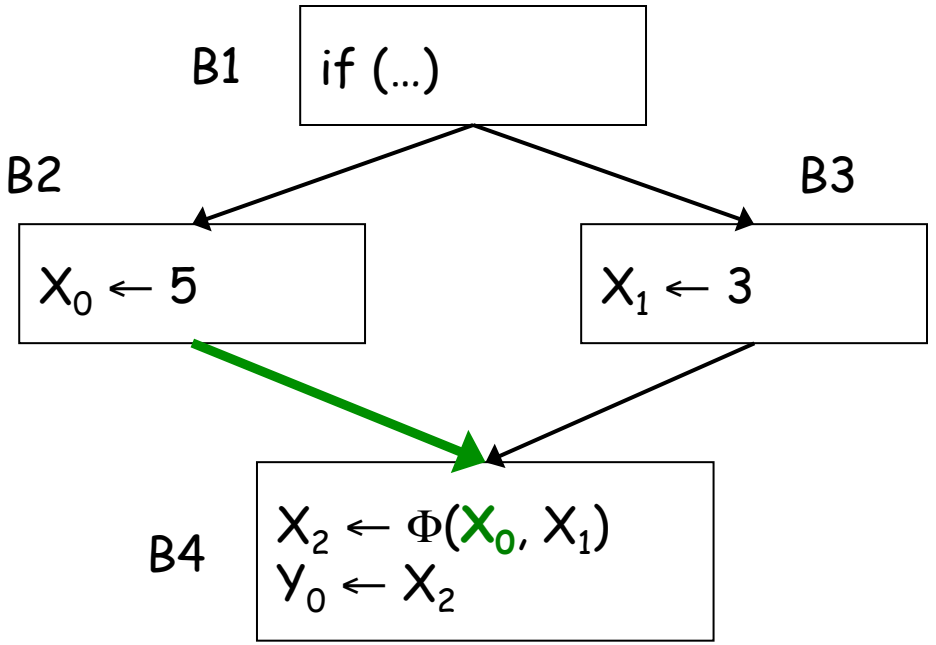


After SSA

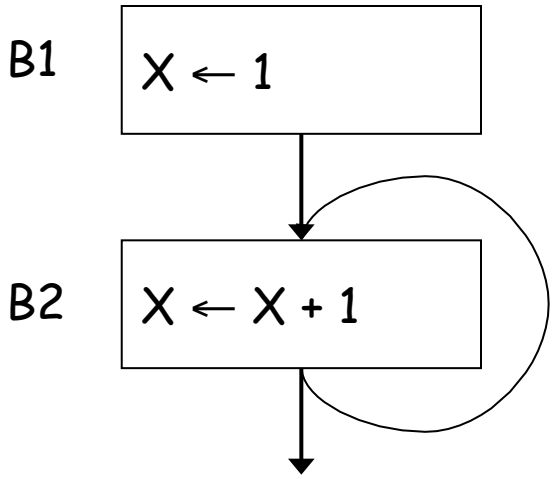
Φ-nodes: combine values from distinct edges (join points)

Φ-Functions

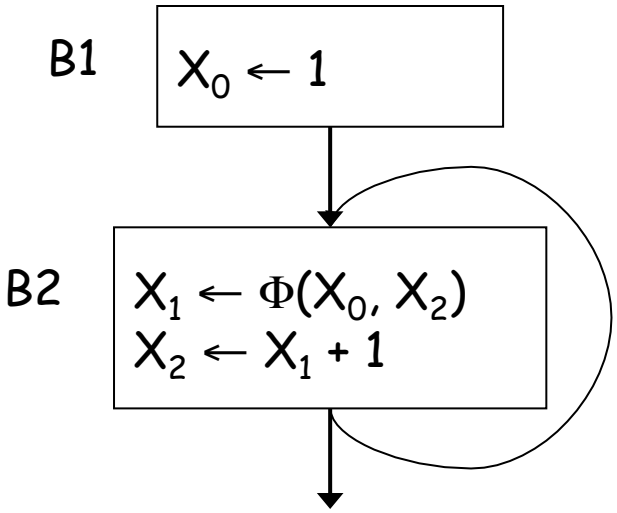
- At each join, add special assignment: "ϕ function":
  - → operands indicate which assignments reach join
  - → j_{th} operand = j_{th} predecessor
- If control reaches join from j_{th} predecessor, then value of ϕ(…) is value of j_{th} operand



What About Control Flow?



Before SSA



After SSA

Class Example


```
if P
  then v = 4
  else v = 6
x=v
```

Class Example


```
if P
  then v = 4
  else v = 6
x=v
```

```
if P
  then v0 = 4
  else v1 = 6
V2 = φ(v0, v1)
X0 = V2
```

Another SSA Example

Original

```
x ← …
y ← …
while (x < k)
    x ← x + 1
    y ← y + x
```

Another SSA Example

Original

```
x ← …
y ← …
while (x < k)
    x ← x + 1
    y ← y + x
```

Same code using gotos

```
    x ← …
    y ← …
    if (x >= k)
        goto next
loop: x ← x + 1
      y ← y + x
      if(x < k)
          goto loop
next:     …
```

Another SSA Example

Original

```
x ← …
y ← …
while (x < k)
    x ← x + 1
    y ← y + x
```

Same code using gotos

```
    x ← …
    y ← …
    if (x >= k)
        goto next
loop: x ← x + 1
      y ← y + x
      if(x < k)
          goto loop
next:     …
```

Another SSA Example

Same code using gotos

```
     x ← …
     y ← …
     if (x >= k)
         goto next
loop: x ← x + 1
      y ← y + x
       if(x < k)
          goto loop
next:      …
```

SSA-form

$$
\begin{array}{l}
\quad x_0 \leftarrow \ldots \\
\quad y_0 \leftarrow \ldots \\
\quad \text{if } (x_0 >= k) \text{ goto next} \\
\text{loop: } x_1 \leftarrow \phi(x_0, x_2) \\
\qquad y_1 \leftarrow \phi(y_0, y_2) \\
\qquad\quad x_2 \leftarrow x_1 + 1 \\
\qquad\quad y_2 \leftarrow y_1 + x_2 \\
\qquad \text{if}(x_2 < k) \text{ goto loop} \\
\text{next:} \qquad \ldots
\end{array}
$$

Another SSA Example

Same code using gotos

```
    x ← …
    y ← …
    if (x >= k)
        goto next
loop: x ← x + 1
      y ← y + x
       if(x < k)
          goto loop
next:      …
```

SSA-form

```
    x0 ← …
    y0 ← …
    if (x0 >= k) goto next
loop: x1 ← φ(x0,x2)
      y1 ← φ(y0,y2)
         x2 ← x1 + 1
         y2 ← y1 + x2
      if(x2 < k) goto loop
next:      …
```

Another SSA Example

Same code using gotos

```
    x ← …
    y ← …
    if (x >= k)
        goto next
loop: x ← x + 1
      y ← y + x
       if(x < k)
          goto loop
next:      …
```

SSA-form

```
    x0 ← …
    y0 ← …
    if (x0 >= k) goto next
loop: x1 ← ϕ(x0,x2)
      y1 ← ϕ(y0,y2)
         x2 ← x1 + 1
         y2 ← y1 + x2
      if(x2 < k) goto loop
next:      …
```

Another SSA Example

Same code using gotos

```
    x ← …
    y ← …
    if (x >= k)
        goto next
loop: x ← x + 1
      y ← y + x
      if(x < k)
        goto loop
next:    …
```

SSA-form

```
    x0 ← …
    y0 ← …
    if (x0 >= k) goto next
loop: x1 ← φ(x0,x2)
      y1 ← φ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
      if(x2 < k) goto loop
next:    …
```

Static Single Assignment Form

Same code using gotos

```
    x ← …
    y ← …
    if (x >= k)
        goto next
loop: x ← x + 1
      y ← y + x
       if(x < k)
          goto loop
next:      …
```

This is new! Inserted at points where diff. control flow merge.

SSA-form

```
    x0 ← …
    y0 ← …
    if (x0 >= k) goto next
loop: x1 ← φ(x0,x2)
      y1 ← φ(y0,y2)
        x2 ← x1 + 1
        y2 ← y1 + x2
      if(x2 < k) goto loop
next:      …
```

Static Single Assignment Form

Same code using gotos

```
    x ← …
    y ← …
    if (x >= k)
        goto next
loop: x ← x + 1
      y ← y + x
       if(x < k)
          goto loop
next:      …
```

Keeps single assign. property.

SSA-form

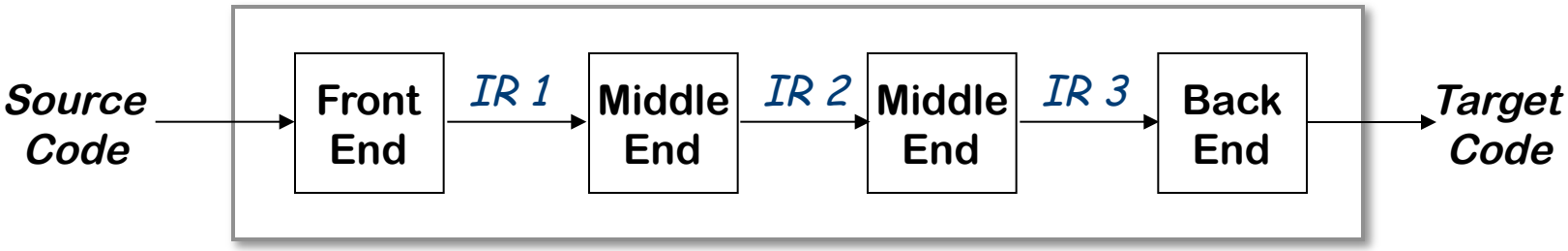
```
    x0 ← …
    y0 ← …
    if (x0 >= k) goto next
loop: x1 ← ϕ(x0,x2)
      y1 ← ϕ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
    if(x2 < k) goto loop
next:      …
```

Static Single Assignment Form Advantages

Strengths of SSA-form

- Sharper analysis because values never redefined
- Simplifies and improves optimizations
- (Sometimes) faster algorithms

Using Multiple Representations



- Repeatedly lower the level of the intermediate representation
  - → Each intermediate representation is suited towards certain optimizations
- Example: the Open64 compiler
  - → WHIRL intermediate format
    - Consists of 5 different *IR*s that are progressively more detailed and less abstract

Memory Models

Two major models

- Register-to-register model
  - → Keep all values that can legally be stored in a register in registers
  - → Ignore machine limitations on number of registers
  - → Compiler back-end must insert loads and stores
- Memory-to-memory model
  - → Keep all values in memory
  - → Only promote values to registers directly before they are used
  - → Compiler back-end can remove loads and stores
- Compilers for RISC machines usually use register-to-register
  - → Reflects programming model
  - → Easier to determine when registers are used

The Rest of the Story…

Representing the code is only part of an *IR*

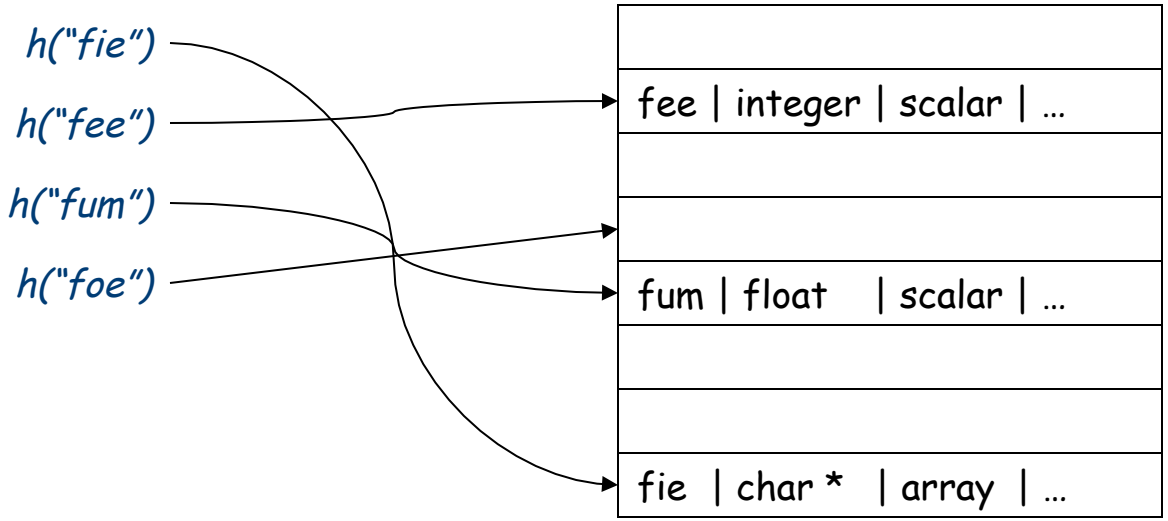
There are other necessary components

- Symbol table
- Constant table
  - → Representation, type
  - → Storage class, offset
- Storage map
  - → Overall storage layout
  - → Overlap information
  - → Virtual register assignments

Symbol Tables

Traditional approach to building a symbol table uses hashing

- One table scheme
  - → Lots of wasted space



Hash table

Symbol Tables

Another approach to building a symbol table uses hashing

- One approach: Two-table scheme
  - → Sparse index to reduce chance of collisions
  - → Dense table to hold actual data
    - Easy to expand, to traverse, to read & write from/to files
- Use chains in index to handle collisions

Collision occurs when h() returns a slot in the sparse index that is already full.