



COMP 520 - Compilers

Lecture 16 – Code/Data Path Analysis

Reminders

- If you submitted PA3 late, make a private post on Piazza so we can determine an appropriate grade.
- Submit to Partial tests, and your first submission to Hidden tests implies you need a grade.
- Midterm 2 on next Thursday, 4/11

Reminders (2)

- As of Lec 15, you have everything you need to do PA4.
- Start sooner rather than later.
- Midterm 2 on next Thursday, 4/11

Compiler Optimization



Dataflow Analysis

Code Analysis

Data Liveness


Expr Liveness

Register
Minimalization

Multiple CodePath
Generation

TODAY

Compilers are **magic**



- This phrase is humorous.
- For the compiler developer, **not so much**.
- What exactly is so **magical** about a compiler?
 - It has the ability to nearly ignore how the programmer wrote code, and instead does something equivalent and more optimized. (*not always a good thing*)

Today

- Data and Expression Liveness analysis
- Algorithms to analyze data usage and memory dependencies

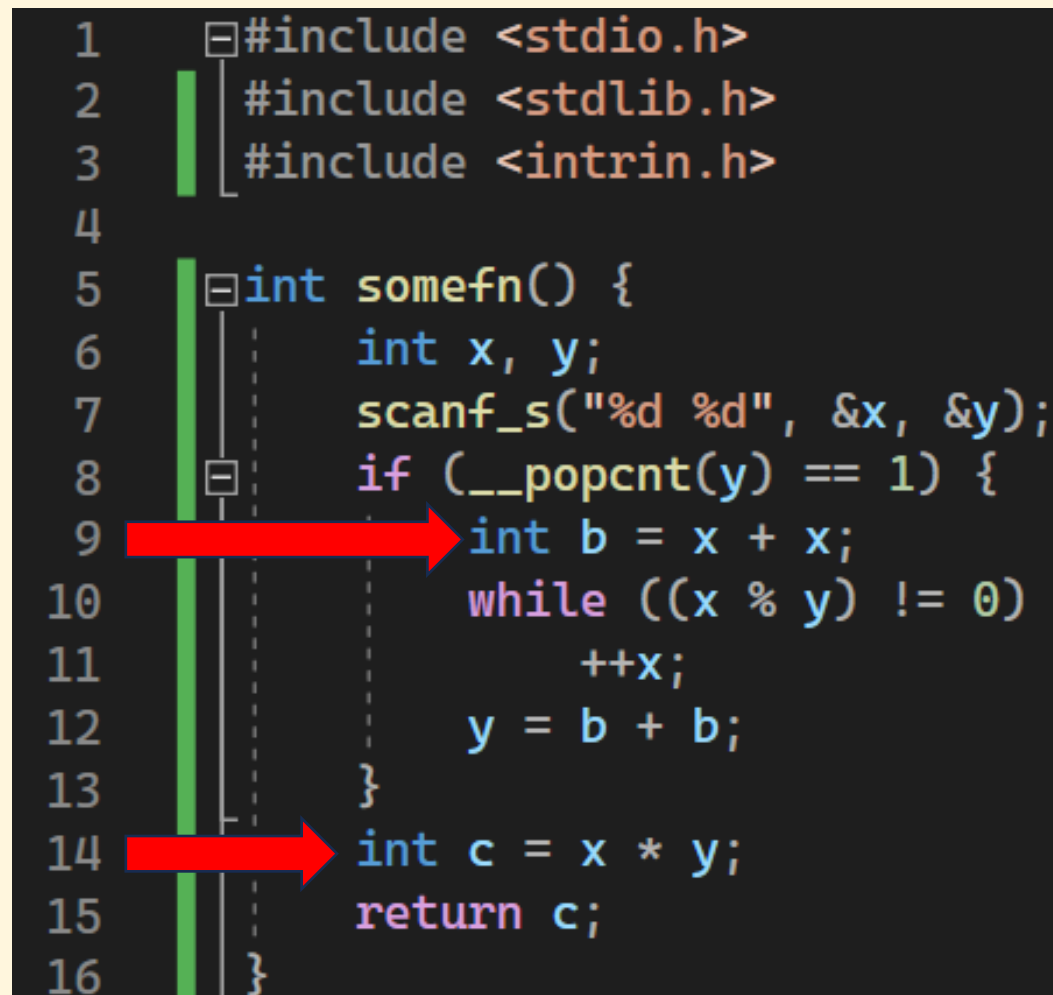
Goal

- Reduce memory usage and instruction count.

Motivation

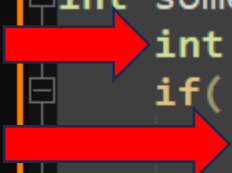
- Variable `b` and `c` are never used at the same time
- Can save space by not keeping both in memory

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <intrin.h>
4
5  int somefn() {
6      int x, y;
7      scanf_s("%d %d", &x, &y);
8      if (__popcnt(y) == 1) {
9          int b = x + x;
10         while ((x % y) != 0)
11             ++x;
12         y = b + b;
13     }
14     int c = x * y;
15     return c;
16 }
```



Motivation (2)

```
1 int somefn() {  
2     int numBytes = recv(...,buf,...);  
3     if( numBytes > 0 ) {  
4         int code;  
5         sscanf(buf,"%d",&code);  
6         return code;  
7     }  
8     return -1;  
9 }
```



- We have two int variables, but after **line 3**, numBytes is never used again
- Ask the developer to change the code?

Motivation (3)

Old

```
1 int somefn() {  
2     int numBytes = recv(...,buf,...);  
3     if( numBytes > 0 ) {  
4         int code;  
5         sscanf(buf,"%d",&code);  
6         return code;  
7     }  
8     return -1;  
9 }
```

New

```
1 int somefn() {  
2     int numBytes = recv(...,buf,...);  
3     if( numBytes > 0 ) {  
4         sscanf(buf,"%d",&numBytes);  
5         return numBytes;  
6     }  
7     return -1;  
8 }
```

- But we now have a problem, the variable name “numBytes” does not actually describe its function
- To support **good coding practices**, we will need to solve how to reduce memory consumption **without** asking the developer to change their programming habits.

Problem Statement

- Programmers create variables whenever.
- They do not want to reuse variables that are available.
- On limited compute capacity machines, we cannot afford to waste memory.



Scoped Data Liveness

A suboptimal but simple solution.

Scoped Data Liveness

- Recall: when a local variable is declared, create stack space for it (simple PA4, Lec14-15)
- Idea: whenever a scope closes, reclaim stack space.

Scoped Data Liveness (2)

- Idea: whenever a scope closes, reclaim stack space.



RIP

```
int x = 0;
if( b > 0 ) {
    int y = 2*b;
    x = y;
}
printf("%d",x);
```



Code Gen

push 0 # Create Stack Space &x = rbp-8
...

Scoped Data Liveness (3)

- Idea: whenever a scope closes, reclaim stack space.

```
int x = 0;  
if( b > 0 ) {  
    int y = 2*b;  
    x = y;  
}  
printf("%d", x);
```

RIP

Code Gen

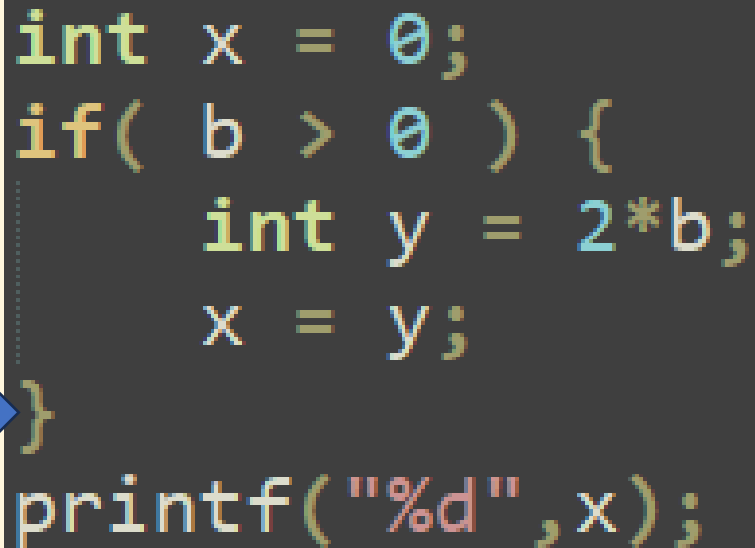
push 0 # Create Stack Space &x = rbp-8

...

push 0 # Create &y = rbp-16

Scoped Data Liveness (4)

- Idea: whenever a scope closes, reclaim stack space.



```
int x = 0;
if( b > 0 ) {
    int y = 2*b;
    x = y;
}
printf( "%d", x );
```

push 0 # Create Stack Space &x = rbp-8

...

push 0 # Create &y = rbp-16

mov rax, 2

imul [b]

mov [y],rax

Code Gen

add rsp,8 # Reclaim y's space

Scoped Data Liveness – Not Optimal

- Why is this not enough?

Scoped Data Liveness – Counterexample

```
1  int fn(int x) {  
2      int a = x + 2;  
3      int b = a * a;  
4      int c = b + x;  
5      return c;  
6  }
```

- At this point, variable `a` is no longer used.
- Thus, some other strategy can be better.

When you have really long methods..

```
1  void main() {  
2      int a = 2;  
3      int b = a * a;  
4      int c = b + b;  
5      int d = c / a;  
6      int e = d + d;  
7      int f = e / 2;  
8      int g = f - e;  
9      ...  
10 }
```

- If the programmer writes bad code, then sure, we have no obligation to make sure it runs.
- But you can't dictate programming habits, and what if some methods just end up being very complicated?
- Also, we would want our compiler to work even if others don't.

Scoped Data Liveness Overview

Overview:

- Reclaim stack space when a scope ends.
- Not optimal (too coarse-grain).
- In PA4: expected to clean up the stack to some degree, and scoped liveness fulfills that requirement.
- We now study better techniques.

Definition: Live Variable

- Let's formalize *data liveness*.

Defn. A variable *x* is *live* before an instruction if *x* is assigned a value before that point, and an instruction will use *x* after that point.

- **Liveness** is overloaded. Liveness also refers to ensuring lock requests are eventually satisfied.
- Instead, we call it Data Liveness, which is a part of Dataflow Analysis.

Optimality Concerns

- Data Liveness Analysis may overly designate variables as “**live**”.
- Better than the opposite.
- Very difficult in some languages.
Example: access variables by memory offsets.

Output:



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <intrin.h>
4
5  class A {
6  private:
7      int x;
8  public:
9      void output() {
10         printf("%d\n", x);
11     }
12 };
13
14 void main() {
15     A a;
16     *(int*)&a = 520; // nasty!
17     a.output();
18 }
19
```

Microsoft Visual Studio Debug Console

520

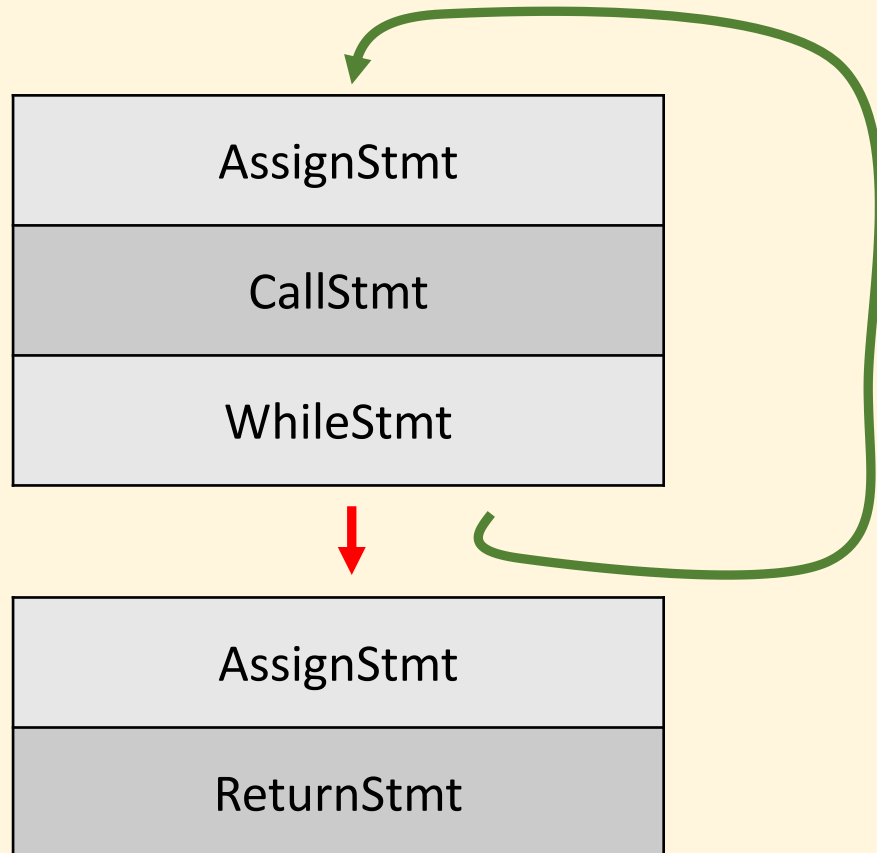
Control Flow Graphs (CFGs)

A super unfortunate acronym.

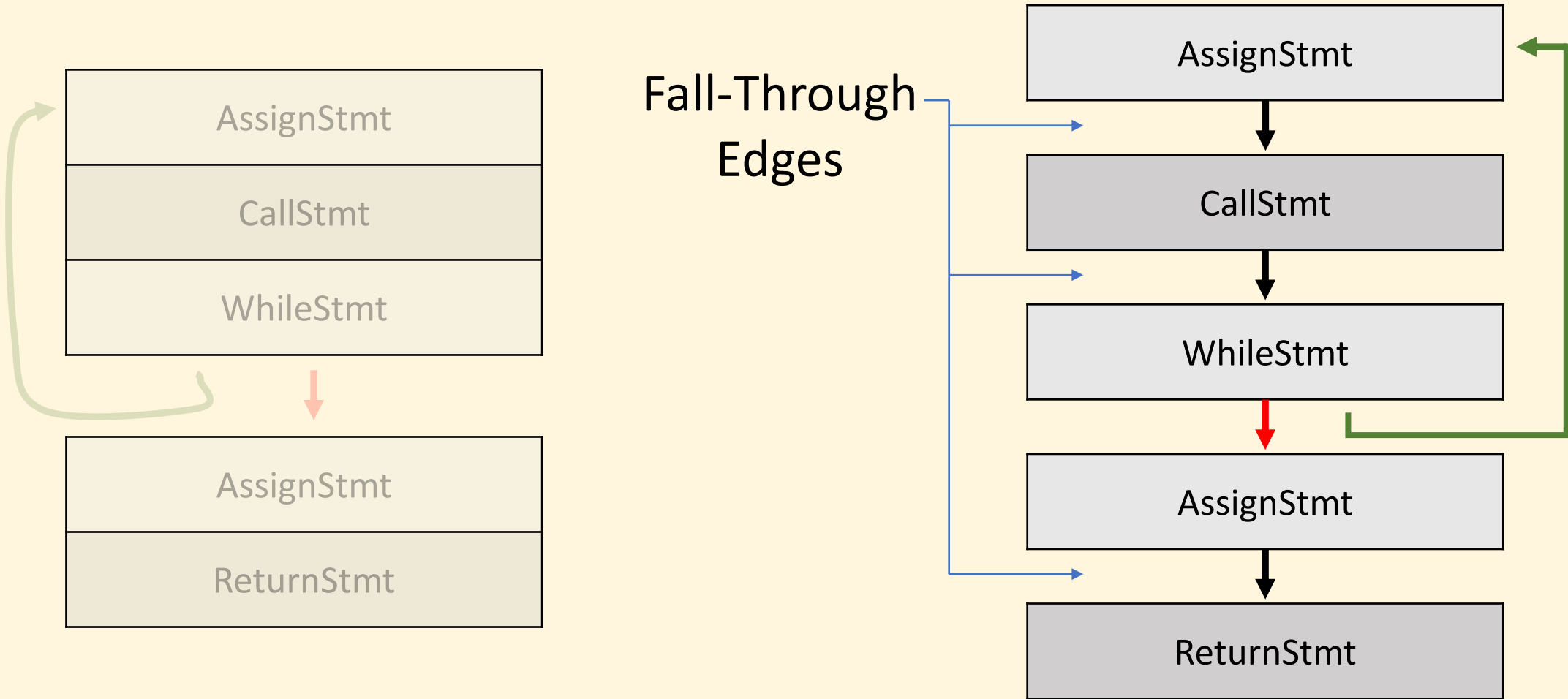
CFG in parsing is context-free grammar.

CFG in code generation is a control flow graph.

Basic Control Flow Graph



Exploded Flow Graph



CFG Edges

Vertex \equiv Operation

(Instruction/Concrete AST)

In-Edge \equiv Directed edge going to the vertex

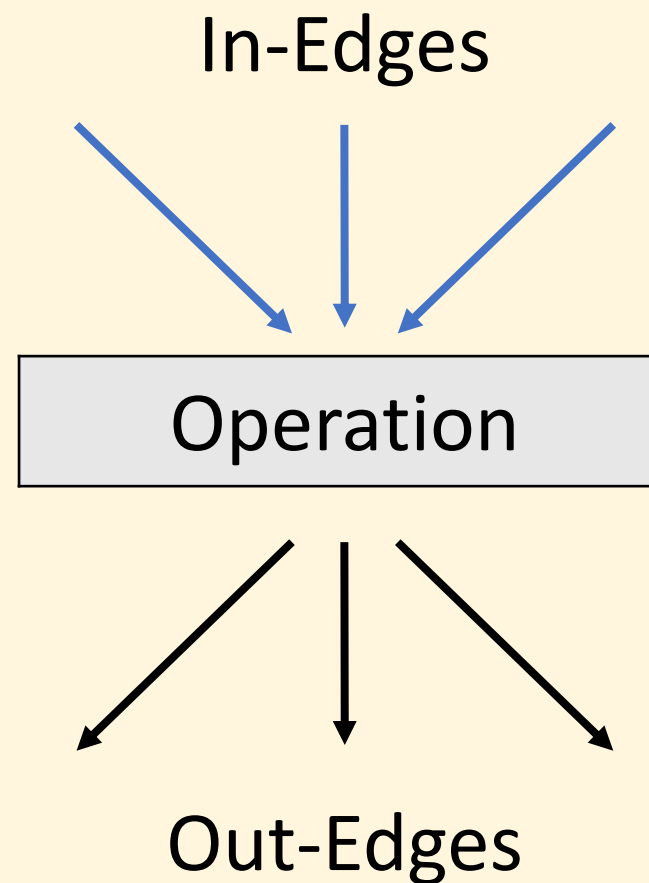
Out-Edge \equiv Directed edge going out of the vertex

Successor/Predecessor \equiv All vertices connected by an out/in-edge

Defn.

Def: in(v) \equiv Set of all variables **live** at In-Edges (before vertex v)

Def: out(v) \equiv Set of all variables **live** at Out-Edges (after vertex v)



Define Data Liveness **at Edges** (before and after)

- Consider a motivating example:

```
int x = y + 1
```

(Assume *y* never used again)

- So both *x* and *y* can use:

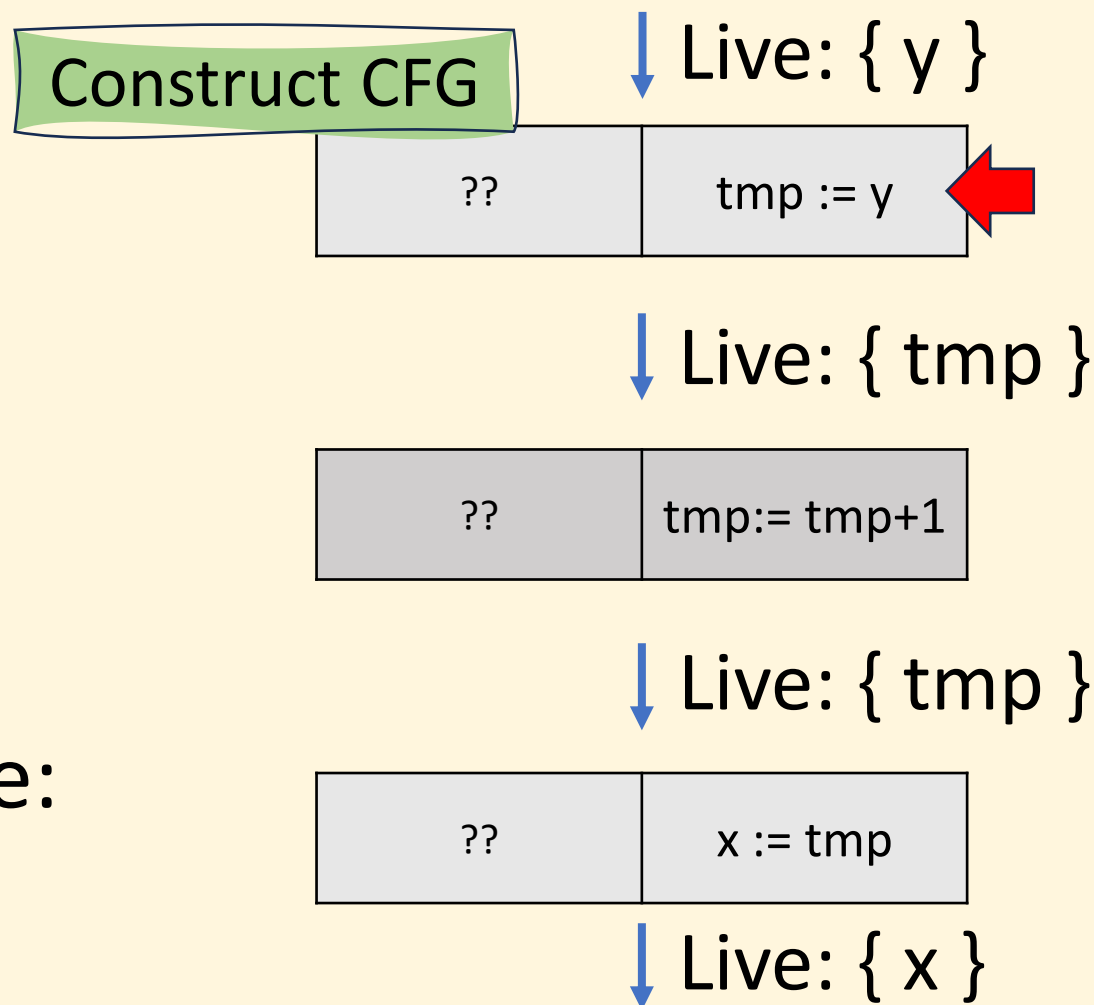
[rbp-8]

Define Data Liveness at Edges (2)

- Consider a motivating example:

`int x = y + 1`
(`y` never used again)

- So both `x` and `y` can use:
[rbp-8]



Note the addition of “temp” variables. This is a part of Dataflow Analysis.

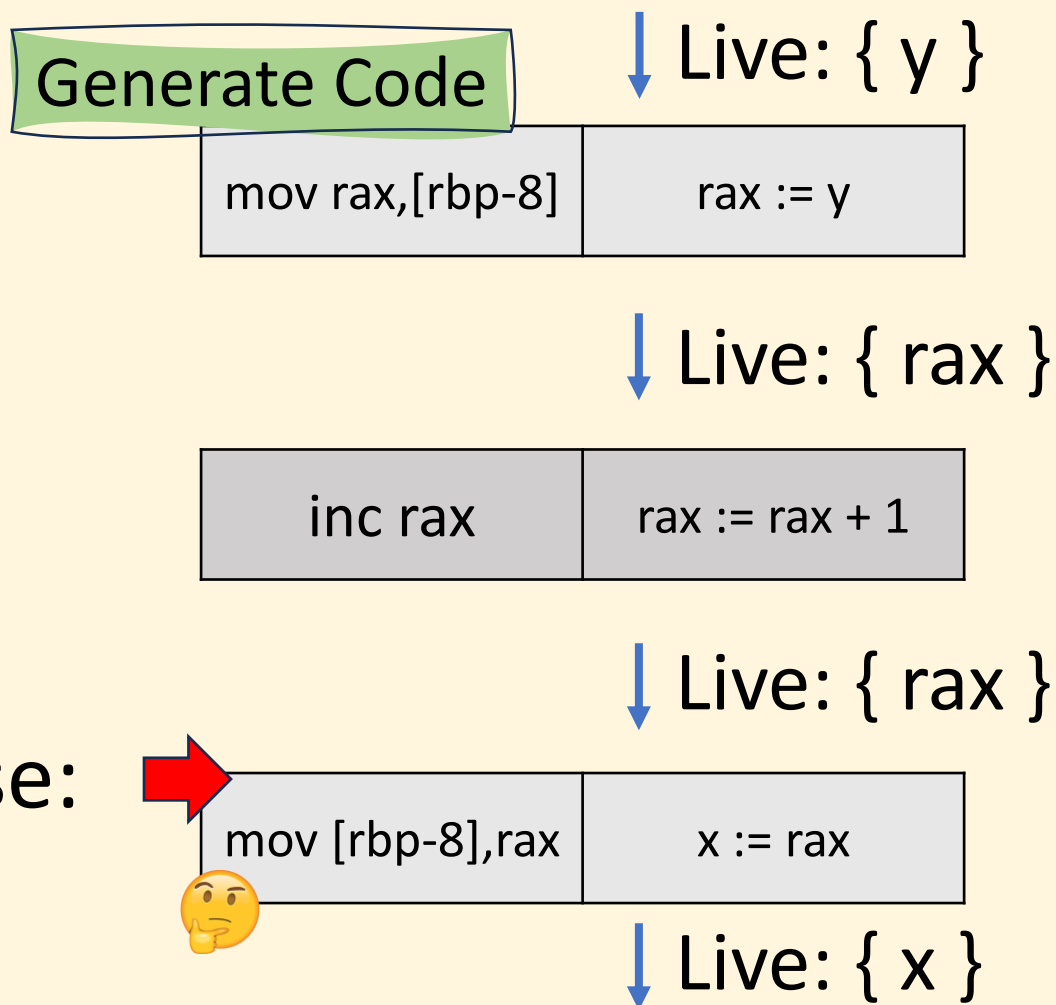
Define Data Liveness at Edges (3)

- Consider a motivating example:

`int x = y + 1`

(`y` never used again)

- So both `x` and `y` will use:
`[rbp-8]`



Temporary
variables
will turn
into
registers
eventually.

Can visually see it, but how can we detect such optimizations?

mov rax,[rbp-8]	rax := y
inc rax	rax := rax + 1
mov [rbp-8],rax	x := rax

 \equiv

inc [rbp-8]	x := y + 1
-------------	------------

...We will need more tools!

Don't worry about such optimizations until you are done with PA4

More Definitions

Set: $\text{use}(v)$

- $\text{use}(v) \equiv$ The set of variables used by vertex v .
- E.g. $v \equiv "z = x + y"$
 - $\text{use}(v) = \{ x, y \}$
 - $\text{def}(v) = \{ z \}$

Set: $\text{def}(v)$

- $\text{def}(v) \equiv$ The set of variables that are defined by vertex v .
- Somewhat of a misnomer, it is variables whose values are **assigned** by the vertex v .
- E.g. $v \equiv "z = z * 2"$
 - $\text{def}(v) = \{ z \}$
 - $\text{use}(v) = \{ z \}$

$\forall v : v \in V ::$

Constraints

$$\text{use}(v) \subseteq \text{in}(v)$$

- Why?

$$\text{out}(v) \setminus \text{def}(v) \subseteq \text{in}(v)$$

- Why?

$$\forall s : s \in \text{successor}(v) :: \text{in}(s) \subseteq \text{out}(v)$$

- Why?

$\forall v : v \in V ::$

Constraints (2)

$$I0 \equiv \text{use}(v) \subseteq \text{in}(v)$$

- If we use the variable, it was live before the vertex is entered.

$$I1 \equiv \text{out}(v) \setminus \text{def}(v) \subseteq \text{in}(v)$$

- If a variable that we didn't assign is live after v , then it was live when we enter v .

$$I2 \equiv \forall s : s \in \text{successor}(v) :: \text{in}(s) \subseteq \text{out}(v)$$

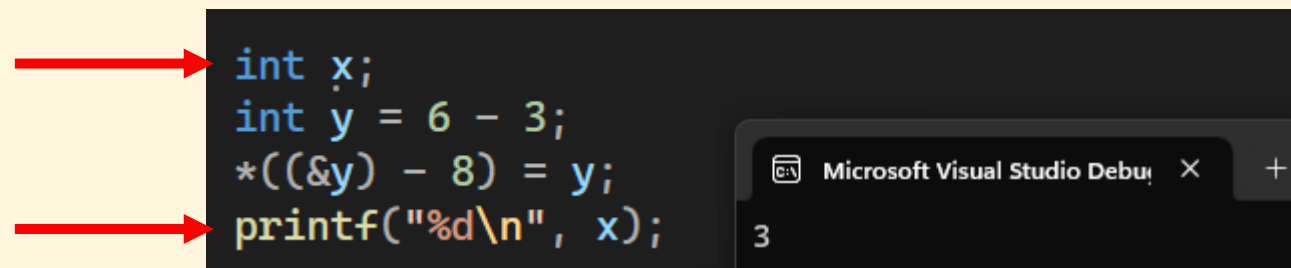
- If a variable is live when entering a successor, then it must be live when exiting the vertex.

$\forall v : v \in V ::$

Other Languages

$$lO \equiv \text{use}(v) \subseteq \text{in}(v)$$

- If we use the variable, it was live before the vertex is entered.
- Not always possible to determine in other languages
- Compile-time error in Java (save for PA5) because x is uninitialized.



```
int x;  
int y = 6 - 3;  
*((&y) - 8) = y;  
printf("%d\n", x);
```

The screenshot shows a code editor window titled "Microsoft Visual Studio Debug" with a tab icon and a close button. The code is in C and contains four lines. Two red arrows point to the first and fourth lines. The first line is `int x;` and the fourth line is `printf("%d\n", x);`. The second line is `int y = 6 - 3;` and the third line is `*((&y) - 8) = y;`. The variable `x` is used in the `printf` statement but has not been assigned a value before, which would cause a compile-time error in Java.

$\forall v : v \in V ::$

Goal

$$I0 \equiv \text{use}(v) \subseteq \text{in}(v)$$

$$I1 \equiv \text{out}(v) \setminus \text{def}(v) \subseteq \text{in}(v)$$

$$I2 \equiv \forall s : s \in \text{successor}(v) :: \text{in}(s) \subseteq \text{out}(v)$$

Can actually use these constraints to our advantage!



Iterative Data Liveness Analysis

Initialization (Base Case)

- Start: $G = (E, V)$
- Initialize:
 - $\forall v : v \in V :: \text{in}(v) := \emptyset$
 - $\forall v : v \in V :: \text{out}(v) := \emptyset$
 - $\forall v : v \in V :: \text{Determine } \text{def}(v), \text{use}(v)$
- Note: constraints are probably not yet satisfied.

Iterative Step

- Evaluate:
- $\text{out}(\textcolor{red}{v}) := \bigcup_{s \in \text{successor}(\textcolor{red}{v})} \text{in}(\textcolor{blue}{s})$
 - What is this doing?
- $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$
 - What is this doing?

Iterative Step (2)

- Evaluate in-order:
- $\text{out}(v) := \bigcup_{s \in \text{successor}(v)} \text{in}(s)$
 - If a successor needs a live variable, then it must be live when exiting v
- $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$
 - What is this doing?

Iterative Step (3)

- $\text{out}(v) := \bigcup_{s \in \text{successor}(v)} \text{in}(s)$
- $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$
 - What is this doing?

Iterative Step (4)

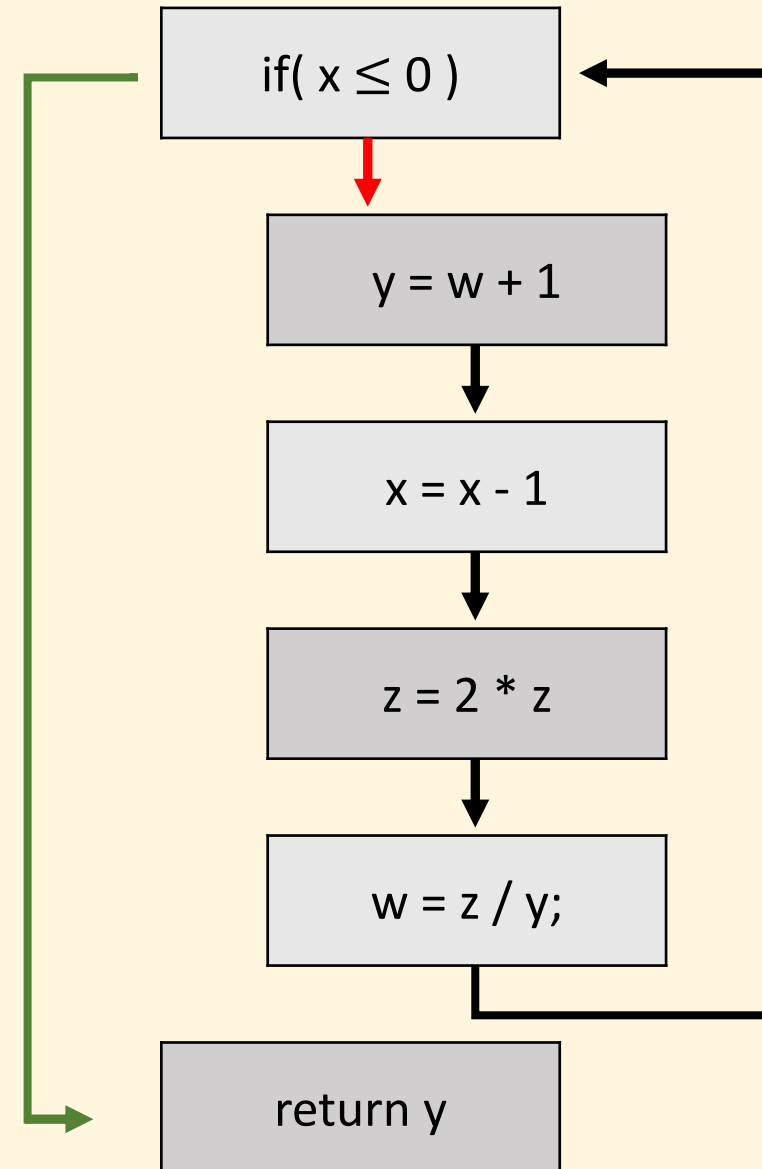
- $\text{out}(v) := \bigcup_{s \in \text{successor}(v)} \text{in}(s)$
- $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$
 - If v uses the variable, it must be live upon entry
 - Union with: variables that must be live afterwards, except the variables that are set by v .
 - We don't need such **assigned** variables live, unless we use their previous value.

Fixed-Point

- Stop when:
 - $I3 \equiv \text{All constraints met } (I0 \wedge I1 \wedge I2) \rightarrow \text{STOP}$
 - $I4 \equiv \text{All sets in/out do not change} \rightarrow \text{STOP}$
- When done by $I4$, $I3$ is too, so only check in/out sets.
 - Curious why? See COMP-735 (Spring 2025)
 - Use well-founded closure rule, eventually, $I4 \rightarrow I3$
 - Analyzing data liveness algorithms not a part of this class

Example:

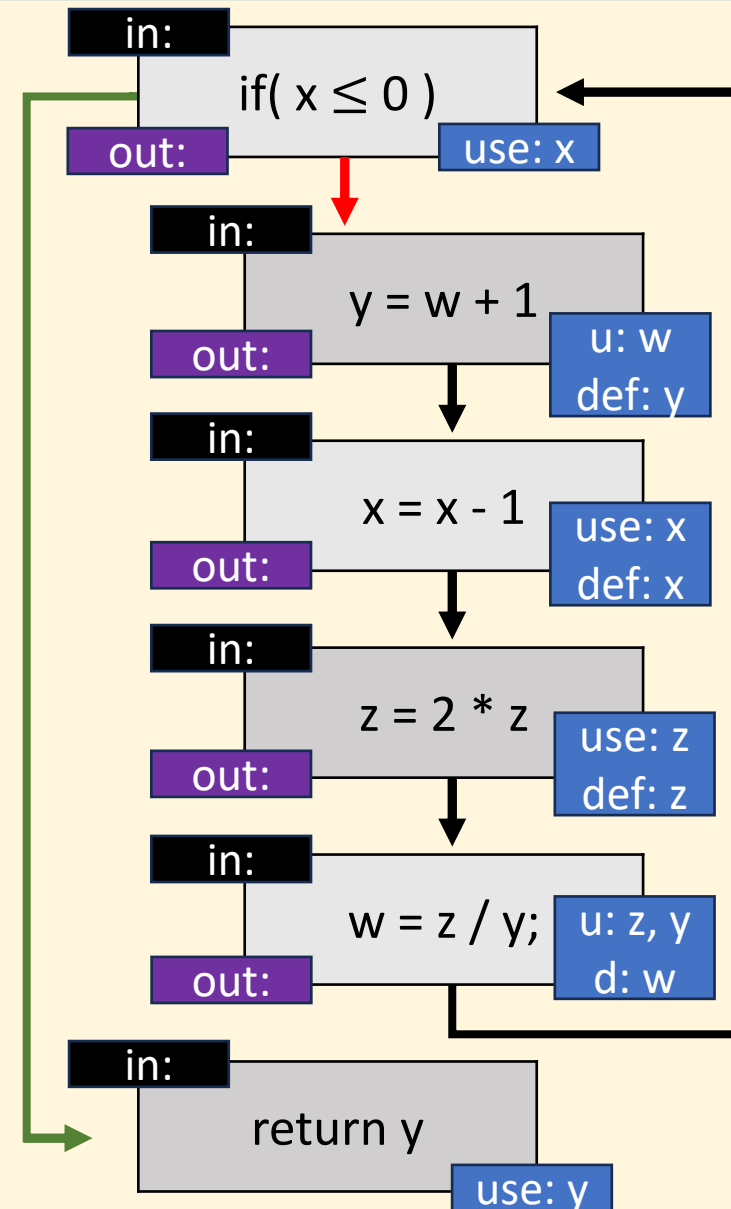
```
y = 0; x = 10; z = 2; w = 0;  
while( x > 0 ) {  
    y = w + 1;  
    x = x - 1;  
    z = 2 * z;  
    w = z / y;  
}  
return y;
```



Initialization.

Determine sets: use/def

Assign all in/out to \emptyset



Iteration 1

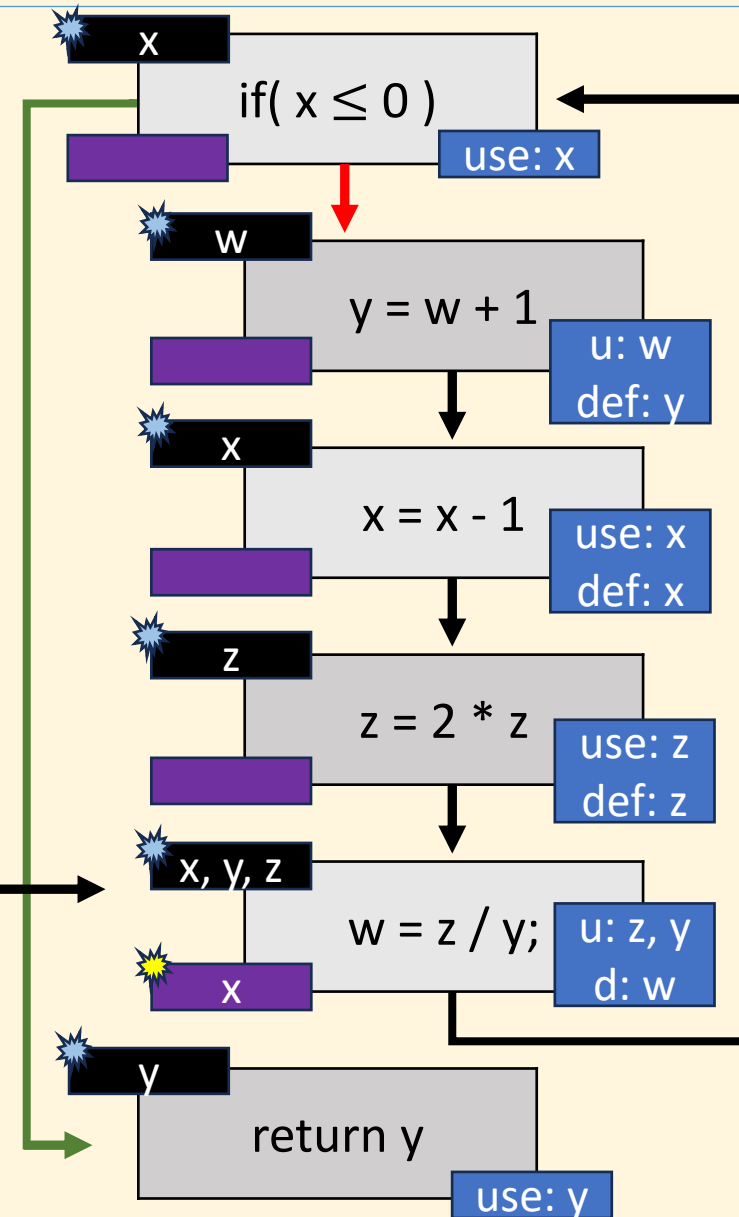
for($v \in V$) {

★ $\text{out}(v) := \bigcup s: \text{in}(s)$

★ $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$

}

If done in-order, then most of the first iteration is easy. Watch out for



Iteration 2

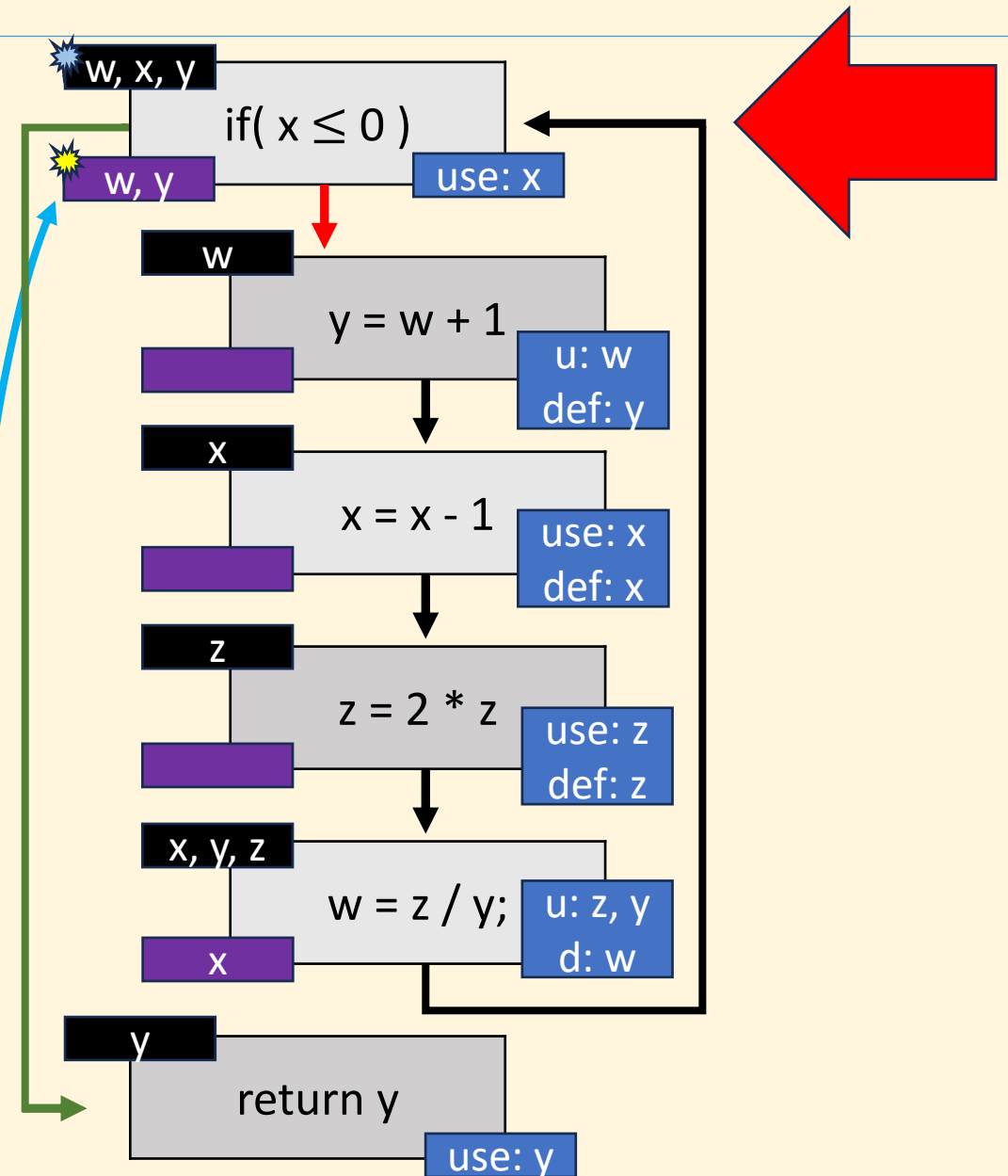
for($v \in V$) {

★ $\text{out}(v) := \bigcup s: \text{in}(s)$

★ $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$

}

Watch out for checking all successors:



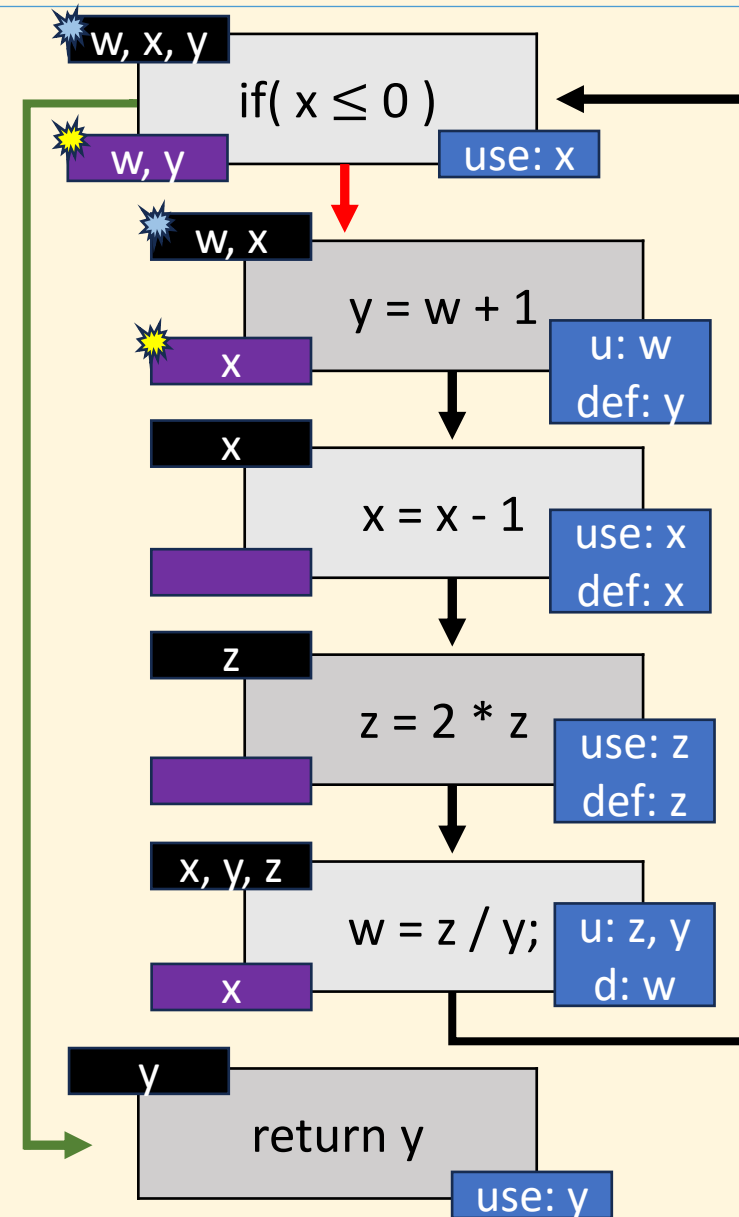
Iteration 2 (2)

for($v \in V$) {

★ $\text{out}(v) := \bigcup s :: \text{in}(s)$

★ $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$

}



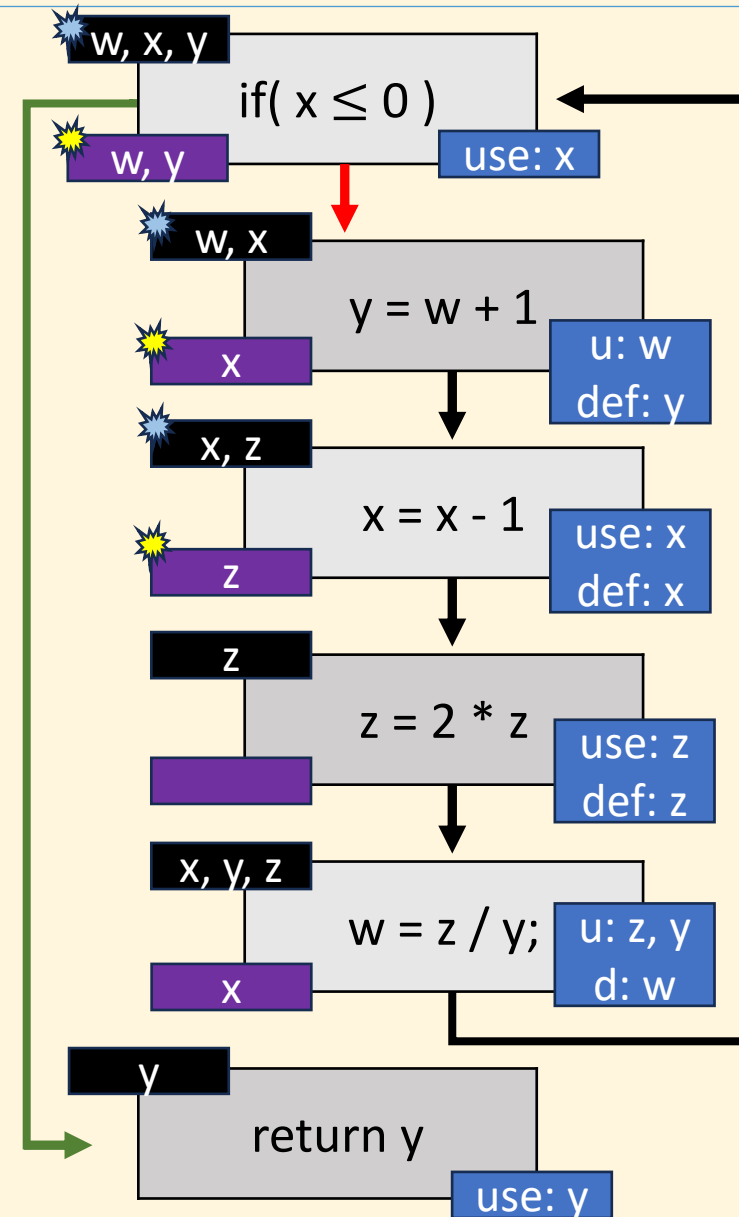
Iteration 2 (3)

for($v \in V$) {

★ $\text{out}(v) := \bigcup s: \text{in}(s)$

★ $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$

}



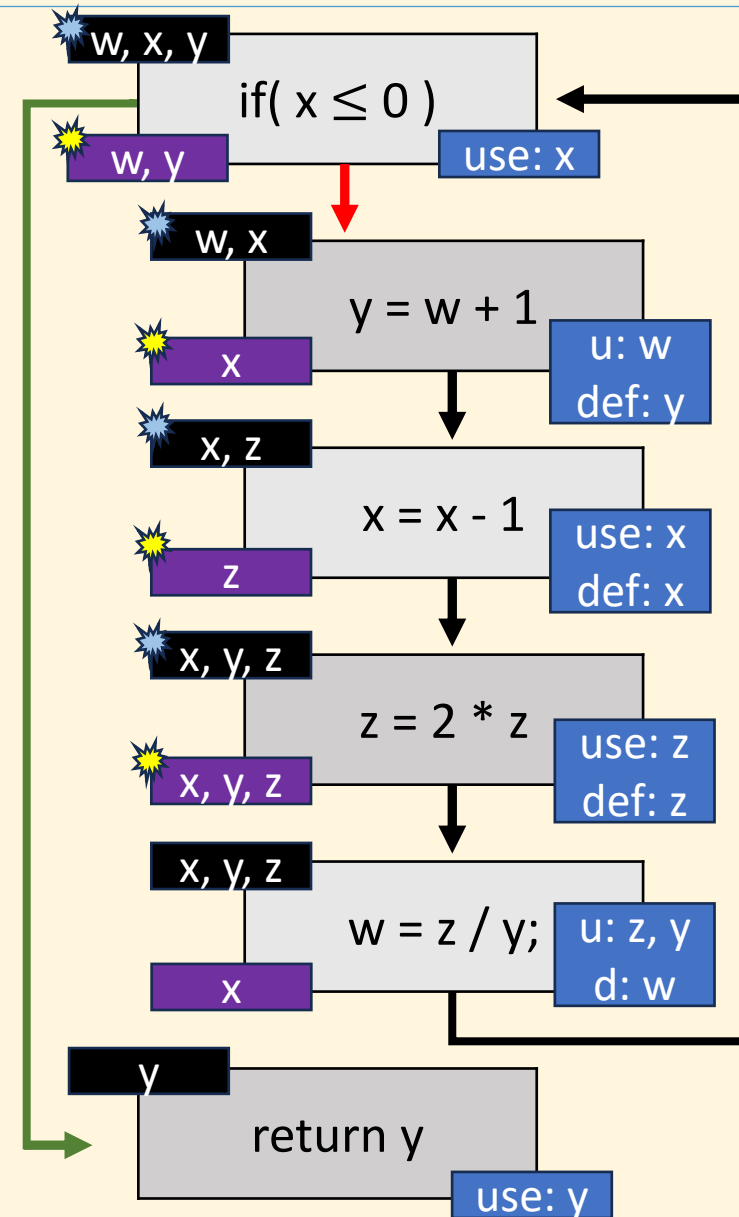
Iteration 2 (4)

for($v \in V$) {

★ $\text{out}(v) := \bigcup s: \text{in}(s)$

★ $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$

}



Iteration 2 (5)

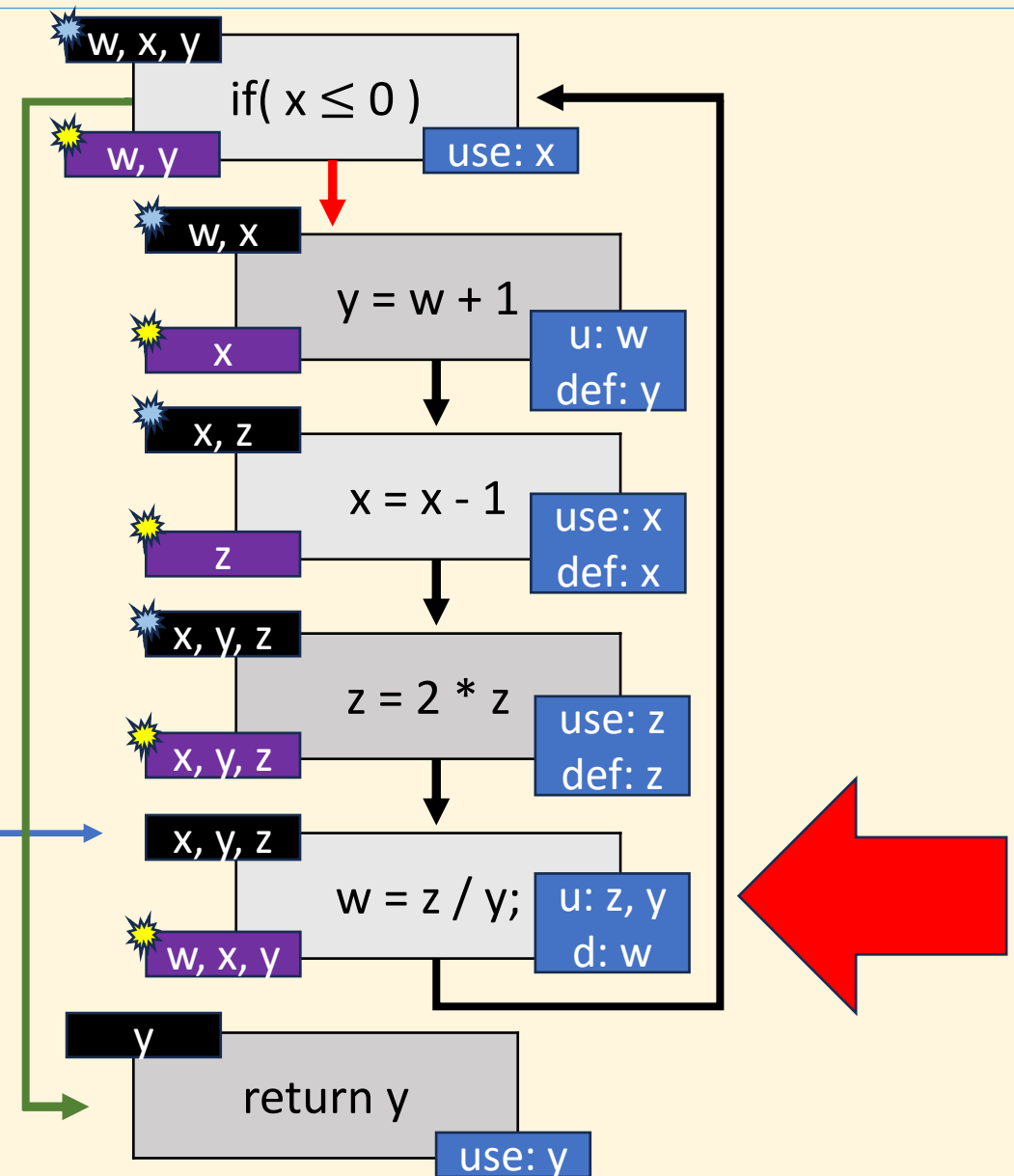
for($v \in V$) {

★ $\text{out}(v) := \bigcup s: \text{in}(s)$

★ $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$

}

Note: no change here



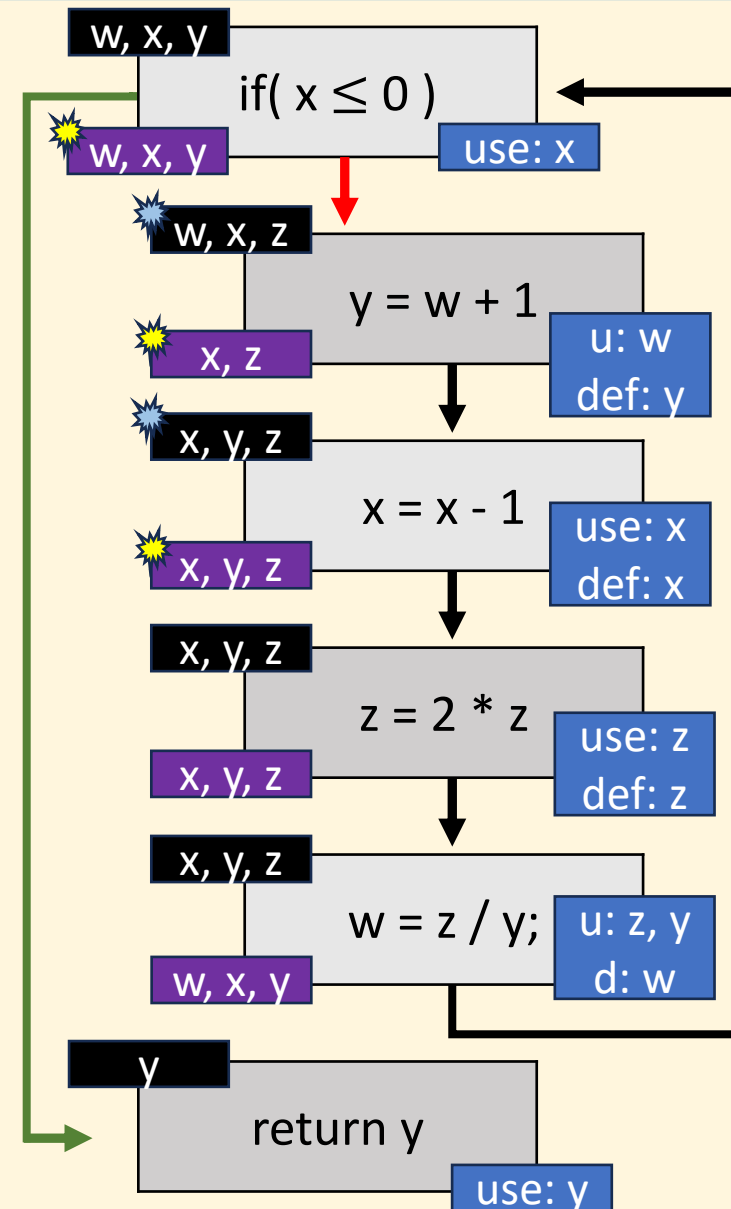
Iteration 3

for($v \in V$) {

★ $\text{out}(v) := \bigcup s: \text{in}(s)$

★ $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$

}



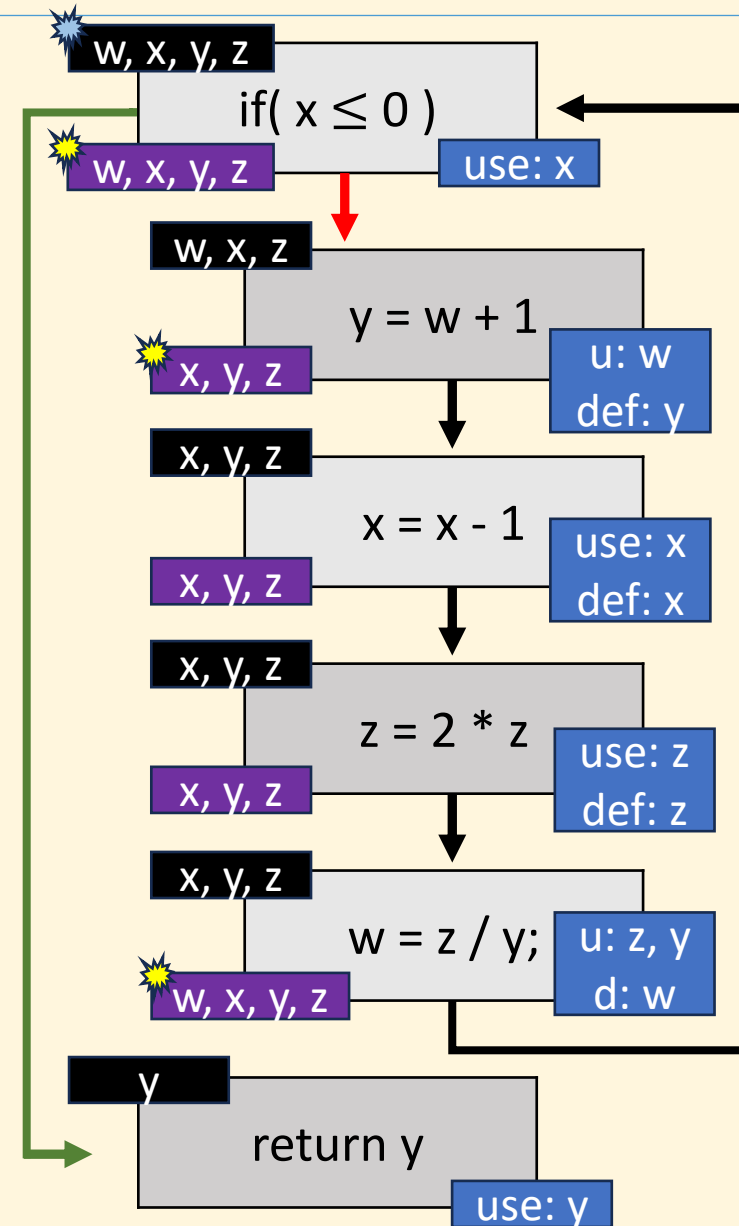
Iteration 4

for($v \in V$) {

★ $\text{out}(v) := \bigcup s: \text{in}(s)$

★ $\text{in}(v) := \text{use}(v) \cup (\text{out}(v) \setminus \text{def}(v))$

}

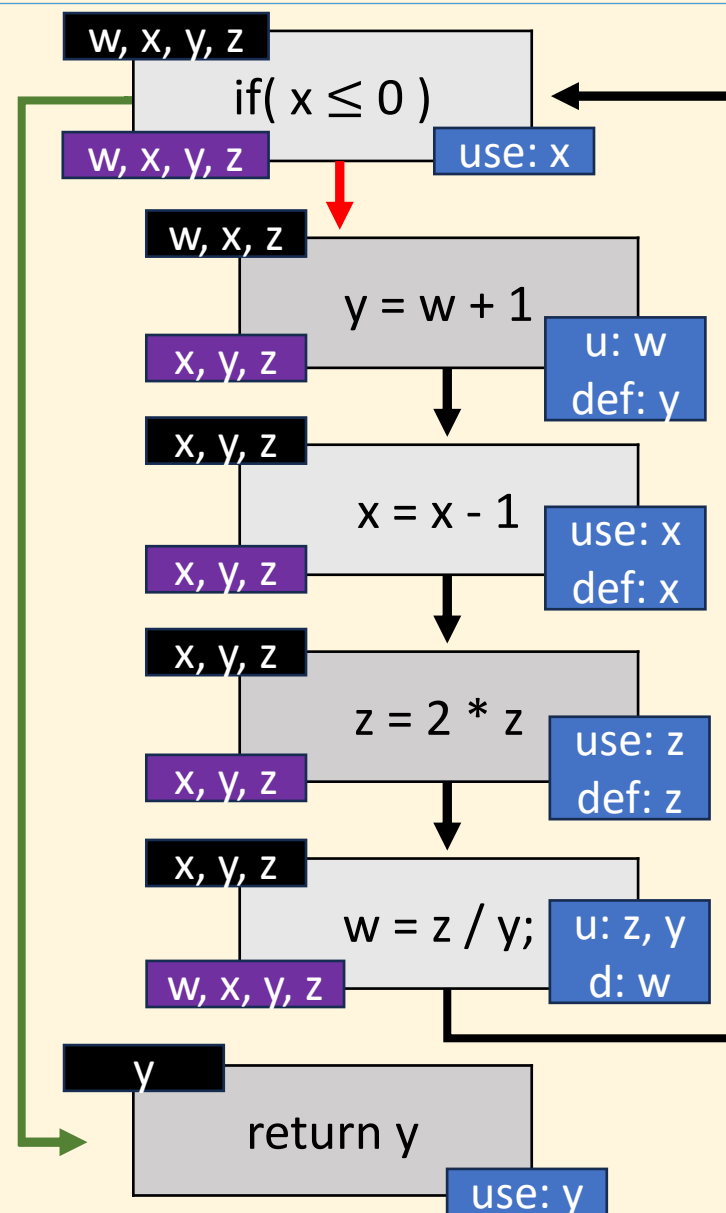


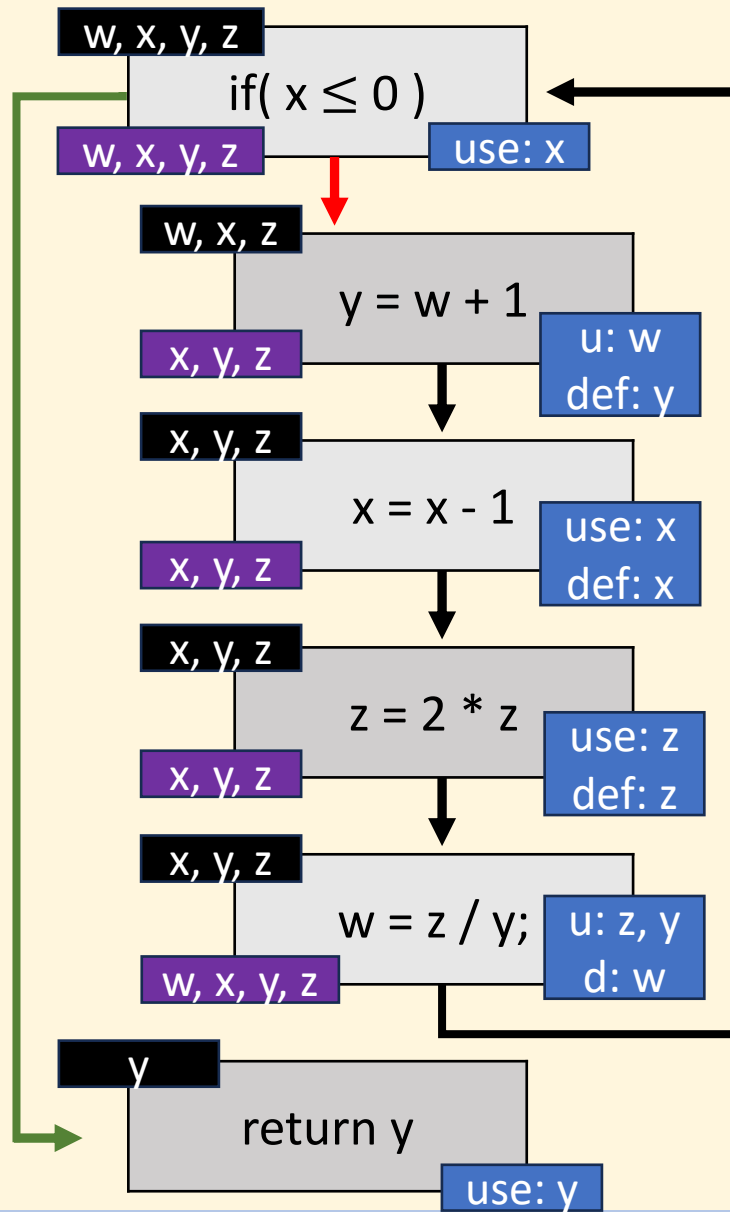
Iteration 5

```
for(  $v \in V$  ) {
    out( $v$ ) :=  $\bigcup s :: \text{in}(s)$ 
    in( $v$ ) := use( $v$ )  $\cup$  ( $\text{out}(v) \setminus \text{def}(v)$ )
}
```

Fixed-Point Reached!

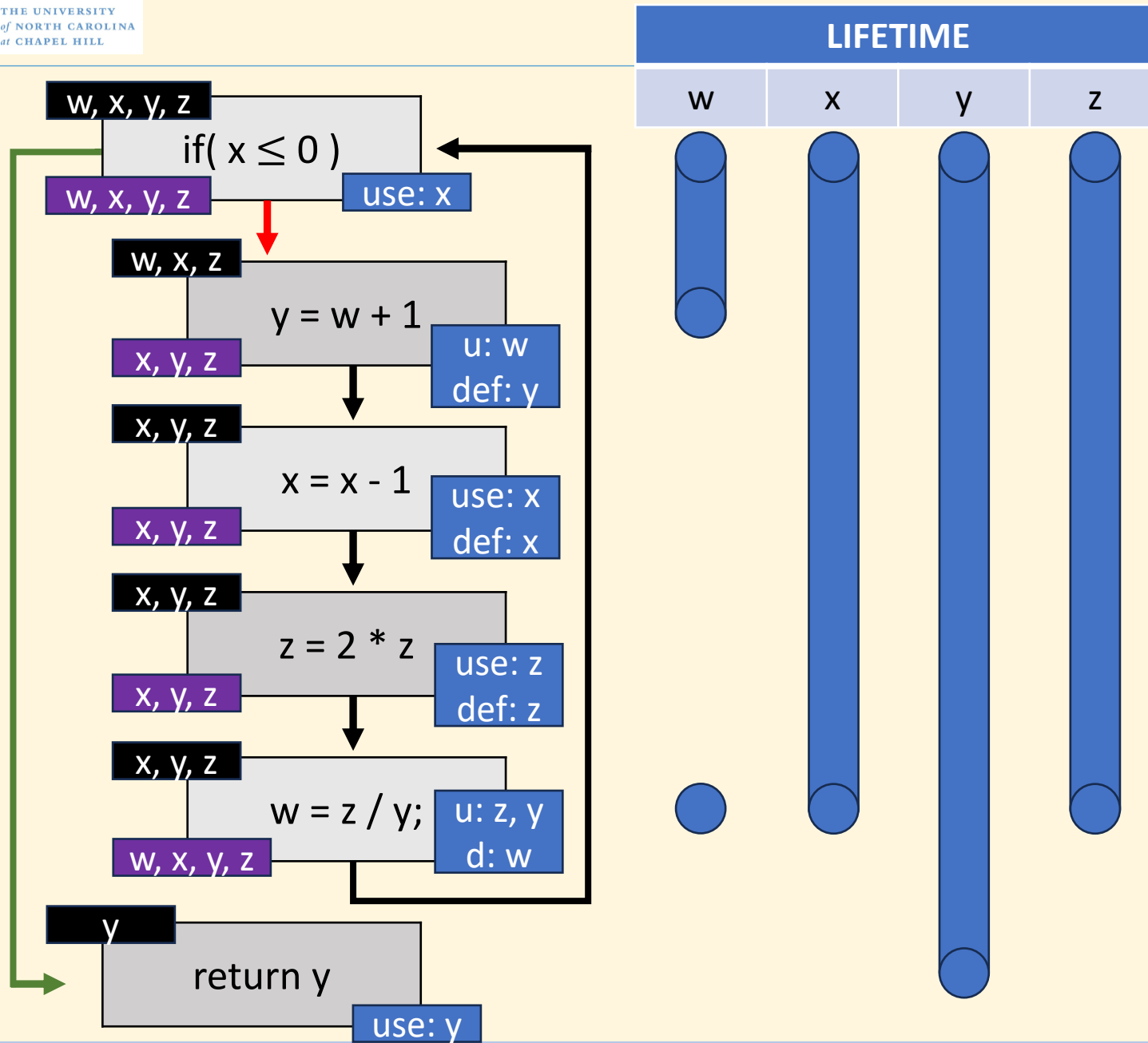
Oh goodness, what a beautiful thing we have evaluated!





Observation 1.

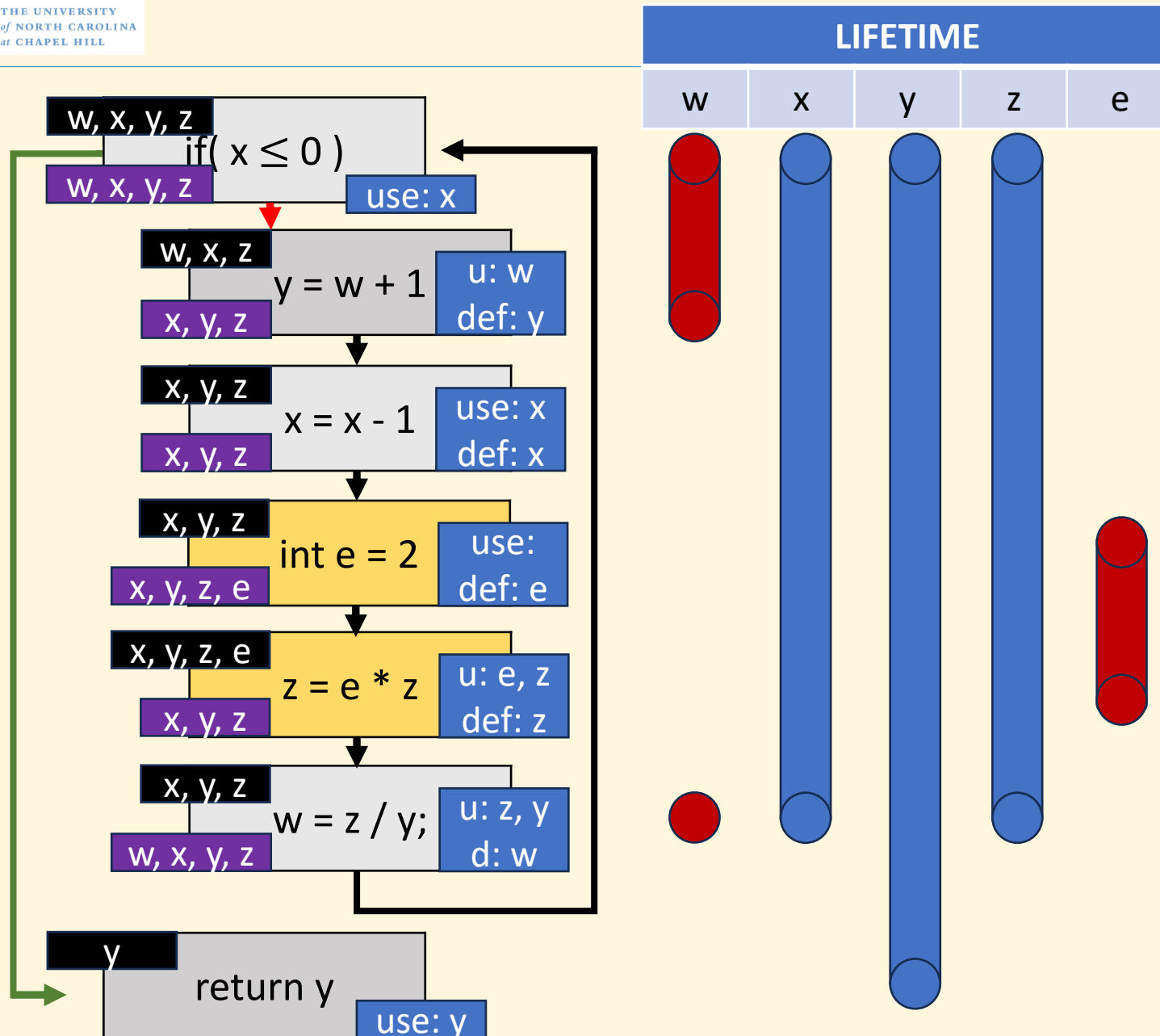
- Parts of the code exist where variable “w” is not needed



Shown:

Data lifetime Graph

No lifetimes are disjoint.



Consider the modifications in yellow.

Note the disjoint lifetime of `e` and `w`.

This is a similar, earlier problem

```
y = 0; x = 10; z = 2; w = 0;  y = 0; x = 10; z = 2; w = 0;
while( x > 0 ) {              while( x > 0 ) {
    y = w + 1;                 y = w + 1;
    x = x - 1;                 x = x - 1;
    int e = 2;                 w = 2;
    z = e * z;                 z = w * z;
    w = z / y;                 w = z / y;
}                               }
return y;                     return y;
```

The diagram illustrates an equivalence between two code blocks. A blue double-headed arrow labeled "EQUIV." connects the two blocks. The left block contains the code snippet within a green box, and the right block contains the code snippet within a green box. A black recycling symbol is present in the right block, indicating a loop or repetition.

How many variables are needed?

Variables: 5

Concurrently Alive: 4

```
y = 0; x = 10; z = 2; w = 0;
while( x > 0 ) {
    y = w + 1;
    x = x - 1;
    int e = 2;
    z = e * z;
    w = z / y;
}
return y;
```

EQUIV.

Variables: 4

Concurrently Alive: 4

```
y = 0; x = 10; z = 2; w = 0;
while( x > 0 ) {
    y = w + 1;
    x = x - 1;
    w = 2;
    z = w * z;
    w = z / y;
}
return y;
```

Primary Observation

Variables with disjoint lifetimes can utilize the same memory space.

- Thus, the code in the earlier example can be done using 4 registers.
- Question: can we keep everything in registers and commit w , x , y , z after the loop ends?

```
y = 0; x = 10;  
z = 2; w = 0;  
while( x > 0 ) {  
    y = w + 1;  
    x = x - 1;  
    int e = 2;  
    z = e * z;  
    w = z / y;  
}  
return y;
```

Side Observation - Multithreaded

- Question: can we keep everything in registers and commit w , x , y , z after the loop ends?
- Only if your target machine has no concurrent threads accessing the memory of w , x , y , z (e is a local, but the other variables could have been global)
- Multi-threaded dataflow analysis is possible. Very helpful when using OpenMP/CUDA/barriers/fences. Optimization is huge.
 - Seen in COMP-735, but in the context of program states and transactions. Also in COMP-633, but not sure when it is offered next.

Need implicit or explicit commit points for proper analysis



Available Expressions / Expression Lifetime Analysis

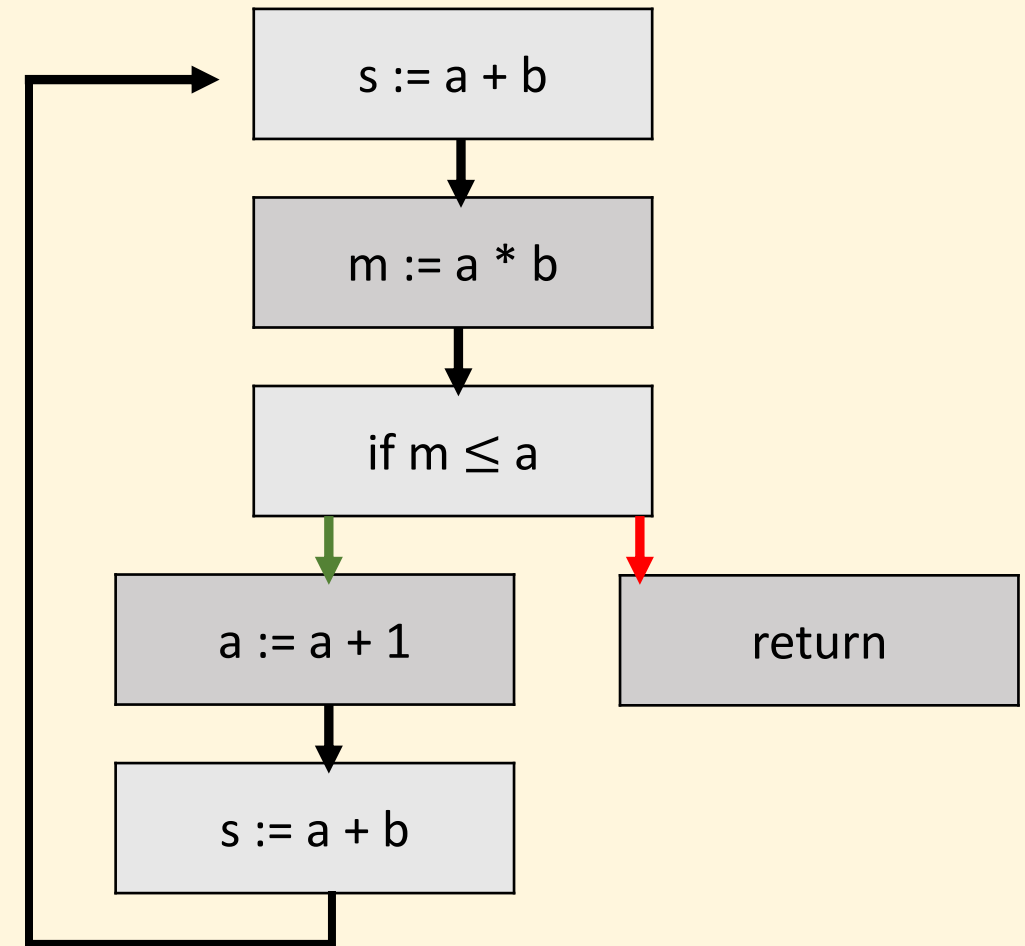
Can also apply lifetime analysis to expressions, not just variables.

Consider the following code:

```
s := a + b;  
m := a * b;  
while( m > a ) {  
    a := a + 1;  
    s := a + b;  
}
```

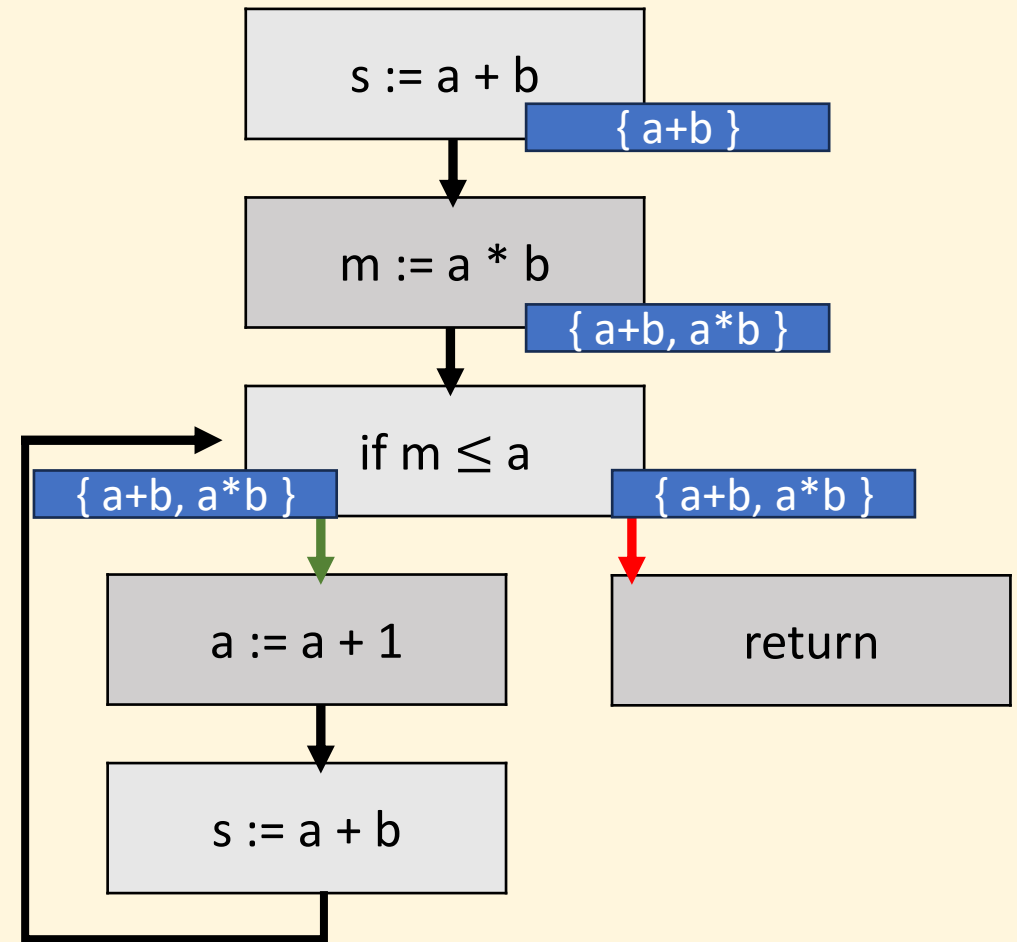
Construct the CFG

```
s := a + b;  
m := a * b;  
while( m > a ) {  
    a := a + 1;  
    s := a + b;  
}
```



When data is *invalidated*, so are all expressions utilizing that data.

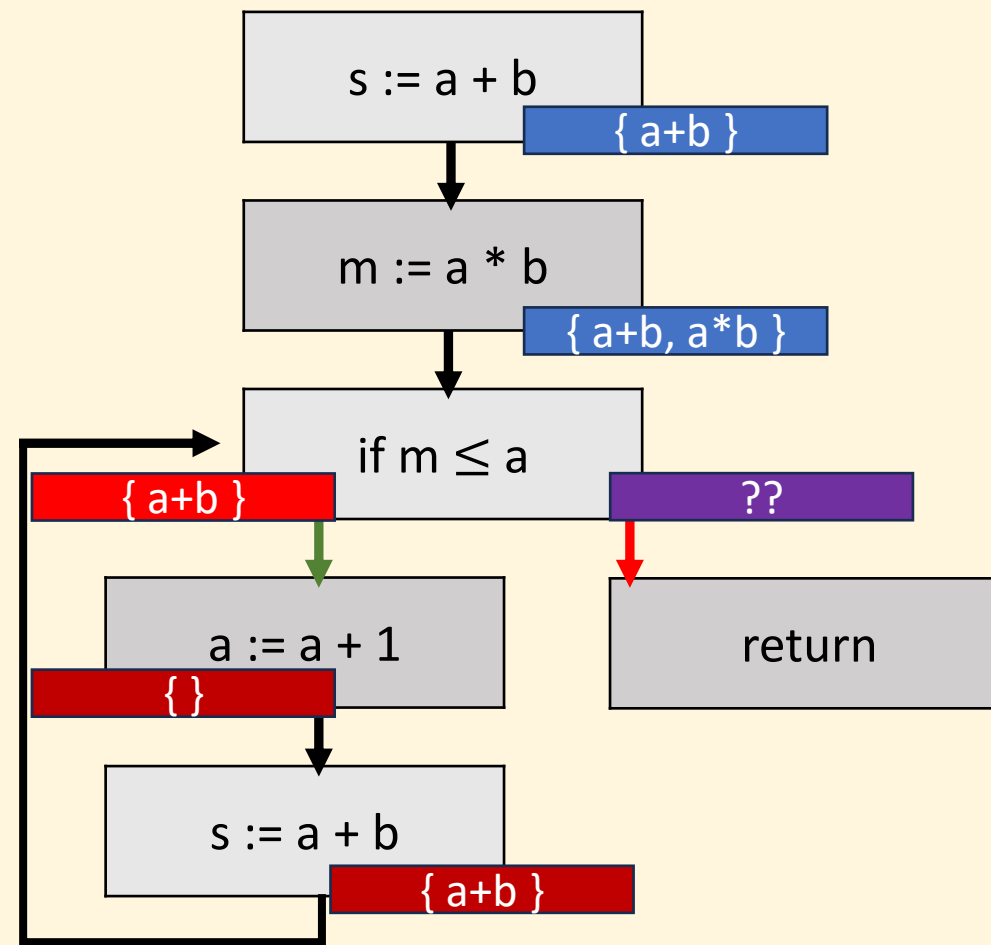
```
s := a + b;  
m := a * b;  
while( m > a ) {  
    a := a + 1;  
    s := a + b;  
}
```



When data is *invalidated*, so are all expressions utilizing that data.

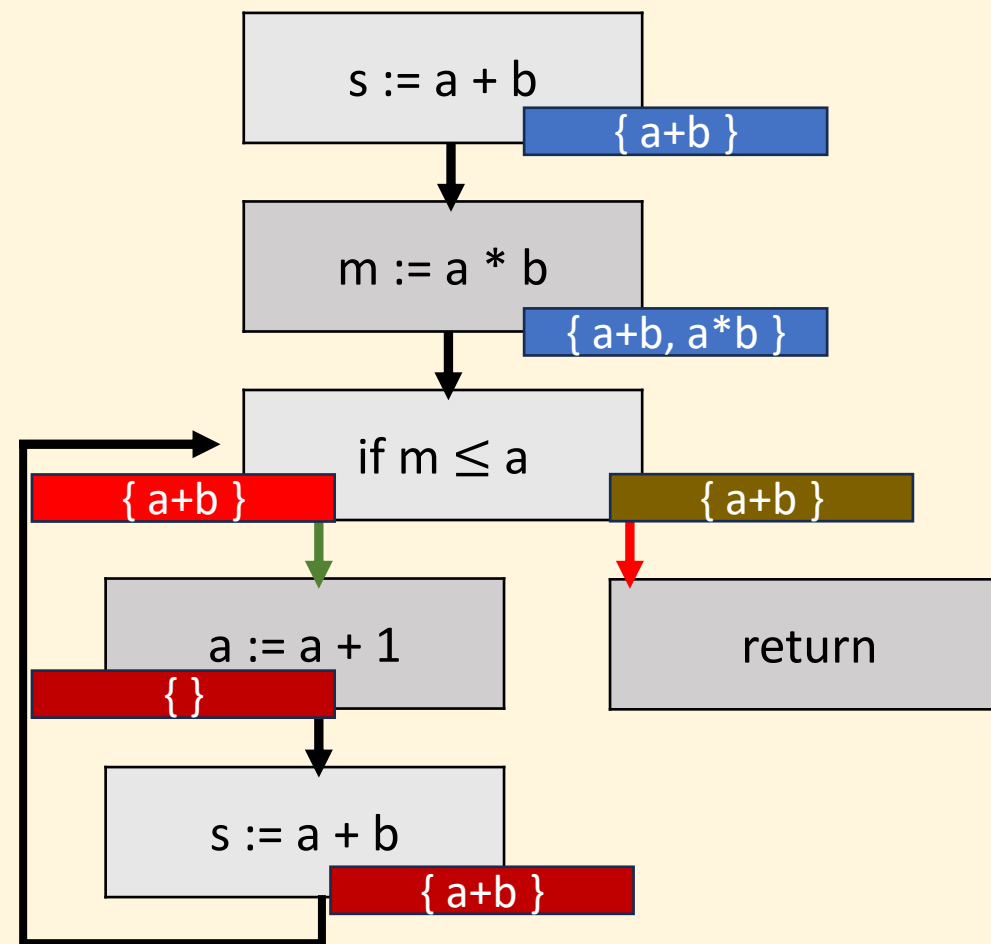
```
s := a + b;  
m := a * b;  
while( m > a ) {  
    a := a + 1;  
    s := a + b;  
}
```

Note: we lost $a*b$ here:



When data is *invalidated*, so are all expressions utilizing that data.

```
s := a + b;  
m := a * b;  
while( m > a ) {  
    a := a + 1;  
    s := a + b;  
}
```

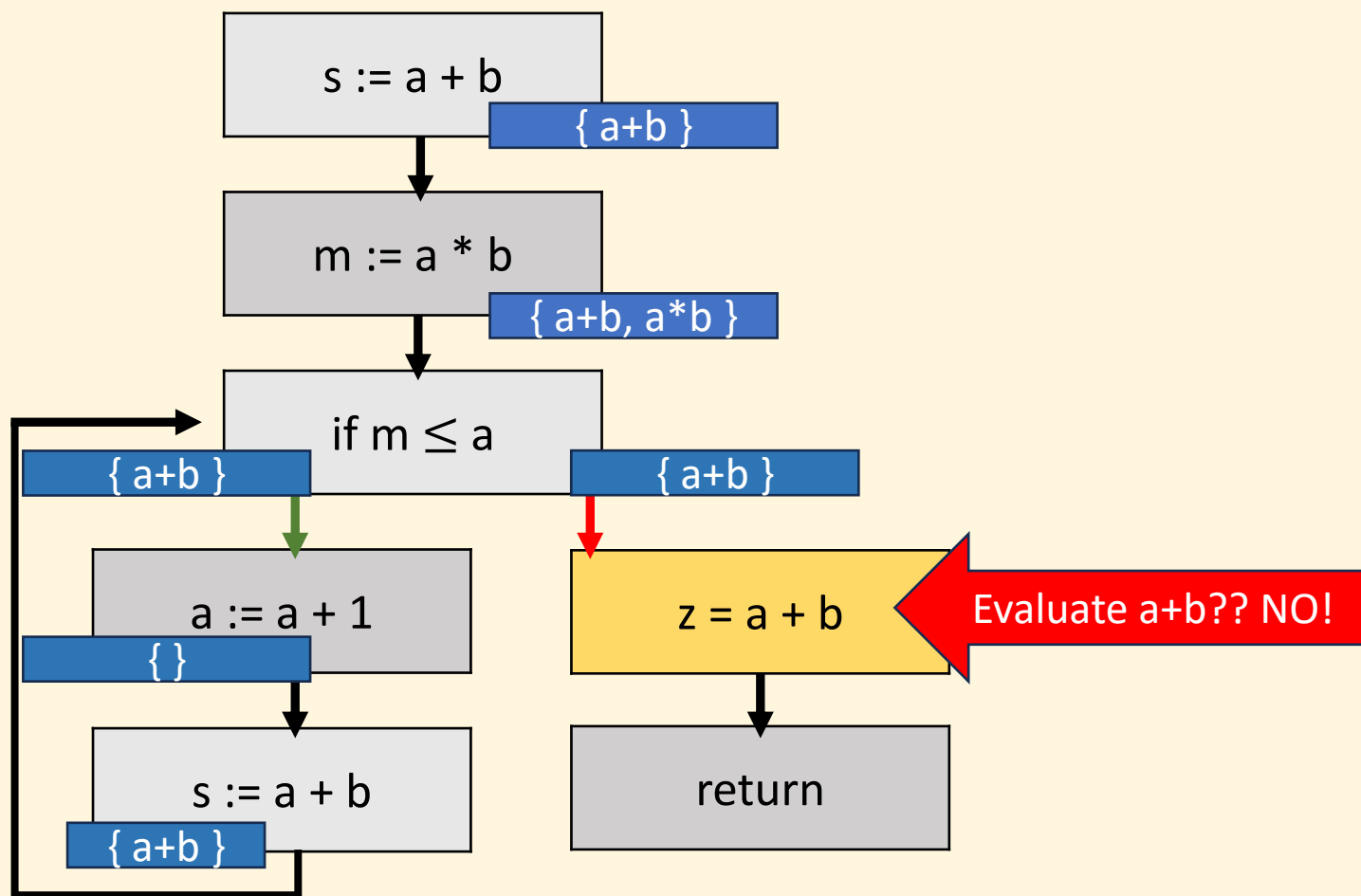


Expression Liveness

- Very useful so that an expression does not have to be re-evaluated.
- Let's look at that example earlier with one minor modification.

No need to re-evaluate **a+b**, because **s** is an alias.

```
s := a + b;  
m := a * b;  
while( m > a ) {  
    a := a + 1;  
    s := a + b;  
}  
z := a + b;
```



Another Description: Data Liveness

- Each vertex generates some “facts”
- Each vertex invalidates some “facts”
- Data Liveness:
 - $\text{gen}_d(v) = \text{use}(v)$
 - $\text{kill}_d(v) = \text{def}(v)$
 - $\text{out}_d(v) = \bigcup_{s \in (\dots)} \text{in}_d(s)$
 - $\text{in}_d(v) = \text{gen}_d(v) \cup (\text{out}_d(v) \setminus \text{kill}(v))$

Formal Description: Expression Liveness

- Each vertex generates some “facts”
- Each vertex invalidates some “facts”
- Expression Liveness:
 - $\text{gen}_e(v)$ = expressions evaluated
 - $\text{kill}_e(v)$ = all expressions that contain $\text{def}(v)$
 - $\text{in}_e(v) = \bigcap_{p \in \text{predecessor}(v)} \text{out}_e(p)$
 - $\text{out}_e(v) = \text{gen}_e(v) \cup (\text{in}_e(v) \setminus \text{kill}_e(v))$

Termination in “Expression Liveness”

- Only re-evaluate vertices when a predecessor has a change in the *out* set.
- Will eventually reach a fixed-point.

Not so simple...

- Problem: what about more complex expressions:

$$(x + y) == (z + w)$$

- We can keep many expressions alive:
 - $x + y, z + w$
 - $(x + y) == (z + w)$
 - Can keep not x, y alive, but instead keep $\alpha = x + y$ alive
 - $\alpha == (z + w)$
 - Etc.

Idea: Break up vertices

- Break every expression into small constituent components. **Generate extra code!**

$$“(x + y) == (z + w)” \Rightarrow \{ x+y, z+w \}$$

Original	Generate Code	
<code>c := (x+y) == (z+w)</code>	<code>a := x+y</code>	
	<code>b := z+w</code>	
<code>d := z+w</code>	<code>c := (x+y)==(z+w)</code>	
	<code>d := z+w</code>	
<code>return c+d</code>		

Apply Expression Liveness Analysis

- Replace expressions with aliased expressions

Original	Generate Code	Apply Aliases
c := (x+y) == (z+w)	a := x+y	a := x+y
	b := z+w	b := z+w
d := z+w	c := (x+y)==(z+w)	c := a==b
	d := z+w	d := b
return c+d		

Apply Data Liveness Analysis

- Reuse variable names

Original	Generate Code	Apply Aliases	x	y	z	w	a	b	c	d	New Data Aliases
c := (x+y) == (z+w)	a := x+y	a := x+y	■	■	■	■	■				
	b := z+w	b := z+w			■	■	■	■			
d := z+w	c := (x+y)==(z+w)	c := a==b					■	■	■		
	d := z+w	d := b						■	■	■	
return c+d									■	■	

Apply Data Liveness Analysis

- Can eliminate redundant operations

Original	Generate Code	Apply Aliases	x	y	z	w	a	b	c	d	New Data Aliases
c := (x+y) == (z+w)	a := x+y	a := x+y	x	y							x := x+y
	b := z+w	b := z+w			z	w					y := z+w
d := z+w	c := (x+y)==(z+w)	c := a==b					x				x := x==y
	d := z+w	d := b						y			y := y
return c+d									x	y	x := x+y ret x

Review

- **Data Liveness Analysis:**

- Reduces the amount of data you need in memory at any given time
- Somewhat related to minimizing register usage (minimizing registers can be done after data+expression liveness)

- **Expression Liveness Analysis:**

- Can eliminate the need to re-process expressions

- **Combined:**

- They can eliminate instructions and reduce memory consumption.

More Optimization?

Statements	# live
$x := x + y$	4 (x,y,z,w)
$y := z + w$	4 (x,y,z,w)
$x := x == y$	2 (x,y)
$x := x + y$	2 (x,y)
ret x	1 (x)

Does that mean we
need 4 registers?

More Optimization?

Statements	# live
$x := x + y$	4 (x,y,z,w)
$y := z + w$	4 (x,y,z,w)
$x := x == y$	2 (x,y)
$x := x + y$	2 (x,y)
ret x	1 (x)

Does that mean we need 4 registers?

Nope! More optimization possible that will be related to the target architecture.

Register Minimalization is not Dataflow/Expression Analysis

Statements	# live	X64	# live
x := x+y	4 (x,y,z,w)	mov rax,[x]	1 (rax)
		add rax,[y]	1 (rax)
y := z+w	4 (x,y,z,w)	mov rcx,[z]	2 (rax,rcx)
		add rcx,[w]	2 (rax,rcx)
x := x==y	2 (x,y)	cmp rax,rcx	2 (rax,rcx)
		xor rax,rax	2 (rax,rcx)
		sete al	2 (rax,rcx)
x := x+y	2 (x,y)	add rax,rcx	2 (rax,rcx)
ret x	1 (x)	ret	1 (rax)

Only needed two registers.

Why? Because x64 can do “load memory” operations inside of instructions!

See You Thursday!

- Intel C Compiler mini-case study.
 - Generating multiple code paths.
 - Rewriting user code to apply *exotic* optimizations.
-
- Remember, start PA4, some content from PA4 will be tested on Midterm 2.

End







