

Loops and Data-Flow Analysis

Introduction

CIS531 — Fall 2025, Syracuse

Kristopher Micinski



Loops: iterated control flow

- ◆ Loops come in many forms: while, do/while, for, etc.
- ◆ Can all be written in terms of one another...
- ◆ All loops generally share common aspects:
 - ◆ Header—evaluates the guard, yields true/false
 - ◆ Body—if the header is true, body executes
 - ◆ Conclusion—after the loop is done, jump to the loop's conclusion

```
// Standard while loop
// run until guard false
while (guard):
    body
```

```
// do block executes
// at least once
do {
    ...
} while guard;
```

```
// C's "for" loop
for (i = 0; i<n; i++) {
    ...
}

// for each in sequence..
for i : sequence {
    ...
}
```

Loops are the cornerstone of imperative programming

- ◆ Many common imperative algorithms use loops in some form
- ◆ Equivalent in expressive power to general recursion (>> than structural recursion)
 - ◆ Quicksort is not structurally recursive
 - ◆ (When translated to use recursion vs. loops)
- ◆ Loops can be very fast: branch prediction, etc.

```
// Insertion sort
i = 0
while (i < n):
    m = i
    for j = i+1 .. n-1:
        if a[j] < a[m]: m = j
    tmp = a[i]
    a[i] = a[m]
    a[m] = tmp
    i++
```

Racket's loops...

```
(for ([i (list 1 2 3)])  
  (println i))
```

Racket's for loop, iterate over lists...

```
(let go ([n 42] [sum 0])  
  (cond  
    [(= n 0) sum]  
    [(even? n) (go (sub1 n) (+ sum n))] ]  
    [else (go (sub1 n) sum)])))
```

“Named let” loop—can also think of as a small tail-recursive helper function

(Tail-recursive functions have the same stack behavior as loops!)

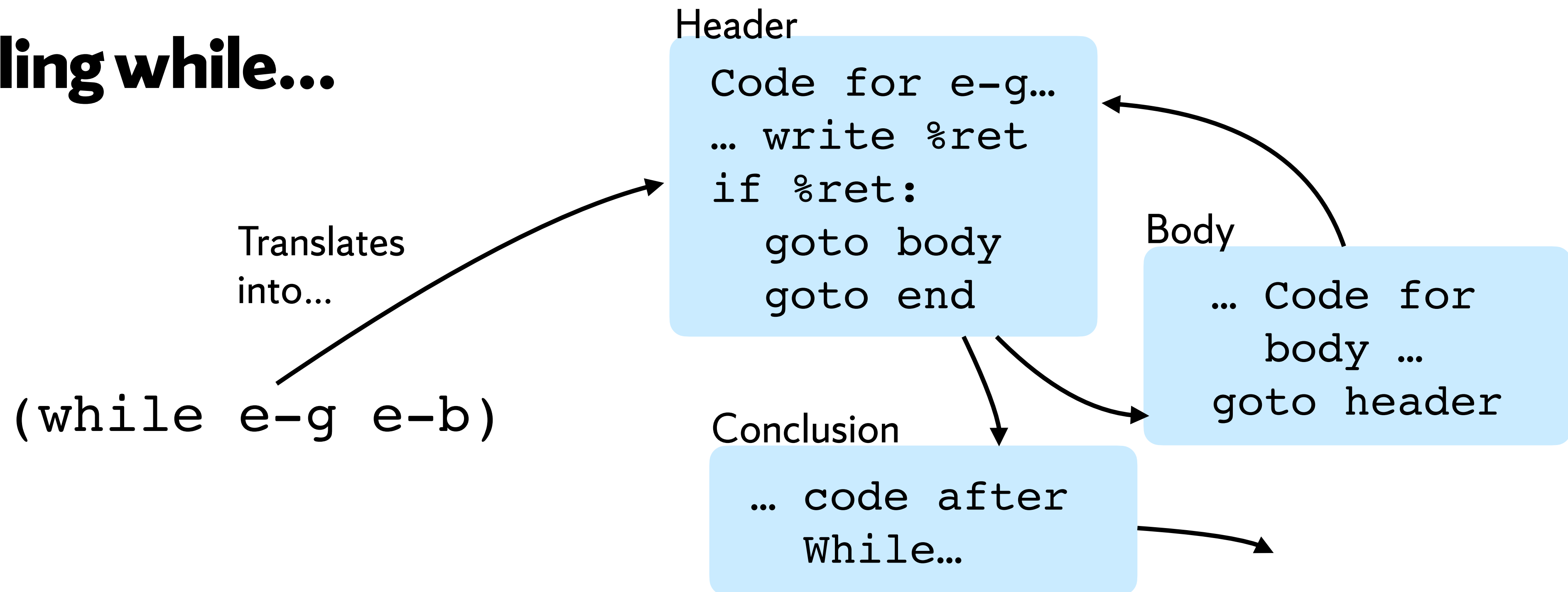
```
(for/fold ([int-list empty]  
          [square-vec (make-vector 3)]  
          [cube-hash (hash)])  
  ([i (list 1 2 3)])  
  (values (append int-list (list i))  
          (begin  
            (vector-set! square-vec (sub1 i) (* i i))  
            square-vec)  
          (hash-set cube-hash i (* i i i)))))
```

for/fold is very powerful and cool but I often forget it...

(**while** e-g e-b)

- ◆ Mostly won't cover Racket's loops; loops not a FP feature
- ◆ Instead, we will implement a traditional **while** loop construct
 - ◆ (`while e-guard e-body`)
- ◆ While the guard evaluates to true, then execute the body
- ◆ After the loop returns, the guard is definitely false
- ◆ Can compile every other form of “C-style” loop into `while` + `goto`
- ◆ We stick with `while` because it feels like the most canonical

Compiling while...



Basic idea, create three blocks:

- The *header*, which (a) tests the guard, (b) if true goto body, (c) else goto *conclusion*
- The *body*, which (a) executes the guard and then jumps back to the header
- The *conclusion* (a jump target of the header) continues execution
 - If nothing follows while, evaluate to (void)
 - Otherwise, `(while ...)` occurs as `(let ([_ (while ...)]) e-b)`, jump to e-b

- Not hard to implement this as an extension of our compiler...
- Need to be able to “catch” the value of the returned value
 - Previously (R2/Lif) all branches simply end in (return ...)
 - Now, we need to be able to take hold of the value generated by the compilation of the guard, and then use that to emit code that branches to the body/conclusion
 - This will require us to extend the **explicate-control** pass
- Otherwise, we simply add extra forms to support while
 - ANF conversion, etc. generally work out similarly
 - After explicate control, we’re in a “blocks” IR full of basic blocks w/ goto
 - Thus, supporting while requires ~no change to the backend

Loops give us Turing-equivalent computation!

- ◆ After understanding Lf/R2, loops are a simple extension
- ◆ However, loops unlock significantly more expressive power
 - ◆ Our language went from being: “boolean formulas over domains of size n ” to being “truly Turing-equivalent computation, capable of expressing any computable function”
- ◆ **This is a significant accomplishment**
 - ◆ However, we will need to discuss *functions* and *closures* (/ *objects*) to arrive at a “fully-featured” language—that will be our topic for next week

Static Analysis

- ◆ Now that we have loops, we're at the last basic building block that we need to start discussing **static analysis**
- ◆ Static analysis is the computation of finite approximations of a program's behavior based on looking at (and computing over) its code, without running it
- ◆ Why is it bad to run the program?
 - ◆ *Can't* run the program in general—state-space explosion!
 - ◆ Fuzzing is great at exploring, but generally doesn't go “deep” into programs
- ◆ Thus, static analysis is a *requirement* if you want to make sound judgments!
 - ◆ *Necessary* for compiler optimizations—absolutely **cannot** break the program!

Available Expressions Analysis

- ◆ Dataflow analysis: a specific kind of static analysis which approximates how data flows through the program.

- ◆ Consider this example on the right...

- ◆ The $b+c$ does not need recomputation

- ◆ b/c don't change!

- ◆ Could lift

- ◆ $a+1$ needs recomputation

- ◆ a got set to 0!

Optimize...

```
x = b + c
y = c * x
z = b + c
x = read_int()
y = b + c
```

```
i = b + c
x = i
y = c * x
z = i
x = read_int()
y = i
```

Available expressions analysis calculates:

What expressions have *stayed the same* since they were last computed

- ◆ **(Definition) Analysis Result for AE:** an analysis result for available expressions, associates with each program statement, stmt, two sets:
 - ◆ $\text{In}(\text{stmt})$ —dataflow facts coming into stmt and $\text{Out}(\text{stmt})$, going out of stmt
- ◆ Definitions (let/assignment/...) generate (GEN) newly-available expressions
- ◆ However, they also KILL expressions mentioning the variable
 - ◆ (Example): If x is changed, then $2 + y * x$ is no longer available
- ◆ The GEN/KILL sets are associated with each statement
- ◆ Sequences of statements work as expected; but the *joins* are interesting!

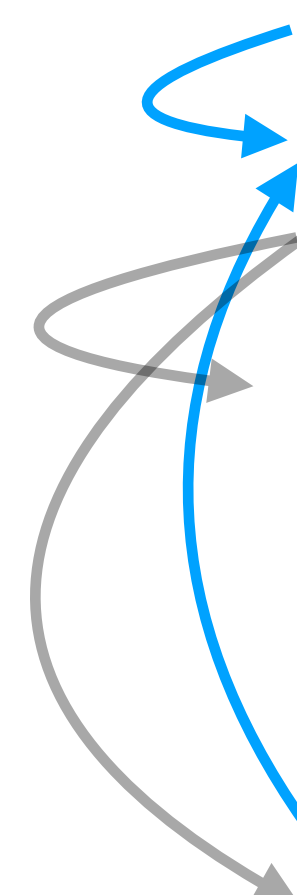
```

1: t1 = b + c // GEN {b+c}
2: t2 = a + 1 // GEN {a+1}
header:
3: if (n <= 0) goto end
body:
4: s  = b + c // GEN {b+c}
5: t3 = a + 1 // GEN {a+1}
6: a  = 0      // KILL exprs mentioning a (kills {a+1})
7: n  = n - 1  // KILL exprs mentioning n
8: goto header
end:
9: u = b + c

```

Definition: Join Point (where multiple paths join together)

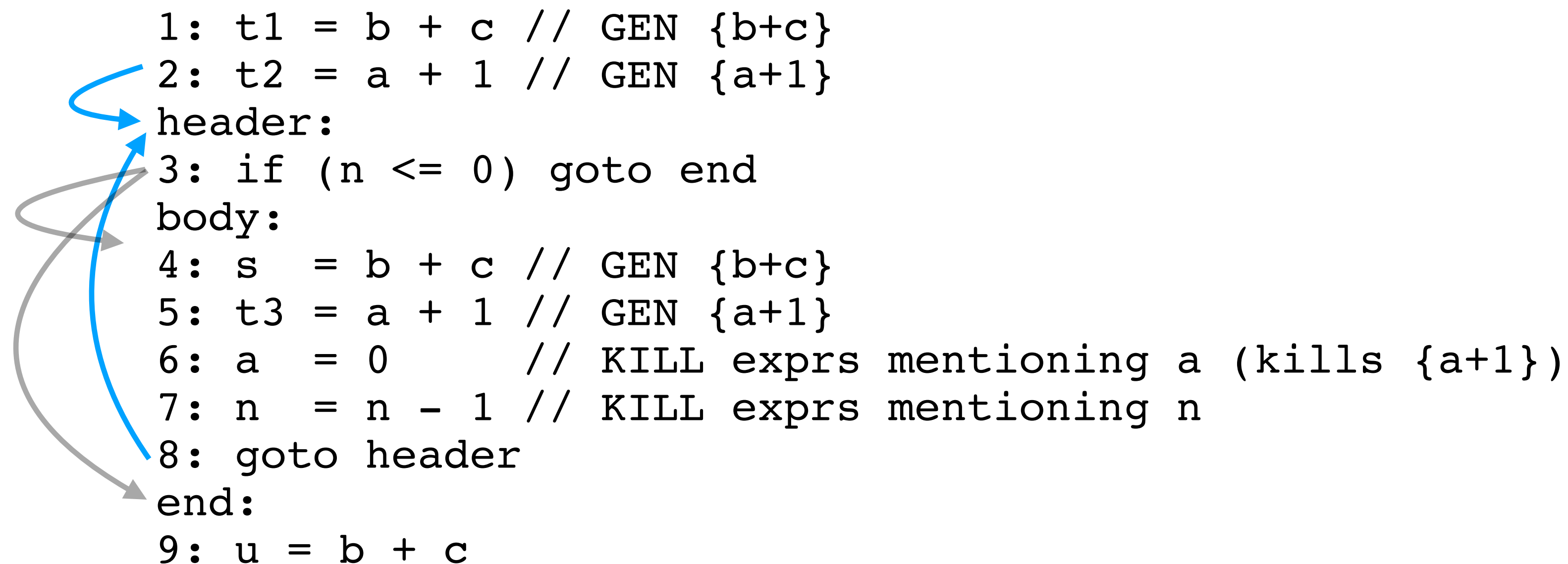
For this exercise, the interesting part is the **join** point (header),
where two paths come together



```
1: t1 = b + c // GEN {b+c}
2: t2 = a + 1 // GEN {a+1}
header: // from line 2 {b+c, a+1}, from 8, {b+c}
3: if (n <= 0) goto end
body:
4: s = b + c // GEN {b+c}
5: t3 = a + 1 // GEN {a+1}
6: a = 0 // KILL exprs mentioning a (kills {a+1})
7: n = n - 1 // KILL exprs mentioning n
8: goto header
end:
9: u = b + c
```

The question is: what happens at the join point?

- Coming into header from line 2, the available expressions are $\{b+c, a+1\}$
- But coming into header from body, the available expressions are $\{b+c\}$
- $a+1$ is killed by line 5, $n-1$ (line 7) doesn't get included because n killed by assignment
- Since we know we could have come from either line 2 or line 8: handle the possibility either branch taken!
- In this case, that means combining available expressions using \cap
 - Expression is only available if it is available along **both** branches!



```

1: t1 = b + c // GEN {b+c}
2: t2 = a + 1 // GEN {a+1}
header:
3: if (n <= 0) goto end
body:
4: s = b + c // GEN {b+c}
5: t3 = a + 1 // GEN {a+1}
6: a = 0 // KILL exprs mentioning a (kills {a+1})
7: n = n - 1 // KILL exprs mentioning n
8: goto header
end:
9: u = b + c

```

The analysis result (for this specific example...)

```
// {}  
1: t1 = b + c // {b+c}  
2: t2 = a + 1 // {b+c,a+1}  
  
// {b+c}  
header:  
3: if (n <= 0) goto end  
  
// {b+c}  
body:  
4: s = b + c // {b+c}  
5: t3 = a + 1 // {b+c,a+1}  
6: a = 0 // {b+c}  
7: n = n - 1 // {b+c} — no change to n  
8: goto header // {b+c}  
  
end:  
// {b+c}  
9: u = b + c
```

Traditional, Bit-Vector Dataflow Analysis

Available expressions is an instance of a very common class of dataflow analysis problems:

- ◆ Analysis results are “sets of facts” (where facts say specific, concrete things)
 - ◆ Set of “all possible facts” must be finite
- ◆ Each statement yields a GEN and KILL set
- ◆ Paths are combined using either \cup or \cap
 - ◆ AE is \cap , we all these “must” analyses, because the property must hold along *all* paths to hold along the join of multiple paths
- ◆ Can implement these analyses efficiently using *bit vectors*

Data-flow Equations for Available Expressions

- ◆ Given GEN/KILL for each statement, along with the control-flow graph (tells us join points, etc.), we can write out a set of flow analysis equations
- ◆ We will call $\text{In}(S)$ the set of analysis facts that hold on the entry to the statement S , and $\text{Out}(S)$ the set of analysis facts holding after S
- ◆ For available expressions...
 - ◆ $\text{GEN}(x = e) = \{e\}$ (if no occurrences of x), $\text{KILL}(x = e) = \{e' \mid e' \text{ contains } x\}$
 - ◆ $\text{In}(S) = \bigcap_{S' \in \text{incoming}(S)} \text{Out}(S')$
 - ◆ $\text{Out}(S) = \text{GEN}(S) \cup (\text{In}(S) - \text{KILL}(S))$

Example: Computing Available Expressions

Step 1:

Initialize all analysis results to {}

```
// {}  
1: a = x + y  
// {}  
2: b = x + y  
// {}  
3: x = 0  
// {}  
4: c = x + y  
// {}  
5: print(b)  
// {}
```

Step 2: Run the flow equations forward

Notice that we discover “more” information: some of the analysis result sets get bigger (more results)

This is a key principle: we’re always ascending a lattice, *monotonically*—result sets never shrink

So we eventually stop: we can’t keep going up forever (bounded set of analysis facts), and we’re always either (a) learning more or (b) no changes, thus **done**

```
// {}  
1: a = x + y  
// {x+y}  
2: b = x + y  
// {x+y}  
3: x = 0  
// {}  
4: c = x + y  
// {x+y}  
5: print(b)  
// {}
```

Step 3: The last step discovered some things in the Out sets, which allow us to propagate some information

Specifically: we need to consider newly-added items and how / if they should be in the out set of each statement...

```
// {}  
1: a = x + y  
// {x+y}  
2: b = x + y  
// {x+y}  
3: x = 0  
// {}  
4: c = x + y  
// {x+y}  
5: print(b)  
// {x+y}
```

Here I have simplified things: in general, Out is not simply equal to the previous statement's In—but in straight-line code, it is...

```
// {}  
1: a = x + y  
// {x+y}  
2: b = x + y  
// {x+y}  
3: x = 0  
// {}  
4: c = x + y  
// {x+y}  
5: print(b)  
// {x+y}
```



```

// AE: {}
1: a = x + y          // GEN {x+y}, KILL {exprs with a}
                        // IN: {}
                        // OUT: {x+y}
2: b = a + 1          // GEN {a+1}, KILL {exprs with b}
                        // IN: {x+y}
                        // OUT: {x+y, a+1}
3: while (b < 10) {    // IN: {x+y, a+1}      <- Merge from 2/6
                        // OUT: {x+y, a+1}
    4: c = x + y        // GEN {x+y}, KILL {exprs with c}
                        // IN: {x+y, a+1}
                        // OUT: {x+y, a+1}
    5: a = b + 2        // GEN {b+2}, KILL {exprs with a}
                        // IN: {x+y, a+1}
                        // OUT: {x+y, b+2}
    6: b = a + 1        // GEN {a+1}, KILL {exprs with b}
                        // IN: {x+y, b+2}
                        // OUT: {x+y, a+1}      <- back to header
}
8: print(b)           // IN: {x+y, a+1}

```

```

// AE: {}
1: p = u * v      // GEN {u*v}, KILL {exprs with p}
                  // OUT: {u*v}
2: q = p - v      // GEN {p-v}, KILL {exprs with q}
                  // IN: {u*v}
                  // OUT: {u*v, p-v}
3: if (flag) {    // IN: {u*v, p-v}
                  // OUT: {u*v, p-v}
4:     u = u + 1  // GEN {u+1}, KILL {exprs with u}
                  // IN: {u*v, p-v}
                  // OUT: {p-v, u+1}
        } else {
6:     v = v + 0  // GEN {v+0}, KILL {exprs with v}
                  // IN: {u*v, p-v}
                  // OUT: {v+0}
        }
8: r = u * v      // GEN {u*v}, KILL {exprs with r}
                  // IN (join of 4 & 6): {}
                  // OUT: {u*v}
9: while (r < 50) { // IN (loop header fixpoint): {u*v}
                  // OUT: {u*v}
10:    s = u * v  // GEN {u*v}, KILL {exprs with s}
                  // IN: {u*v}
                  // OUT: {u*v}
11:    v = v + 2  // GEN {v+2}, KILL {exprs with v}
                  // IN: {u*v}
                  // OUT: {v+2}
12:    r = u * v  // GEN {u*v}, KILL {exprs with r}
                  // IN: {v+2}
                  // OUT: {v+2, u*v} ← back to header
        }
13: print(r)      // IN (after loop): {u*v}

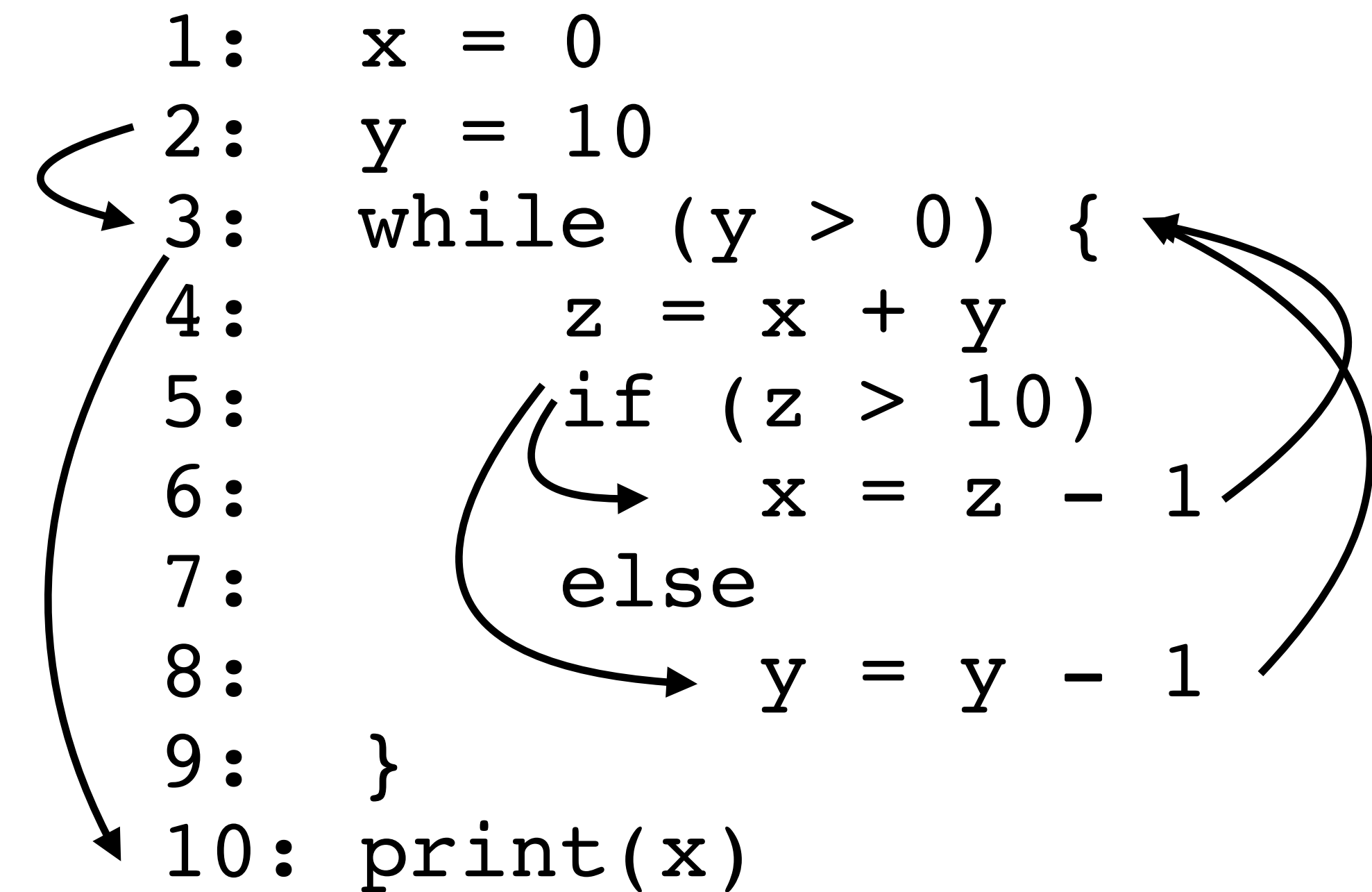
```

Live Variable Analysis

- Asks: which variables are “live” at each program point?
- Variable live when it is *later used* along some program path
- Used in *register allocation*
- Analysis facts: set of program variables
- $GEN(s) = \{\text{all variables read by } s\}$
- $KILL(x=e) = \{x\}$
- Live variables propagates information **backward** rather than forward
 - ◆ $In(S) = GEN(S) \cup (Out(S) - KILL(S))$
- It's a **May** (\cup) analysis
 - Live if a var used along **any** future path

```
// {}  
1: a = 1  
// {a}  
2: b = a + 2  
// {b, a}  
3: c = b + a  
// {c}  
4: b = read()  
// {c, b}  
5: c = b + c  
// {c}  
6: print(c)
```

- What variables live before line 3? {x,y}
 - x (used at 10 & 4)
 - y (used line 3)
- Before line 8? {x,y}
 - 3 is successor
 - y is *used on line 8 before assignment*
- After line 8? Also {x,y}
 - y used by guard, x by print(x)



Live Variables is a **backwards may** analysis:

- Information flows backwards
- Analysis facts: sets of variables
- When joining paths (backwards), combine w/ \cup

1: x = 1 + y	// GEN {y}, KILL {x}
	// OUT: {x}
	// IN: {y}
2: z = x + 2	// GEN {x}, KILL {z}
	// OUT: {z}
	// IN: {x}
3: y = z + 3	// GEN {z}, KILL {y}
	// OUT: {y}
	// IN: {z}
4: print(y)	// GEN {y}, KILL {}
	// OUT: {}
	// IN: {y}

```

1: i = 0          // GEN {},      KILL {i}
                  // OUT: {i,sum}
                  // IN:  {sum}

2: sum = 0        // GEN {},      KILL {sum}
                  // OUT: {i,sum}
                  // IN:  {i}

3: while (i < 3) { // GEN {i},    KILL {}
                  // OUT: {i,sum}
                  // IN:  {i,sum}

4:     sum = sum + i // GEN {sum,i}, KILL {sum}
                  // OUT: {i,sum}
                  // IN:  {i,sum}

5:     i = i + 1    // GEN {i},    KILL {i}
                  // OUT: {i,sum}    ← feeds back to 3
                  // IN:  {i,sum}

}
6: print(sum)      // GEN {sum},   KILL {}
                  // OUT: {}
                  // IN:  {sum}

```

There is a general framework...

- ◆ Big observation—all of these *different* analyses are fundamentally doing the same thing: iterating a set of dataflow equations until they get an answer
- ◆ Gives us a systematic **recipe** for how to design dataflow analyses:
 - ◆ Pick a set of “possible analysis facts,” must be bounded for termination
 - ◆ Pick GEN/KILL sets
 - ◆ Decide: forward or backward?
 - ◆ If forward, then $\text{Out}(S) = \text{GEN}(S) \cup (\text{In}(S) - \text{KILL}(S))$
 - ◆ If backwards, then $\text{In}(S) = \text{GEN}(S) \cup (\text{Out}(S) - \text{KILL}(S))$
 - ◆ Decide: may (combine using \cup) or must (combine using \cap)
- ◆ Once we have these, we get the analysis **for free**

Standard Intraprocedural Data Flow Analyses...

	May	Must
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

And what they mean...

- ◆ Available expressions: which expressions have not changed since they were last computed — forward MUST analysis (must have been the same along all branches)
- ◆ Common-subexpression elimination—compute a value **once**
 - ◆ But what about $2 * x = x * 2$? (Can't handle this entirely, not possible)
- ◆ Reaching definitions: which variables may reach a point without being overwritten
- ◆ Dead code elim, constant propagation, computing use/def chains
- ◆ Live variables: which variables are demanded and unchanged since last assignment
- ◆ Register allocation: non-live variables don't need to be in registers
- ◆ Very busy expressions: expressions which will necessarily be used along all paths
- ◆ Loop invariant code motion—no need to recompute in the loop body

Now, let's generalize...!

- ◆ Can compute **any** analysis by computing the *fixed point* of functions
- ◆ **Definition (Fixed Point of a Function):**
 - ◆ A fixed point of a function is a point X such that $f(X) = X$
- ◆ **Key issue:** need to know type of X for this statement to have any meaning!
 - ◆ Intuitively, X is FlowSolution (AnalysisResult, etc...)
- ◆ For the case of bit-vector dataflow analyses, X represents:
 - ◆ All of the In/Out sets for each statement in the program
 - ◆ (Avil. Expr.) X is finite map from statement (in/out) to set of AEs
- ◆ X (the type) must possess a notion of “no information” (written as \perp)
 - ◆ For AE \perp is this map: “for every statement X , the In/Out set are both $\{\}$ ”

What is f...?

- ◆ **Definition (Fixed Point of a Function):**

- ◆ A fixed point of a function is a point X such that $f(X) = X$
- ◆ Established that X is a map from statement In/Out sets to sets
 - ◆ I.e., $X = \text{Statement} \rightarrow \text{Set} \times \text{Set} = \text{FlowSolution}$
- ◆ What is f ?
 - ◆ Needs to be of type $\text{FlowSolution} \rightarrow \text{FlowSolution}$
 - ◆ f runs the **whole** analysis, for *each* statement, all at once
- ◆ Per-statement flow functions discussed previously:
 - ◆ Fwd: $\text{Out}(S) = \text{GEN}(S) \cup (\text{In}(S) - \text{KILL}(S))$, $\text{In}(S) = \bigcap_{S' \in \text{incoming}(S)} \text{Out}(S')$
 - ◆ Lift per-statement flows to “whole procedure” flows (not whole program)

Live Variables: Computing the Fixpoint

We can compute fix point via *iteration*

x_0 is \perp , which (for live variables) maps everything to $\{\}$

$x_0 =$

```
// IN:  {}  
1: i := 0  
// OUT: {}
```

```
// IN:  {}  
2: j := 1  
// OUT: {}
```

```
// IN:  {}  
3: sum := 0  
// OUT: {}
```

```
// IN:  {}  
Header: sum := sum + i  
// OUT: {}
```

```
// IN:  {}  
5: i := i + 1  
// OUT: {}
```

```
// IN:  {}  
6: if (i < j) goto Header  
// OUT: {}
```

```
// IN:  {}  
7: print sum  
// OUT: {}
```

Available expressions backwards propagates per-statement information on the first iteration...

$$f(x_0) = x_1 =$$

```
// IN:  {}
1: i := 0
// OUT: {}

// IN:  {}
2: j := 1
// OUT: {}

// IN:  {}
3: sum := 0
// OUT: {}

// IN:  {sum, i}
Header: sum := sum + i
// OUT: {}

// IN:  {i}
5: i := i + 1
// OUT: {}

// IN:  {i, j}
6: if (i < j) goto Header
// OUT: {}

// IN:  {sum}
7: print sum
// OUT: {}
```

In available expressions, we propagate backwards, from in (back to) out...

$$f(x_1) = x_2 =$$

```
// IN:  {}  
1: i := 0  
// OUT: {}
```

```
// IN:  {}  
2: j := 1  
// OUT: {}
```

Notice how In flows to Out

```
// IN:  {i}  
3: sum := 0  
// OUT: {sum, i}
```

```
// IN:  {sum, i}  
Header: sum := sum + i  
// OUT: {i}
```

```
// IN:  {i, j} i and j propagated from 6->5  
5: i := i + 1  
// OUT: {i, j}
```

```
// IN:  {i, j}  
6: if (i < j) goto Header  
// OUT: {sum}
```

```
// IN:  {sum}  
7: print sum  
// OUT: {}
```

sum propagated from 7->6

Because we're running the analysis backwards, we propagate information from In sets of *branch targets* to Out sets of branches

$$f(x_2) = x_3 =$$

Since this is a “May” analysis, we merge In sets via \cup to calculate Out set

In set is calculated by GEN/KILL

```
// IN:  {}
1: i := 0
// OUT: {}

// IN:  {}      i propagated up
2: j := 1
// OUT: {i} ←

// IN:  {i}
3: sum := 0
// OUT: {sum, i}

// IN:  {sum, i}
Header: sum := sum + i
// OUT: {i, j} ←      j backward propagates...

// IN:  {i, j}
5: i := i + 1
// OUT: {i, j}

// IN:  {i, j, sum}
6: if (i < j) goto Header
// OUT: {sum, i} ←      {sum} ∪ {sum, i} = {sum, i}

// IN:  {sum}
7: print sum
// OUT: {}
```

Continued propagation of results eventually yields this analysis result...

$$f(x_N) =$$

```
// IN:  {}
1: i := 0      Usages of i/j eventually flow all the way
// OUT: {i}    up...

// IN:  {i}
2: j := 1
// OUT: {i,j}

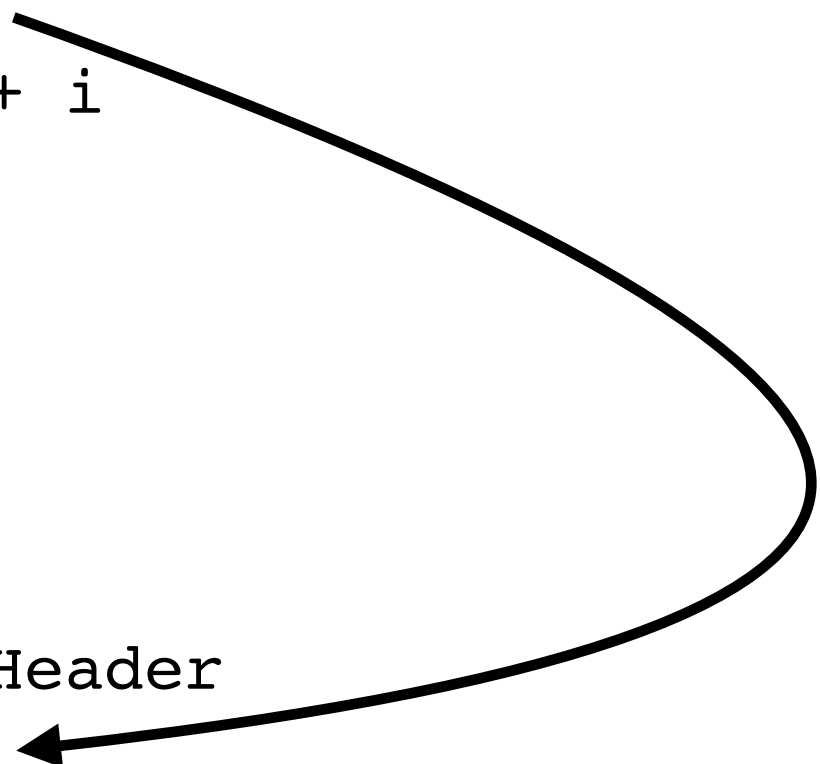
// IN:  {i,j}
3: sum := 0
// OUT: {sum, i, j}  j eventually reaches Header's In set—
                    propagates back to Out of 6...

// IN:  {sum, i, j}
Header: sum := sum + i
// OUT: {i, j, sum}

// IN:  {i, j, sum}
5: i := i + 1
// OUT: {i, j, sum}

// IN:  {i, j, sum}
6: if (i < j) goto Header
// OUT: {sum, i, j}

// IN:  {sum}
7: print sum
// OUT: {}
```



This result has an interesting property...

$$f\left(\begin{array}{l} // \text{ IN: } \{\} \\ 1: i := 0 \\ // \text{ OUT: } \{i\} \\ \\ // \text{ IN: } \{i\} \\ 2: j := 1 \\ // \text{ OUT: } \{i, j\} \\ \\ // \text{ IN: } \{i, j\} \\ 3: \text{sum} := 0 \\ // \text{ OUT: } \{\text{sum}, i, j\} \\ \\ // \text{ IN: } \{\text{sum}, i, j\} \\ \text{Header: sum} := \text{sum} + i \\ // \text{ OUT: } \{i, j, \text{sum}\} \\ \\ // \text{ IN: } \{i, j, \text{sum}\} \\ 5: i := i + 1 \\ // \text{ OUT: } \{i, j, \text{sum}\} \\ \\ // \text{ IN: } \{i, j, \text{sum}\} \\ 6: \text{if } (i < j) \text{ goto Header} \\ // \text{ OUT: } \{\text{sum}, i, j\} \\ \\ // \text{ IN: } \{\text{sum}\} \\ 7: \text{print sum} \\ // \text{ OUT: } \{\} \end{array}\right) = \begin{array}{l} // \text{ IN: } \{\} \\ 1: i := 0 \\ // \text{ OUT: } \{i\} \\ \\ // \text{ IN: } \{i\} \\ 2: j := 1 \\ // \text{ OUT: } \{i, j\} \\ \\ // \text{ IN: } \{i, j\} \\ 3: \text{sum} := 0 \\ // \text{ OUT: } \{\text{sum}, i, j\} \\ \\ // \text{ IN: } \{\text{sum}, i, j\} \\ \text{Header: sum} := \text{sum} + i \\ // \text{ OUT: } \{i, j, \text{sum}\} \\ \\ // \text{ IN: } \{i, j, \text{sum}\} \\ 5: i := i + 1 \\ // \text{ OUT: } \{i, j, \text{sum}\} \\ \\ // \text{ IN: } \{i, j, \text{sum}\} \\ 6: \text{if } (i < j) \text{ goto Header} \\ // \text{ OUT: } \{\text{sum}, i, j\} \\ \\ // \text{ IN: } \{\text{sum}\} \\ 7: \text{print sum} \\ // \text{ OUT: } \{\} \end{array}$$

This is the ***fixed point*** of the analysis

At this point, **no new analysis results are discoverable**

$$f(x_{Result}) = f(x_{Result}) =$$

```
// IN:  {}
1: i := 0
// OUT: {}

// IN:  {}
2: j := 1
// OUT: {}

// IN:  {i}
3: sum := 0
// OUT: {sum, i}

// IN:  {sum,i, j}
Header: sum := sum + i
// OUT: {i, j, sum}

// IN:  {i, j, sum}
5: i := i + 1
// OUT: {i, j, sum}

// IN:  {i, j, sum}
6: if (i < j) goto Header
// OUT: {sum, i, j}

// IN:  {sum}
7: print sum
// OUT: {}
```

How do we make this rigorous?

- ◆ I just described one way of computing analysis solutions
 - ◆ The method I presented is called “chaotic iteration”
 - ◆ Claim: the process ***always terminates*** and ***always yields a fixed point***
- ◆ **But:** How do we make this claim rigorous? Need a bit more formalism...
- ◆ We will show...
 - ◆ Monotonic functions always have a fixed-point
 - ◆ Which can always be computed via chaotic iteration!
 - ◆ First must define what a “monotonic” function is...

Definition: Lattice

- ◆ A *lattice* is a set E of lattice elements, along with...
 - ◆ A partial order \sqsubseteq , which relates elements ($e_0 \sqsubseteq e_1$) or not
 - ◆ Reflexive, antisymmetric, and transitive
 - ◆ An operation $e_0 \sqcup e_1$, the “join” (least upper bound) of e_0 and e_1
 - ◆ The least upper bound: the least element e' s.t., $e_0 \sqsubseteq e'$ and $e_1 \sqsubseteq e'$
 - ◆ An operation $e_0 \sqcap e_1$, the “meet” (greatest lower bound) of e_0 and e_1
 - ◆ The greatest lower bound: the greatest element e' s.t., $e' \sqsubseteq e_0$ and $e' \sqsubseteq e_1$
 - ◆ Two named elements: \top (“top”) and \perp (“bot”), s.t., $e \sqsubseteq \top$ and $\perp \sqsubseteq e$ for all e
- ◆ (Trivia—Complete lattice: arbitrary joins/meets, every subset rather than binary joins/meets)

- ◆ You can think of a lattice as an “Interface” or “Typeclass” from programming
 - ◆ Q: Why do we care to generalize?
 - ◆ A: Many results hold over **any** lattice—gives us reusable framework, you bring the lattice, get the program analysis “for free” (induced by construction)
- ◆ For any set S , the power set (set of subsets of S) is a lattice:
 - ◆ \sqcup is \cup , \sqcap is \cap , \top is S , \perp is $\{\}$
- ◆ Positive integers, ordered as follows: $x \sqsubseteq y$ iff x divides y
 - ◆ Meet (glb) $x \sqcap y = \gcd(x, y)$
 - ◆ Join (lub) $x \sqcup y = \text{lcm}(x, y)$
 - ◆ All integers, ordered by divisor does **not** work (why? Ask: is \sqsubseteq antisymmetric?)
- ◆ $\{0, 1\}$ is a lattice (Boolean lattice): min is \sqcap , max is \sqcup , \top is 1 , \perp is 0

- ◆ Propositional formulas ordered by logical implication: \Rightarrow , \top is \wedge , $\sqcup = \vee$, $\top = \text{True}$, ...

- ◆ The lattice of integer intervals: $[a,b]$ s.t., $a \leq b$
 - ◆ $[x_1, x_2] \sqsubseteq [y_1, y_2]$ iff $y_1 \leq x_1 \wedge x_2 \leq y_2$
 - ◆ \top is $[-\infty, +\infty]$ (largest possible interval)
 - ◆ \perp is the empty interval $[+\infty, -\infty]$ (impossible interval)
 - ◆ $[x_1, x_2] \sqcup [y_1, y_2] = [\min(x_1, y_1), \max(x_2, y_2)]$
 - ◆ $[x_1, x_2] \sqcap [y_1, y_2] = [\max(x_1, y_1), \min(x_2, y_2)]$

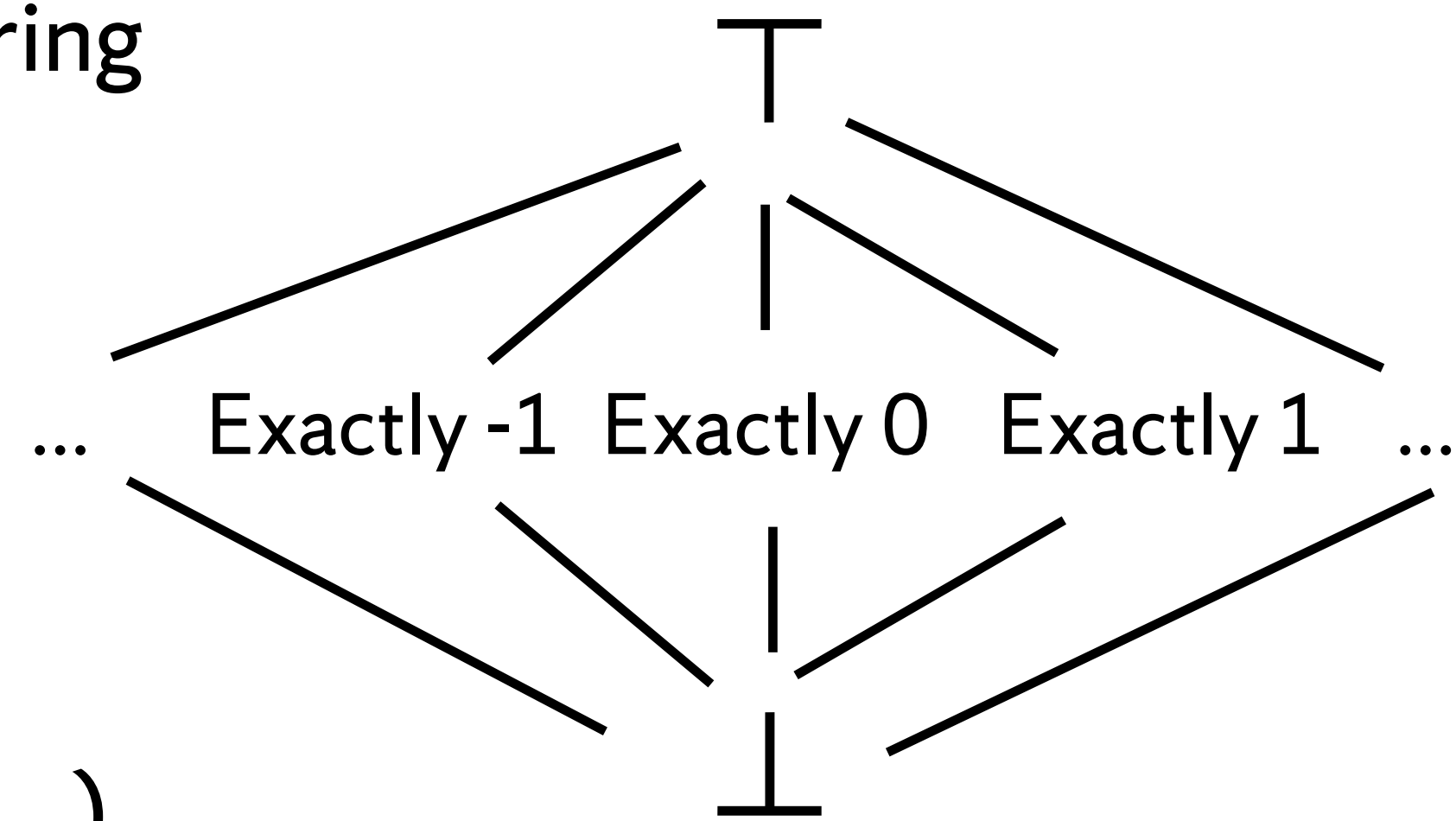
- ◆ The **constant propagation lattice (right)**
 - ◆ \top is “all possible values,” \perp is “no possible values.”
 - ◆ Also, “exactly N” for all values N:
 - ◆ “exactly N” \sqcup “exactly N” is “exactly N”, “exactly N” \sqcup “exactly K” = \top

Hasse Diagrams

Can visualize any lattice as a “Hasse diagram” as sketched on right

Joins / meets are apparent by organization

Things higher on the page are larger in the ordering



(Constant propagation lattice pictured...)

Definition: Monotonicity

- ◆ Lattices give us “information orders,” allowing us to compare / join information
- ◆ We now define monotonic functions...
 - ◆ Let L be a lattice
 - ◆ If f is a function over L , i.e., $L \rightarrow L$, then f is **monotonic** precisely when:
 - ◆ For all $e \in L$, $f(e) = e'$ implies $e \sqsubseteq e'$
- ◆ A monotonic function *never moves you down the lattice*
 - ◆ It may not move you up, either—when $f(e)=e$, e is a fixed point!
 - ◆ But, if $f(e) \neq e$, then we **know** that $e \sqsubseteq f(e)$
 - ◆ I.e., running f will always yield more information

Big Result: The Knaster-Tarski theorem

- ◆ Let (L, \sqsubseteq) be a lattice, and let $f : L \rightarrow L$ be a monotonic function over L
 - ◆ The set of *fixed points* of f **also** forms a lattice
 - ◆ Since it's a lattice, there is a *least* fixed point (lattices always have \perp)
 - ◆ In particular, f has a least fixed point **in L**
 - ◆ Which *may be computed* $f(\dots(f(\perp)\dots))$
- ◆ Trivial to prove: if L is **finite** then analysis will **definitely terminate**
 - ◆ There are no infinite increasing chains $\perp, f(\perp), f(f(\perp)), \dots$
 - ◆ f is monotone, and L is finite—either you hit a fix point, or you “run out” of L

Computing Fixed Points

- ◆ Easy: compute via “Chaotic iteration:”

```
(define (fix f)
  (define (h x)
    (if (equal? x (f x)) x (h (f x))))
  h)
```

- ◆ Unfortunately, this is slow:
 - ◆ Everything from ***every previous iteration*** rediscovered each iteration
 - ◆ Observation: we can often *incrementalize* the analysis
 - ◆ Great for bit-vector analyses; more on this later

What is not covered by this lecture...

- ◆ Kildall-style dataflow analyses are ***inherently intraprocedural***
 - ◆ *Useful* for some common compiler tasks (register allocation, use/def chains)
 - ◆ They offer **no** story for heap-allocated data
 - ◆ Issue is aliasing: pointer writes $*(x) = y$ might totally invalidate analysis facts
 - ◆ In general, need to reason about aliasing via pointer analysis
 - ◆ Not possible to reason across functions in general:
 - ◆ What do you do with recursive functions?
 - ◆ What if called function invalidates analysis facts?
- ◆ We'll cover functions next, then we'll discuss a bit of interprocedural analysis!

Summary

- ◆ Introduction to lattices, monotonic functions, fix points
- ◆ Kildall-style dataflow analysis is a great story:
 - ◆ You tell me: analysis fact structure, forward / backward, GEN/KILL functions
 - ◆ Get analysis “for free,” efficient implementation via bit vectors
 - ◆ May vs. Must (how to combine analysis results at join points)
- ◆ Know the popular analyses: reaching definitions, available expressions, very busy expressions, and live variables
- ◆ What is a lattice, why do we care about them?
 - ◆ What are top, bottom, meet, and join?
- ◆ How do we use lattices in dataflow analysis? (Why did we learn them?)
 - ◆ The Fixed-Point Theorem: every monotonic function has a fixed point
 - ◆ kkmicins@syr.edu