NTNU – Trondheim
Norwegian University of
Science and Technology

Live Variables

Where we were

- Control flow graphs represent the computation and control flow of a program
- Nodes are basic blocks, representing the computation
- Arcs represent the control flow between basic blocks
- CFGs can be built from high or low level IR

Uses of CFG representation

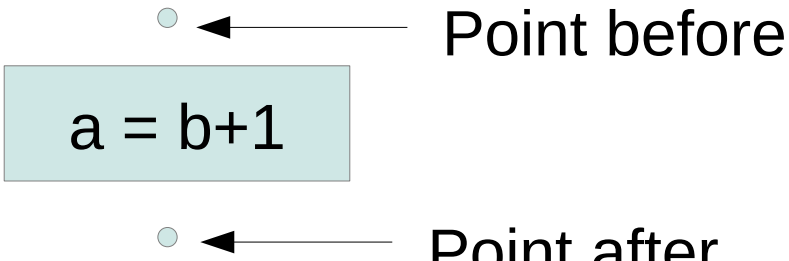
- The purpose is to statically extract information about the program at compile time
- Reasoning about the run-time values of variables and expressions in every possible execution enables optimizations
- We can illustrate this by finding *live variables*

Liveness

- A live variable is one which holds a value that may still be used at a later point
- Conversely, a dead variable is guaranteed to see no further use (until its next assignment)
- This means we’re searching for ranges of instructions in the program where variables hold values that matter to the execution
- In order to find ranges of instructions, we need to define program points the ranges can span across

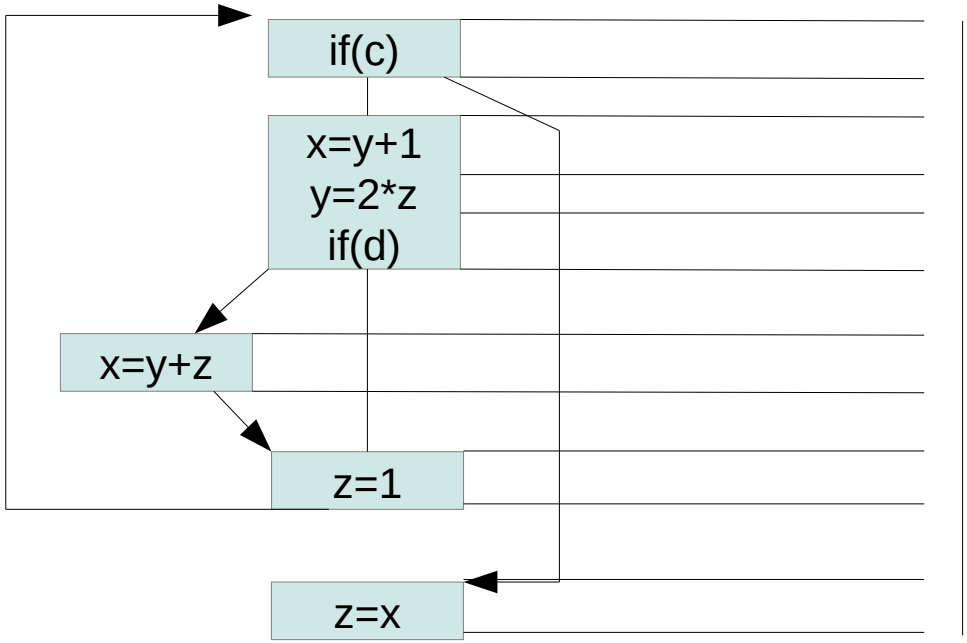
Program points

- As we want to capture how state is changed through an instruction, we need to talk about the state before and the state after, and describe the difference
- Hence, there is one program point before and one after each instruction



- For basic blocks, they're the points
  - After the prececessor(s)
  - Before the successor(s)

Program points in the example we saw before



These are all the points

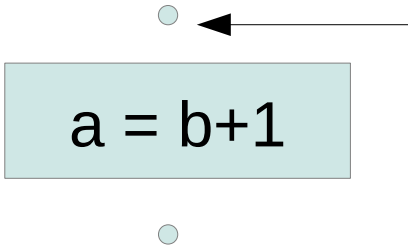
The two things to mind

- How does an instruction affect the state at the points immediately before and after it?
  - In other words, what is the effect of an instruction?
- How does state propagate between program points?
  - In other words, what is the effect of control flow?

- If we can tell which variables are live at one point, we can compute which ones are live by its neighbors

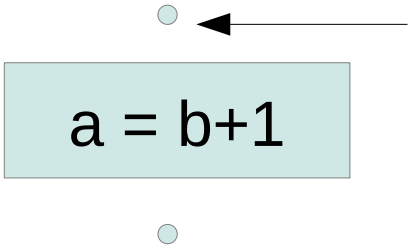
Which instructions affect liveness?

- If a variable is used in an expression, it must be kept at the preceding program point:



b must be live here, we need it in a moment

- If a variable is defined by an expression, it was dead at the preceding program point



a will not be used again here, the next thing we do is redefine it

Systematizing it a little

- For an instruction I, define two sets of variables
  - in[I] = set of live variables at point before I
  - out[I] = set of live variables at point after I
- This extends naturally to basic blocks
  - in[B] = set of live variables at point before B
  - out[B] = set of live variables at point after B

...so if I_1 and I_2 are the first and last instructions in B,

- $in[B] = in[I_1]$
- $out[B] = out[I_2]$

Before & after *vs.* instructions

- All variables used by an instruction must be live before it can use them
- Variables defined by an instruction are not live at the last point before the instruction
- That is,

  live before = live after – defined vars + used vars

  or

  in[I] = out[I] – def(I) + use(I)

Before & after *vs.* control flow

- All variables used along the path of any successor must be live after the predecessor
  - You never know which path will be taken, one of them might need it
- Where control flows split,

  live after = live before successor #1 + live before successor #2 +…

  or

  $out[I] = in[I_1] + in[I_2]$

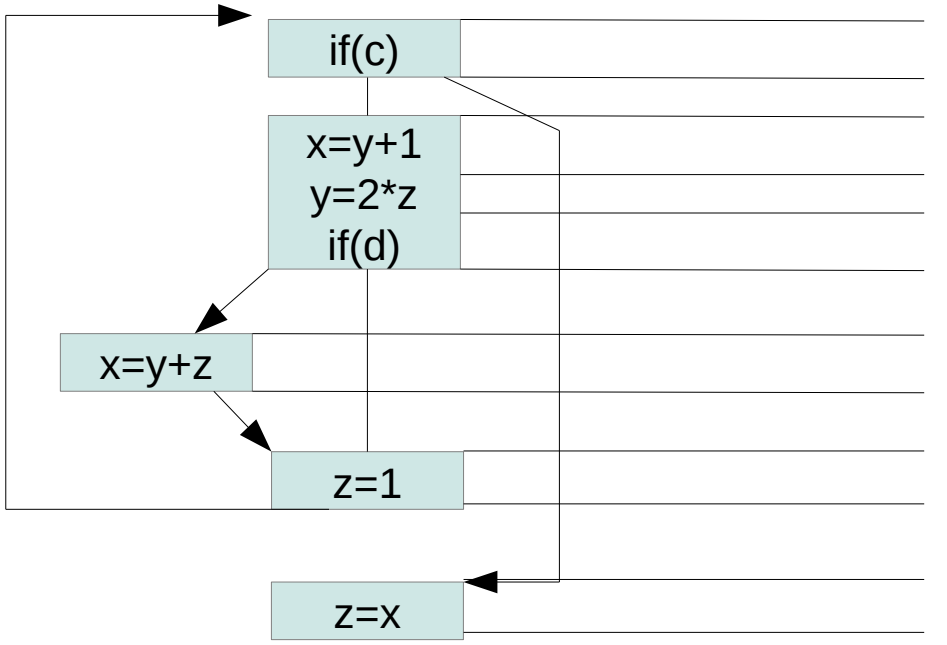
  where I_1, I_2 are successors of I

Liveness flows backwards

- As you may notice, we're defining the in-sets in terms of the out-sets
- This means we need out-sets to start working with
- In the name of safety, assume that every variable is live until it has been determined otherwise
- This gives us a final out-state to start working from, so that we can examine the CFG in reverse

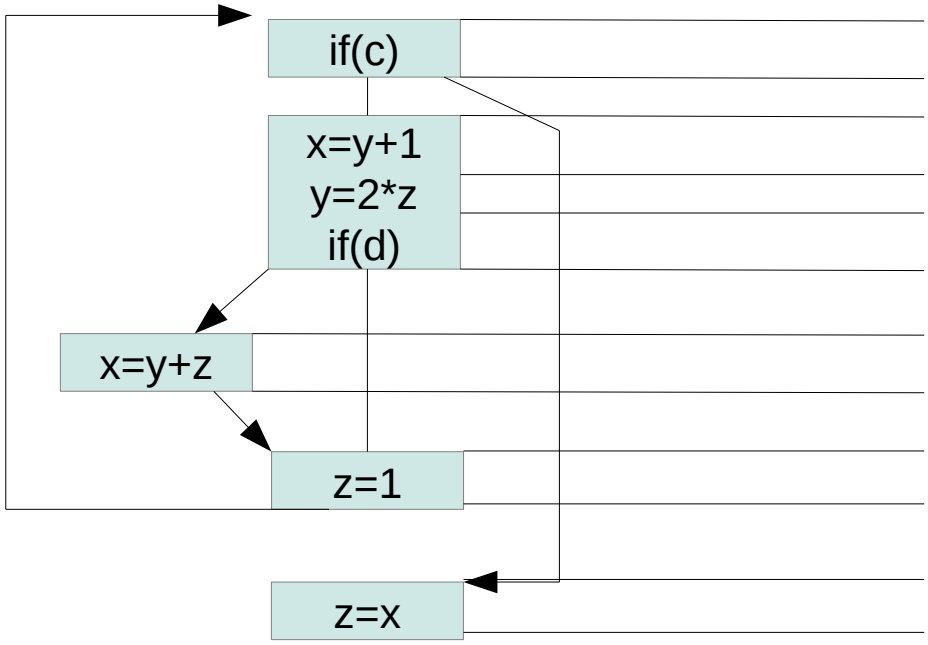
Start from the back

Conservative assumption: everything is live at the end



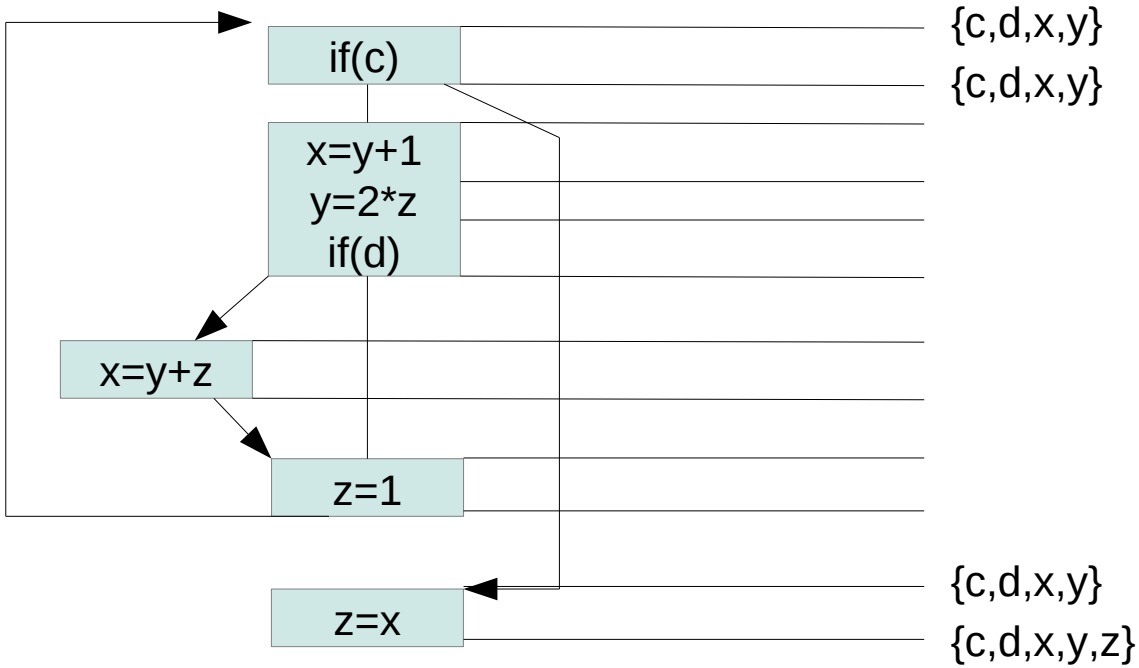
Iteration 1

Last statement defines z



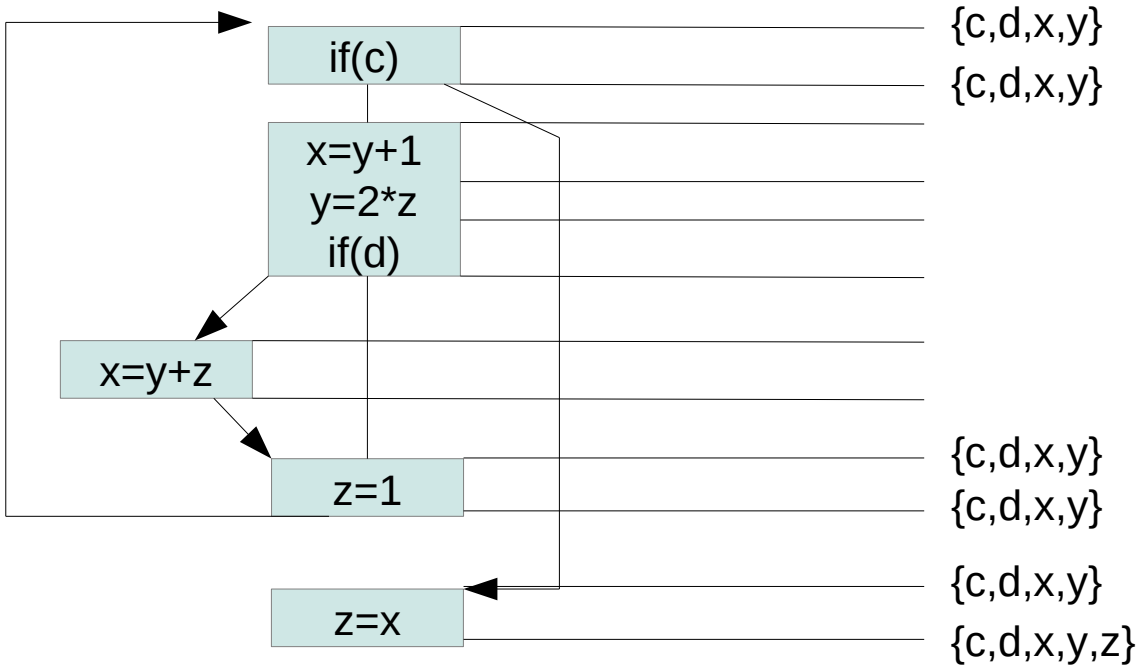
Iteration 1

Predecessor doesn't define anything



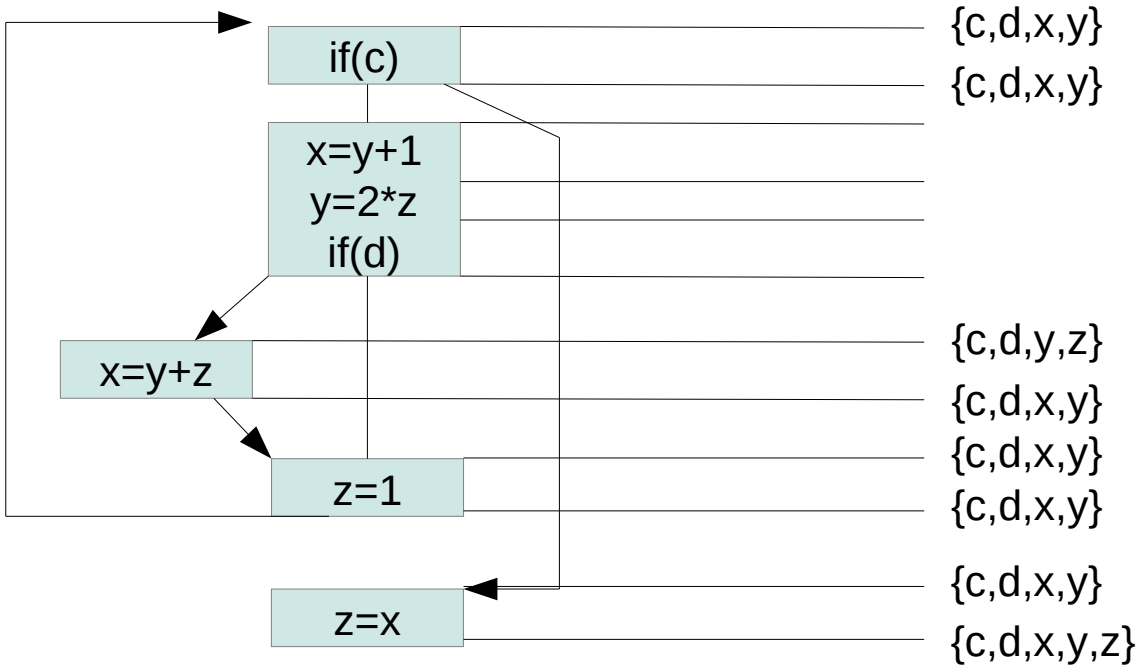
Iteration 1

Predecessor defines z, but it wasn't live anyway



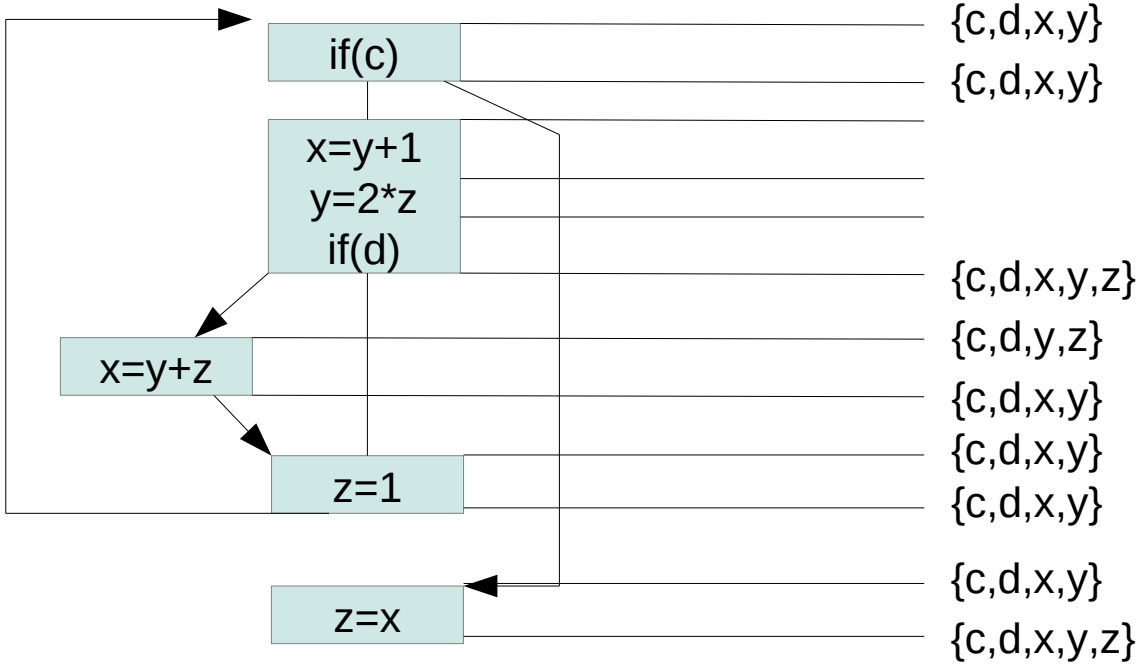
Iteration 1

Predecessor *uses* z, it is live again



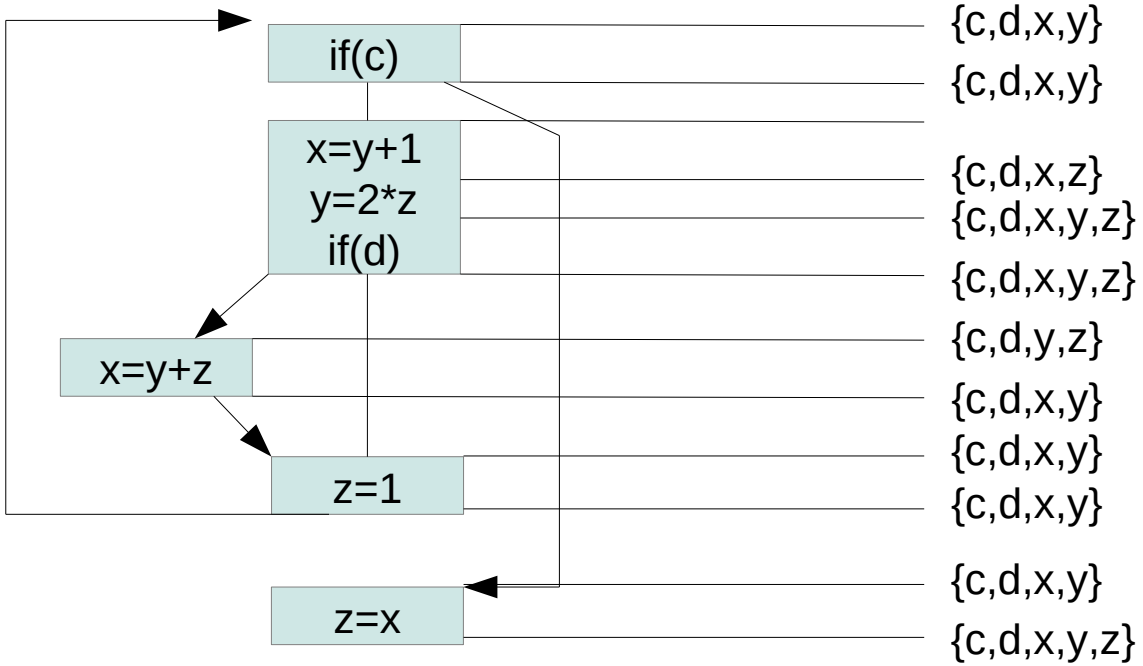
Iteration 1

Predecessor of two successors (control flow):
must assume union of the live variables of each succ.



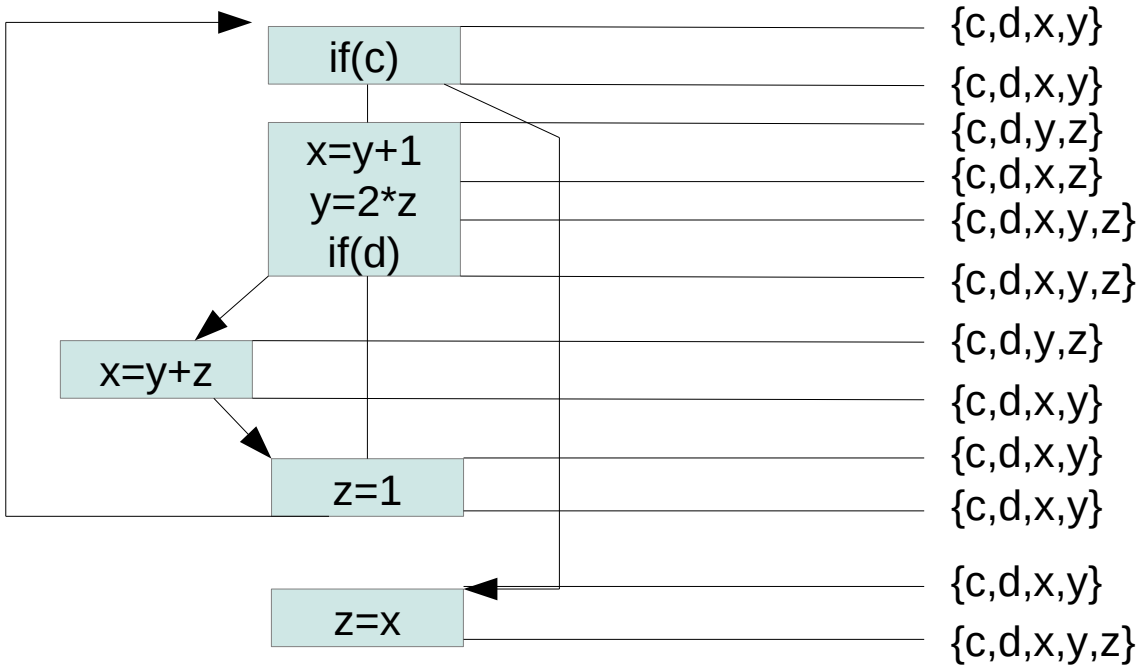
Iteration 1

Definition of y



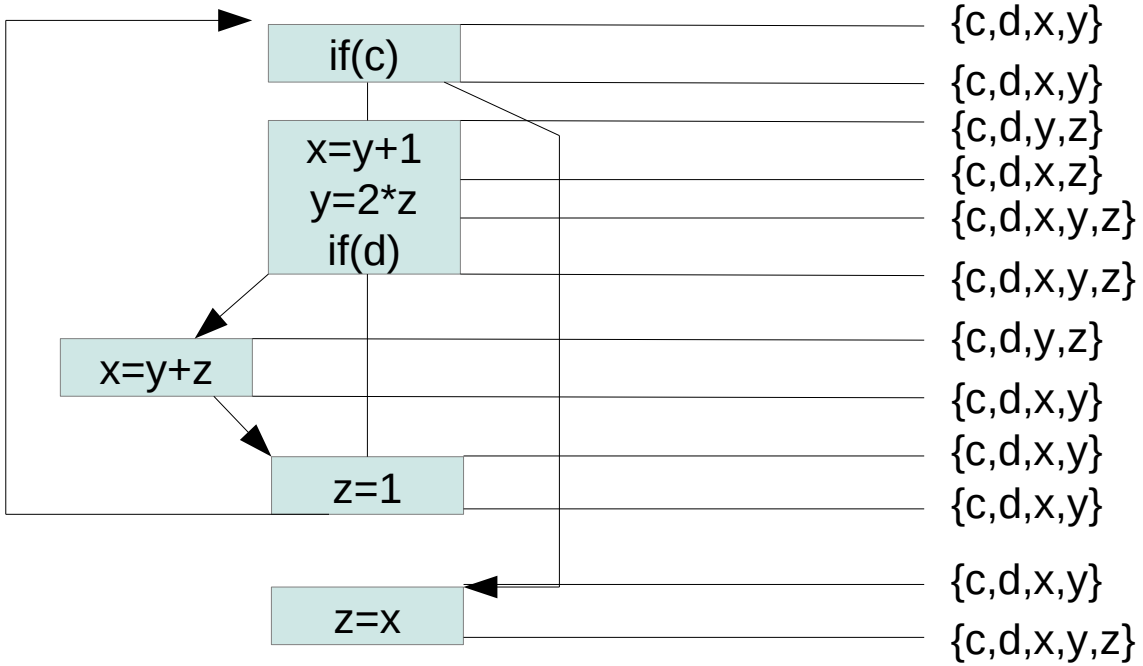
Iteration 1

Use of y, definition of x



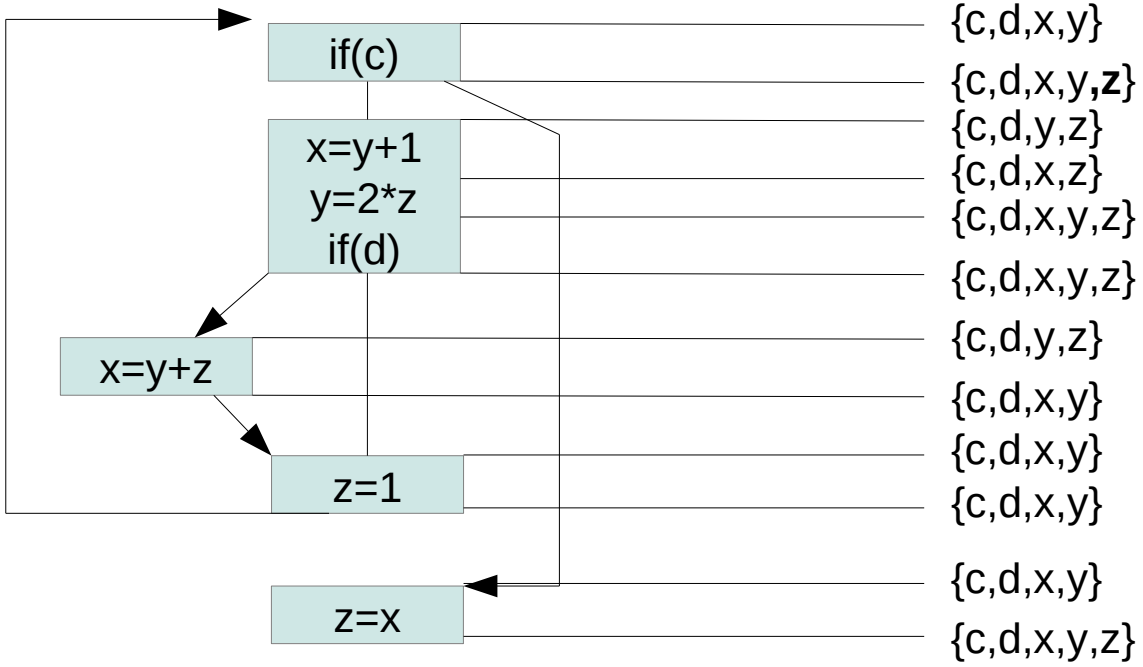
End of iteration 1

We’ve covered all points, but *something changed* - Repeat from the start...



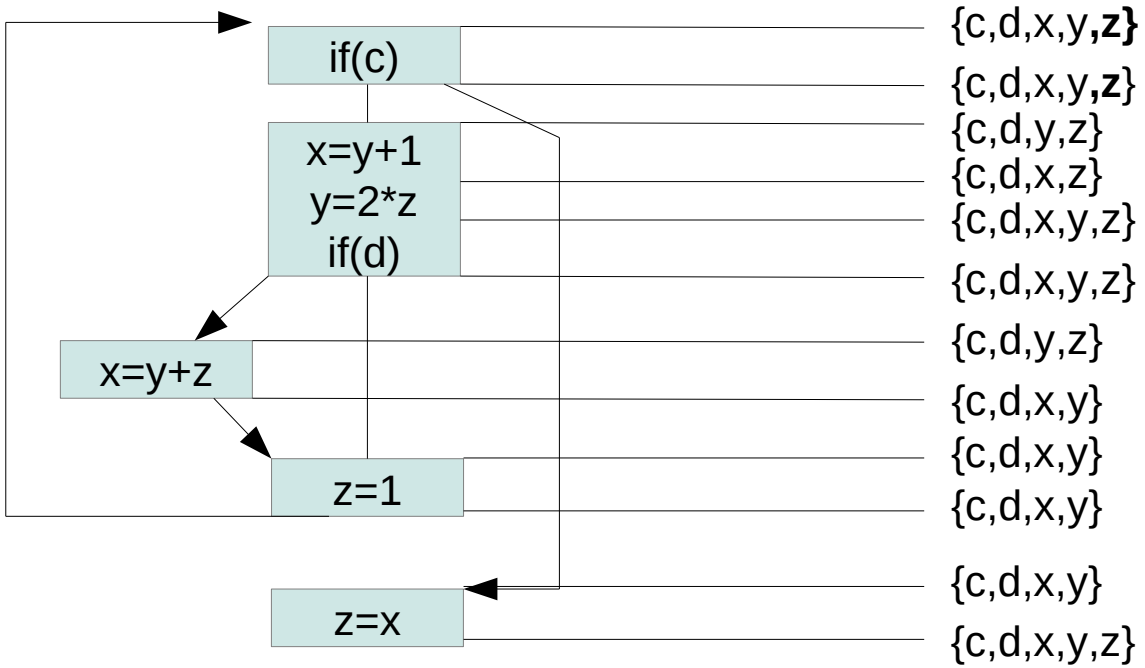
Iteration 2

The union of the two successors here is different



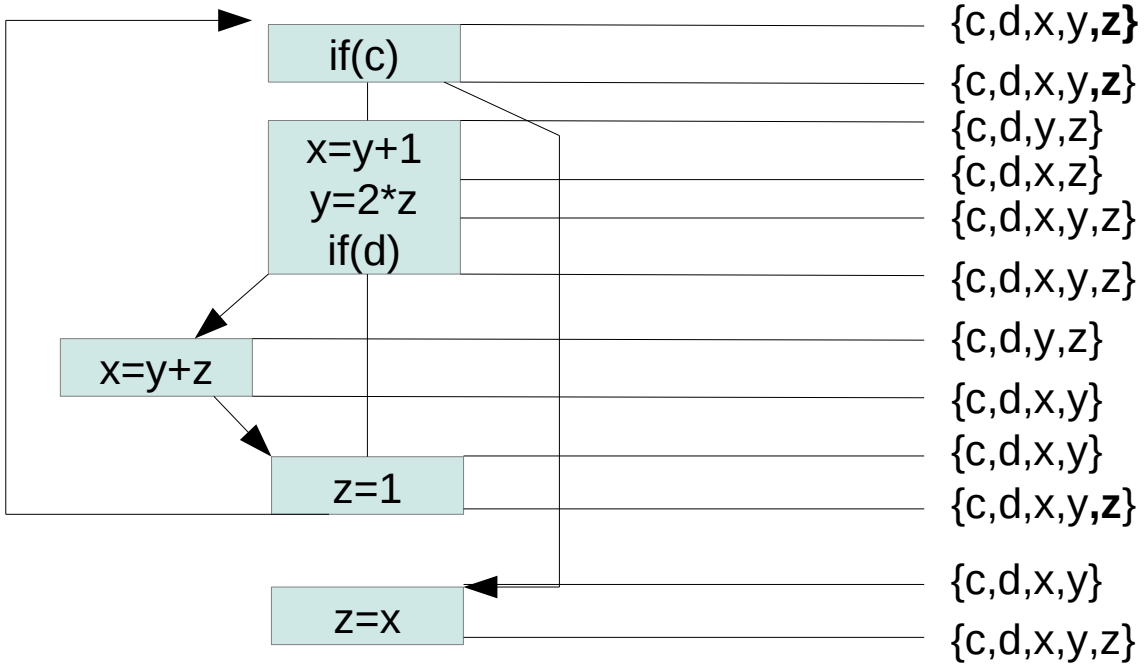
Iteration 2

Propagate it to predecessor



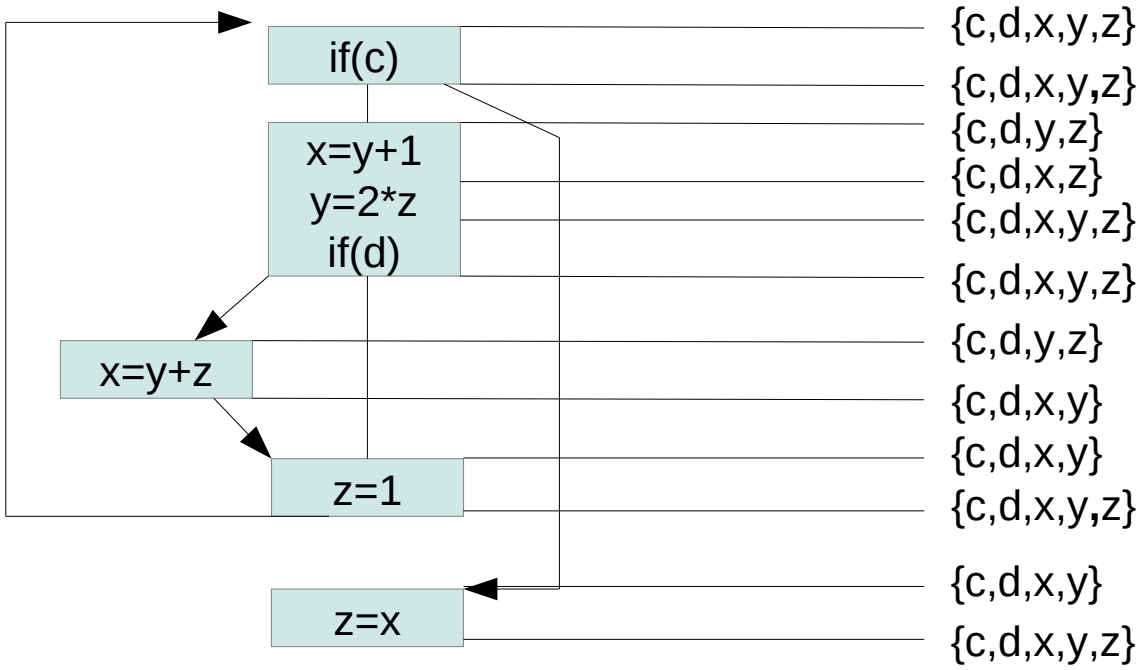
Iteration 2

...and again, until we've been through all nodes…
(then repeat, because something changed)



Iteration 3

Nothing changes, we have reached a *fixed point*



Between the lines

- Every instruction implies a constraint equation
  - Live before = live after – what it defines + what it uses
- Everywhere control flows join, there is another constraint equation
  - Live after = sum of what's live at all successors

- The framework for data flow analysis is just different instances of this pattern
  - Different constraint equations capture different information
  - Different split/join behavior follows from the type of information
  - May work forward or backward (liveness propagates backwards)
- We'll look at a handful of instances