NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 17: Optimizations in detail

Michael Engel

Overview

- Optimizations
  - Control-flow graphs
  - Liveness of variables

Optimization

- We wish to apply various program transformations to improve its non-functional properties without changing its meaning
- Transformations can apply either at IR or lower levels

- Optimizations have to be ***safe***
    - the optimized program must give the same results as the un-optimized program **for every possible execution**

- We need some structured approaches to ensure this…

The meaning of programs

- Information required for performing optimizations often is not explicitly contained in the source code
    - So we have to extract information
- Consider the following code:

```
x = y + 1;
y = 2 * z;
x = y + z;
z = 1;
z = x;
```

- Are all these statements necessary?

Program meaning is implicit

- Some of the statements are ***dead code***

```
x = y + 1;     ← This assignment of x
y = 2 * z;     ← …is not used in any intermediate statement…
x = y + z;     ← …until x is assigned again here
z = 1;         ← This assignment of z…
z = x;         ← …is immediately overwritten
```

- Knowing this, we can construct a shorter **identical** program

```
y = 2 * z;

x = y + z;

z = x;
```

- Control flow is linear here, so the dead state is obvious
- It becomes harder to tell when control flow is involved

Conditions complicate everything

- If we add some **control flow**…

```
x = y + 1;    ← is this statement still dead?
y = 2 * z;
if (c) { x = y + z; }
z = 1;        ← what about that one?
z = x;
```

- …the first assignment to **x** may or may not be used again:

```
x = y + 1;    ← x is reused in a loop here
y = 2 * z;
if (c) { x = y + z; }
z = 1;        ← This still makes no difference
z = x;
```

- This assignment becomes relevant when the value of **c** is false

Loops complicate even more…

- If we **insert a loop**…

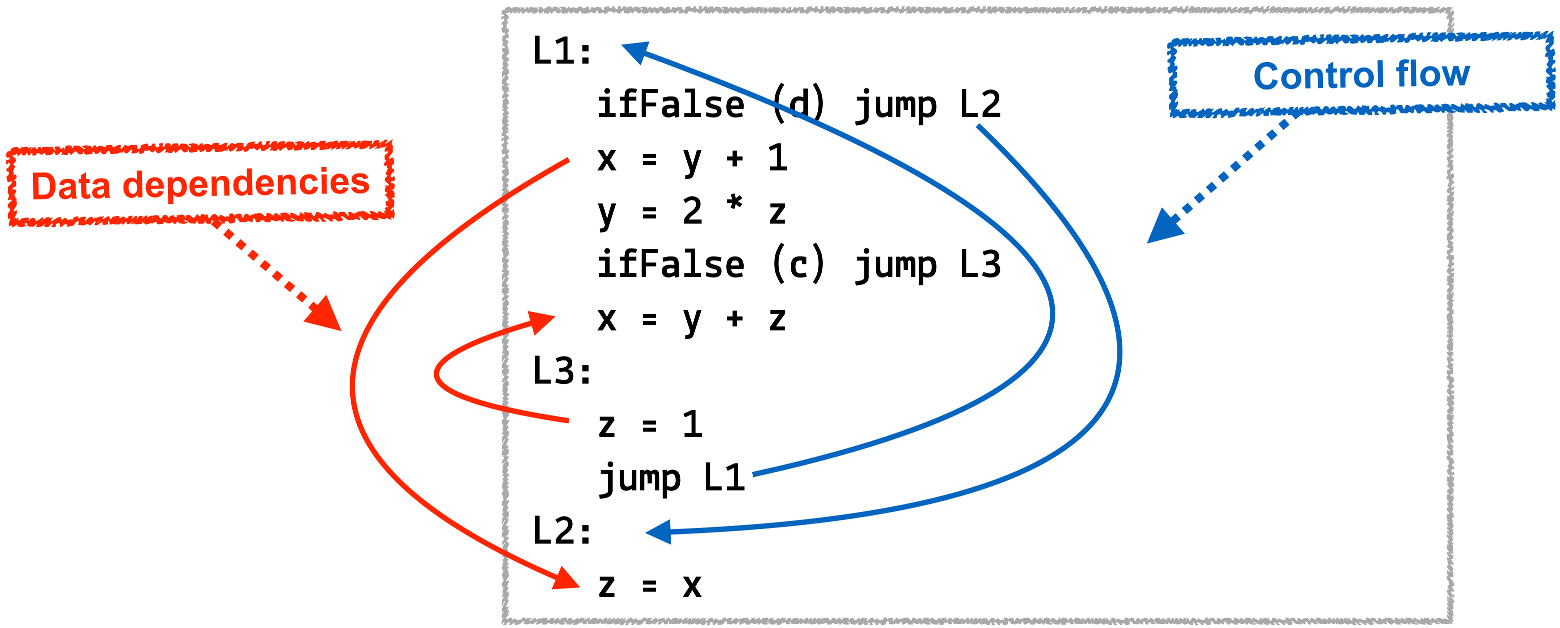
```
while (d) {
   x = y + 1;    ← is this statement still dead?
   y = 2 * z;
   if (c) { x = y + z; }
   z = 1;        ← is this statement still dead?
}
z = x;
```

- …neither statement can be omitted!

```
while (d) {
   x = y + 1;
   y = 2 * z;            ← assignment z=1 becomes relevant
   if (c) { x = y + z; }   if there is an additional
   z = 1;                  iteration of the loop!
}
z = x;
```

Low-level code makes it worse…

- Control flow is more obvious from source code syntax than from its translation into jumps and labels:



What do we need?

- Methods to compute information that are
    - implicit in the program
    - static (so that it can be found at compile time)
    - valid for every possible dynamic situation (at runtime)

- A data structure that can represent every possible control flow
    - Different branches taken (conditionals)
    - Branches taken different numbers of times (loops)

- Problem is similar to that of NFA:
  "What are all the possible paths I can take from here?"

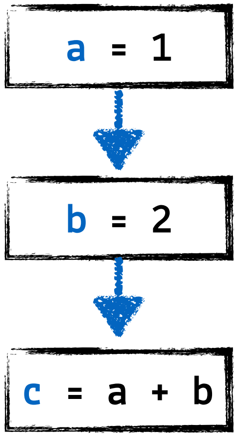
Control Flow Graphs (CFGs)

- Program control flow can be captured in a directed graph, where statements make nodes and their sequencing follows the arcs
- Movement of data can be inferred by traversing a structure like this
    - By far the most common approach in present compilers (It is also possible to graph data movement and infer control, but let's stick to the control flow view)
- Multiple paths emerge since nodes can have multiple incoming/ outgoing arcs

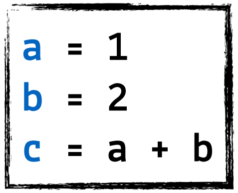
Linear code sequences

- Rather simple…

```
a = 1;
b = 2;
c = a + b;
```



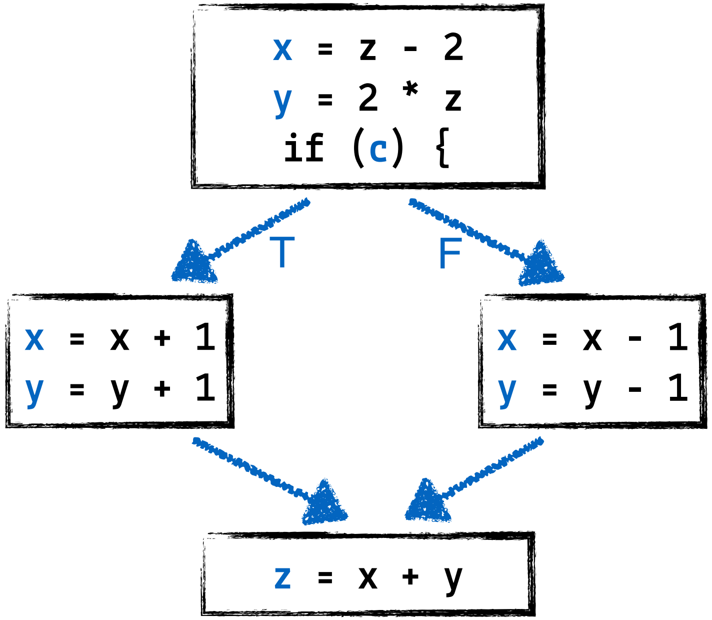
- Therefore, we contract them to ***basic blocks***
  - but remember that there are separate statements inside...



Branches end basic blocks

- This code needs multiple basic blocks:

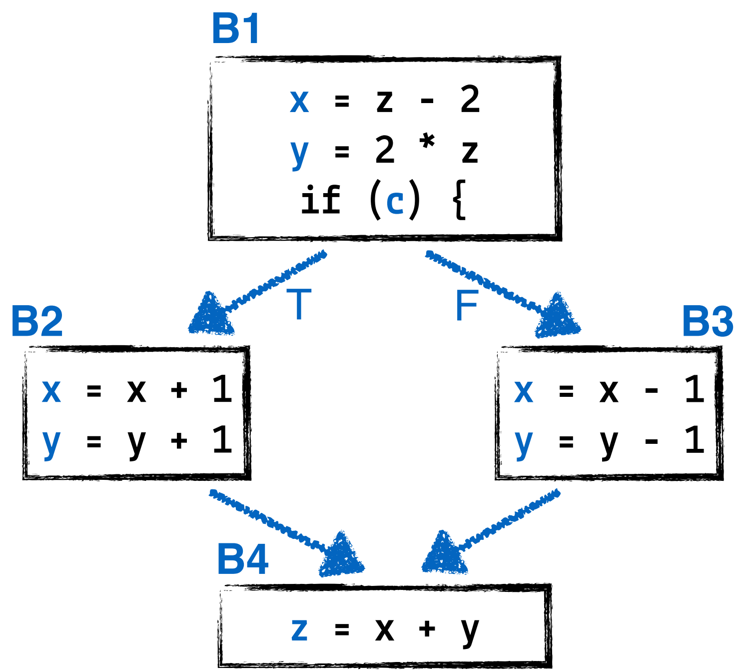
```
x = z - 2;
y = 2 * z;
if (c) {
   x = x + 1;
   y = y + 1;
} else {
   x = x - 1;
   y = y - 1;
}
z = x + y;
```



Multiple paths

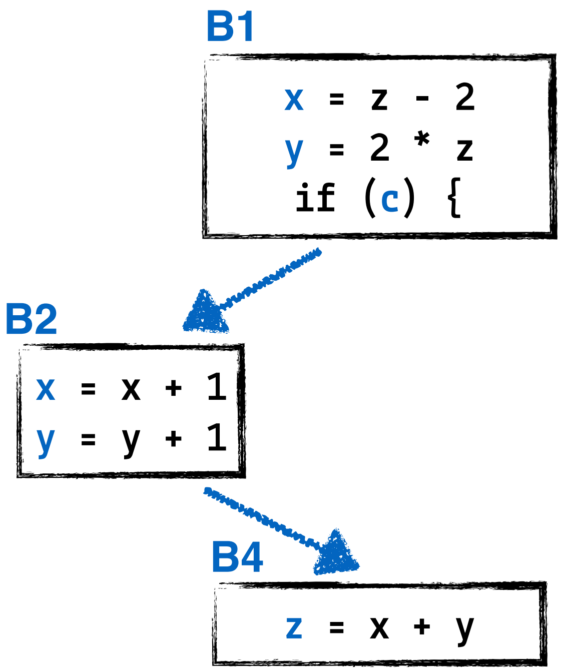
- Every possible execution is encoded in the CFG

- Each path corresponds to a run of the program

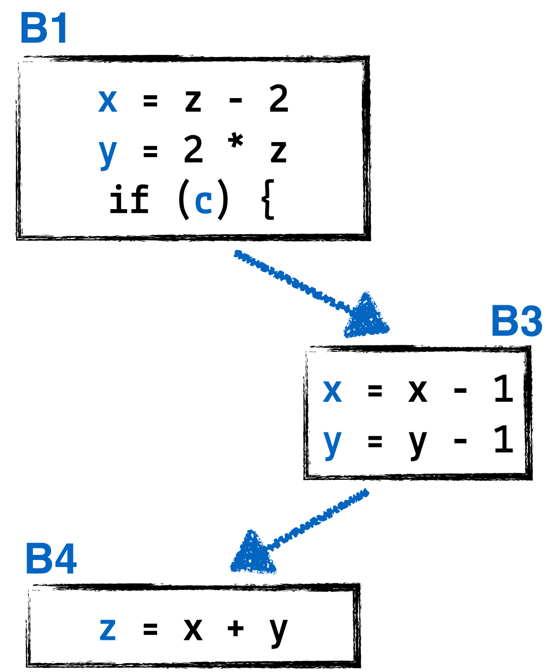


Choose your path…

When `c` is `true`:

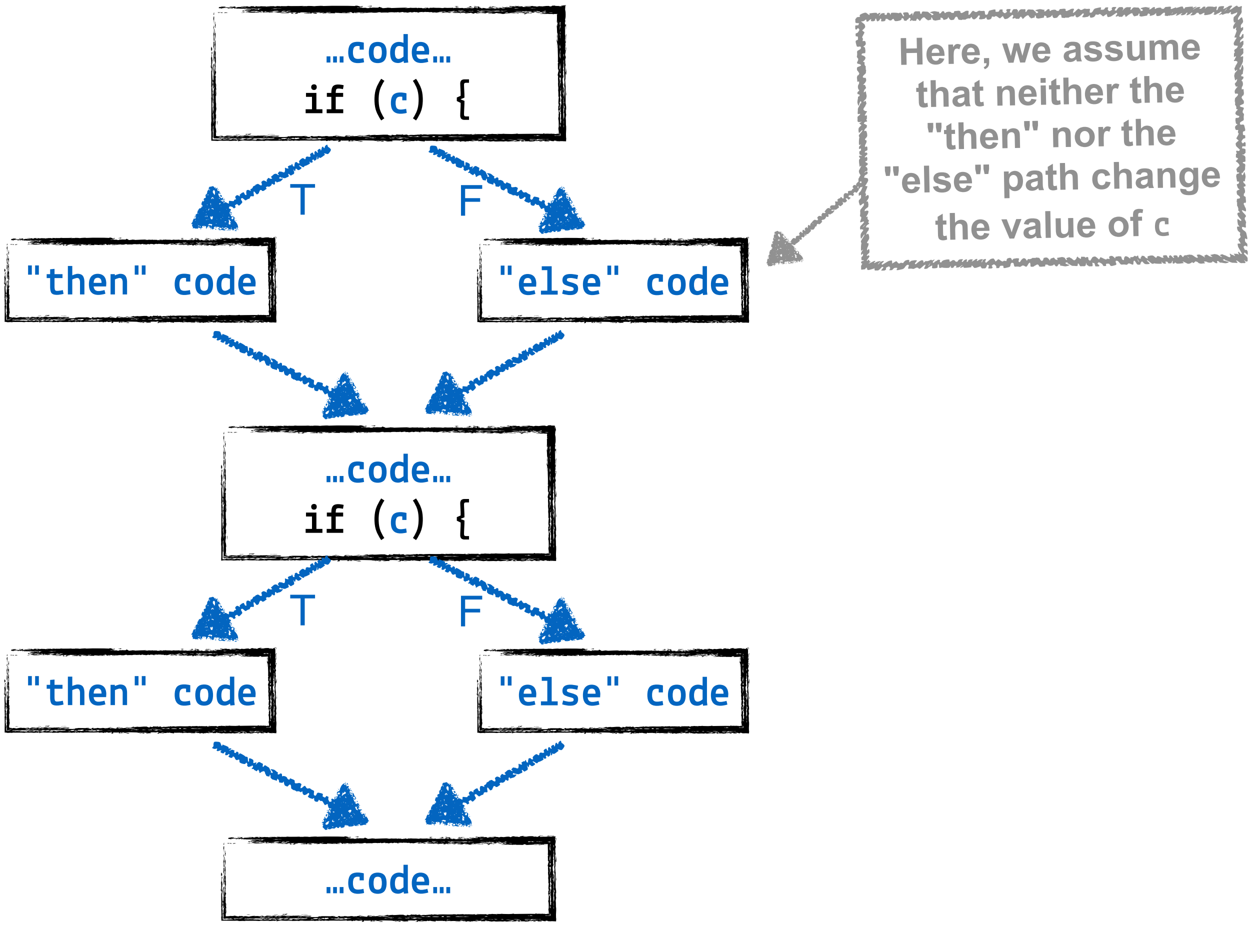


When `c` is `false`:



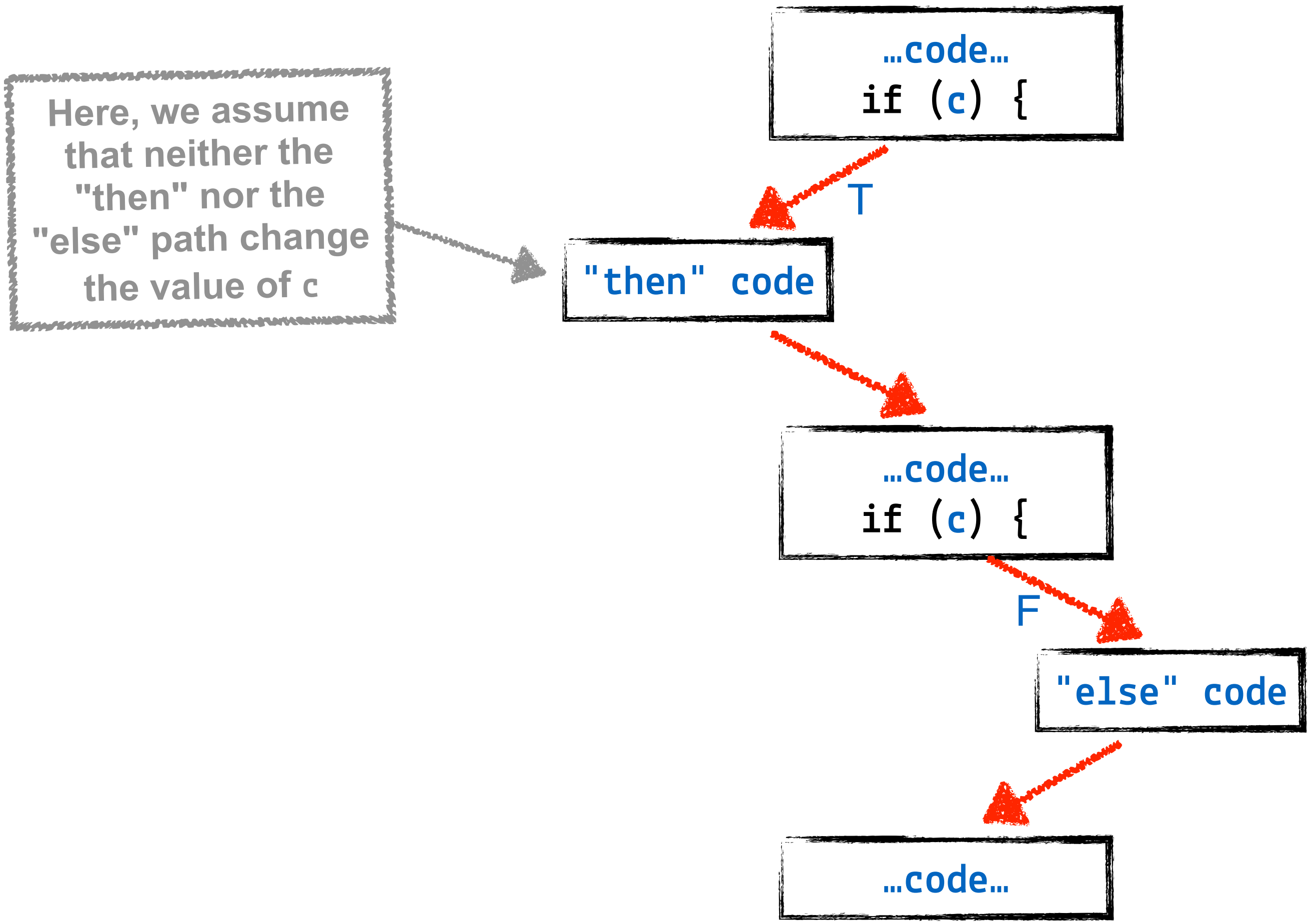
Infeasible executions

Some paths may not correspond to any possible run:



Infeasible executions

⇒ This path is ***infeasible***, even though it is part of the CFG!

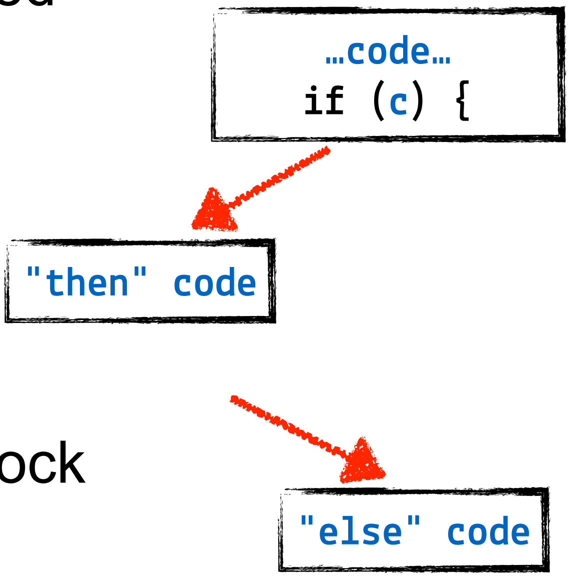


Interpretation of arcs

- Without pruning infeasible paths (which may require run-time information), the analysis will remain conservative/safe as long as every actual path is also represented

- Outgoing arcs mean that their destination ***may be*** a successor to a basic block

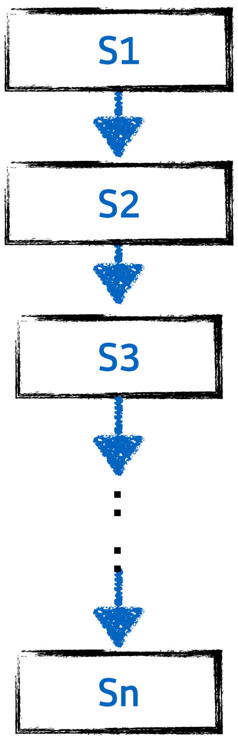
- Incoming arcs mean that any of the source blocks ***may be*** a predecessor to a basic block



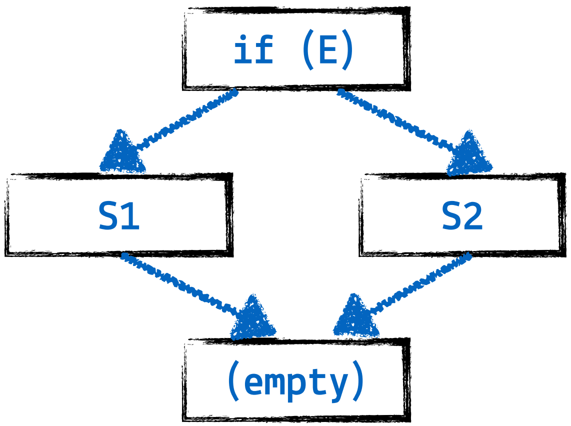
Recursive CFG construction

- At high level, CFGs can be built by a syntax directed scheme
  - Similar to our translation to TAC

`CFG(S1; S2; … ; Sn) =`

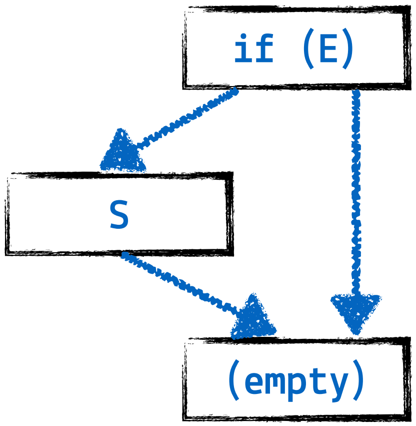


`CFG(if (E) S1 else S2) =`

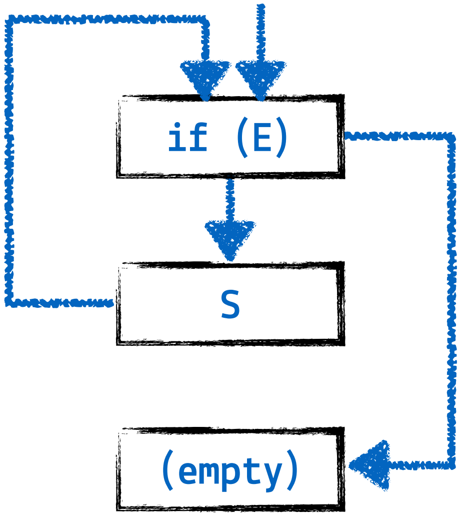


Recursive CFG construction: if/while

`CFG(if (E) S) =`



`CFG(while (E) S) =`

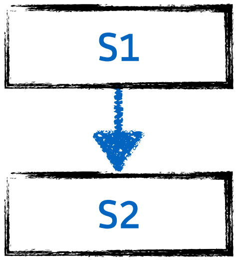


Recursive application

- We are analyzing statements recursively to refine our CFG:

```
while (c) {
  x = y +1;
  y = 2 * z;
  if (d) x = y + z;
  z = 1;
}
z = x;
```

(S1; S2)

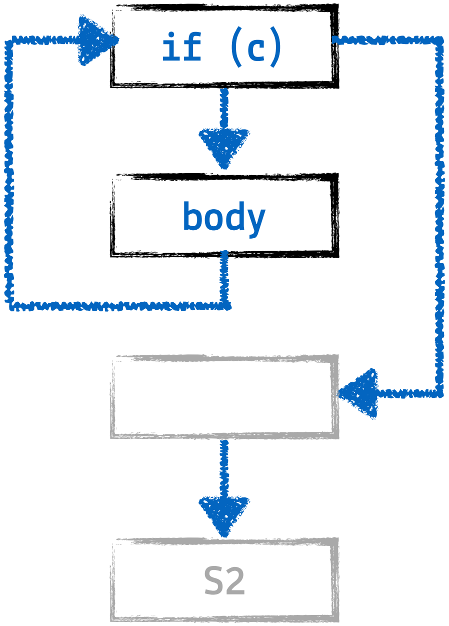


Recursive application

- We are analyzing statements recursively to refine our CFG:

```
while (c) {
    x = y +1;
    y = 2 * z;
    if (d) x = y + z;
    z = 1;
}
z = x;
```

(while)

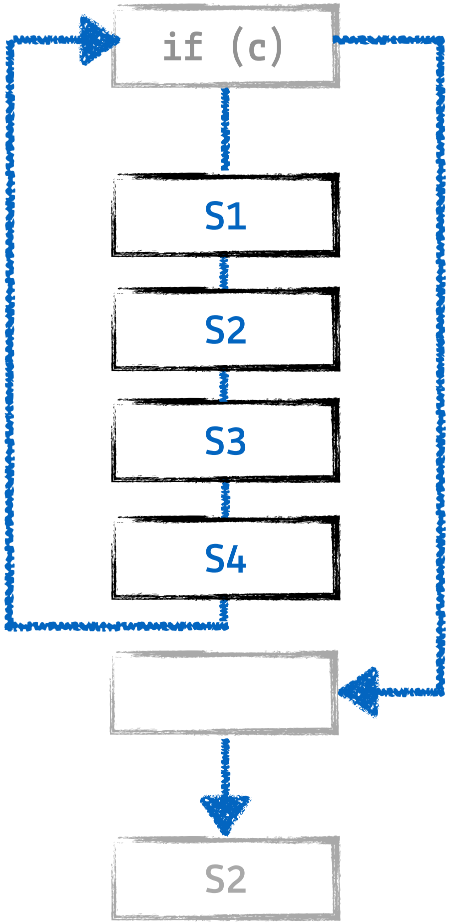


Recursive application

- We are analyzing statements recursively to refine our CFG:

```
while (c) {
    x = y +1;
    y = 2 * z;
    if (d) x = y + z;
    z = 1;
}
z = x;
```

(S1; S2; S3; S4)

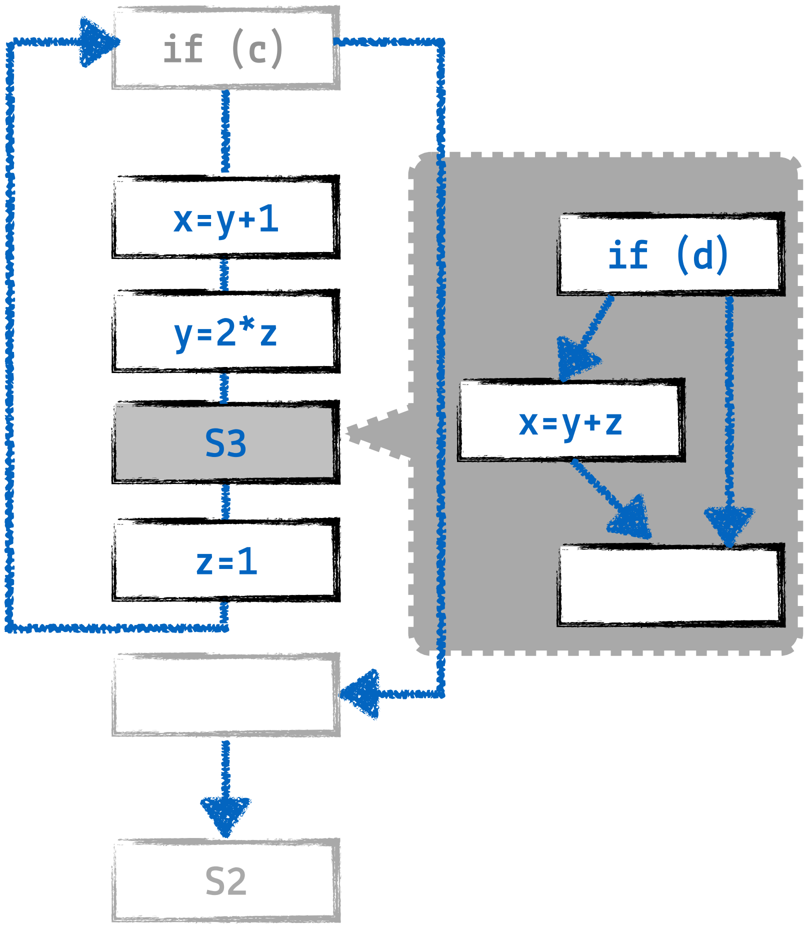


Recursive application

- We are analyzing statements recursively to refine our CFG:

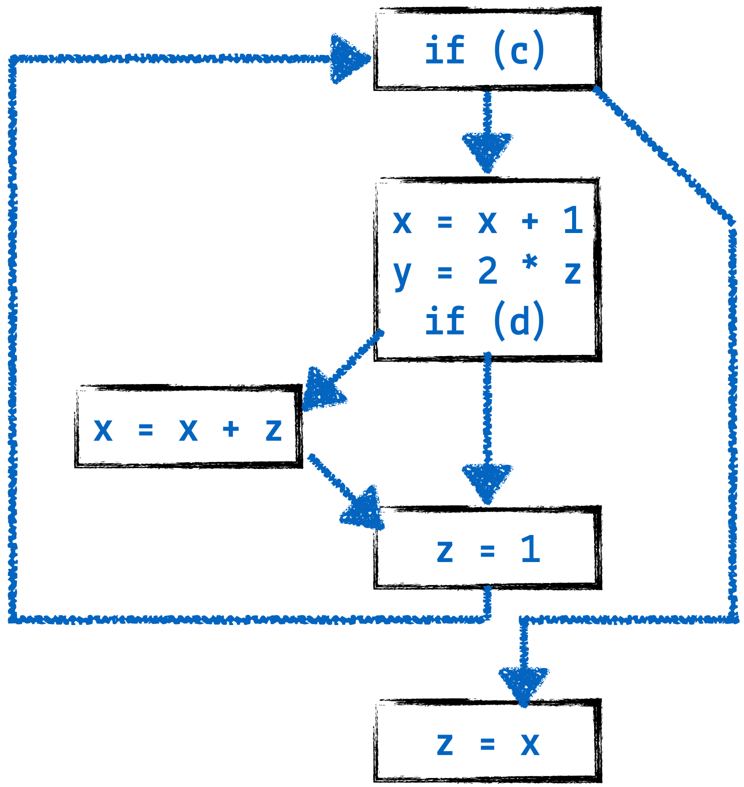
```
while (c) {
    x = y +1;
    y = 2 * z;
    if (d) x = y + z;
    z = 1;
}
z = x;
```

(S1; S2; S3; S4)



Efficiency

- Empty blocks and sequences can be pruned after or during construction of the CFG



Efficiency

- These graphs grow large
  - It’s good to have as few basic blocks as possible
  - They should be as large as possible

- Merge linear subgraphs if
  - B2 is a successor of B1
  - B1 has one outgoing edge
  - B2 has one incoming edge
  - B1→B2 should be a block

- Remove empty blocks

At low-level IR

- Split the operation sequence at labels and jumps
    - Labels can have incoming control flow
    - Jumps have outgoing control flow

```
L1:
    ifFalse (c) jump L2
    x = y + 1
    y = 2 * z
    ifFalse (d) jump L3
    x = y + z
L3:
    z = 1
    jump L1
L2:
    z = x
```

```
L1:
    ifFalse (c) jump L2
```

```
    x = y + 1
    y = 2 * z
    ifFalse (d) jump L3
```

```
    x = y + z
```

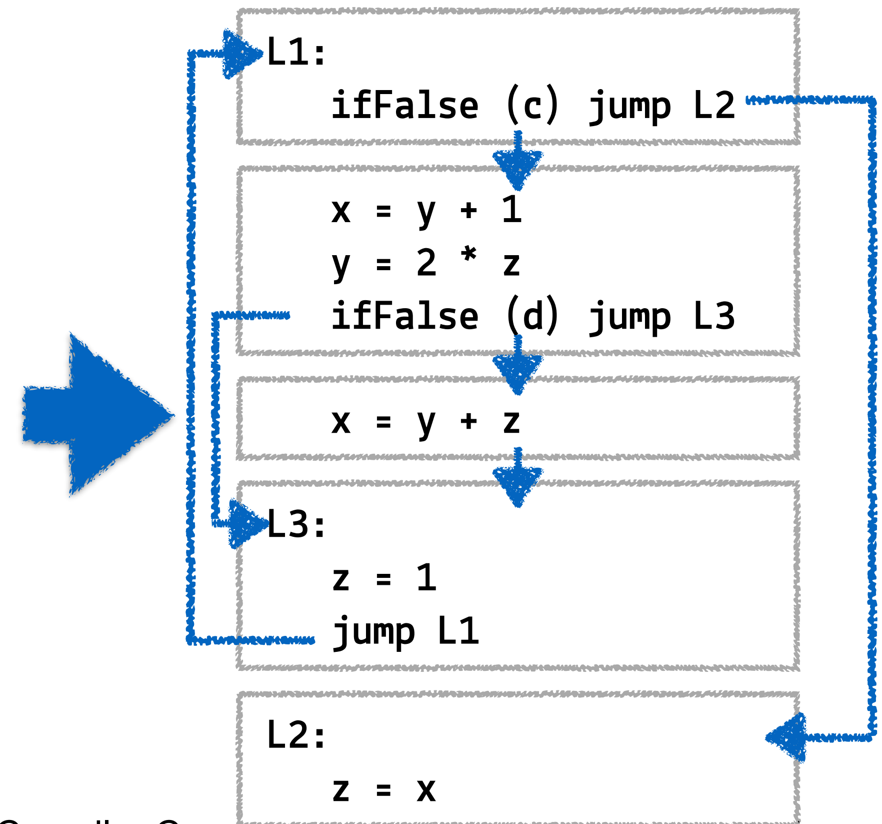
```
L3:
    z = 1
    jump L1
```

```
L2:
    z = x
```

At low-level IR

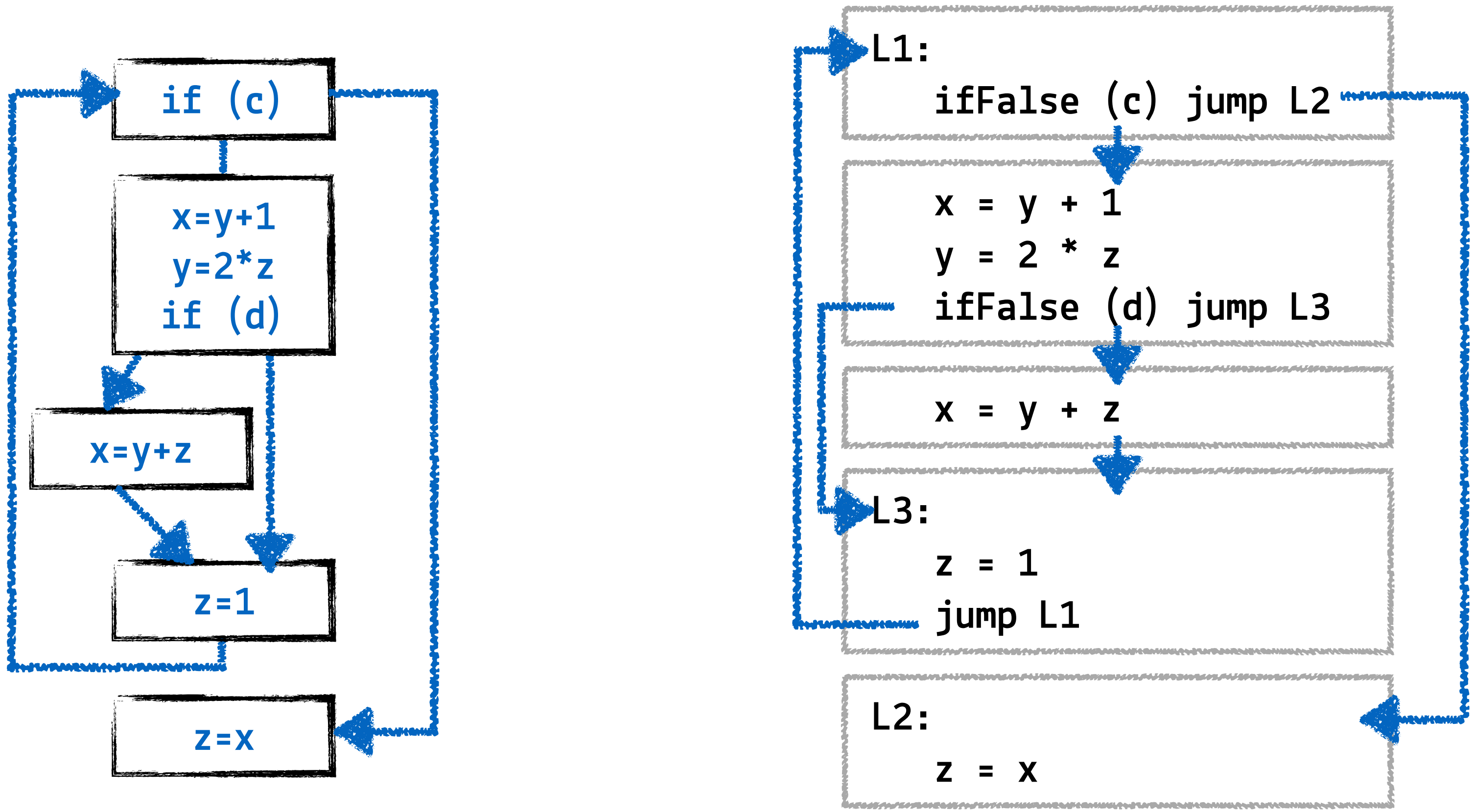
- Conditional jump = 2 successors
- Unconditional jump = 1 successor

```
L1:
   ifFalse (c) jump L2
   x = y + 1
   y = 2 * z
   ifFalse (d) jump L3
   x = y + z
L3:
   z = 1
   jump L1
L2:
   z = x
```



The outcome is the same

- Both procedures give us the equivalent program logic:



Live variables and the CFG

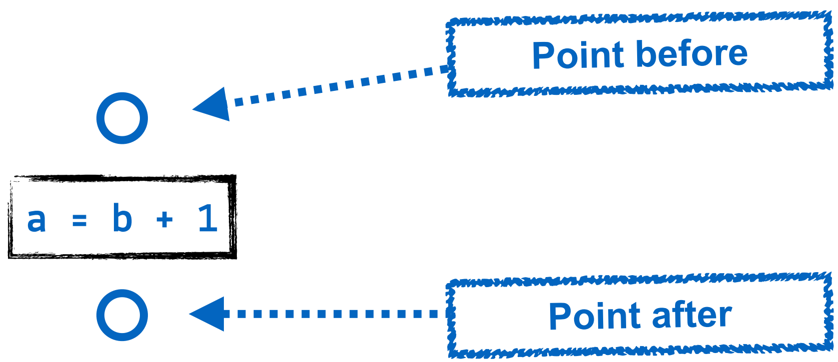
- The purpose of using the CFG is to statically extract information about the program at compile time
- Reasoning about the run-time values of variables and expressions in every possible execution enables optimizations
- We can illustrate this by finding ***live variables***

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

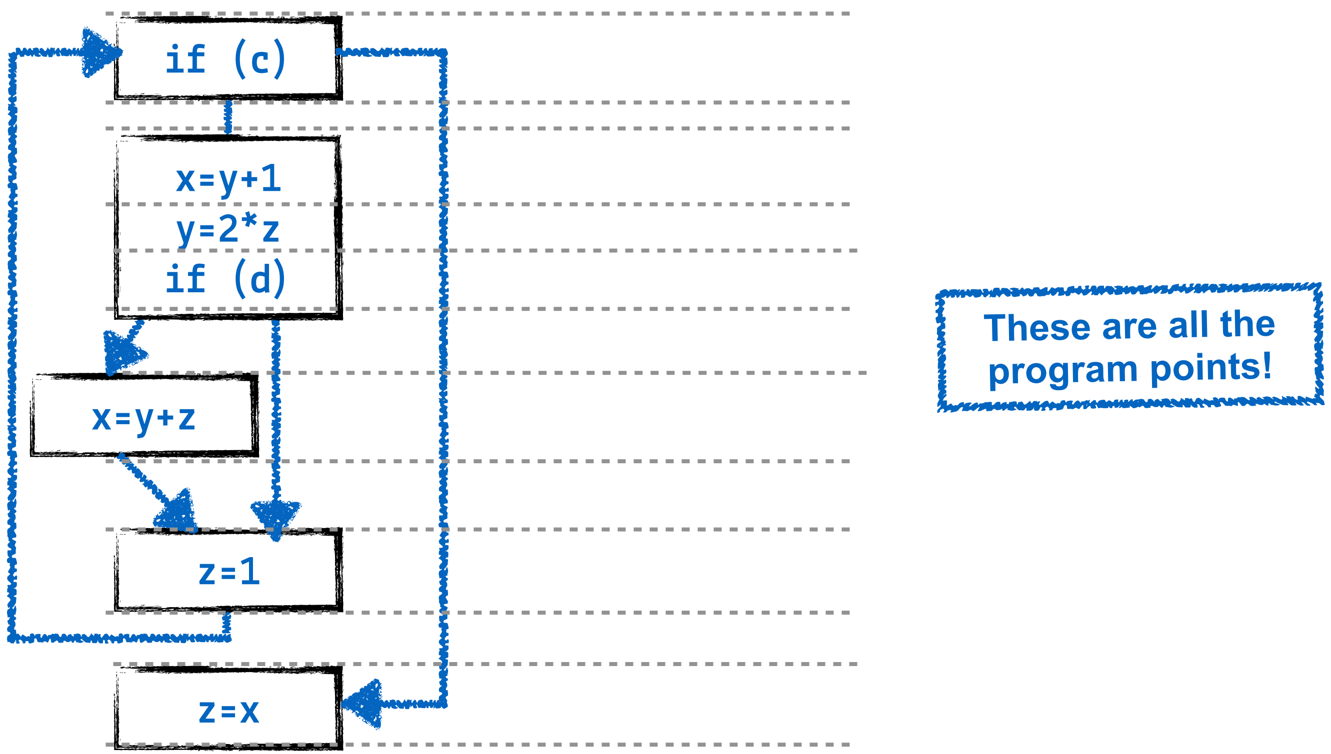
- As we want to capture how state is changed through an instruction, we need to talk about the state before and the state after, and describe the difference
- Hence, there is one program point before and one after each instruction



- For basic blocks, these are the points
  - after the predecessor(s)
  - before the successor(s)

Program points in our previous ex.

- We mark the before and after points with dashed lines here:



Two things to consider

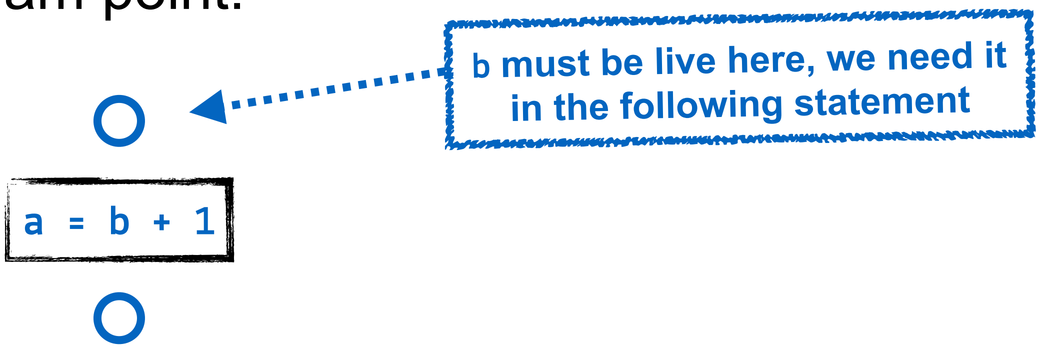
- How does an instruction affect the state at the points immediately before and after it?
    - In other words, what is the effect of an instruction?

- How does state propagate between program points?
    - In other words, what is the effect of control flow?

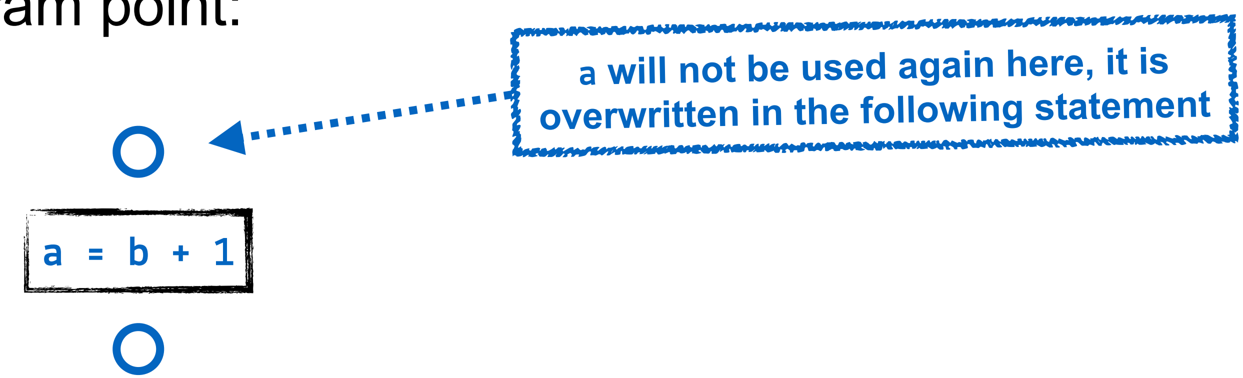
- If we can tell which variables are life at one point, we can compute which ones are live by its neighbors

Which instructions affect liveness?

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



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



Doing it systematically

- 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 `I1` and `I2` are the first and last instructions in `B`,
  - `in[B] = in[I1]`
  - `out[B] = out[I2]`

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
- So,

  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[I1] + in[I2]`

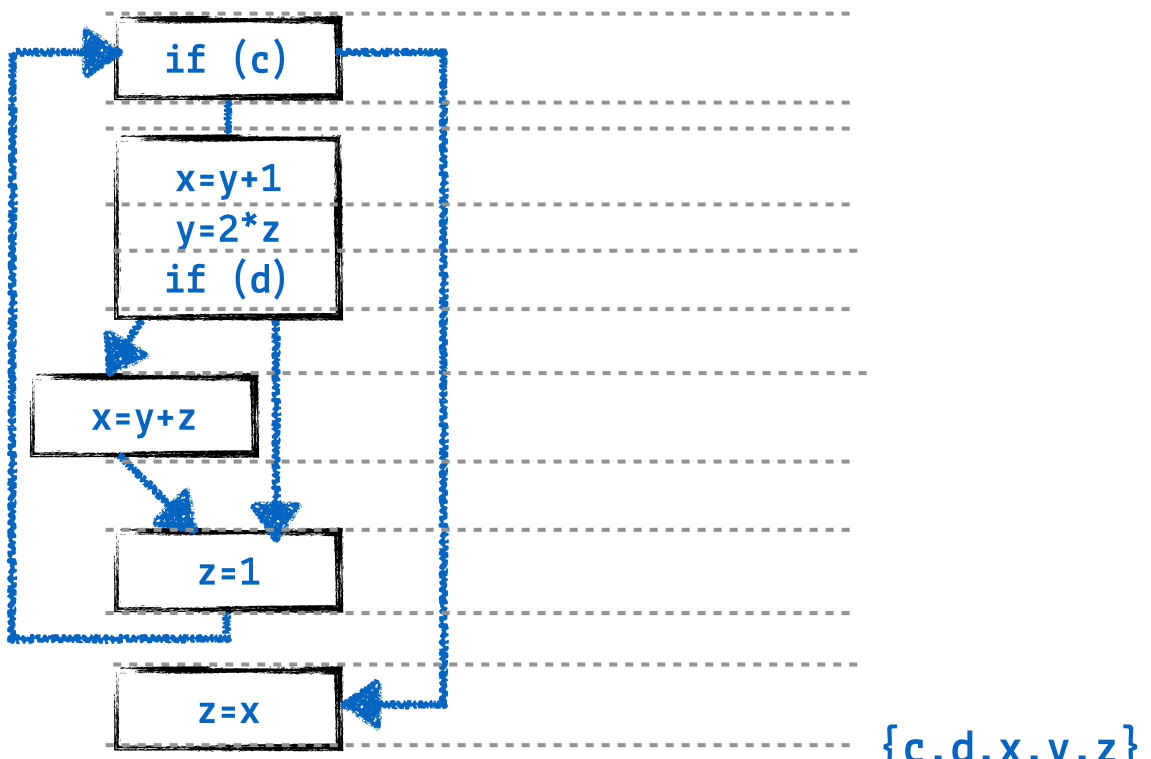
where `I1`, `I2` are successors of `I`

Liveness flows backwards

- We define the in-sets in terms of the out-sets
- This means we need out-sets to start our analysis
- In the name of safety, assume that every variable is live until it has been determined otherwise
- This results in a final out-state to start working from, so that we can ***examine the CFG in reverse***

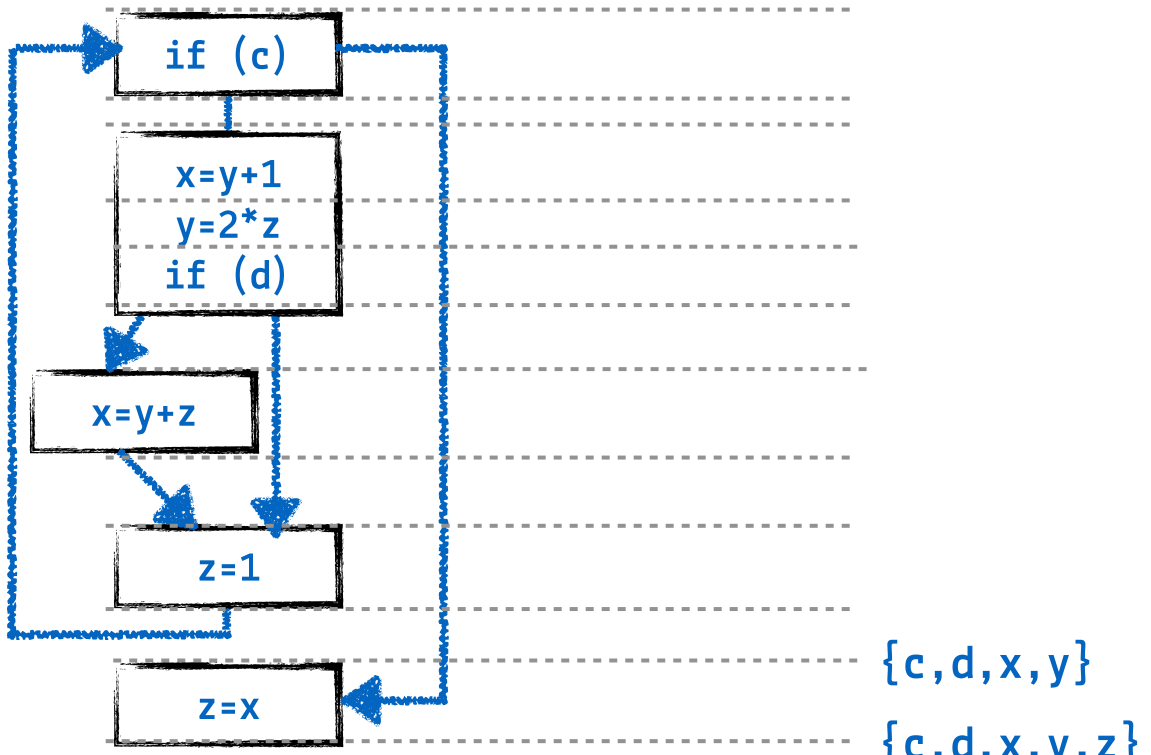
Start at the end…

- Conservative assumption: everything is live at the end



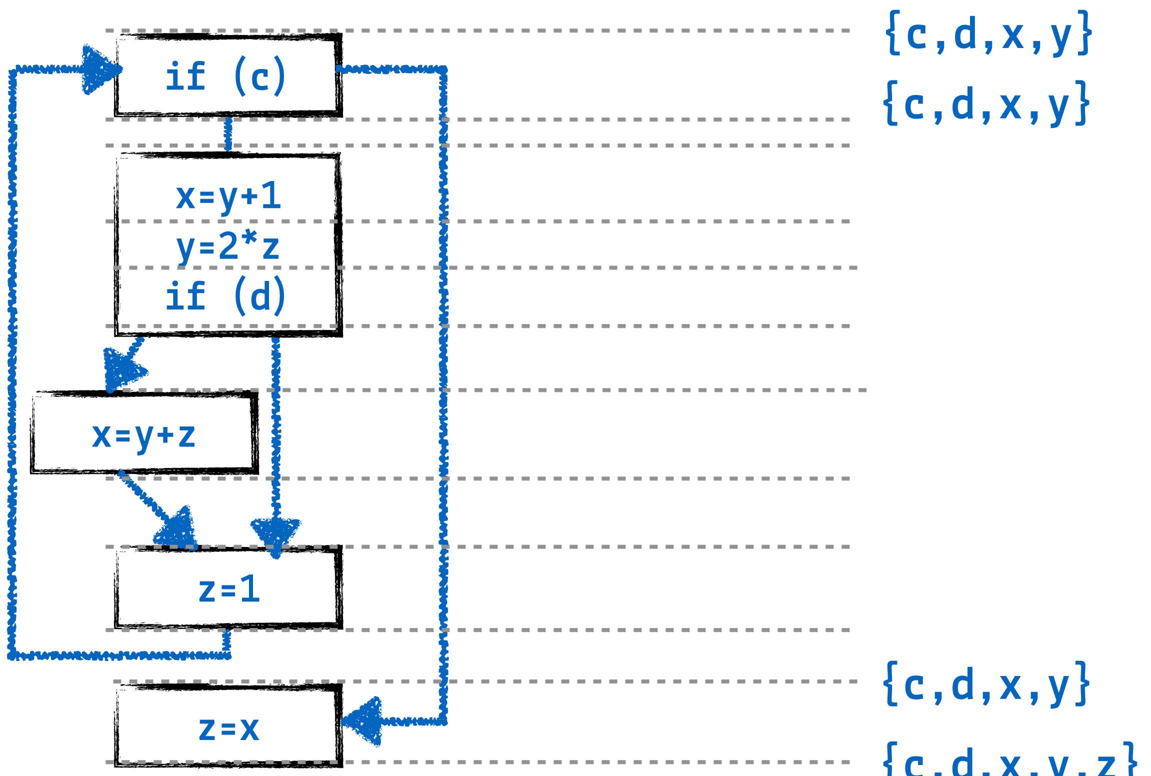
Iteration 1

- The last statement defines `z`



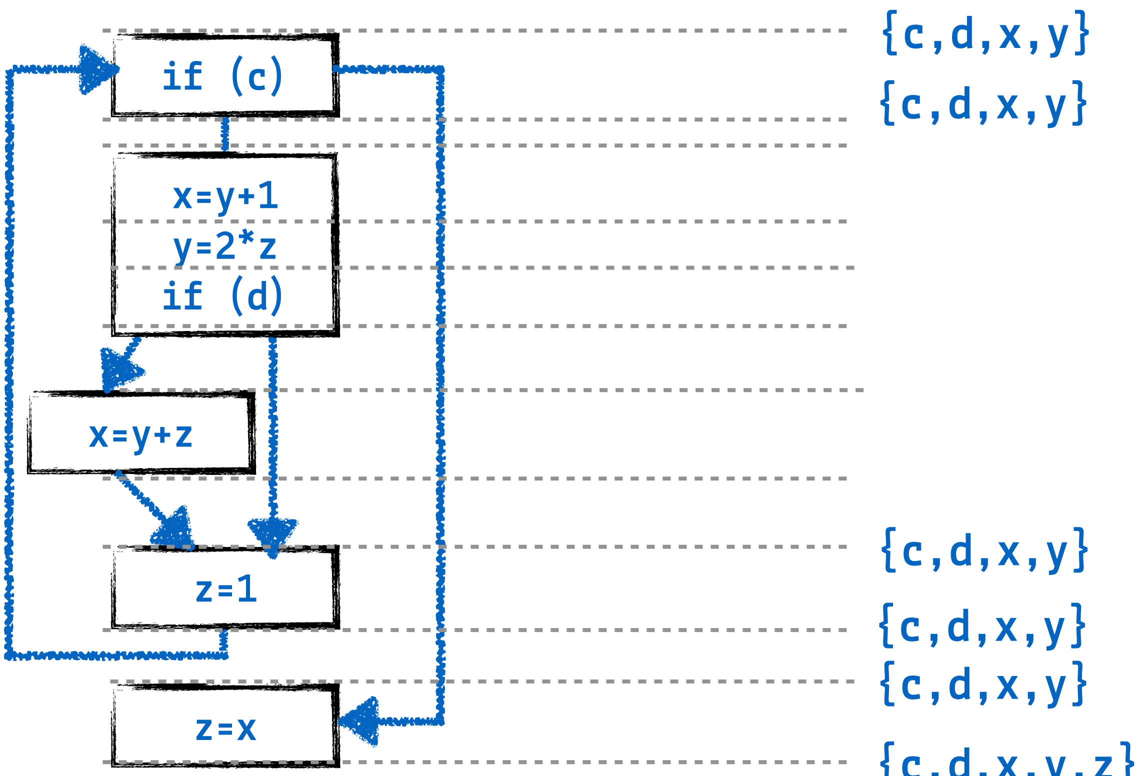
Iteration 1

- Its predecessor doesn't define anything



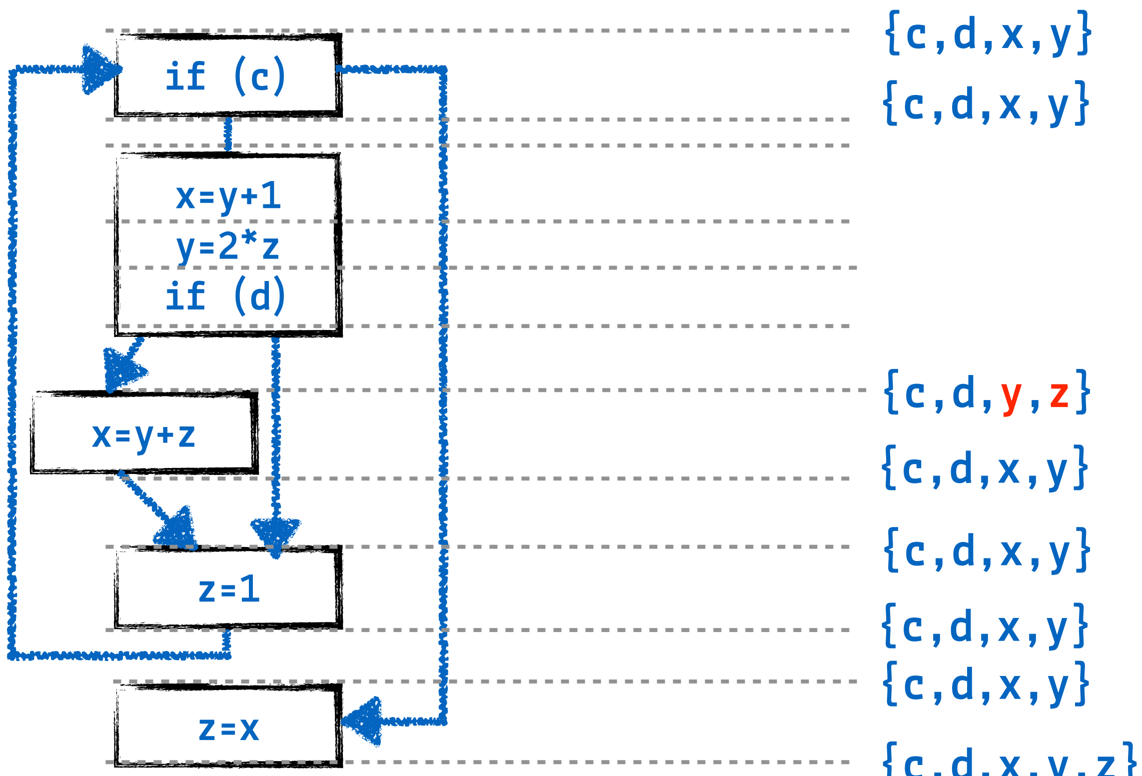
Iteration 1

- Predecessor defines `z`, but it wasn't live anyway



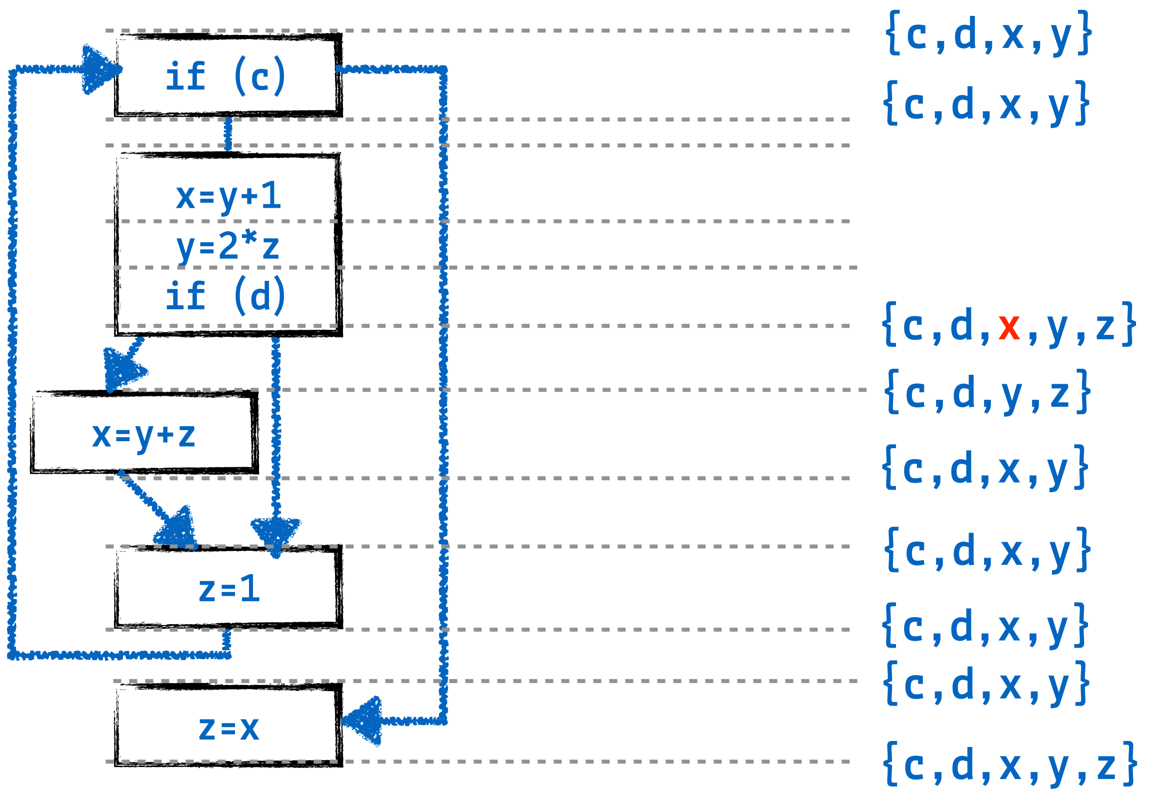
Iteration 1

- Predecessor ***uses*** `z`, it is live again



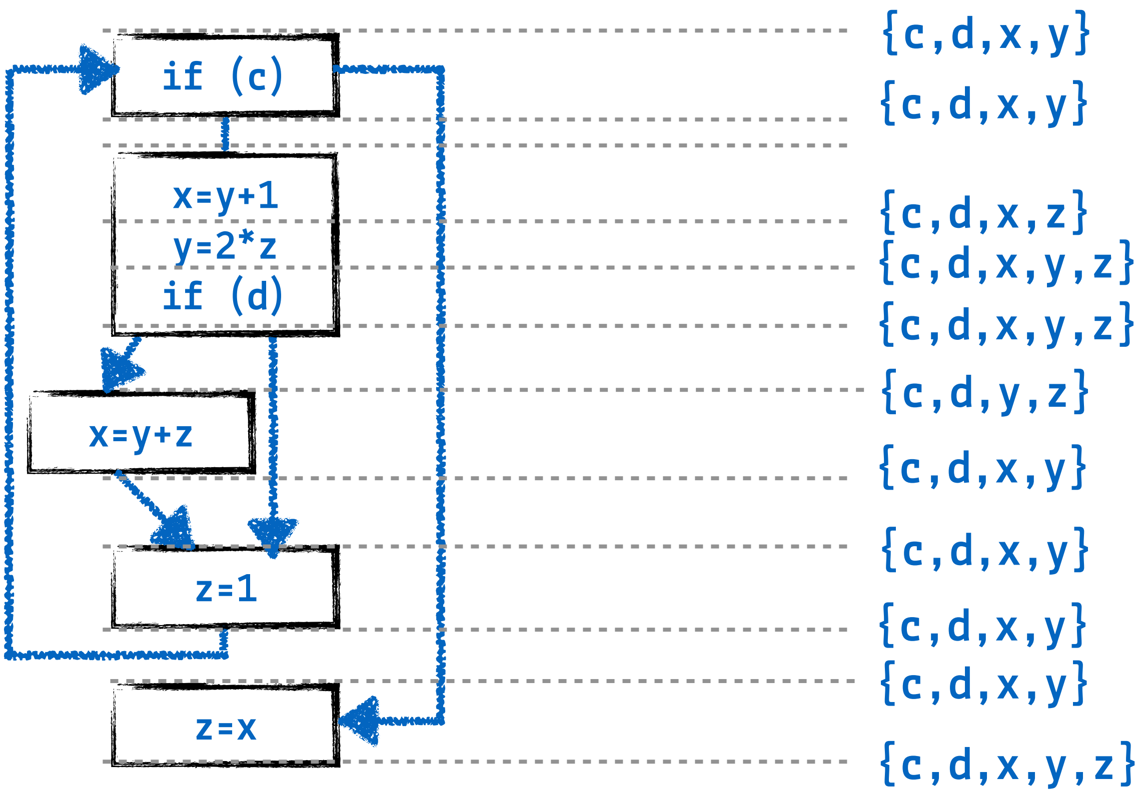
Iteration 1

- Predecessor of two successors (control flow): must assume ***union*** of the live variables of each successor



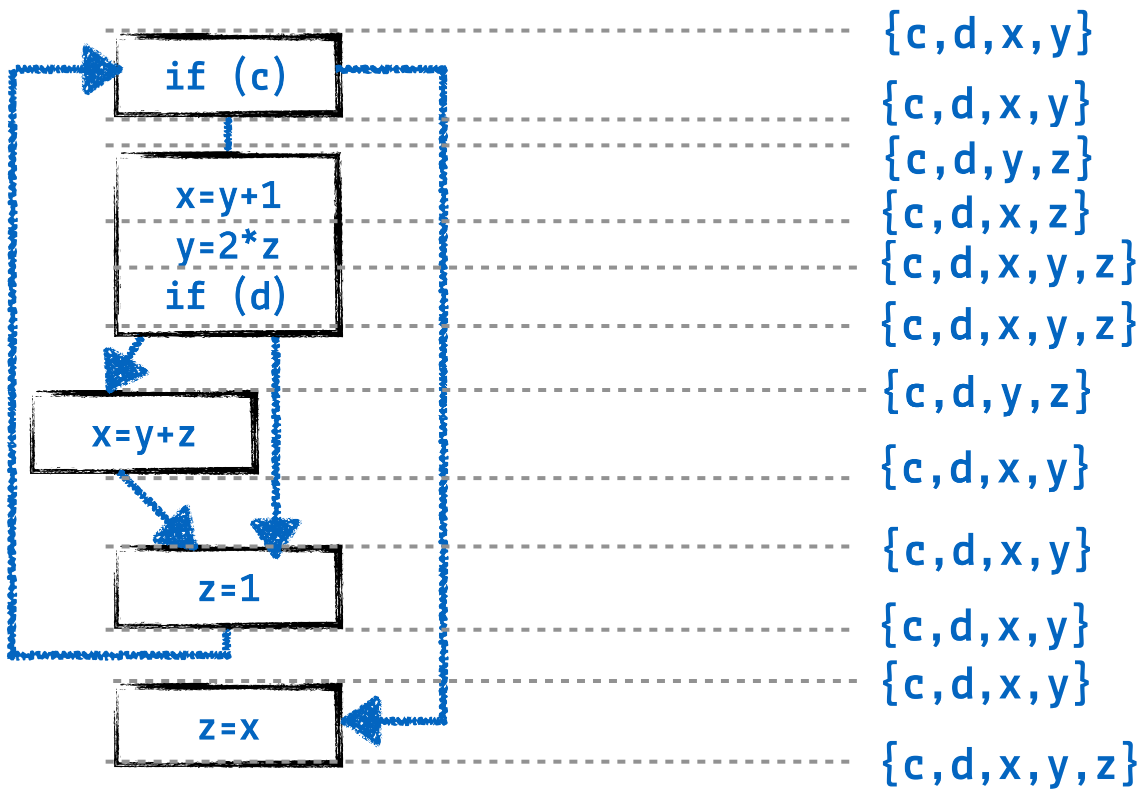
Iteration 1

- Definition of y



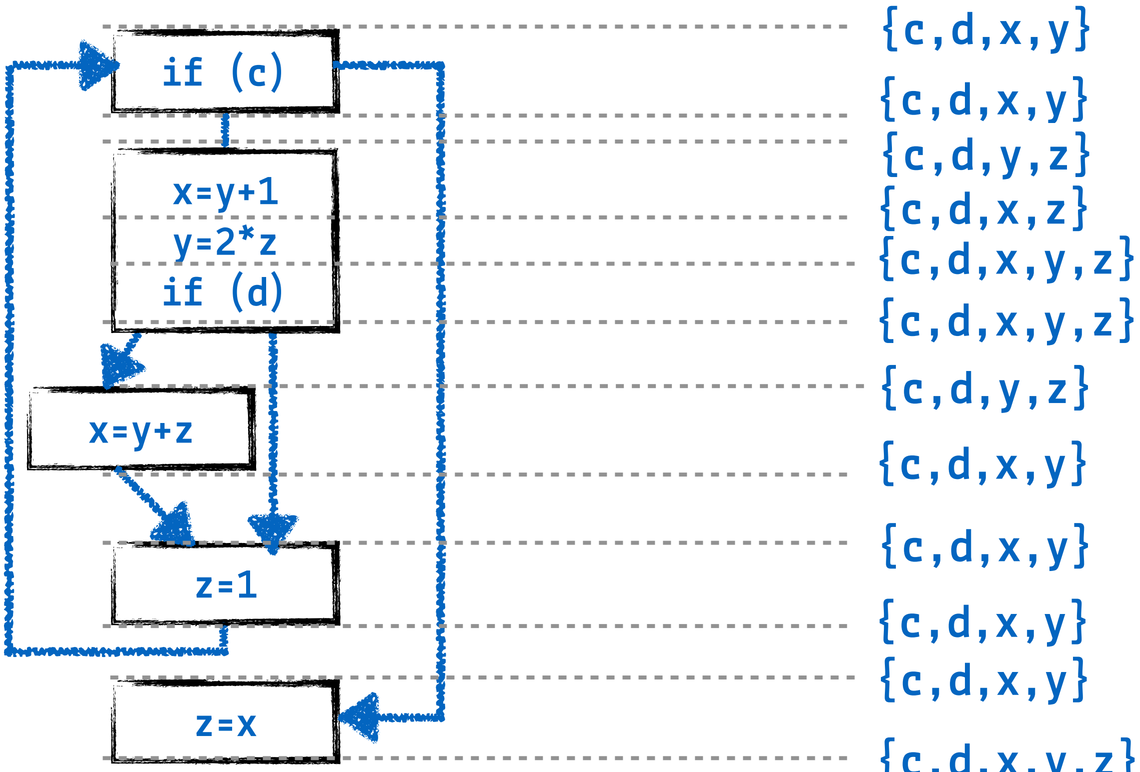
Iteration 1

- Use of y, definition of z



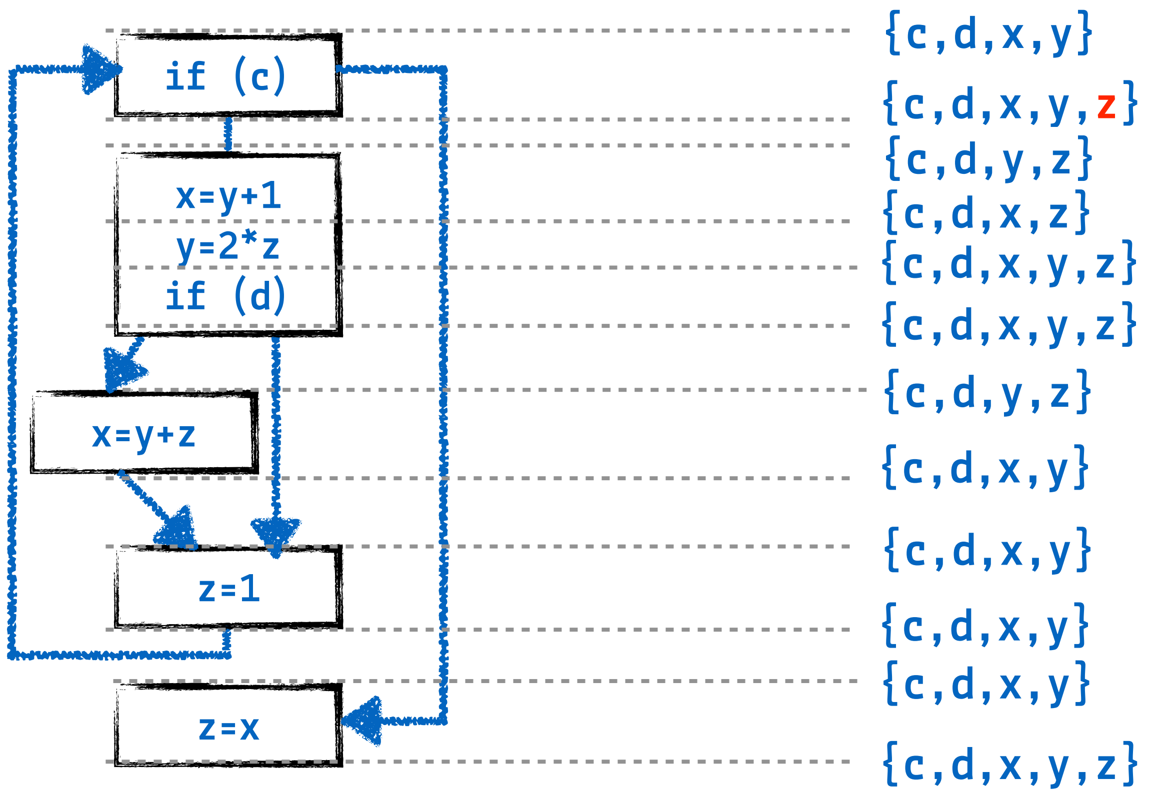
End of iteration 1

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



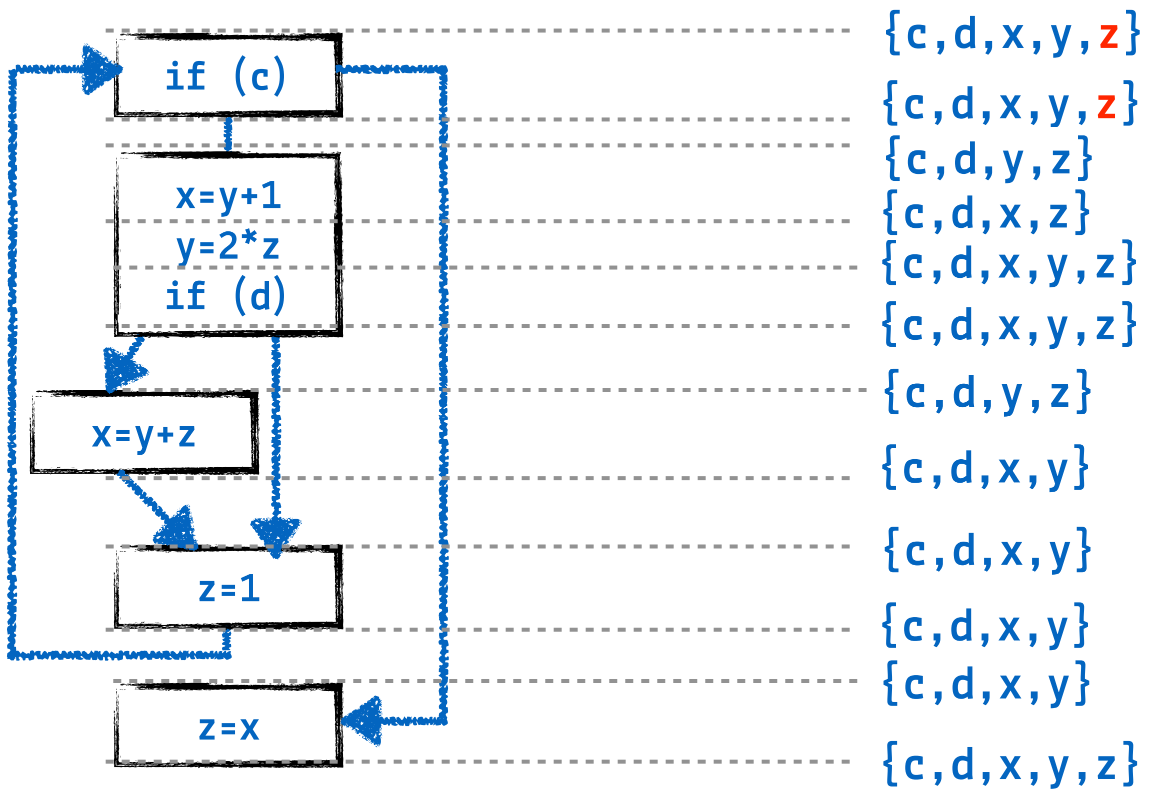
Iteration 2

- The ***union*** of the two successors here is different



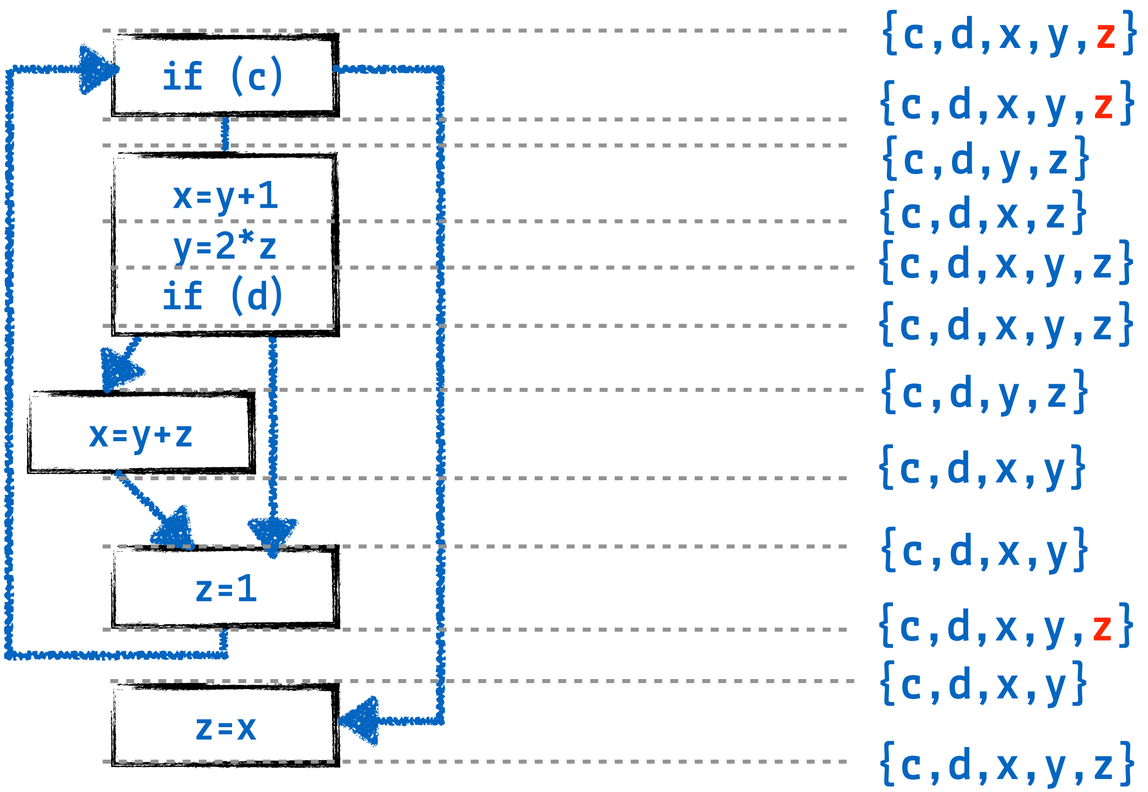
Iteration 2

- Propagate it to the predecessor



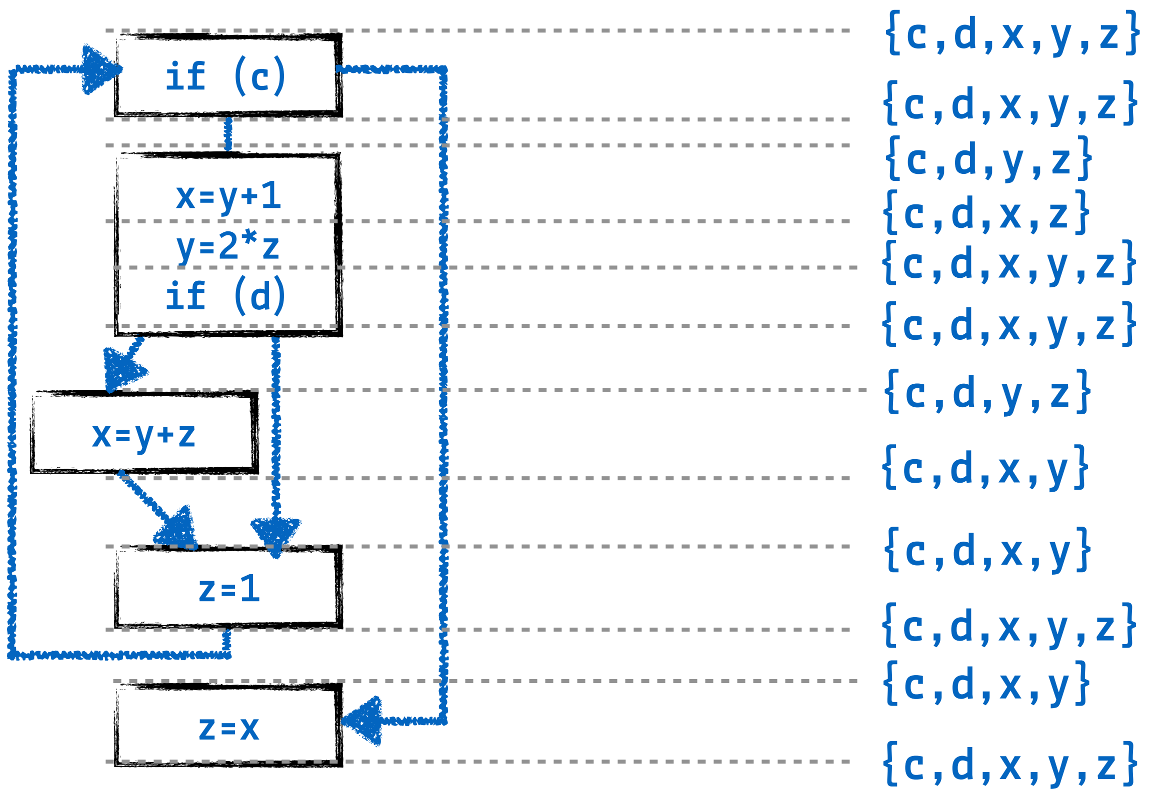
Iteration 2

- ...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 simply uses 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 next week

What's next?

- Optimizations in detail: data-flow analyses

References

[1] Frances E. Allen. 1970.
Control flow analysis. SIGPLAN Not. 5, 7 (July 1970), 1–19.
DOI:https://doi.org/10.1145/390013.808479