CS412/CS413

Introduction to Compilers

Tim Teitelbaum

Lecture 25: Liveness and Copy Propagation

26 March 08

Control Flow Graphs

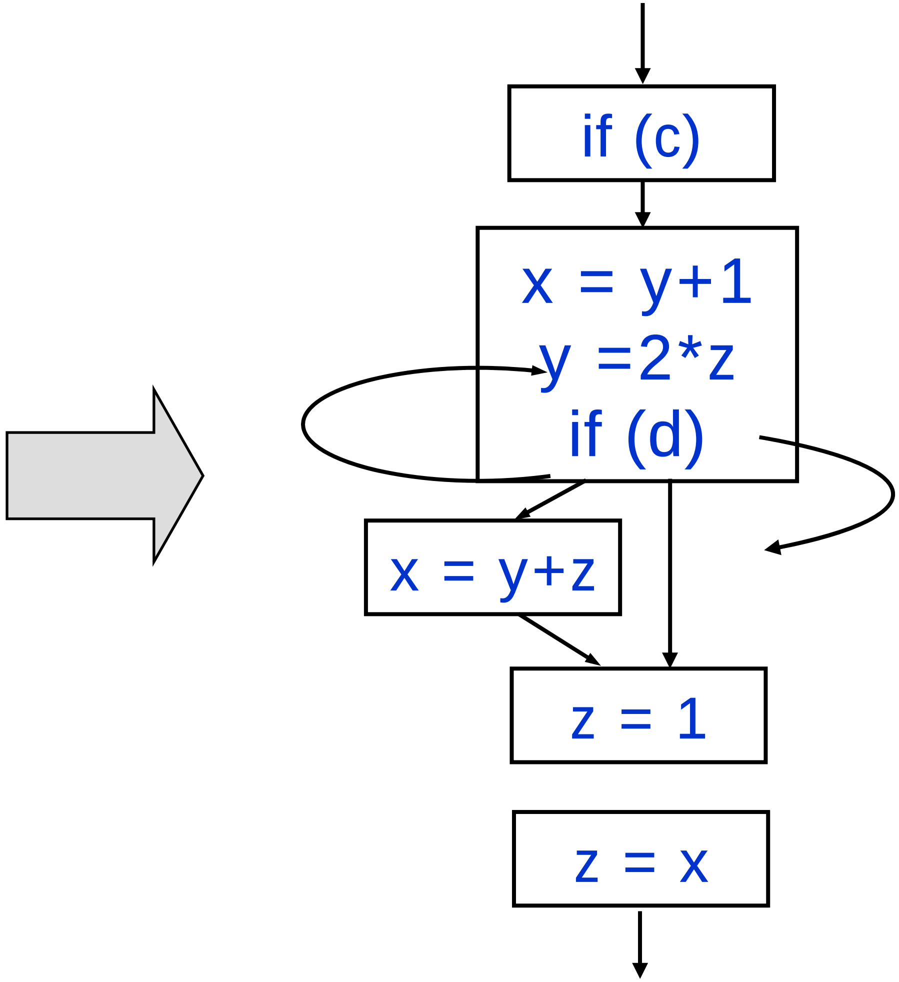
- Control Flow Graph (CFG) = graph representation of computation and control flow in the program
  – framework to statically analyze program control-flow

- In a CFG:
  – Nodes are basic blocks; they represent computation
  – Edges characterize control flow between basic blocks

- Can build the CFG representation either from the high IR or from the low IR

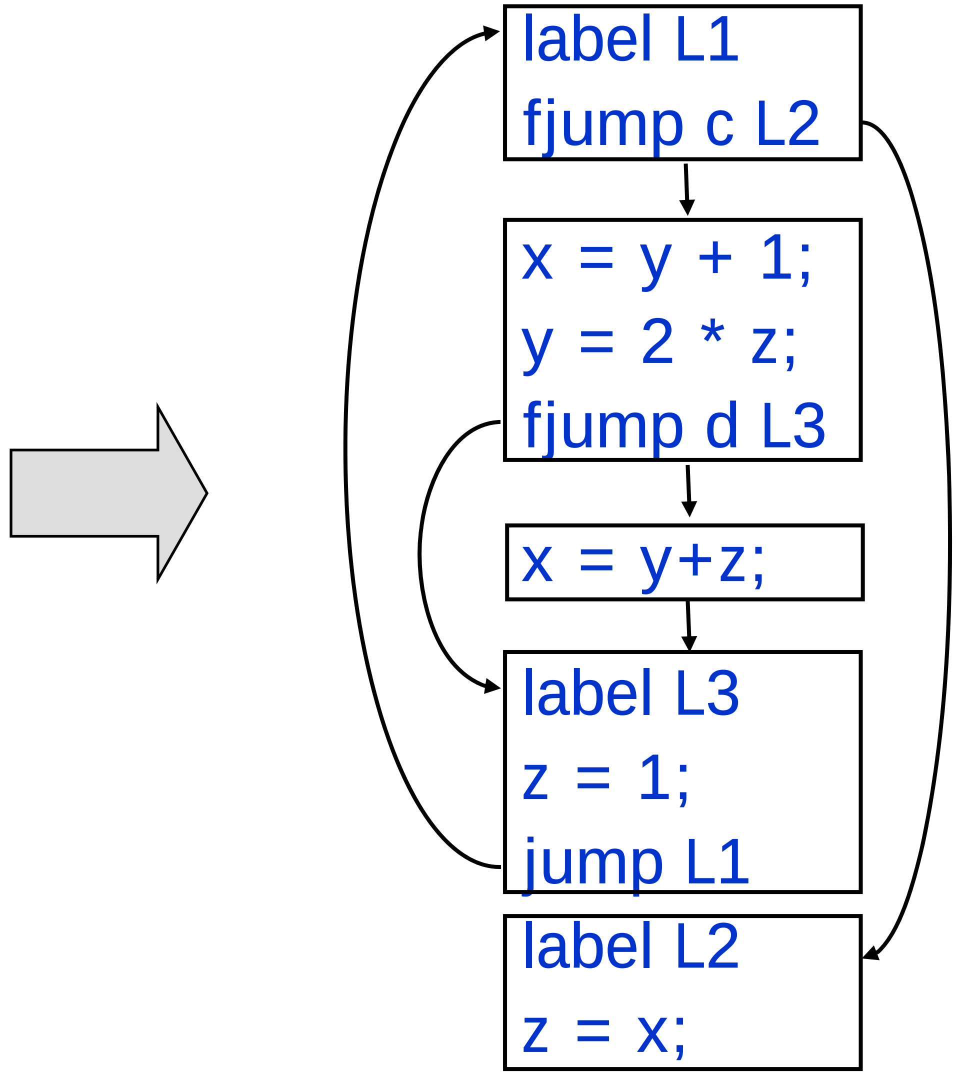
Build CFG from High IR

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



Build CFG from Low IR

```
label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y+z;
label L3
z = 1;
jump L1
label L2
z = x;
```

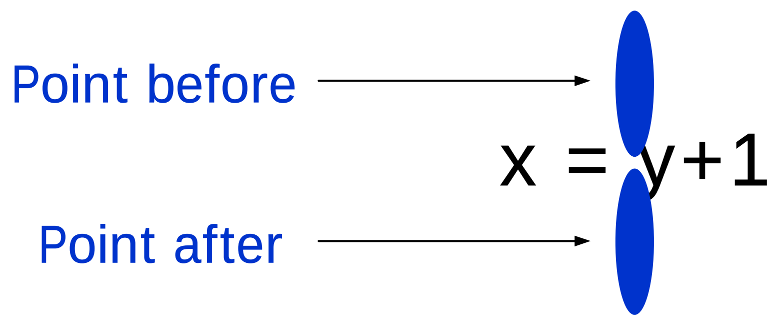


Using CFGs

- Next: use CFG representation to statically extract information about the program
  - Reason at compile-time
  - About the run-time values of variables and expressions in all program executions
- Extracted information example: live variables
- Idea:
  - Define program points in the CFG
  - Reason statically about how the information flows between these program points

Program Points

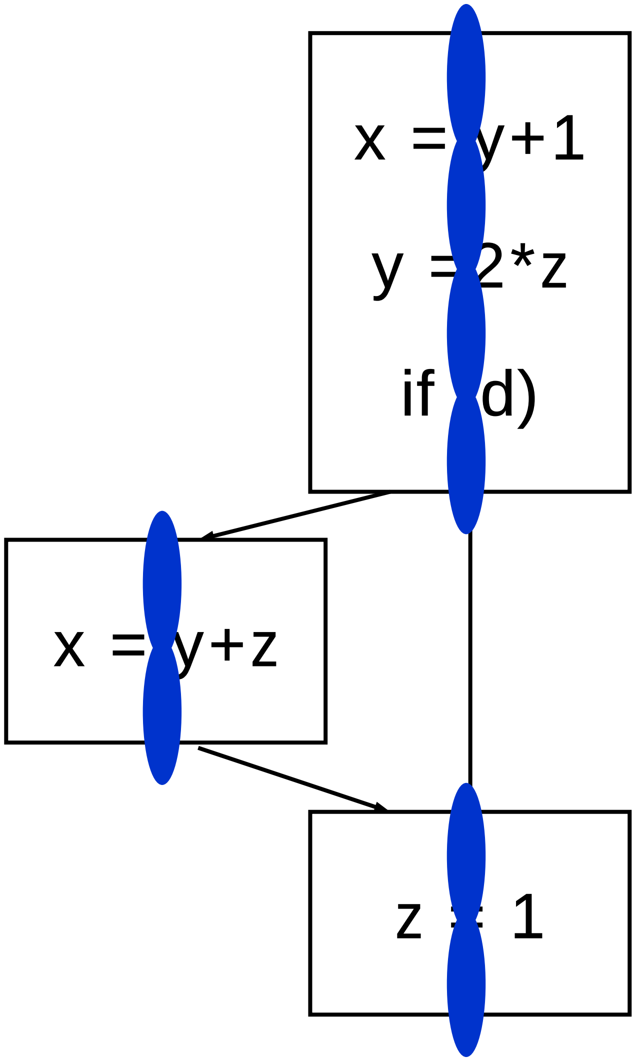
- Two program points for each instruction:
  - There is a program point before each instruction
  - There is a program point after each instruction



- In a basic block:
  - Program point after an instruction = program point before the successor instruction

Program Points: Example

- Multiple successor blocks means that point at the end of a block has multiple successor program points
- Depending on the execution, control flows from a program point to one of its successors
- Also multiple predecessors
- How does information propagate between program points?

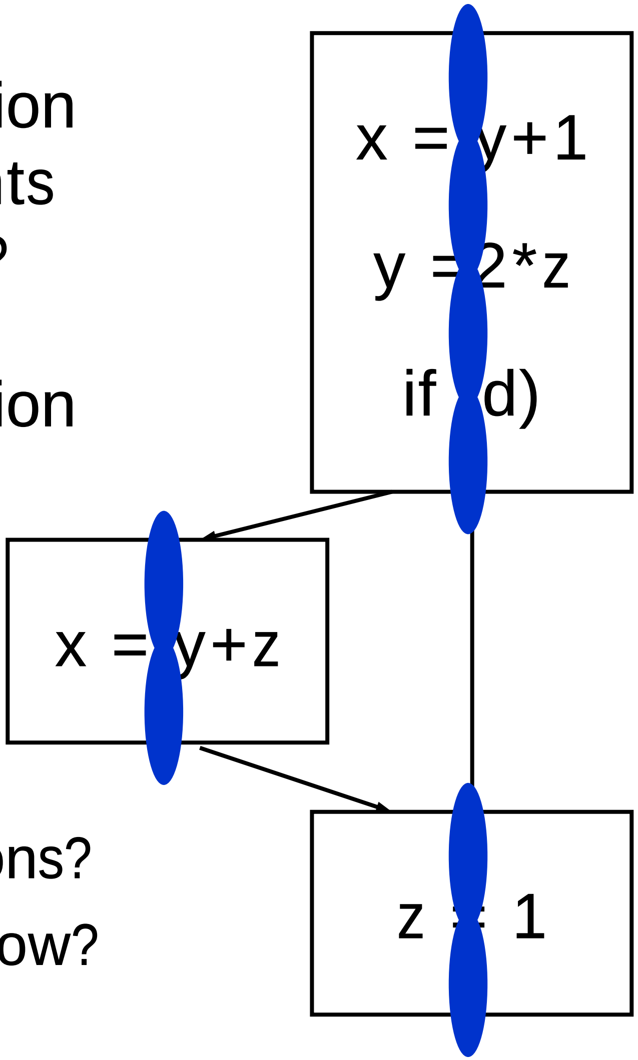


Flow of Extracted Information

- Question 1: how does information flow between the program points before and after an instruction?
- Question 2: how does information flow between successor and predecessor basic blocks?
- ... in other words:

  Q1: what is the effect of instructions?

  Q2: what is the effect of control flow?



Using CFGs

- To extract information: reason about how it propagates between program points

- Rest of this lecture: how to use CFGs to compute information at each program point for:
  - Live variable analysis, which computes which variables are live at each program point
  - Copy propagation analysis, which computes the variable copies available at each program point

Live Variable Analysis

- Computes live variables at each program point
  - I.e., variables holding values that may be used later (in some execution of the program)

- For an instruction I, consider:
  - in[I] = live variables at program point before I
  - out[I] = live variables at program point after I

- For a basic block B, consider:
  - in[B] = live variables at beginning of B
  - out[B] = live variables at end of B

- If I = first instruction in B, then in[B] = in[I]
- If I' = last instruction in B, then out[B] = out[I']

How to Compute Liveness?

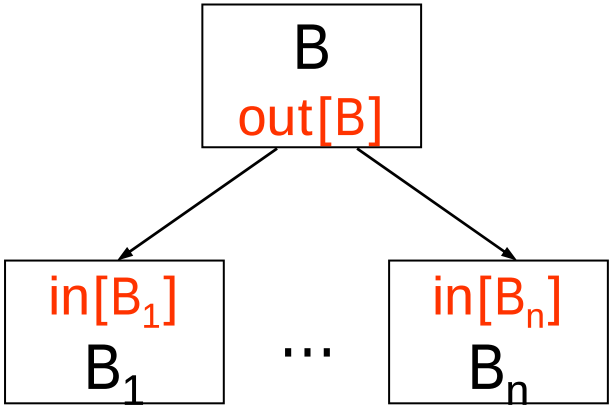
- Answer question 1: for each instruction I, what is the relation between in[I] and out[I] ?

in[I]

I

out[I]

- Answer question 2: for each basic block B with successor blocks B_1, ..., B_n, what is the relation between out[B] and in[B_1], ..., in[B_n]?



Part 1: Analyze Instructions

- Question: what is the relation between sets of live variables before and after an instruction?

in[I]
I
out[I]

- Examples:

conclude	in[I] = {y,z}	in[I] = {y,z,t}	in[I] = {x,t}
	x = y+z;	x = y+z;	x = x+1;
assume	out[I] = {z}	out[I] = {x,t}	out[I] = {x,t}

- ... is there a general rule?

Analyze Instructions

- Yes: knowing variables live after I, can compute variables live before I:
  - Each variable live after I is also live before I, unless I defines (writes) it
  - Each variable that I uses (reads) is also live before instruction I

in[I]

I

out[I]

- Mathematically:

$$in[I] = (\ out[I] - def[I]\ ) \cup use[I]$$

where:

- def[I] = variables defined (written) by instruction I
- use[I] = variables used (read) by instruction I

Computing Use/Def

- Compute use[I] and def[I] for each instruction I:

if I is $x = y \text{ OP } z$:	use[I] = {y, z}	def[I] = {x}
if I is $x = \text{OP } y$:	use[I] = {y}	def[I] = {x}
if I is $x = y$:	use[I] = {y}	def[I] = {x}
if I is $x = \text{addr } y$:	use[I] = {}	def[I] = {x}
if I is if (x) :	use[I] = {x}	def[I] = {}
if I is return x :	use[I] = {x}	def[I] = {}
if I is $x = f(y_1, \ldots, y_n)$:	use[I] = $\{y_1, \ldots, y_n\}$	def[I] = {x}

(For now, ignore load and store instructions)

Example

- Example: block B with three instructions I1, I2, I3:

  Live1 = in[B] = in[I1]

  Live2 = out[I1] = in[I2]

  Live3 = out[I2] = in[I3]

  Live4 = out[I3] = out[B]

- Relation between Live sets:

  Live1 = (Live2-{x}) $\cup$ {y}

  Live2 = (Live3-{y}) $\cup$ {z}

  Live3 = (Live4-{}) $\cup$ {d}

Block B

	Block B
	Live1
I1	x = y+1
	Live2
I2	y =2*z
	Live3
I3	if (d)
	Live4

Backward Flow

- Relation:

  $in[I] = (out[I] - def[I]) \cup use[I]$

in[I]

I

out[I]

- The information flows backward!

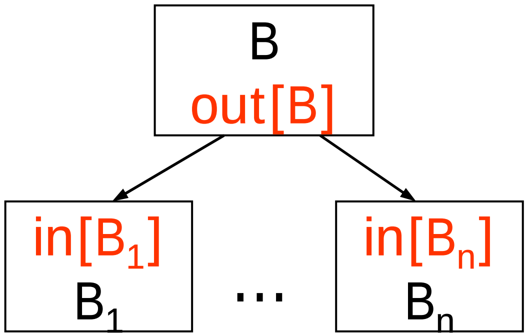
- Instructions: can compute in[I] if we know out[I]

- Basic blocks: information about live variables flows from out[B] to in[B]

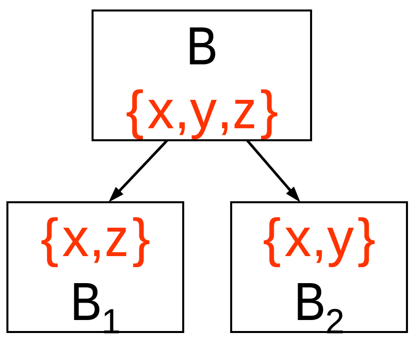
In[B]
x = y+1
y =2*z
if (d)
out[B]

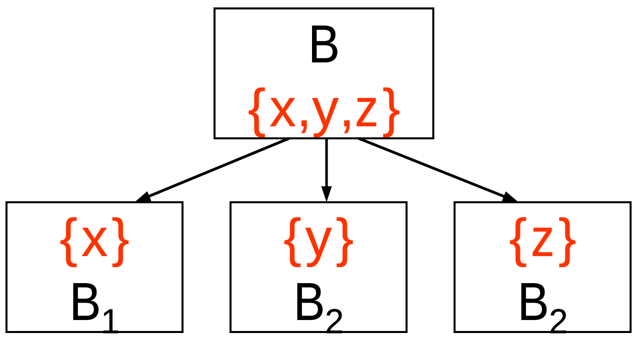
Part 2: Analyze Control Flow

- Question: for each basic block B with successor blocks B_1, ..., B_n, what is the relation between out[B] and in[B_1], ..., in[B_n]?



- Examples:



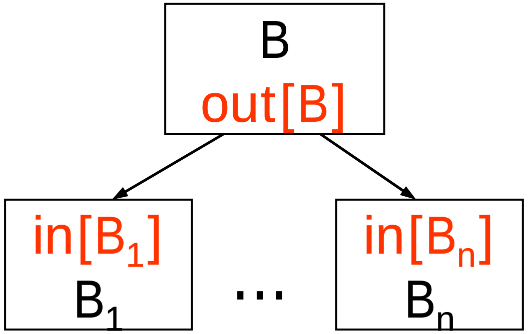


- What is the general rule?

Analyze Control Flow

- Rule: A variables is live at end of block B if it is live at the beginning of one (or more) successor blocks
- Characterizes all possible program executions
- Mathematically:

$$out[B] = \bigcup_{B' \in succ(B)} in[B']$$



- Again, information flows backward: from successors B' of B to basic block B

Constraint System

- Put parts together: start with CFG and derive a system of constraints between live variable sets:

$$\begin{cases} in[I] = (\ out[I] - def[I]\ ) \cup use[I] & \text{for each instruction I} \\ out[B] = \bigcup_{B' \in succ(B)} in[B'] & \text{for each basic block B} \end{cases}$$

- Solve constraints:
  - Start with empty sets of live variables
  - Iteratively apply constraints
  - Stop when we reach a fixed point

Constraint Solving Algorithm

for all instructions I **do** in[I] = out[I] = $\emptyset$;

repeat

select an instuction I (or a basic block B) such that

$in[I] \neq (out[I] - def[I]) \cup use[I]$

or (respectively)

$out[B] \neq \bigcup_{B' \in succ(B)} in[B']$

and update in[I] (or out[B]) accordingly

until no such change is possible

Example

def = {}, use = {c}

def = {x}, use = {y}

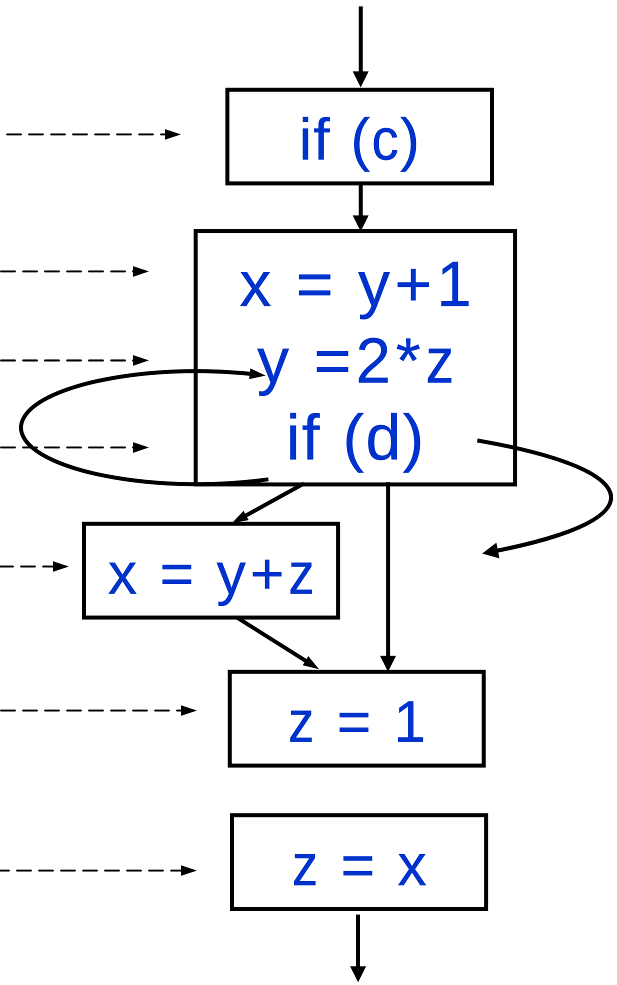
def = {y}, use = {z}

def = {}, use = {d}

def = {x}, use = {y,z}

def = {z}, use = {}

def = {z}, use = {x}



Example

def = {}, use = {c}

def = {x}, use = {y}

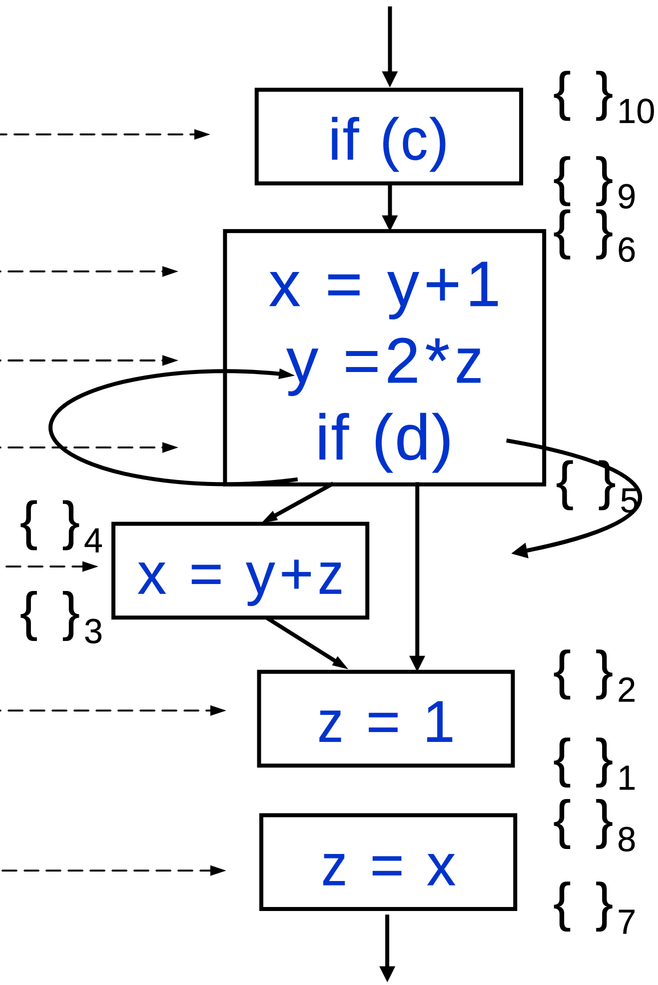
def = {y}, use = {z}

def = {}, use = {d}

def = {x}, use = {y,z}

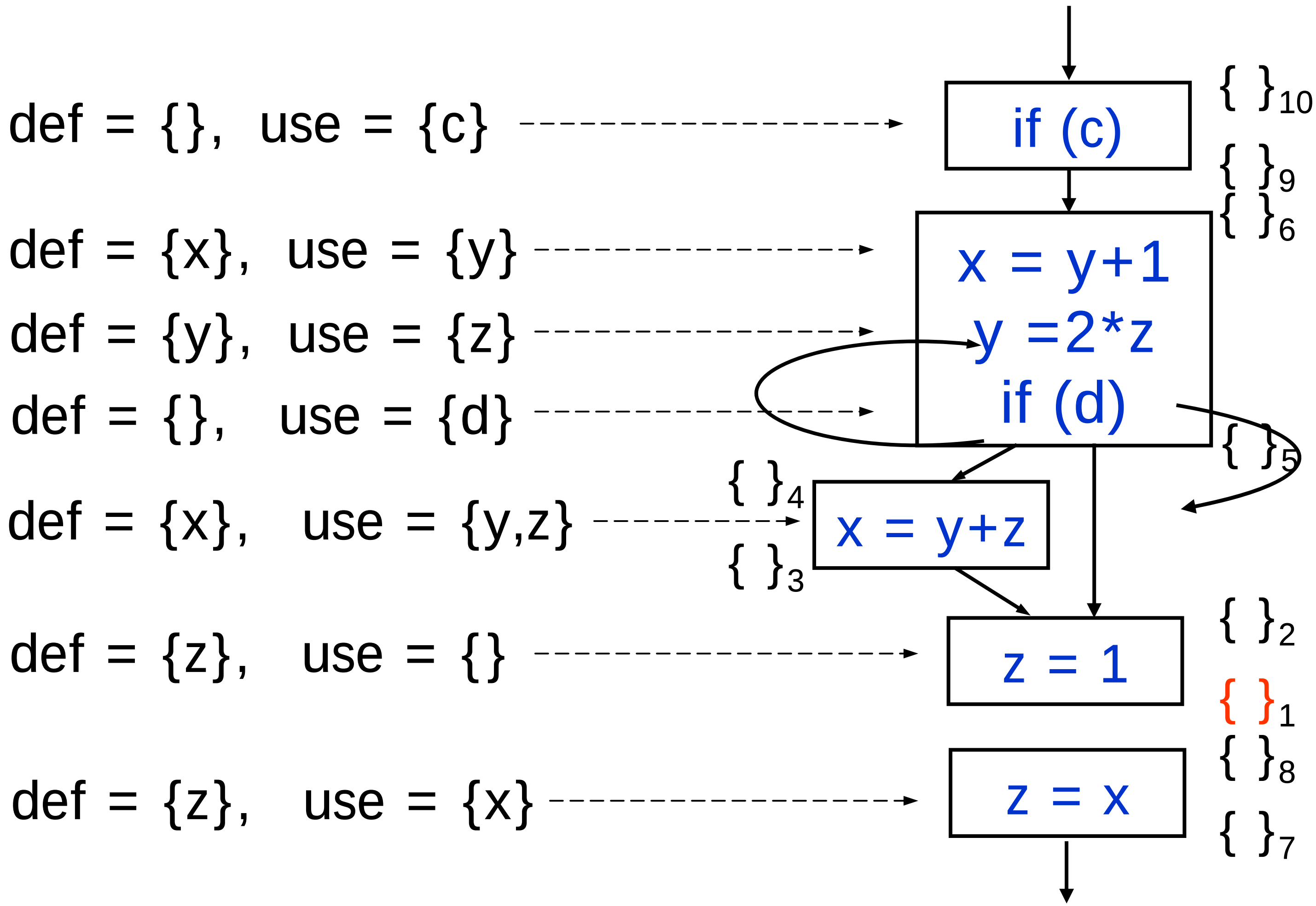
def = {z}, use = {}

def = {z}, use = {x}

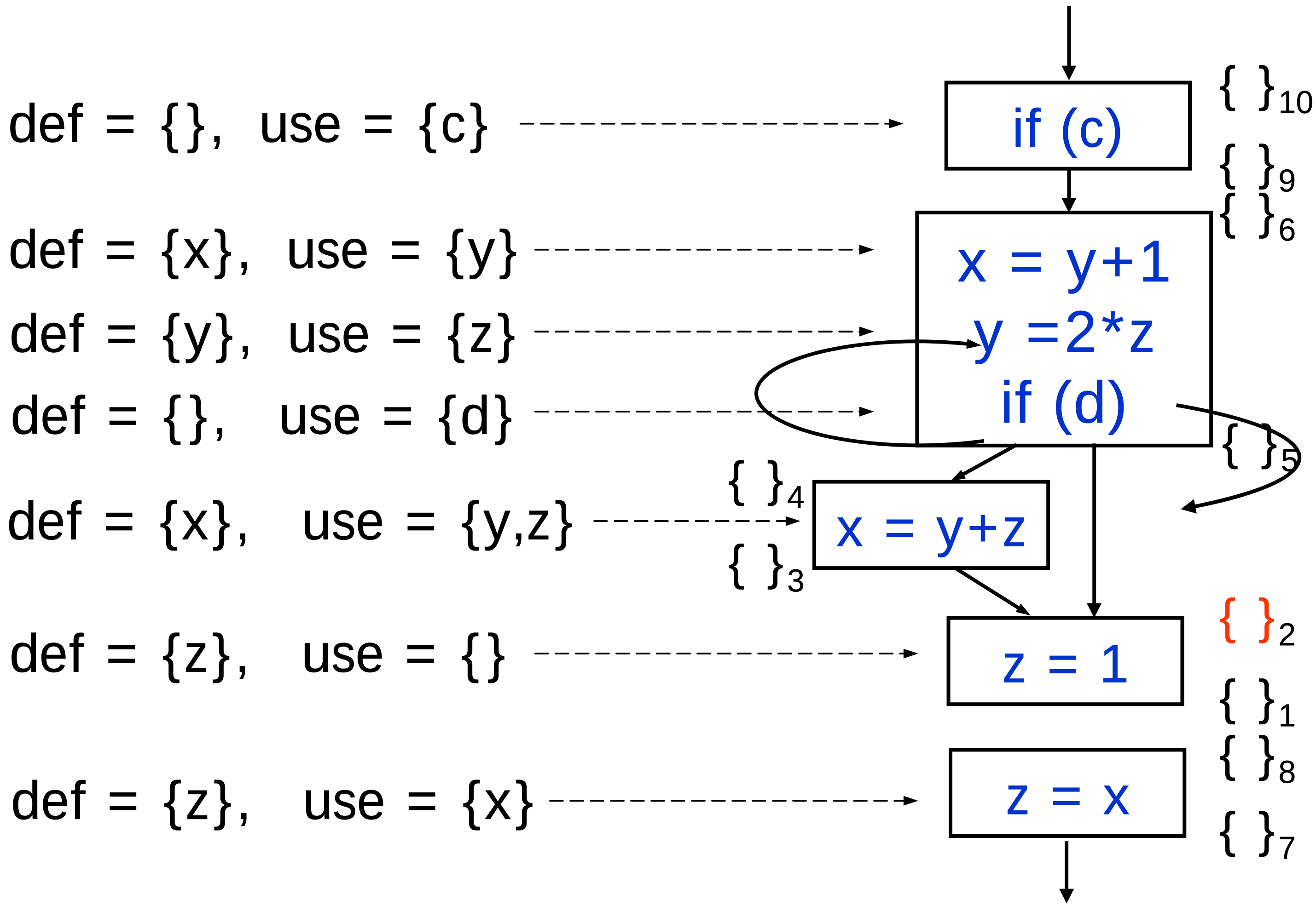


strategy: pick program points in postorder

Example


def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}
def = {z}, use = {}
def = {z}, use = {x}
if (c)
x = y+1
y =2*z
if (d)
x = y+z
z = 1
z = x
{ }10
{ }9
{ }6
{ }5
{ }4
{ }3
{ }2
{ }1
{ }8
{ }7

Example



def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}
def = {z}, use = {}
def = {z}, use = {x}
if (c)
x = y+1
y =2*z
if (d)
x = y+z
z = 1
z = x
{ }10
{ }9
{ }6
{ }5
{ }4
{ }3
{ }2
{ }1
{ }8
{ }7

Example

def = {}, use = {c}

def = {x}, use = {y}

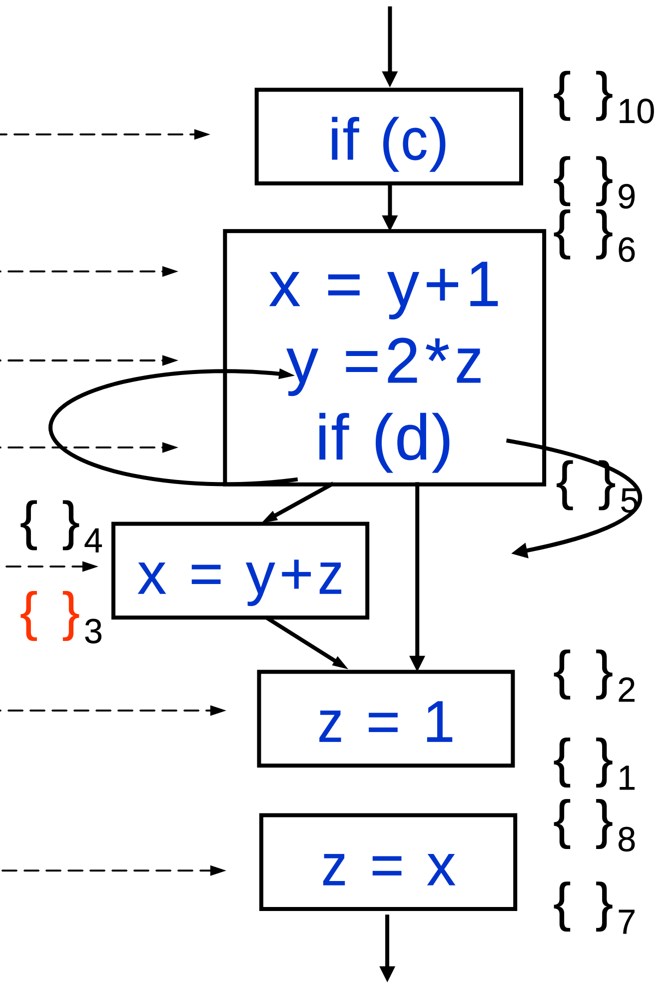
def = {y}, use = {z}

def = {}, use = {d}

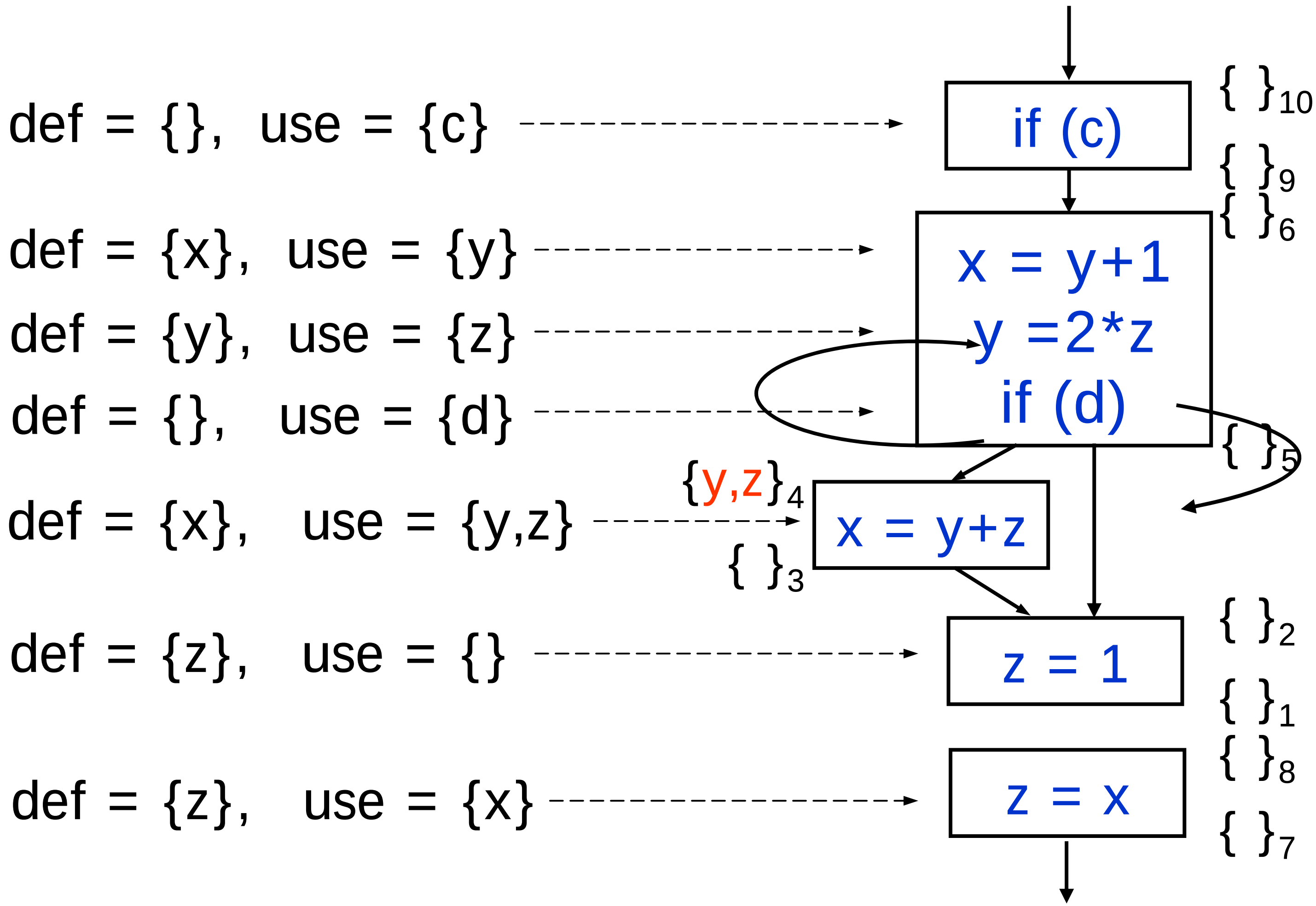
def = {x}, use = {y,z}

def = {z}, use = {}

def = {z}, use = {x}



Example



def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}
def = {z}, use = {}
def = {z}, use = {x}
if (c)
x = y+1
y =2*z
if (d)
x = y+z
z = 1
z = x
{ }10
{ }9
{ }6
{ }5
{y,z}4
{ }3
{ }2
{ }1
{ }8
{ }7

Example

def = {}, use = {c}

def = {x}, use = {y}

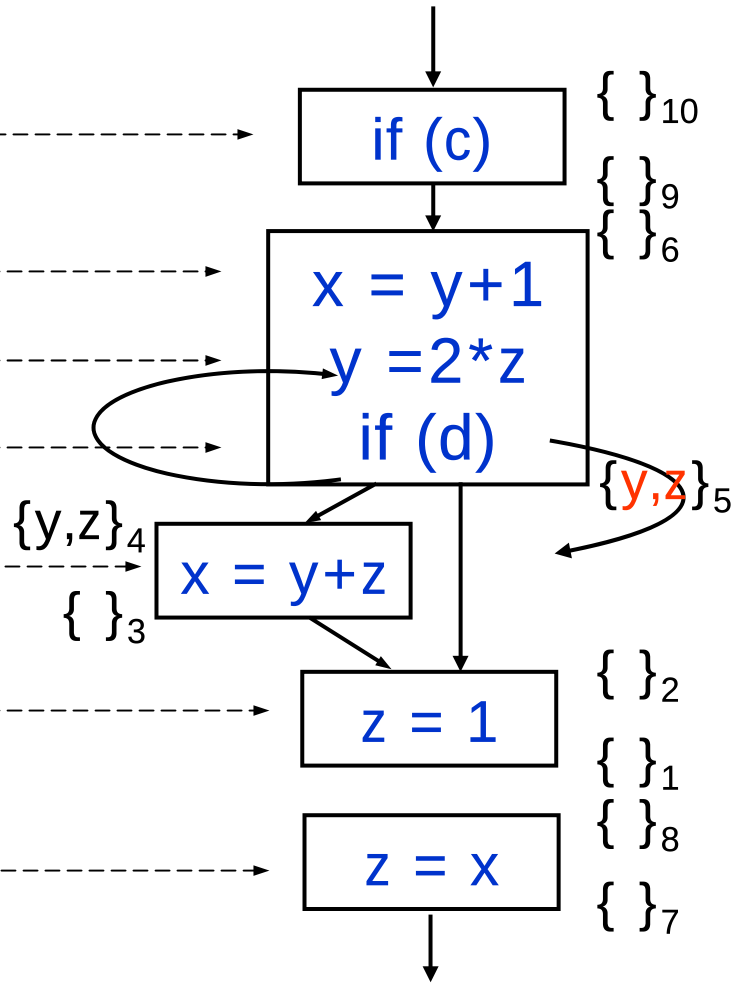
def = {y}, use = {z}

def = {}, use = {d}

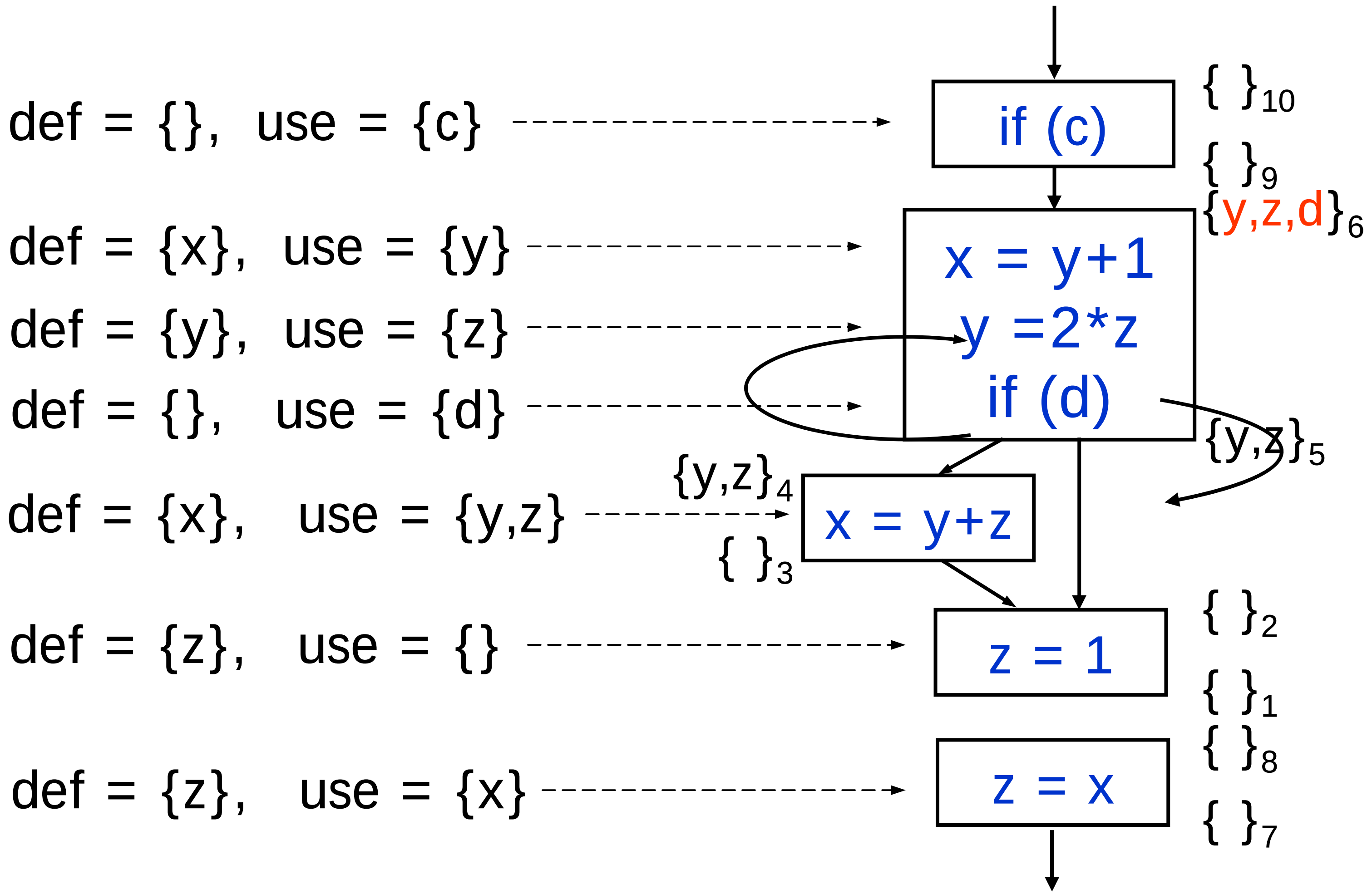
def = {x}, use = {y,z}

def = {z}, use = {}

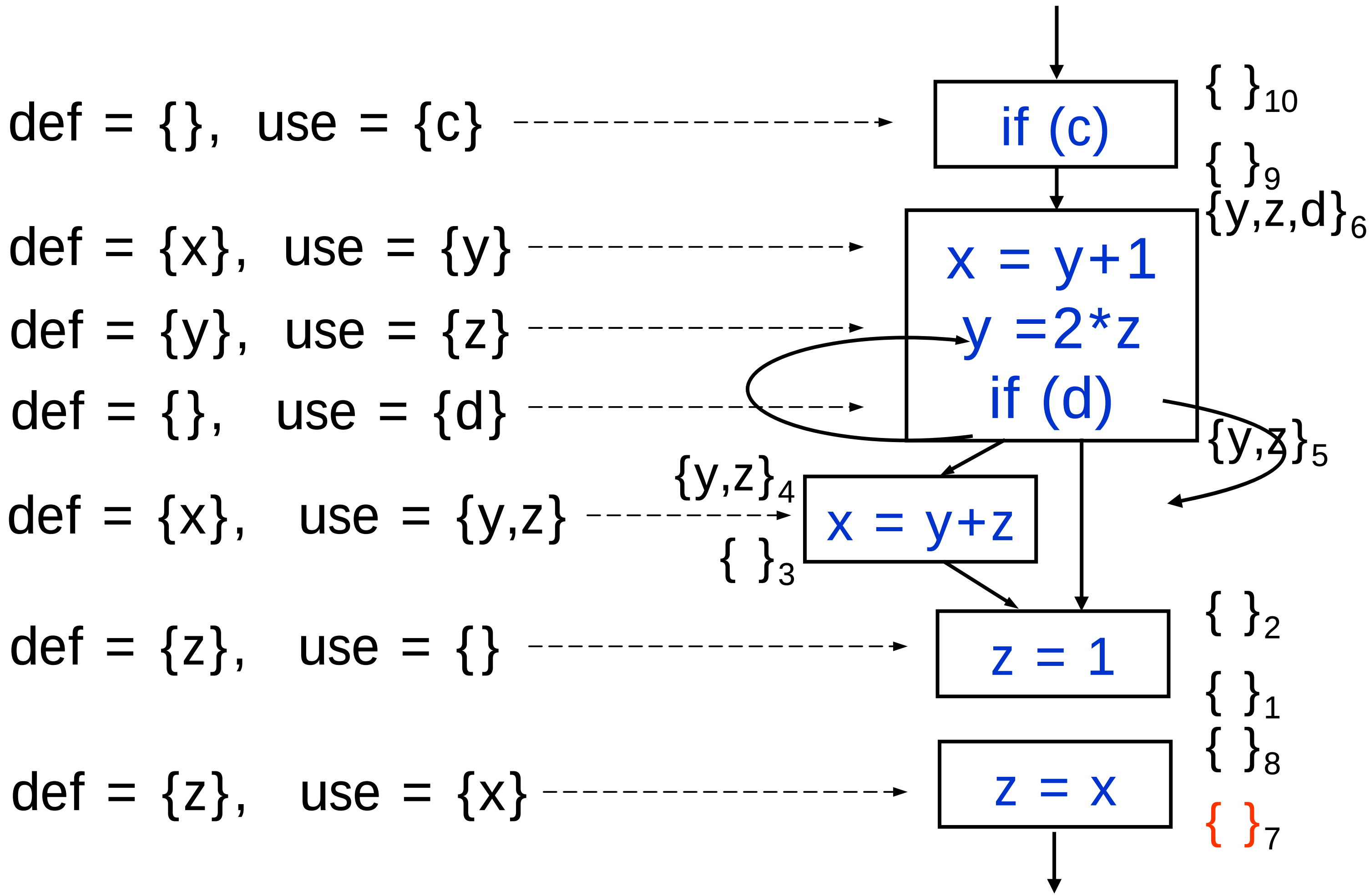
def = {z}, use = {x}



Example


def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}
def = {z}, use = {}
def = {z}, use = {x}
if (c)
x = y+1
y =2*z
if (d)
x = y+z
z = 1
z = x
{ }10
{ }9
{y,z,d}6
{y,z}5
{y,z}4
{ }3
{ }2
{ }1
{ }8
{ }7

Example


def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}
def = {z}, use = {}
def = {z}, use = {x}
if (c)
x = y+1
y =2*z
if (d)
x = y+z
z = 1
z = x
{ }10
{ }9
{y,z,d}6
{y,z}5
{y,z}4
{ }3
{ }2
{ }1
{ }8
{ }7

Example

def = {}, use = {c}

def = {x}, use = {y}

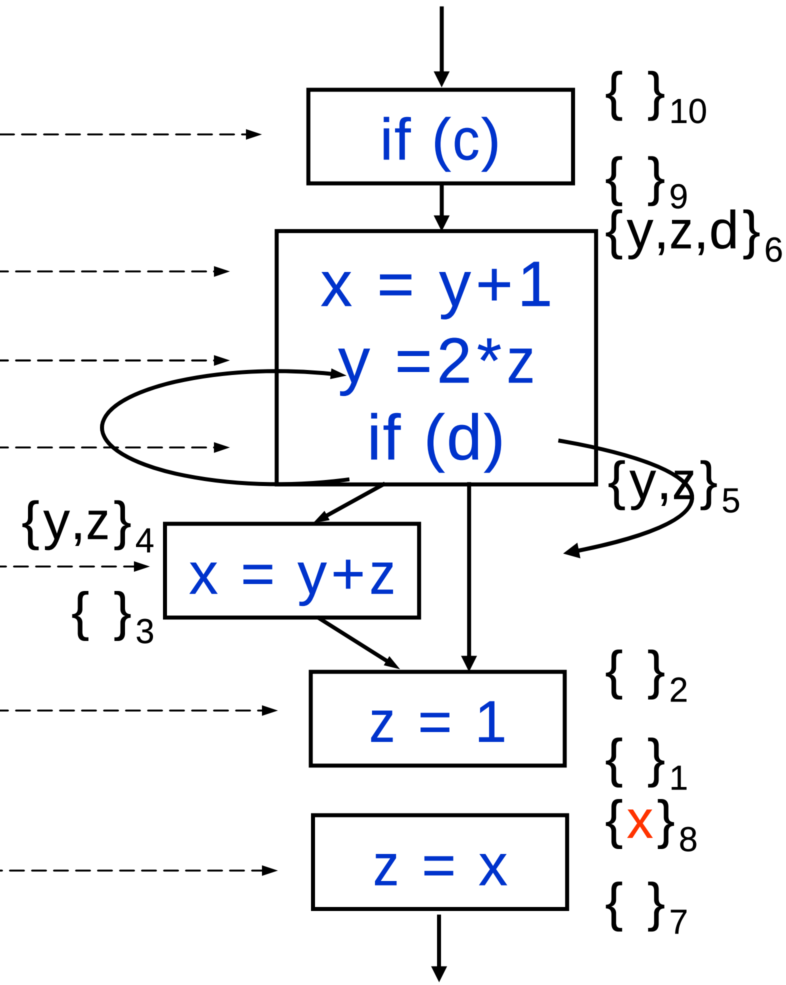
def = {y}, use = {z}

def = {}, use = {d}

def = {x}, use = {y,z}

def = {z}, use = {}

def = {z}, use = {x}



Example

def = {}, use = {c}

def = {x}, use = {y}

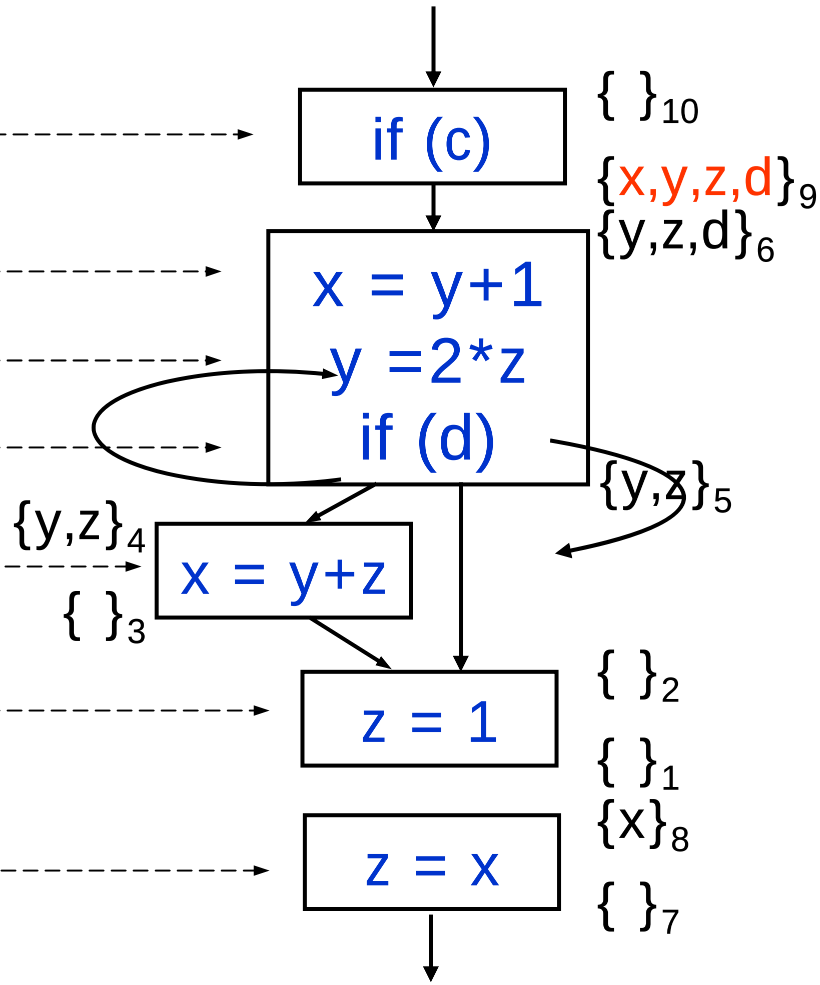
def = {y}, use = {z}

def = {}, use = {d}

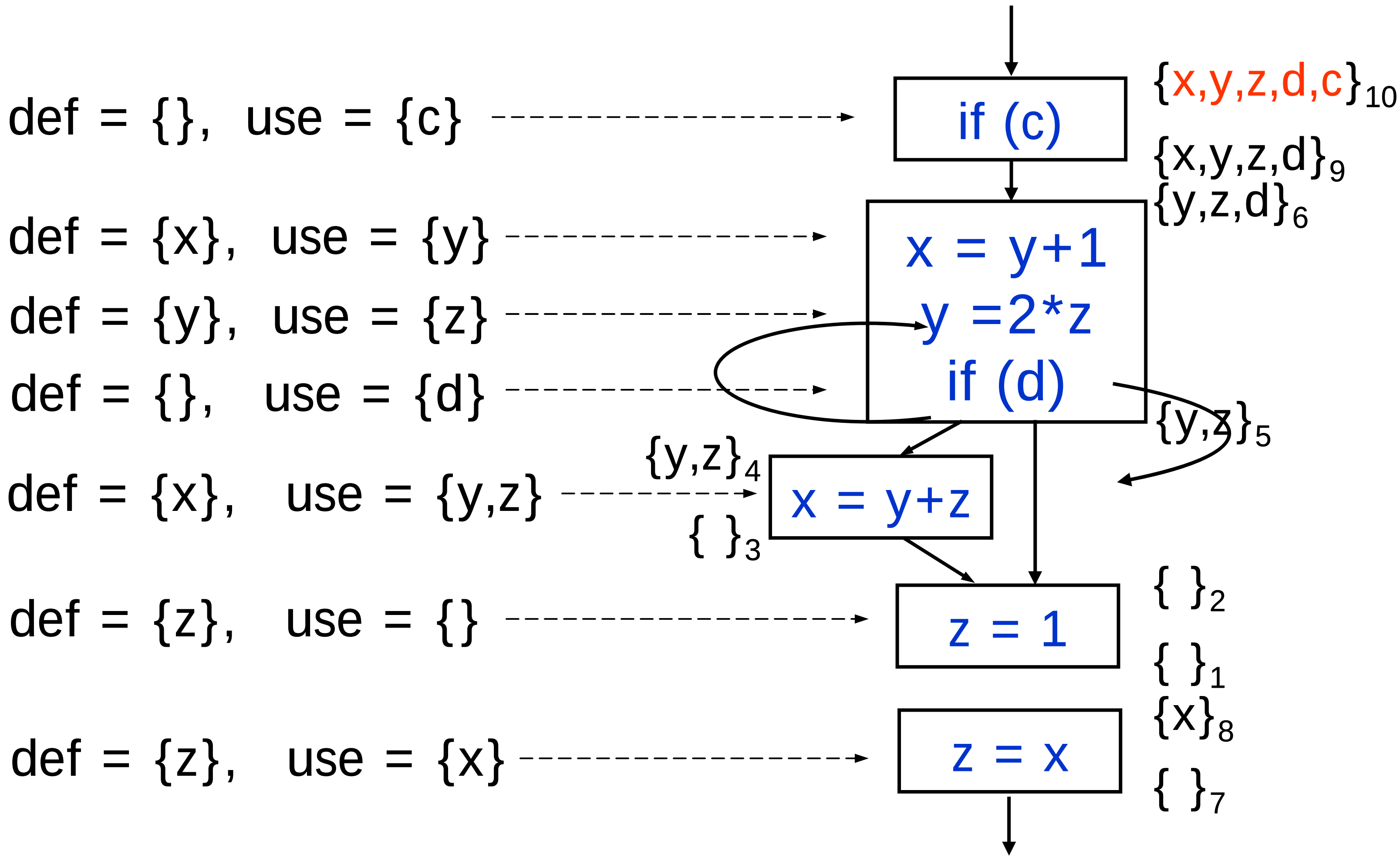
def = {x}, use = {y,z}

def = {z}, use = {}

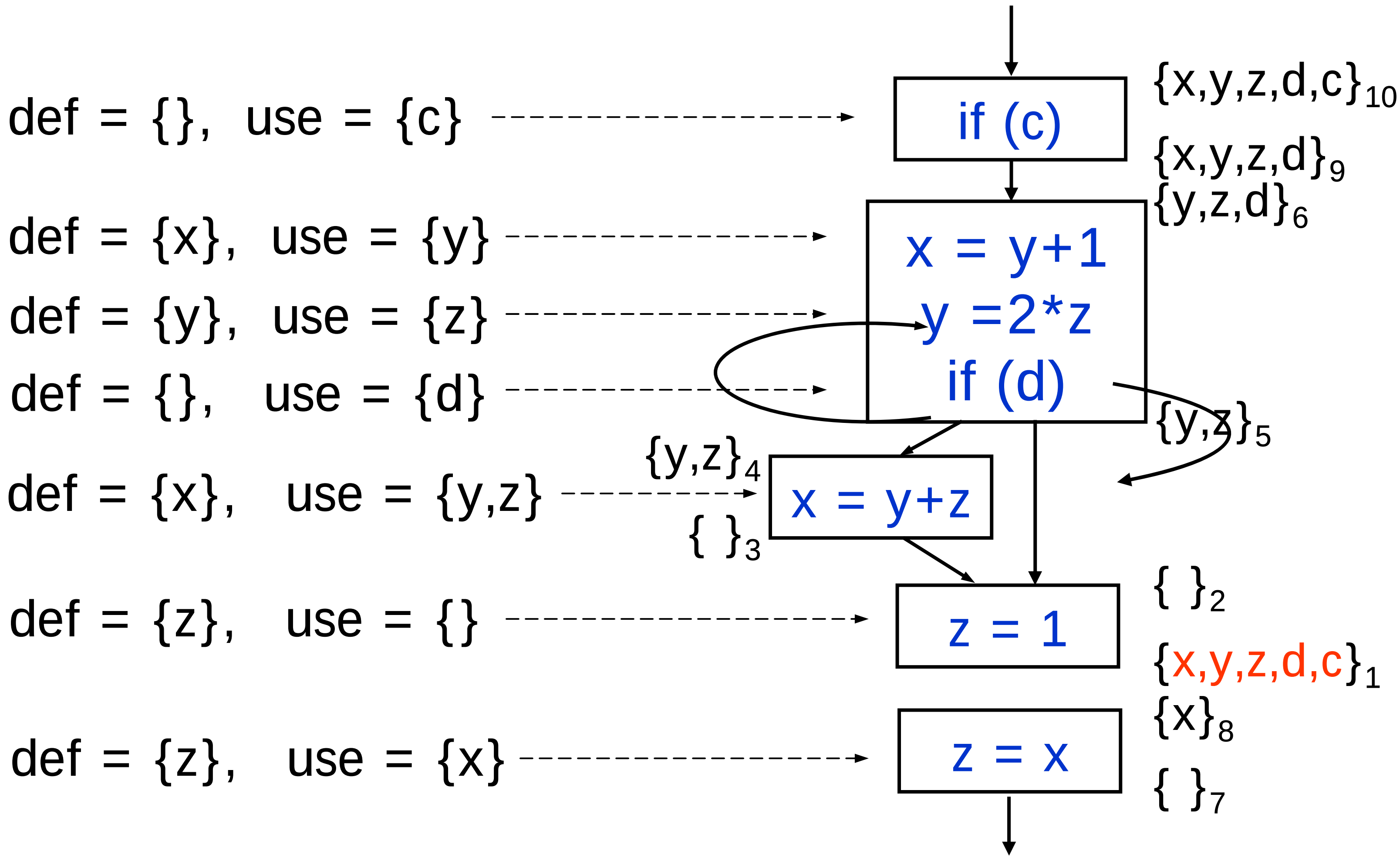
def = {z}, use = {x}



Example


def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}
def = {z}, use = {}
def = {z}, use = {x}
if (c)
x = y+1
y =2*z
if (d)
x = y+z
z = 1
z = x
{x,y,z,d,c}10
{x,y,z,d}9
{y,z,d}6
{y,z}5
{y,z}4
{ }3
{ }2
{ }1
{x}8
{ }7

Example



def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}
def = {z}, use = {}
def = {z}, use = {x}
if (c)
x = y+1
y =2*z
if (d)
x = y+z
z = 1
z = x
{x,y,z,d,c}10
{x,y,z,d}9
{y,z,d}6
{y,z}5
{y,z}4
{ }3
{ }2
{x,y,z,d,c}1
{x}8
{ }7

Example

def = {}, use = {c}

def = {x}, use = {y}

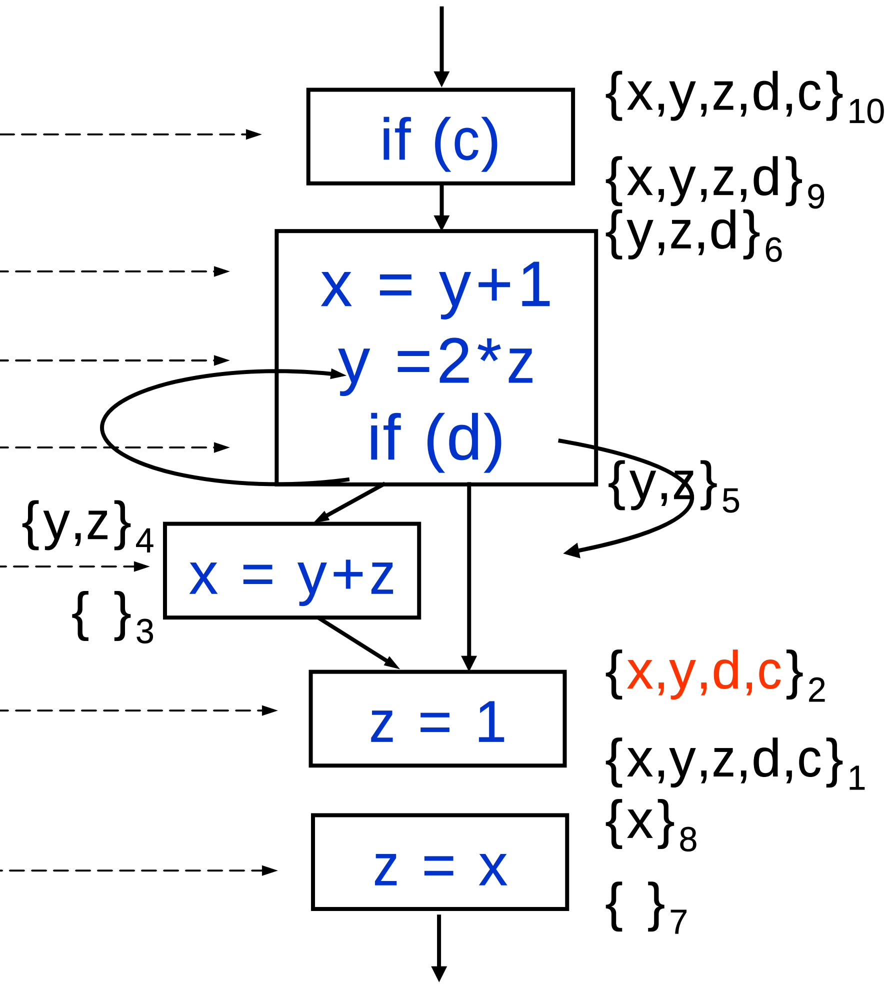
def = {y}, use = {z}

def = {}, use = {d}

def = {x}, use = {y,z}

def = {z}, use = {}

def = {z}, use = {x}



Example

def = {}, use = {c}

def = {x}, use = {y}

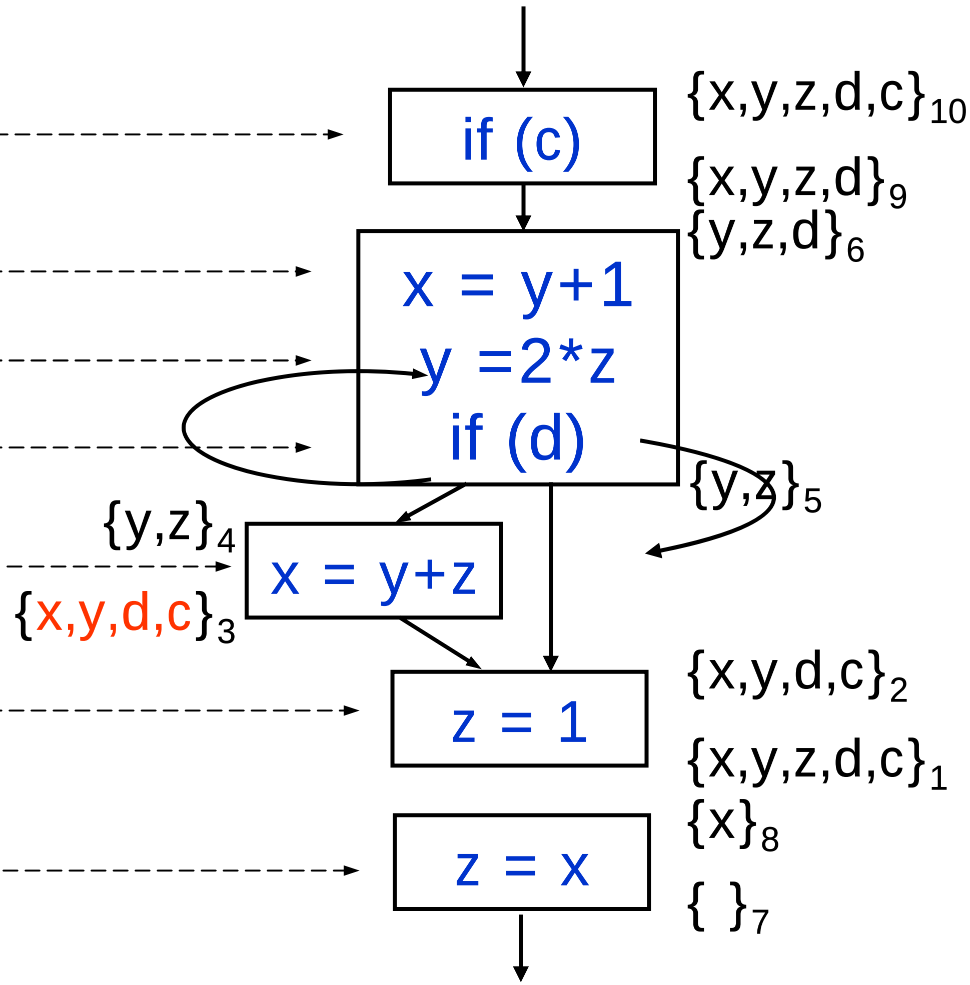
def = {y}, use = {z}

def = {}, use = {d}

def = {x}, use = {y,z}

def = {z}, use = {}

def = {z}, use = {x}



Example

def = {}, use = {c}

def = {x}, use = {y}

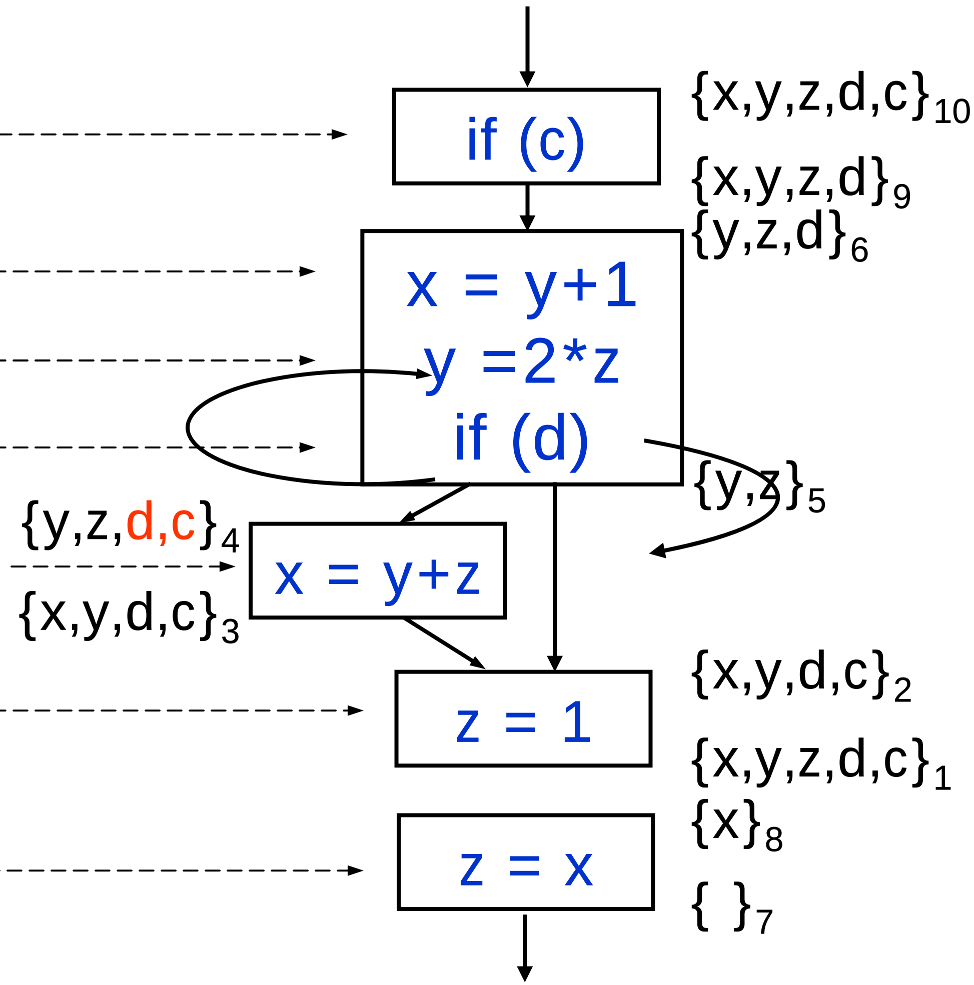
def = {y}, use = {z}

def = {}, use = {d}

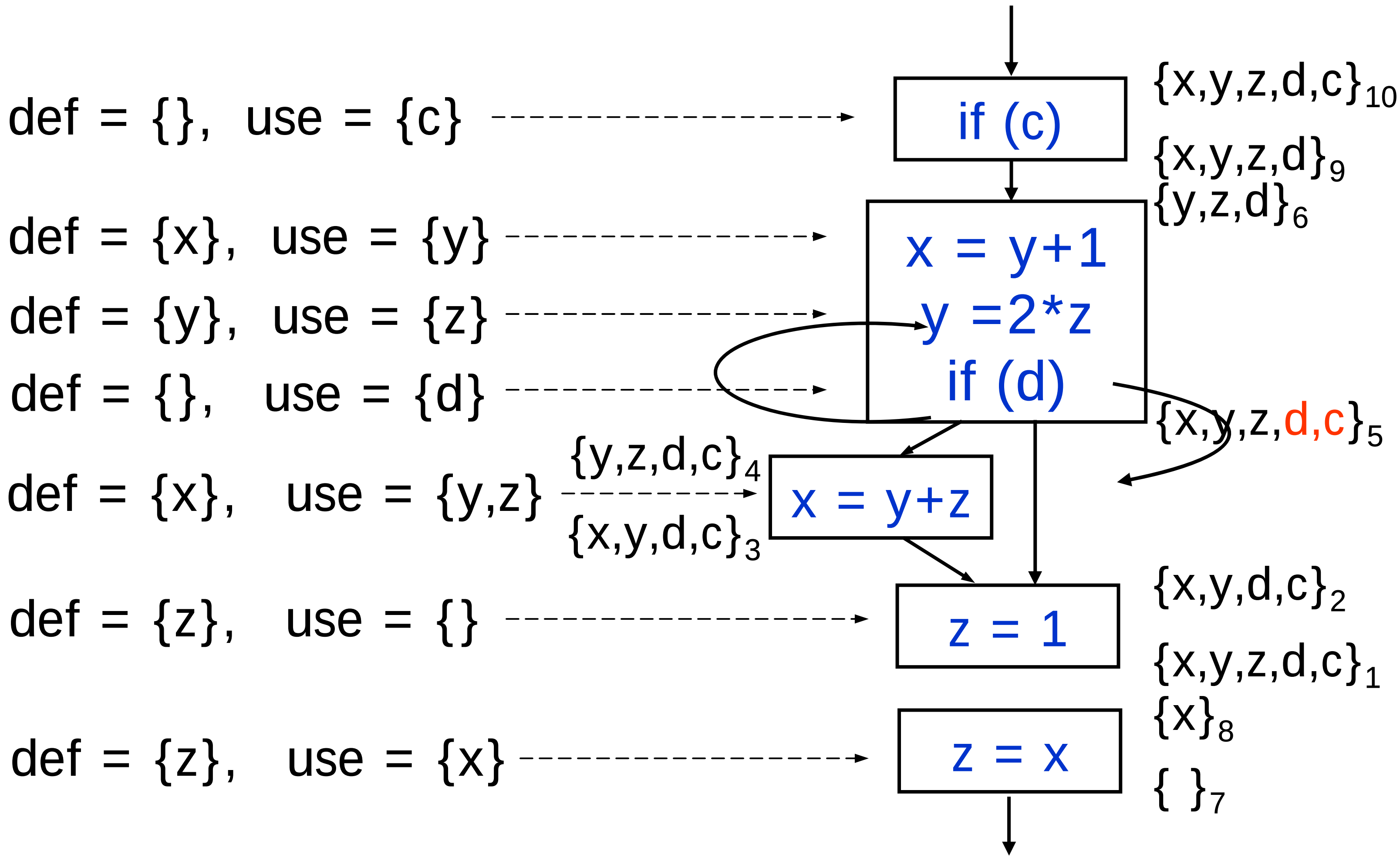
def = {x}, use = {y,z}

def = {z}, use = {}

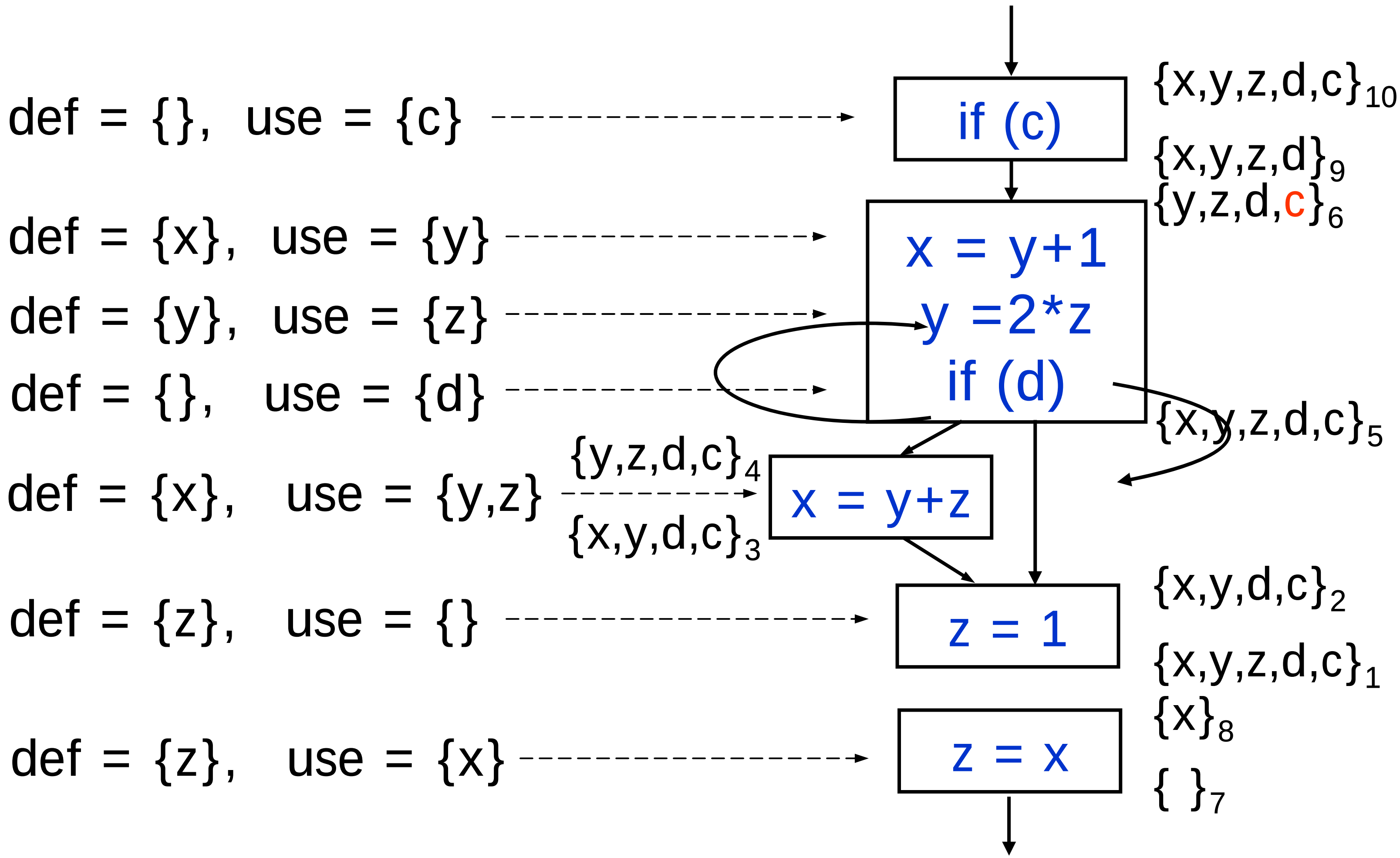
def = {z}, use = {x}



Example


def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}
def = {z}, use = {}
def = {z}, use = {x}
if (c)
x = y+1
y =2*z
if (d)
x = y+z
z = 1
z = x
{x,y,z,d,c}10
{x,y,z,d}9
{y,z,d}6
{x,y,z,d,c}5
{y,z,d,c}4
{x,y,d,c}3
{x,y,d,c}2
{x,y,z,d,c}1
{x}8
{ }7

Example



def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}
def = {z}, use = {}
def = {z}, use = {x}
if (c)
x = y+1
y =2*z
if (d)
x = y+z
z = 1
z = x
$\{x,y,z,d,c\}_{10}$
$\{x,y,z,d\}_{9}$
$\{y,z,d,c\}_{6}$
$\{x,y,z,d,c\}_{5}$
$\{y,z,d,c\}_{4}$
$\{x,y,d,c\}_{3}$
$\{x,y,d,c\}_{2}$
$\{x,y,z,d,c\}_{1}$
$\{x\}_{8}$
$\{\ \}_{7}$

Example

def = {}, use = {c}

def = {x}, use = {y}

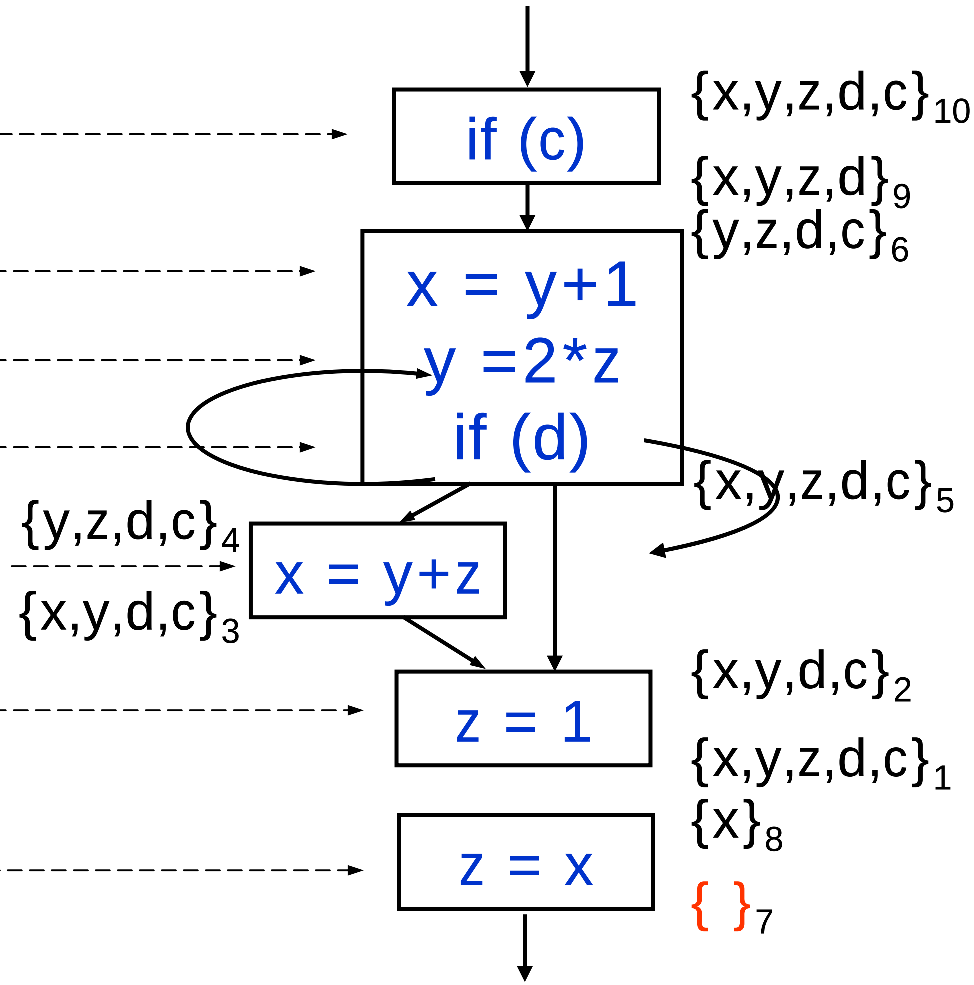
def = {y}, use = {z}

def = {}, use = {d}

def = {x}, use = {y,z}

def = {z}, use = {}

def = {z}, use = {x}



Example

def = {}, use = {c}

def = {x}, use = {y}

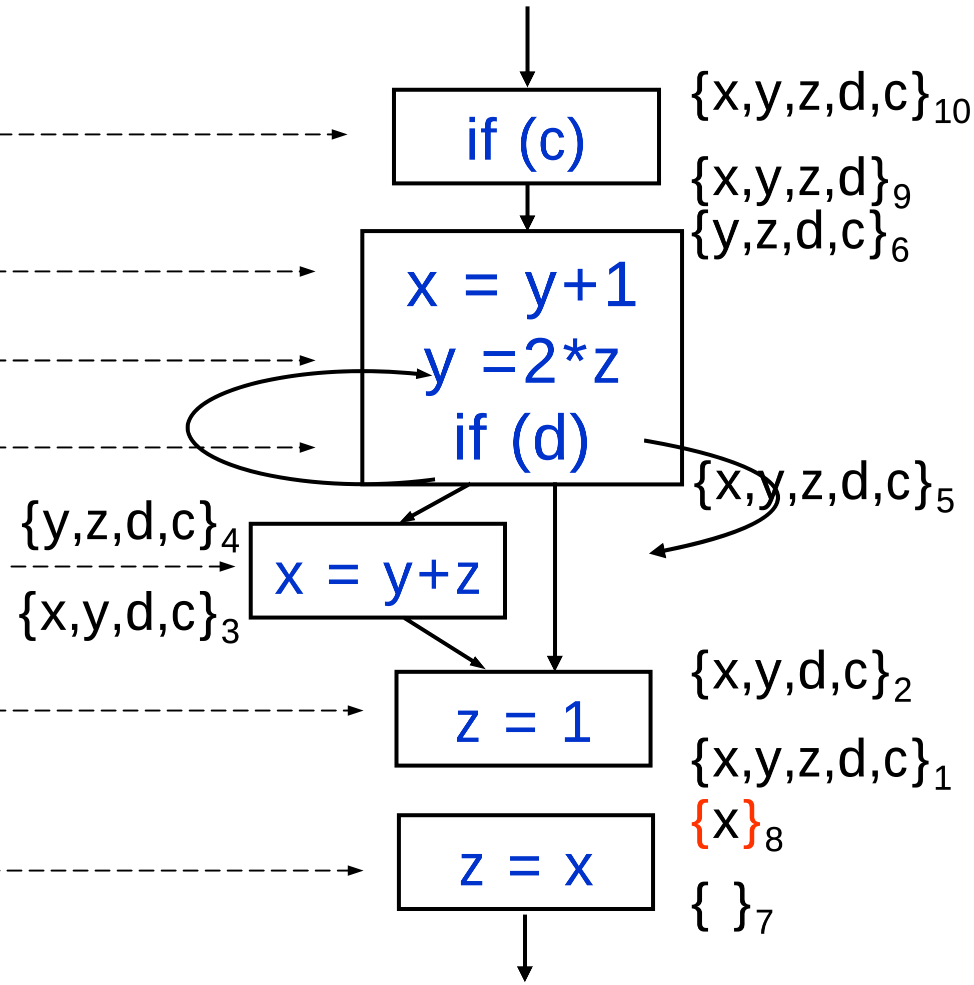
def = {y}, use = {z}

def = {}, use = {d}

def = {x}, use = {y,z}

def = {z}, use = {}

def = {z}, use = {x}



Example

def = {}, use = {c}

def = {x}, use = {y}

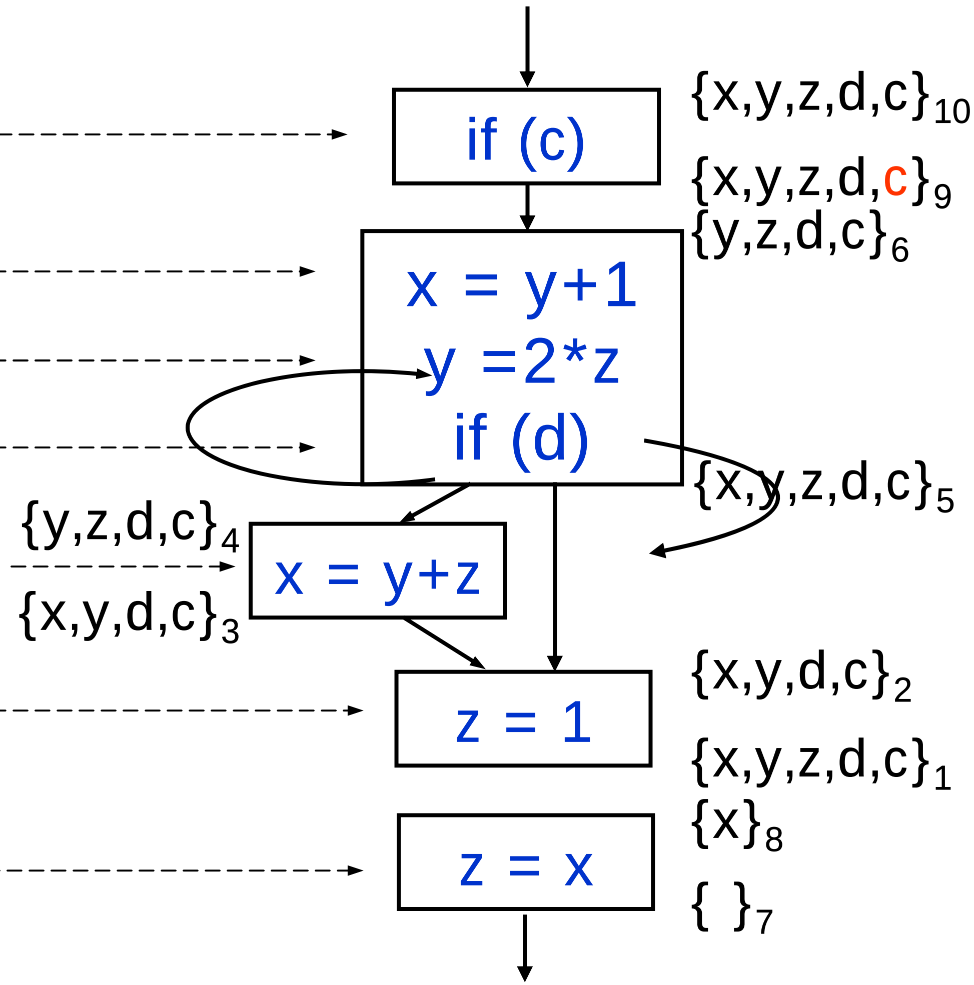
def = {y}, use = {z}

def = {}, use = {d}

def = {x}, use = {y,z}

def = {z}, use = {}

def = {z}, use = {x}



Example

def = {}, use = {c}

def = {x}, use = {y}

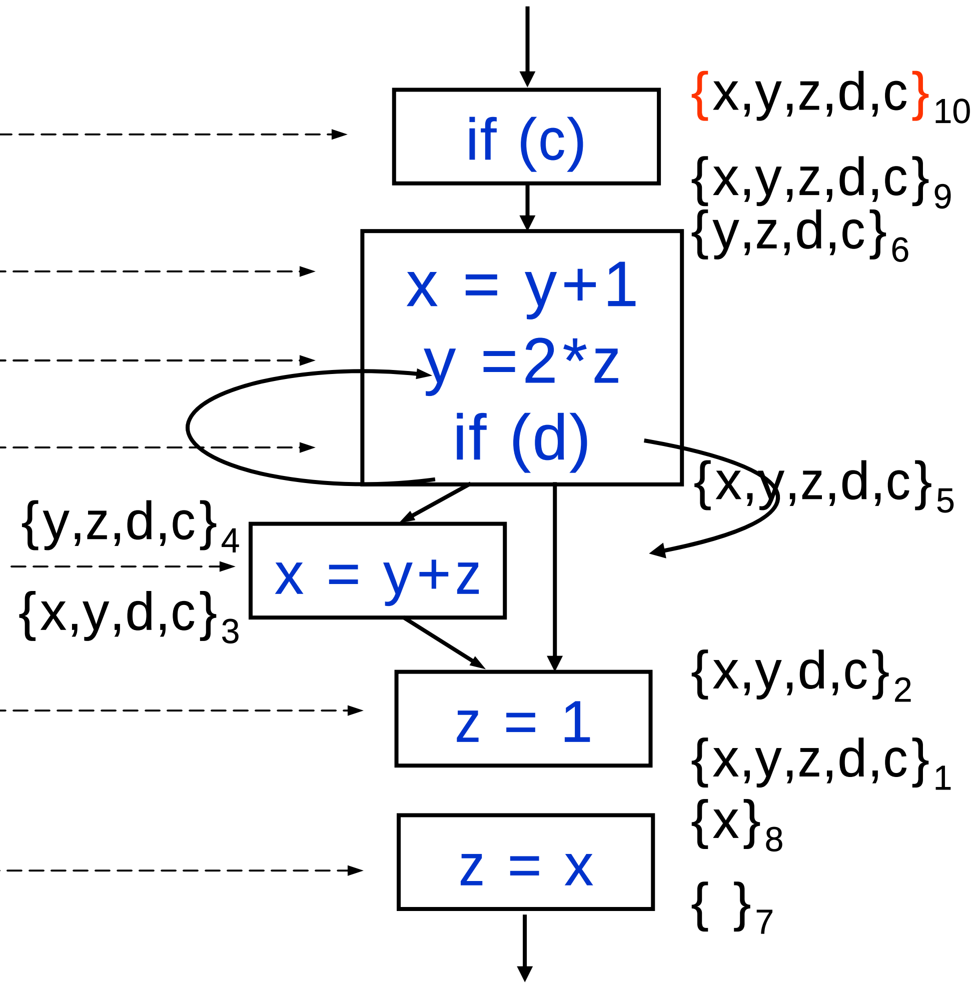
def = {y}, use = {z}

def = {}, use = {d}

def = {x}, use = {y,z}

def = {z}, use = {}

def = {z}, use = {x}



Fixed Point Reached

def = {}, use = {c}

def = {x}, use = {y}

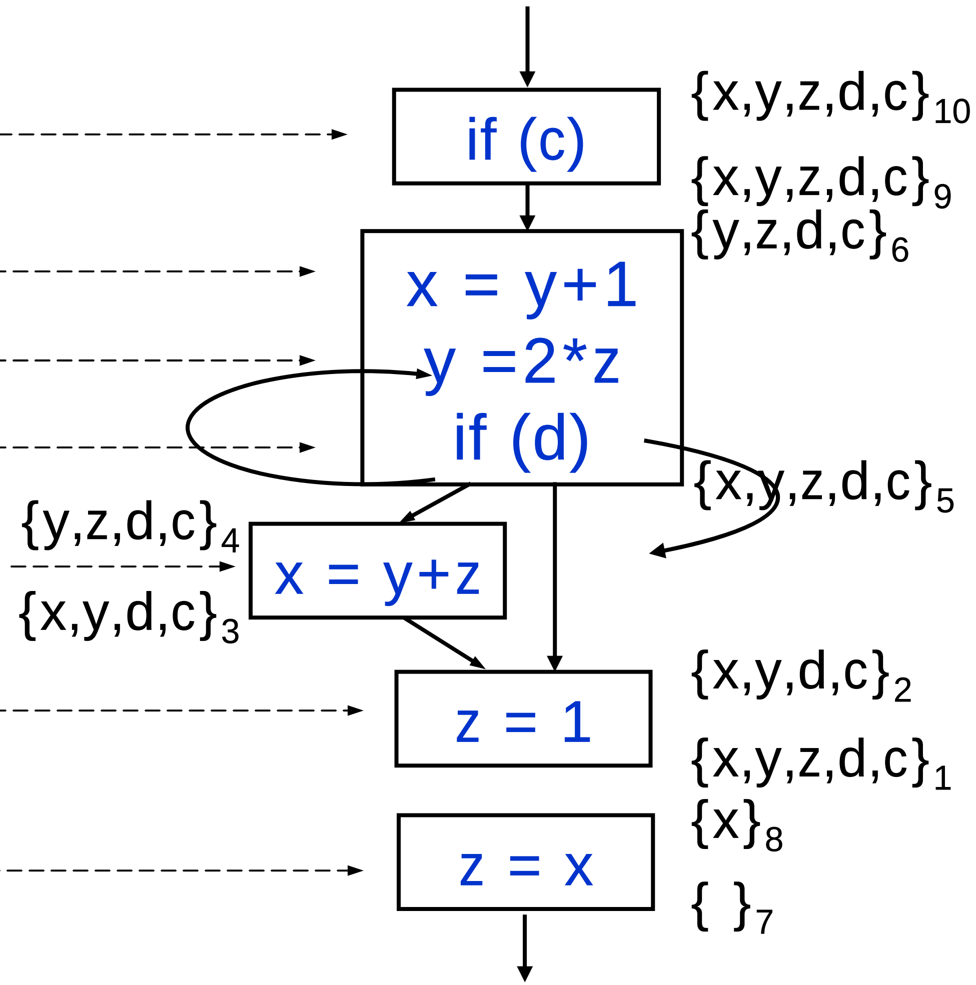
def = {y}, use = {z}

def = {}, use = {d}

def = {x}, use = {y,z}

def = {z}, use = {}

def = {z}, use = {x}



Copy Propagation

- Goal: determine copies available at each program point
- Information: set of copies <x=y> at each point
- For each instruction I:
  - in[I] = copies available at program point before I
  - out[I] = copies available at program point after I
- For each basic block B:
  - in[B] = copies available at beginning of B
  - out[B] = copies available at end of B
- If I = first instruction in B, then in[B] = in[I]
- If I’ = last instruction in B, then out[B] = out[I’]

Same Methodology

1. Express flow of information (i.e., available copies):
   – For points before and after each instruction (in[I], out[I])
   – For points at exit and entry of basic blocks (in[B], out[B])

2. Build constraint system using the relations between available copies

3. Solve constraints to determine available copies at each point in the program

Analyze Instructions

- Knowing in[I], can compute out[I]:
  - Remove from in[I] all copies <u=v> if variable u or v is written by I
  - Keep all other copies from in[I]
  - If I is of the form x=y, add it to out[I]

in[I]

I

out[I]

- Mathematically:

$$out[I] = (in[I] - kill[I]) \cup gen[I]$$

where:

- kill[I] = copies "killed" by instruction I
- gen[I] = copies "generated" by instruction I

Computing Kill/Gen

- Compute kill[I] and gen[I] for each instruction I:

  if I is x = y OP z : gen[I] = {} kill[I] = {u=v|u or v is x}

  if I is x = OP y : gen[I] = {} kill[I] = {u=v|u or v is x}

  if I is x = y : gen[I] = {x=y} kill[I] = {u=v|u or v is x}

  if I is x = addr y : gen[I] = {} kill[I] = {u=v|u or v is x}

  if I is if (x) : gen[I] = {} kill[I] = {}

  if I is return x : gen[I] = {} kill[I] = {}

  if I is $x = f(y_1, \ldots, y_n)$: gen[I] = {} kill[I] = {u=v| u or v is x}

  (again, ignore load and store instructions)

Forward Flow

- Relation:

  out[I] = (in[I] – kill[I]) ∪ gen[I]

- The information flows forward!

- Instructions: can compute out[I] if we know in[I]

- Basic blocks: information about available copies flows from in[B] to out[B]

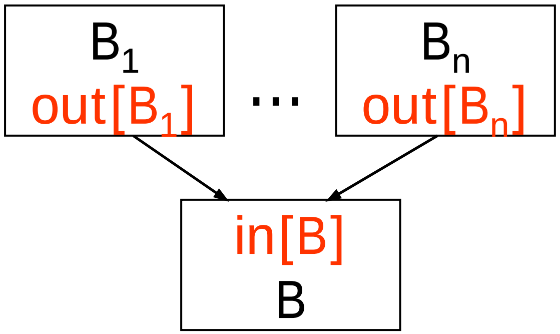
in[I]
I
out[I]

In[B]
x = y
y =2*z
if (d)
out[B]

Analyze Control Flow

- Rule: A copy is available at beginning of block B if it is available at the end of all predecessor blocks
- Characterizes all possible program executions
- Mathematically:

$$in[B] = \bigcap_{B' \in pred(B)} out[B']$$



- Information flows forward: from predecessors B' of B to basic block B

Constraint System

- Build constraints: start with CFG and derive a system of constraints between sets of available copies:

$$\begin{cases} out[I] = (in[I] - kill[I]) \cup gen[I] & \text{for each instruction I} \\ in[B] = \bigcap_{B' \in pred(B)} out[B'] & \text{for each basic block B} \end{cases}$$

- Solve constraints:
  - Start with empty set of available copies at start and universal set of available copies everywhere else
  - Iteratively apply constraints
  - Stop when we reach a fixed point

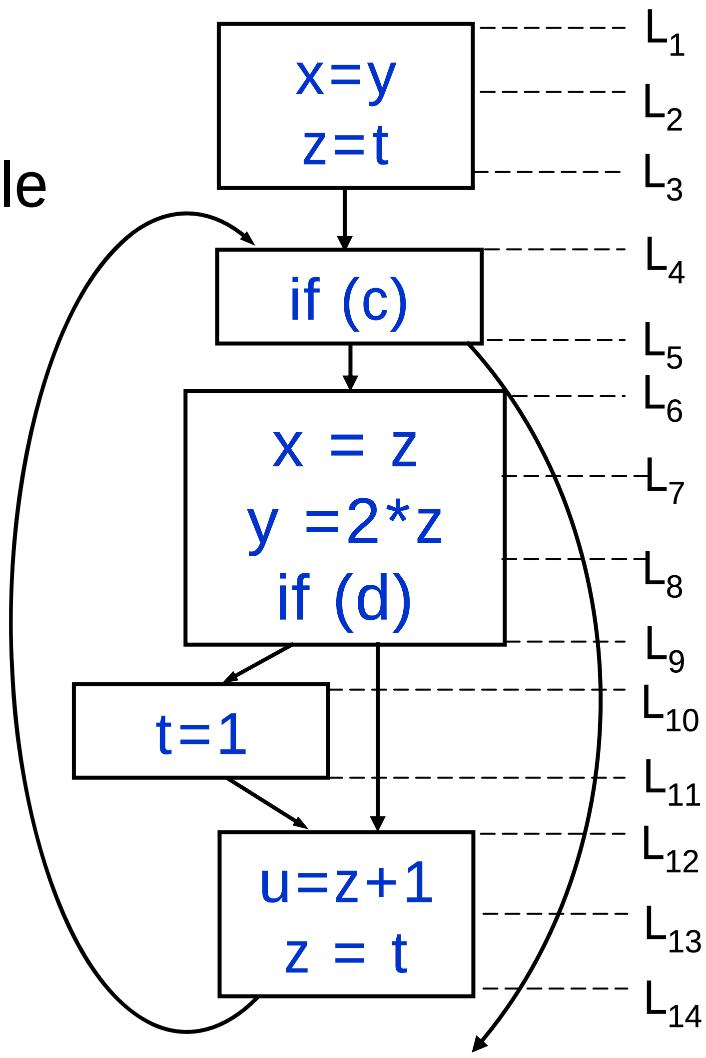
Example

- What are the available copies at the end of the program?

x=y?

z=t?

x=z?



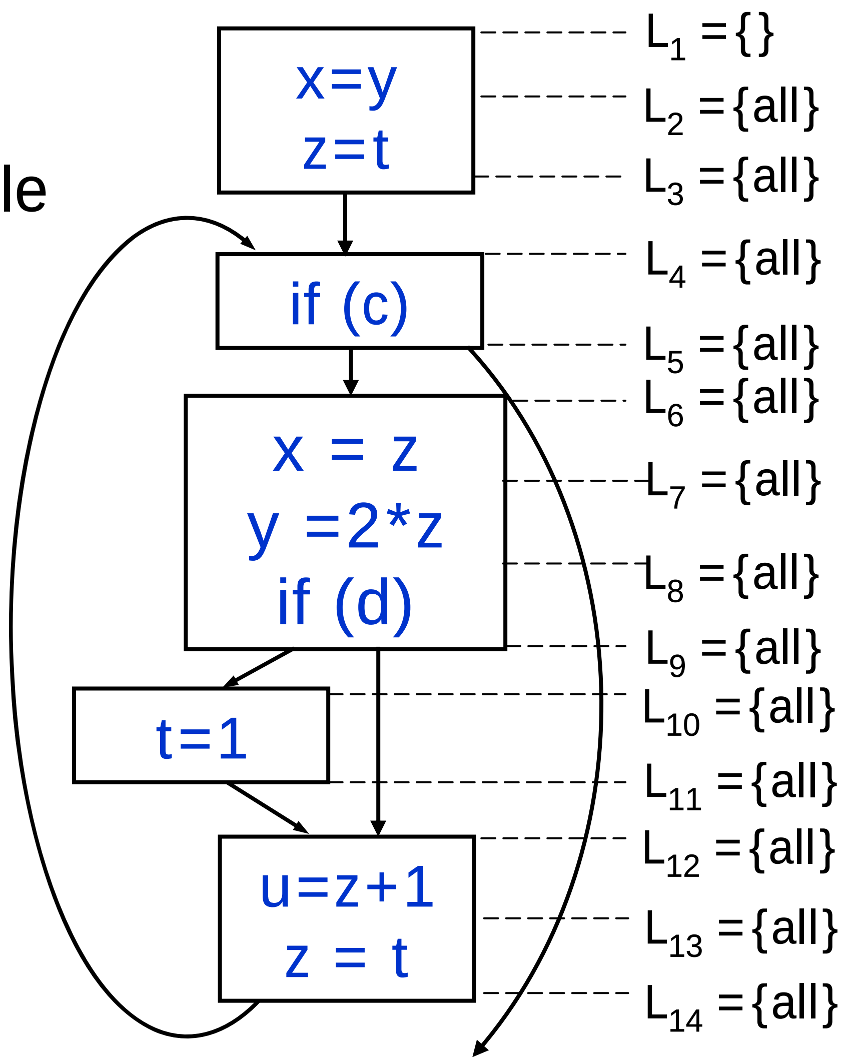
Example

- What are the available copies at the end of the program?

x=y?

z=t?

x=z?



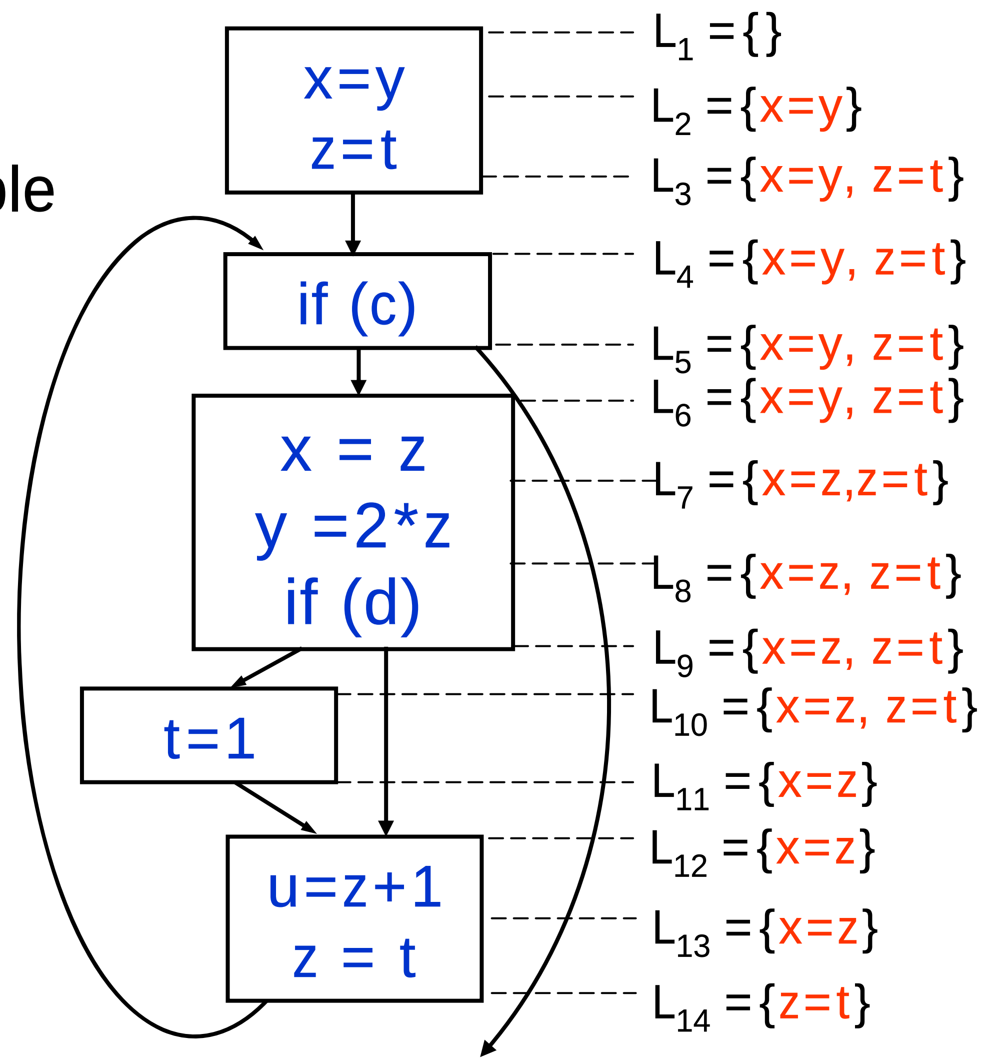
Iteration 1

- What are the available copies at the end of the program?

x=y?

z=t?

x=z?



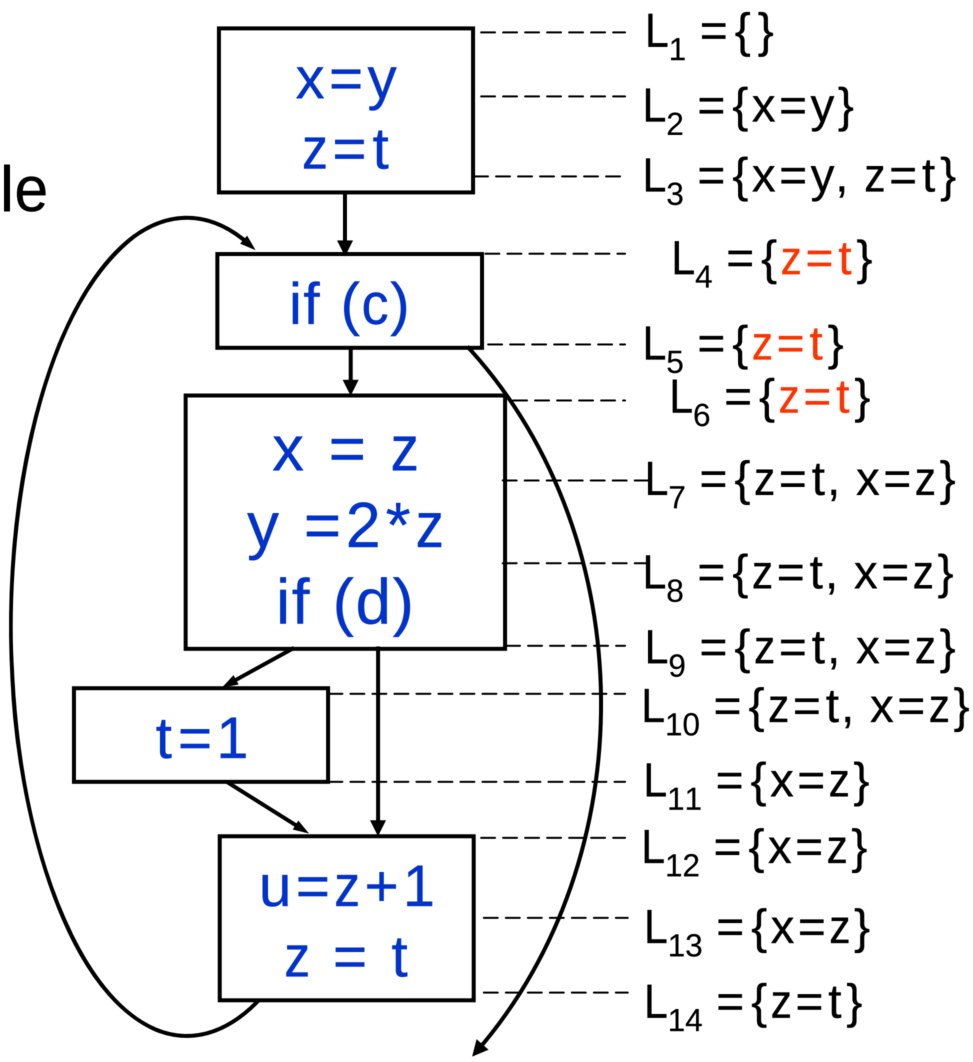
Iteration 2

- What are the available copies at the end of the program?

x=y?

z=t?

x=z?



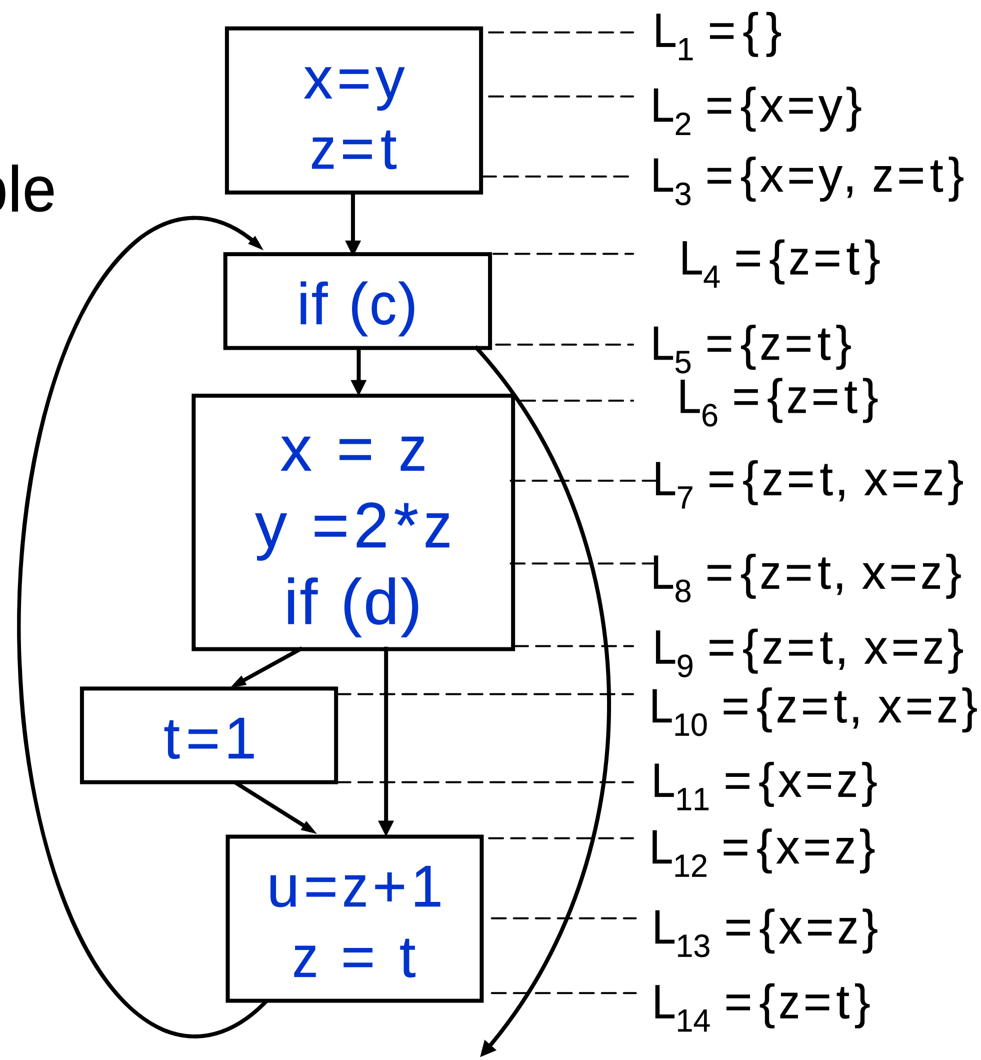
Fixed Point Reached!

- What are the available copies at the end of the program?

x=y? NO

z=t? YES

x=z? NO



Summary

- Extracting information about live variables and available copies is similar
  - Define the required information
  - Define information before/after instructions
  - Define information at entry/exit of blocks
  - Build constraints for instructions/control flow
  - Solve constraints to get needed information

- ...is there a general framework?
  - Yes: dataflow analysis!