Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-16/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 15

- Dataflow Analysis for Global Optimization

Summary

- From local to global optimization: Liveness analysis
- Other examples of global optimizations:
  - Available expressions
  - Very busy expressions
  - Reaching definitions
- A general framework for Dataflow Analysis

(Local) Liveness/Next-Use Information

We have seen how to compute *next-use/liveness information* in a basic block:

- Next-use is computed by a backward scan of a basic block and performing the following actions on statement

  L: *x* := *y* op *z*

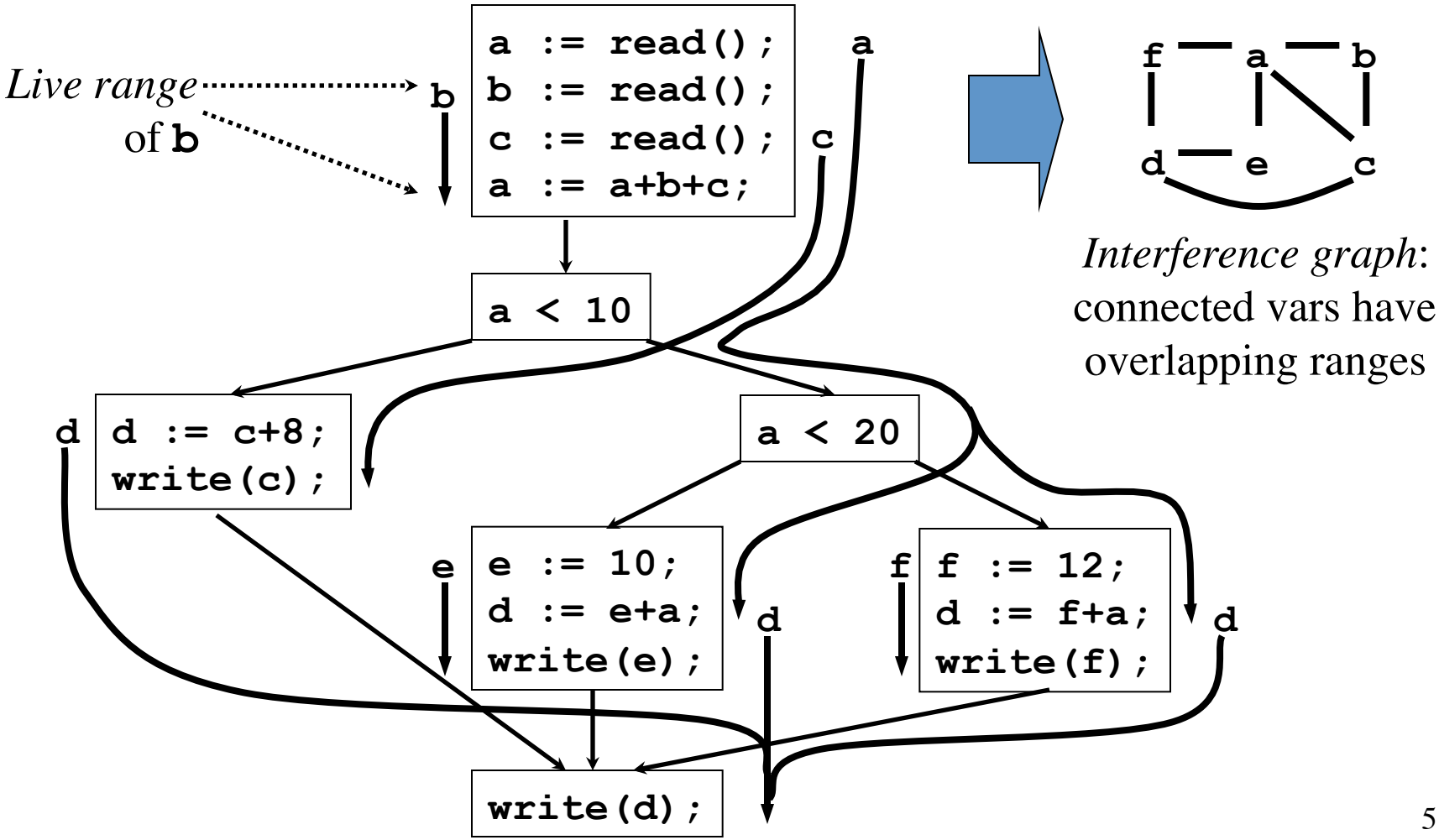
  - Add liveness/next-use info on *x*, *y*, and *z* to statement ***L***
    - This info can be stored in the symbol table
  - Before going up to the previous statement (scan up):
    - Set *x* info to "not live" and "no next use"
    - Set *y* and *z* info to "live" and the "next uses" of *y* and *z* to ***L***

Local Next-Use/Liveness Analysis: an example

i: **b := b + 1**	[$live(\mathbf{b}) = \text{true}, nextuse(\mathbf{b}) = j$]
j: **a := b + c**	[$live(\mathbf{a}) = \text{true}, live(\mathbf{b}) = \text{true}, live(\mathbf{c}) = \text{true},$ $nextuse(\mathbf{a}) = k, nextuse(\mathbf{b}) = k, nextuse(\mathbf{c}) = \text{none}$]
k: **t := a + b**	[$live(\mathbf{a}) = \text{true}, live(\mathbf{b}) = \text{true}, live(\mathbf{t}) = \text{true},$ $nextuse(\mathbf{a}) = \text{none}, nextuse(\mathbf{b}) = \text{none}, nextuse(\mathbf{t}) = \text{none}$]

- In absence of global information, all variables are assumed to be live at the end of the block

Determination of **Global** Live Ranges for Register Allocation with Graph Coloring



The Dataflow Framework

- There exists a general algorithm to extract information from programs.
  - This algorithm solves what we will call dataflow analysis.
- Many static analyses are dataflow analysis:
  - Liveness, available expressions, very busy expressions, reaching definitions.
- We start with *liveness,* then present some other kind of analyses, and will eventually derive a common pattern for them.

A Simple Example

```
var x,y,z;

x = input;

while (x > 1) {

    y = x / 2;

    if (y > 3)

        x = x - y;

    z = x - 4;

    if (z > 0)

        x = x / 2;

    z = z - 1;

}

output x;
```

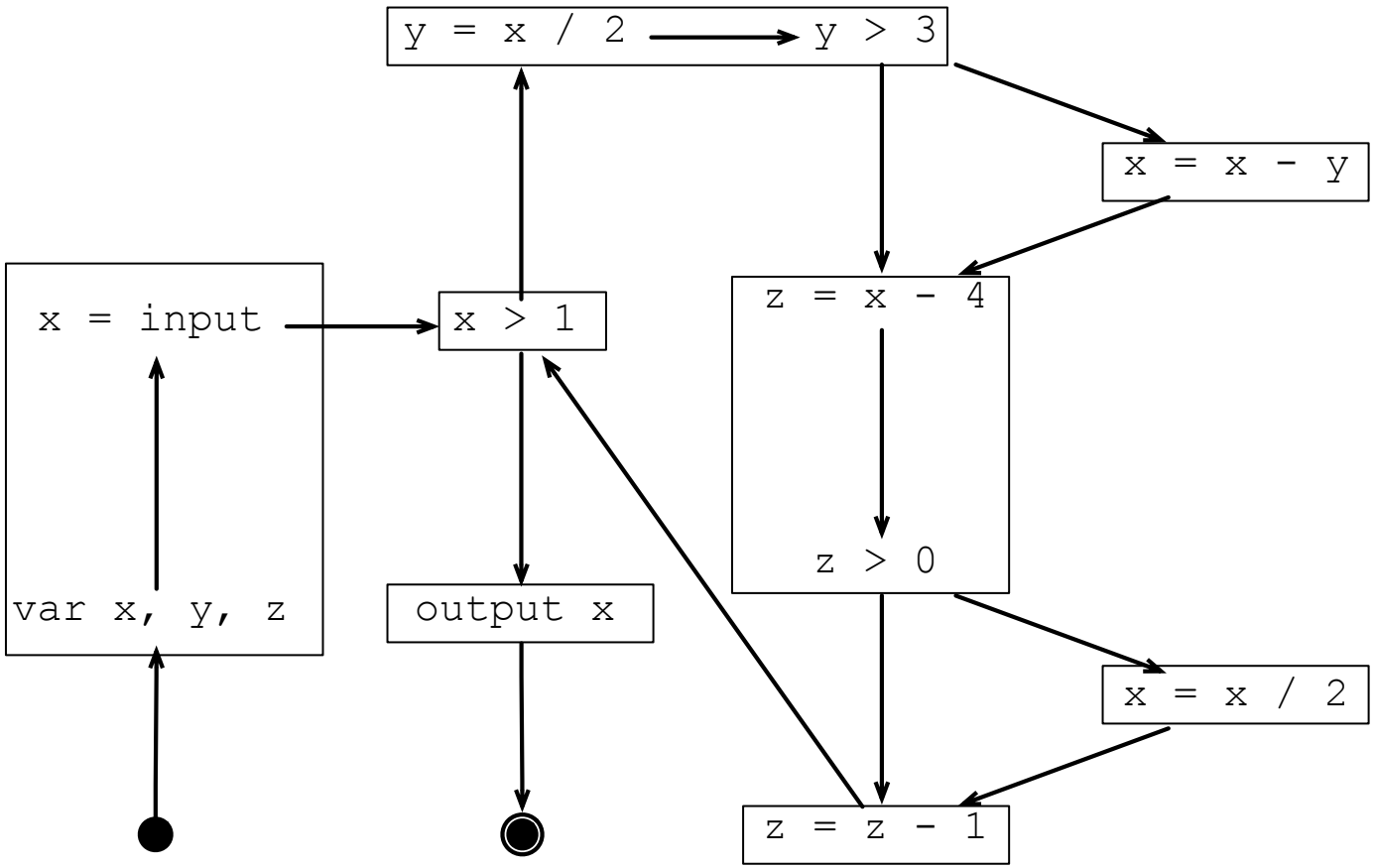
How does the control flow graph of this program look like?

- Determine the leaders
  - First statement
  - Each target of a goto
  - Each statement following a goto

- One block from each leader to the line preceding the next leader

A simple example with Control Flow Graph

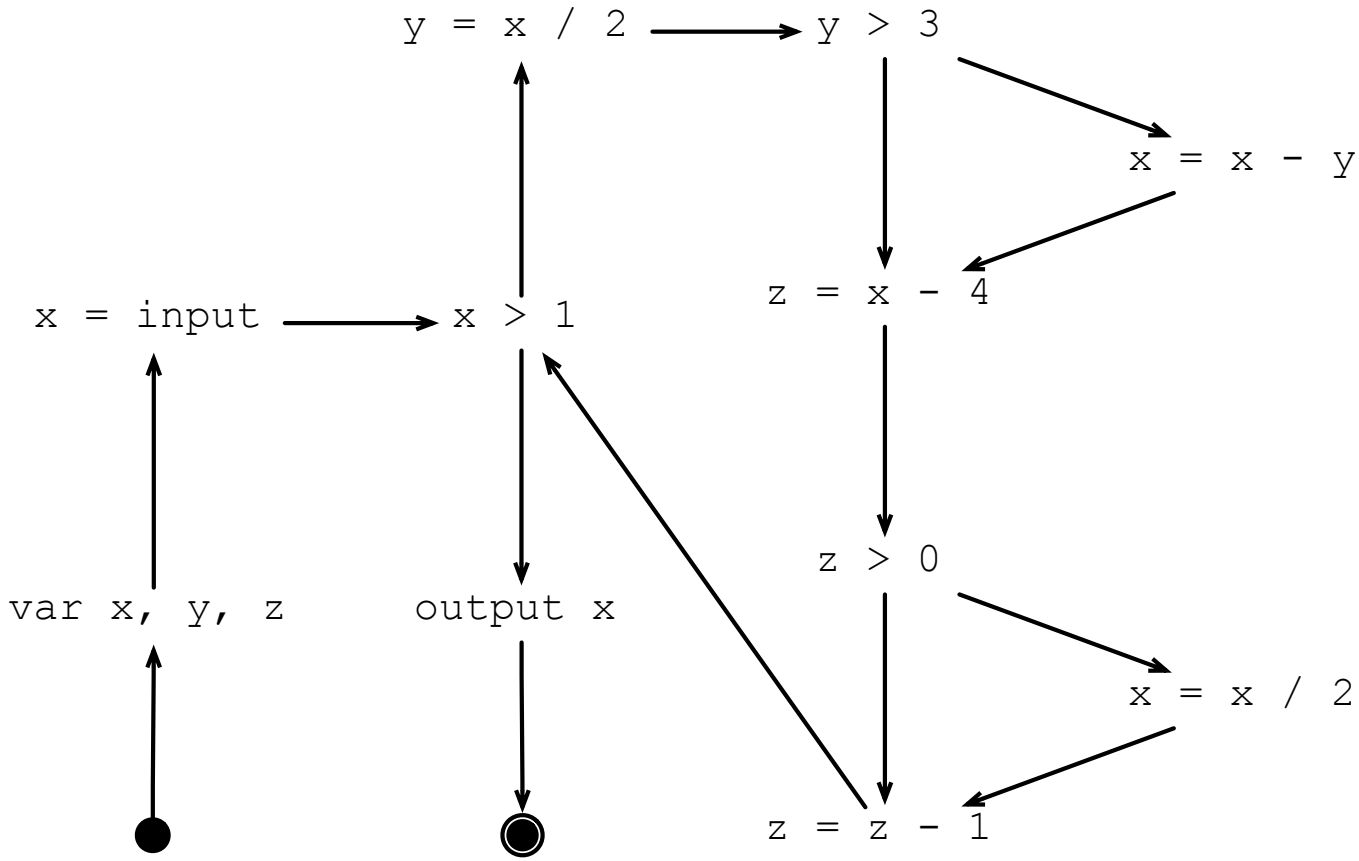
```
var x,y,z;

x = input;

while (x > 1) {
    y = x / 2;
    if (y > 3)
        x = x - y;
    z = x - 4;
    if (z > 0)
        x = x / 2;
    z = z - 1;
}

output x;
```



We can safely consider one block for each statement, omitting the boxes

A simple example with Control Flow Graph

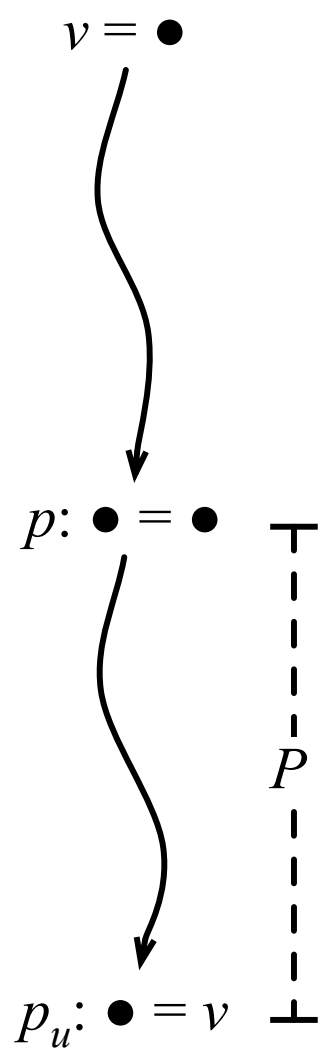
```
var x,y,z;

x = input;

while (x > 1) {

    y = x / 2;

    if (y > 3)

        x = x - y;

    z = x - 4;

    if (z > 0)

        x = x / 2;

    z = z - 1;

}

output x;
```



- How many registers do I need to compile this program?
- We can answer with (Global) Liveness Analysis

Liveness

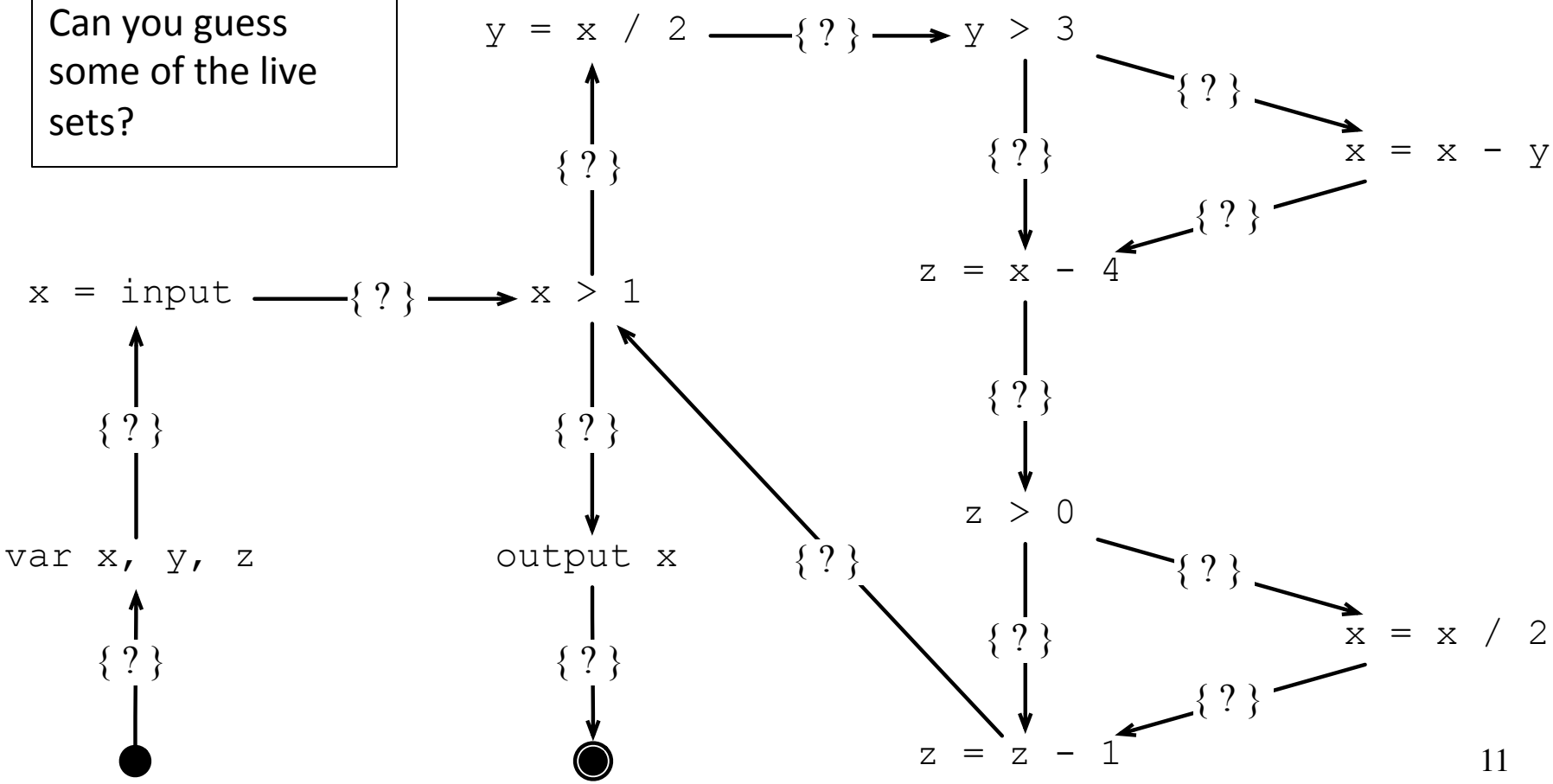
- If we assume an infinite set of registers, then a variable *v* should be in a register at a program point *p*, whenever:
  1. There is a path *P* from *p* to another program point p_u, where *v* is used.
  2. The path *P* does not include any definition of *v*.
- Conditions 1 and 2 determine when a variable *v* is <u>alive</u> at a program point *p*.



Liveness Analysis

- Given a control flow graph G, find, for each edge E in G, the set of variables alive at E.

Can you guess some of the live sets?

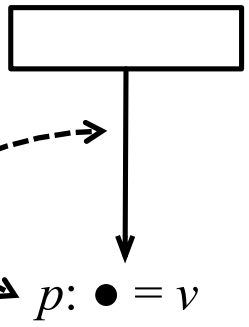


Determining Live Sets

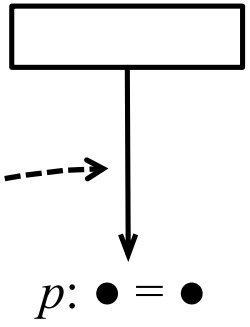
- To determine the sets of live variables on all edges of the control flow graph we consider
  - The origin of information: where liveness holds because of immediate observation
  - The propagation of information: how liveness info is passed along a sequence of statements ("locally")
  - Joining information: how liveness info is passed across the boundaries of basic blocks ("globally")
- The same pattern will be used for other kind of analyses, and will be the essence of the Dataflow Analysis Framework

The Origin of Information

If a variable is used
at a program point *p*,
then it must be alive
<u>immediately</u> before *p*.



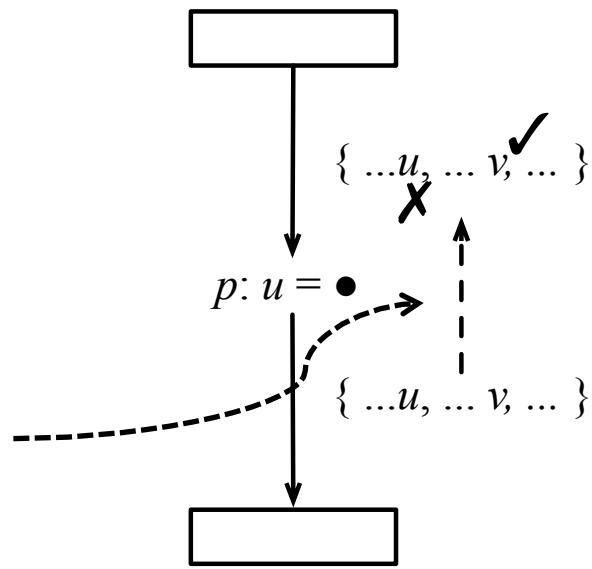
If the variable is not
used at *p*, when is it
going to be alive
immediately before *p*?



The Propagation of Information

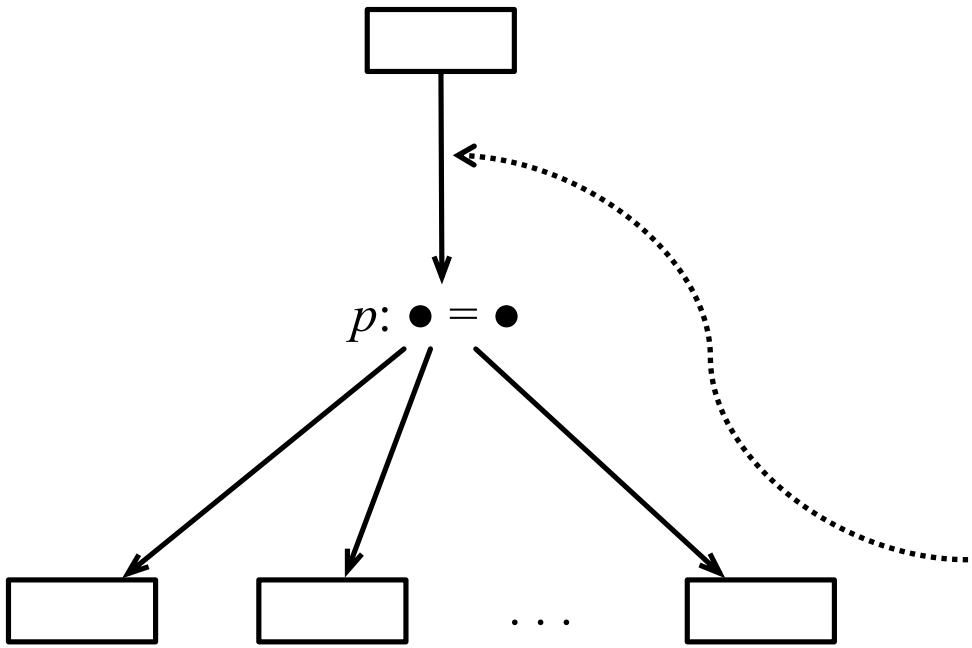
- A variable is alive immediately before a program point *p* if, and only if:
  1. It is alive immediate after *p*.
  2. It is not redefined at *p*.
- or
  1. It is used at *p*.

What if a program point has multiple successors?



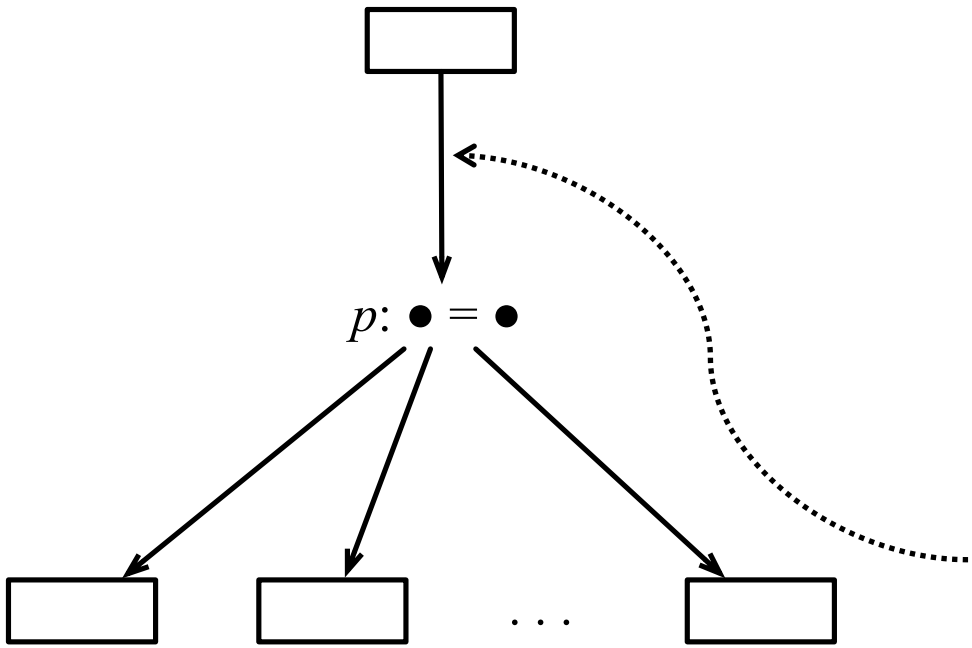
This is the direction through which information propagates.

Joining Information



How do we find the variables that are alive at this program point?

Joining Information



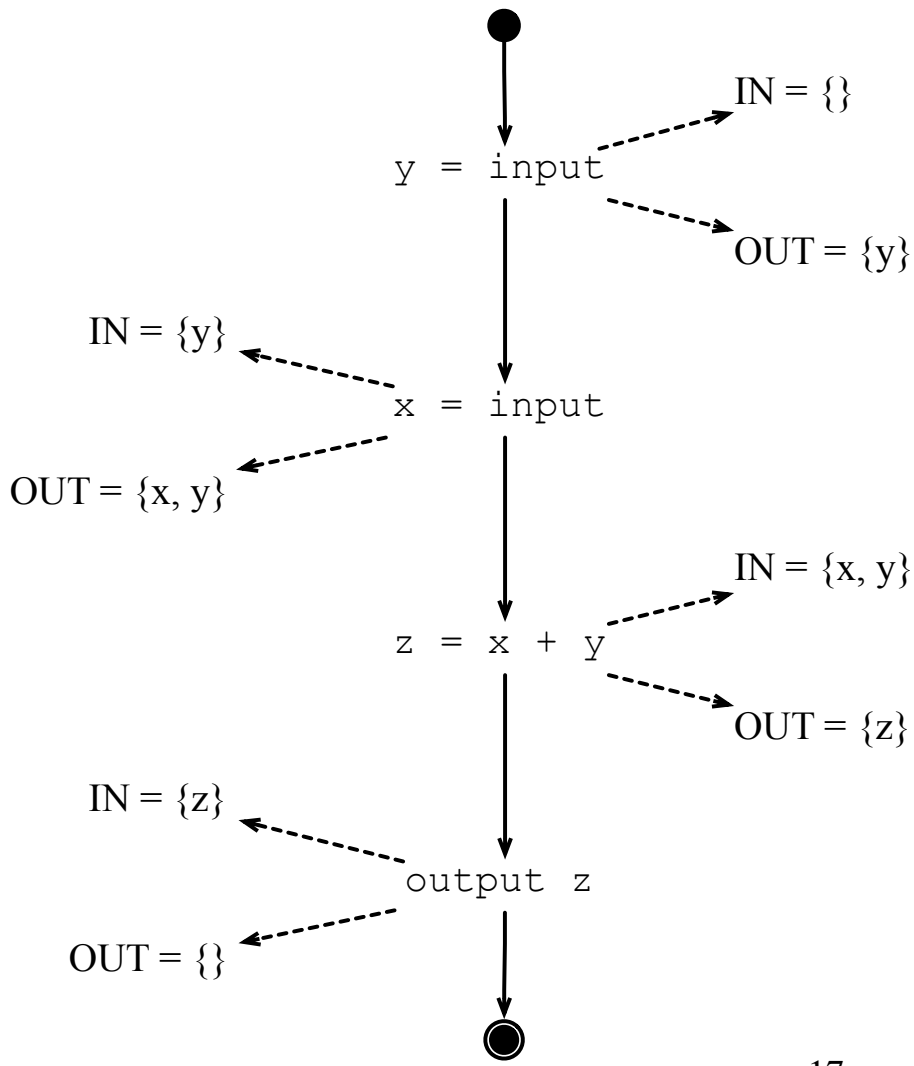
But, what if a program point has multiple successors?

How do we find the variables that are alive at this program point?

If a variable v is alive immediately before any predecessor of *p*, then it must be alive immediately after *p*.

IN and OUT Sets for Liveness

- To solve the liveness analysis problem:
- We associate with each program point *p* two sets, IN and OUT.
  - IN is the set of variables alive immediately **before** *p*.
  - OUT is the set of variables alive immediately **after** *p*.
- We state equations that relate these sets
- We compute the sets as solutions of a system of equations



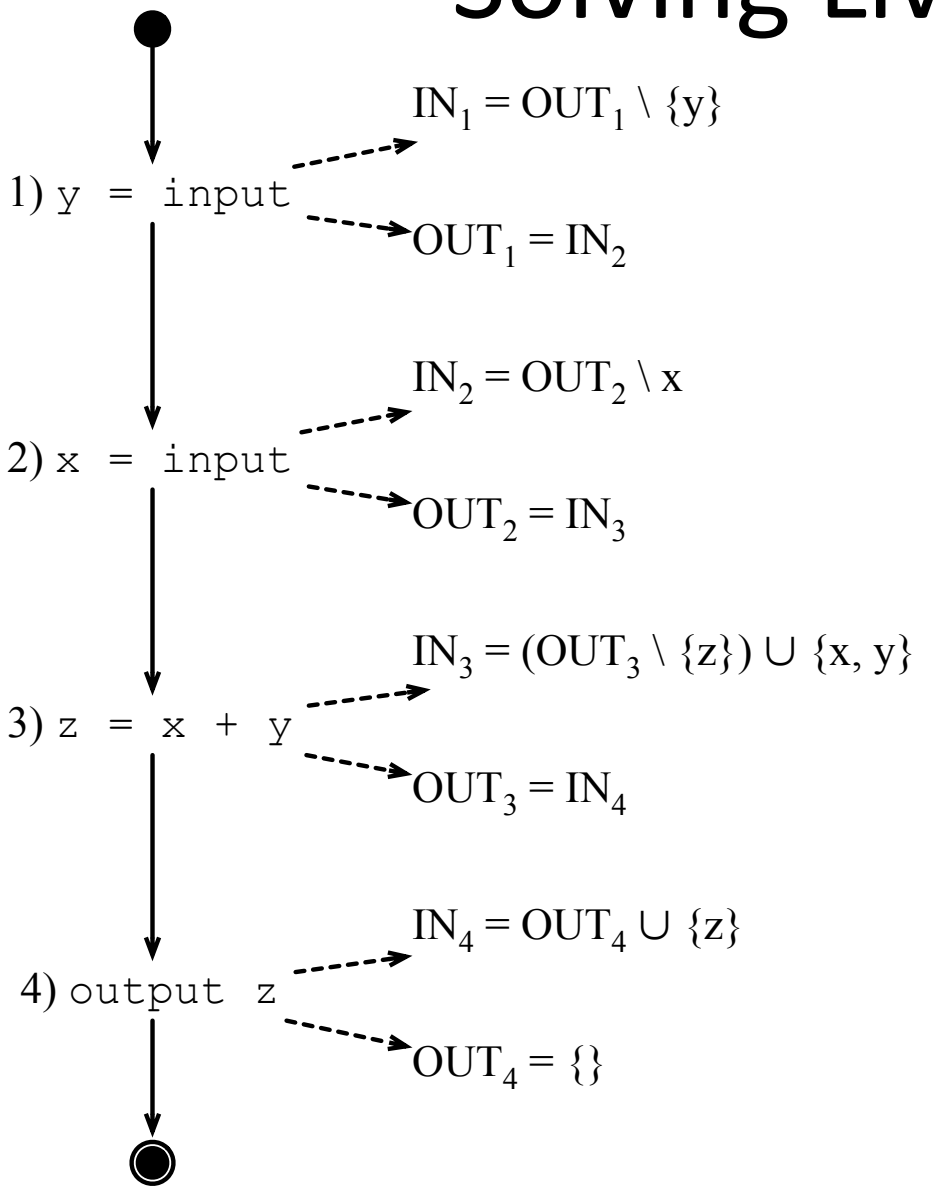
Dataflow Equations for Liveness Analysis

$$p : v = E$$

$$\begin{aligned} IN(p) &= (OUT(p) \setminus \{v\}) \cup vars(E) \\ OUT(p) &= \bigcup IN(p_s), p_s \in succ(p) \end{aligned}$$

- *IN(p)* = set of variables alive immediately before p
- *OUT(p)* = set of variables alive immediately after p
- *vars(E)* = the variables that appear in E
- *succ(p)* = the set of control flow nodes that are successors of *p*

Solving Liveness

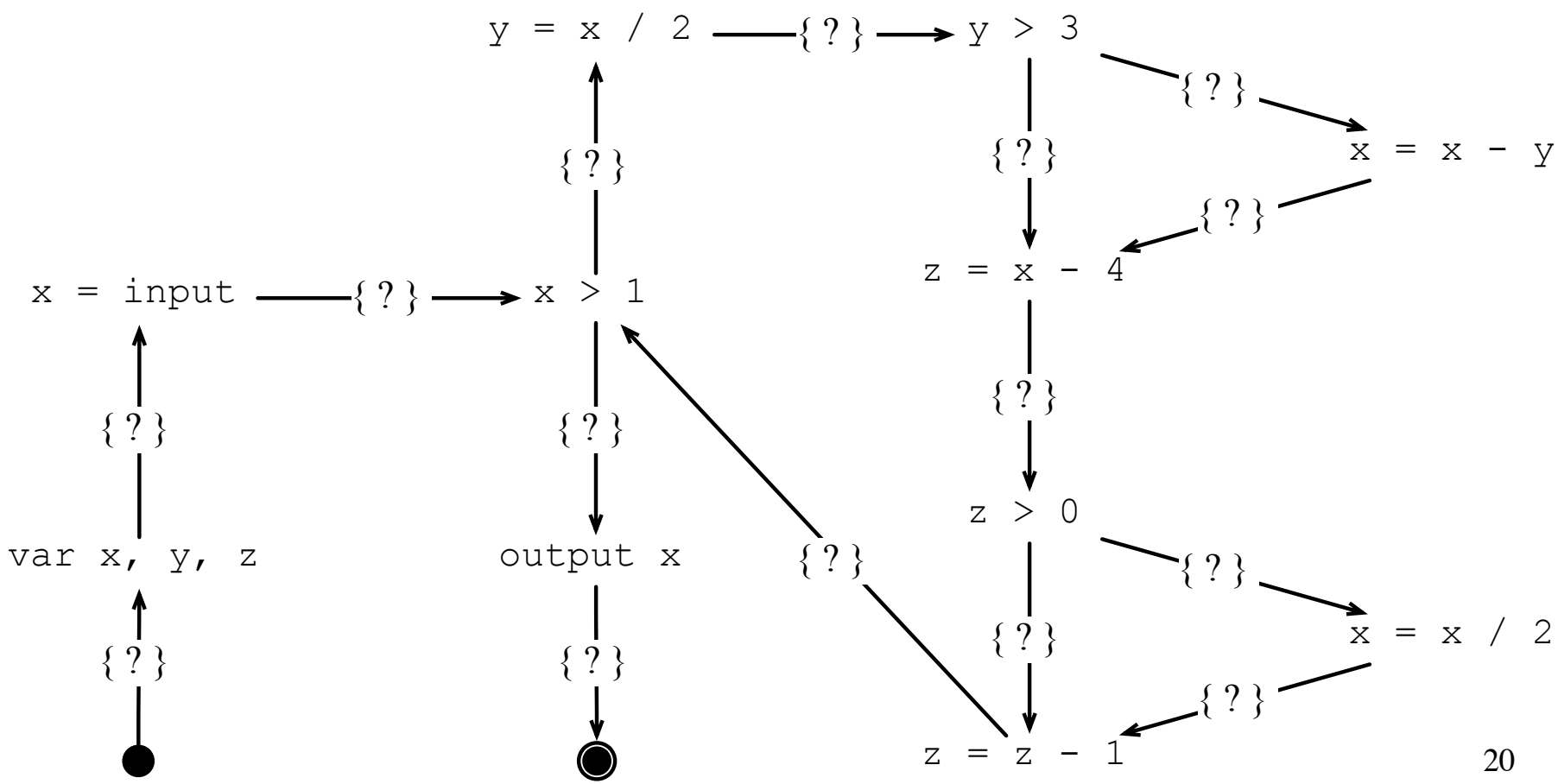


1. We initialize each IN and OUT set to empty
2. We evaluate all the equations
3. If any IN or OUT set has changed during this evaluation, then we repeat (2), otherwise we are done

- Is termination guaranteed?

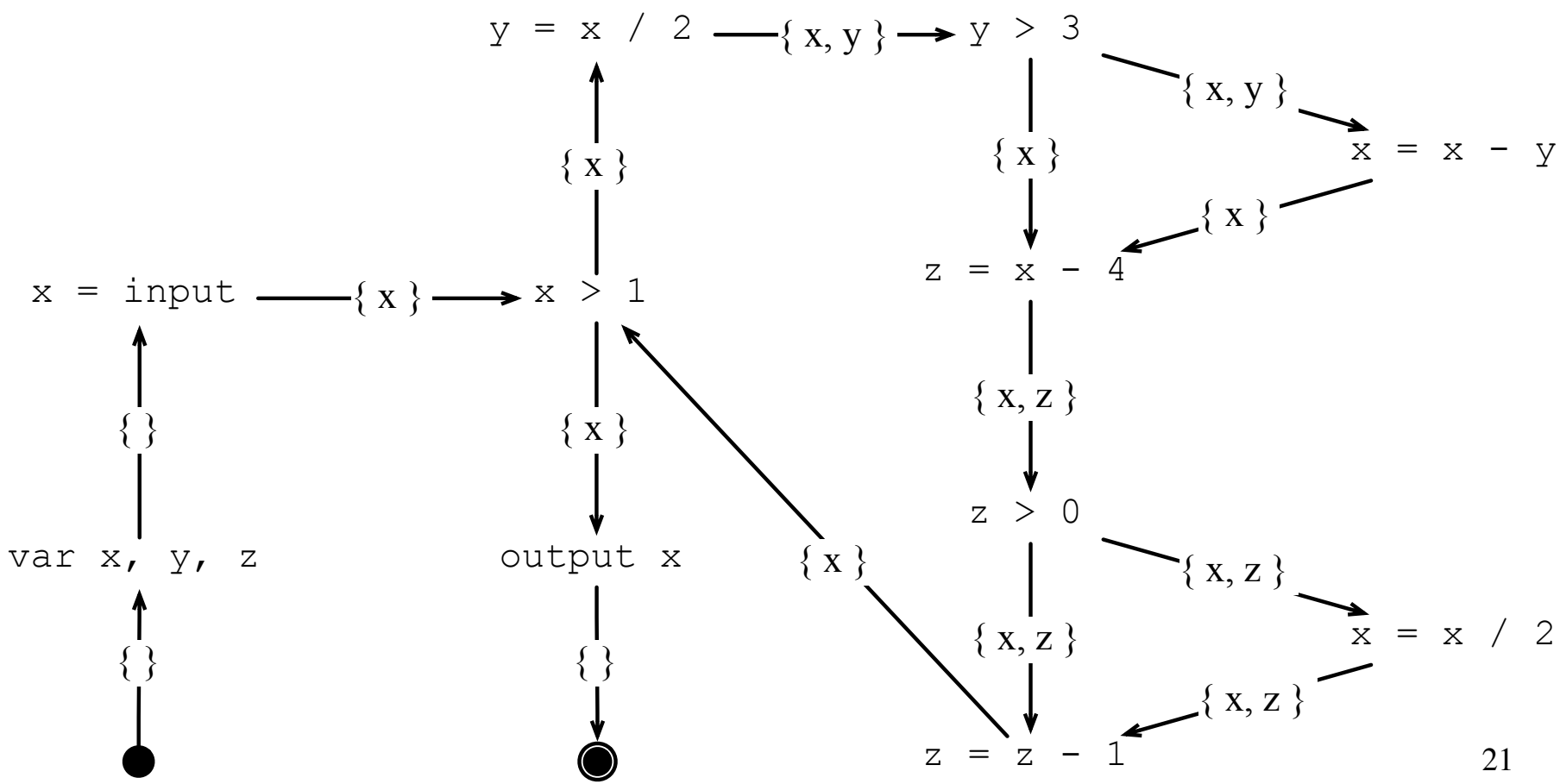
Example of Liveness Problem

- Let's solve liveness analysis for the program below:



Example of Liveness Problem

- Let's solve liveness analysis for the program below:

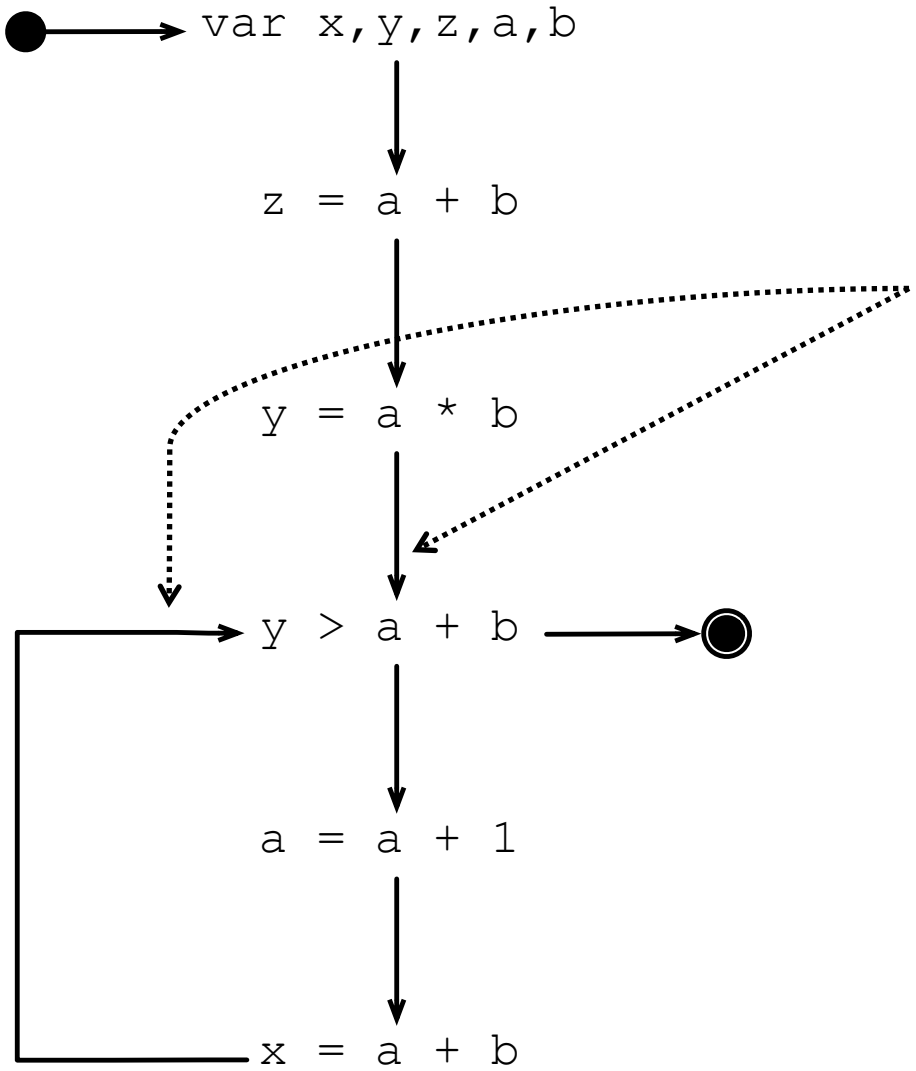


Another Global Analysis: Available Expressions

```
var x, y, z, a, b;

z = a + b

y = a * b

while (y > a + b) {

    a = a + 1

    x = a + b

}
```

- Consider the program on the left. How could we optimize it?
- Which information does this optimization require?
- How is the control flow graph of this program?

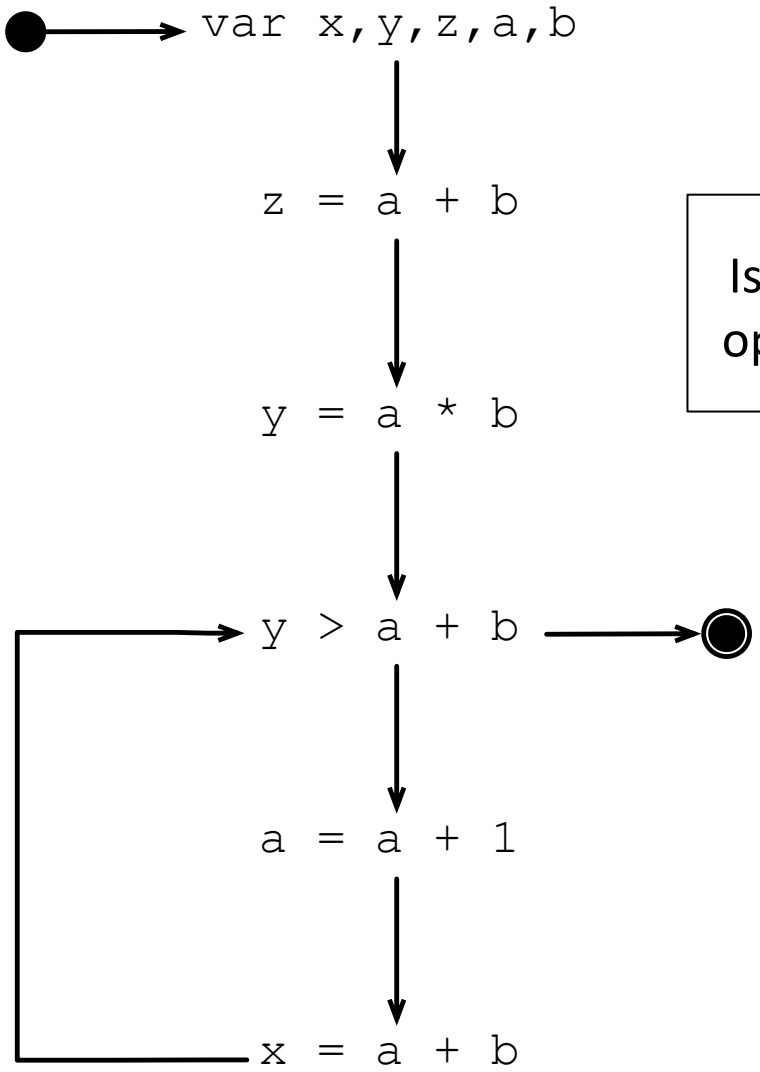
Available Expressions



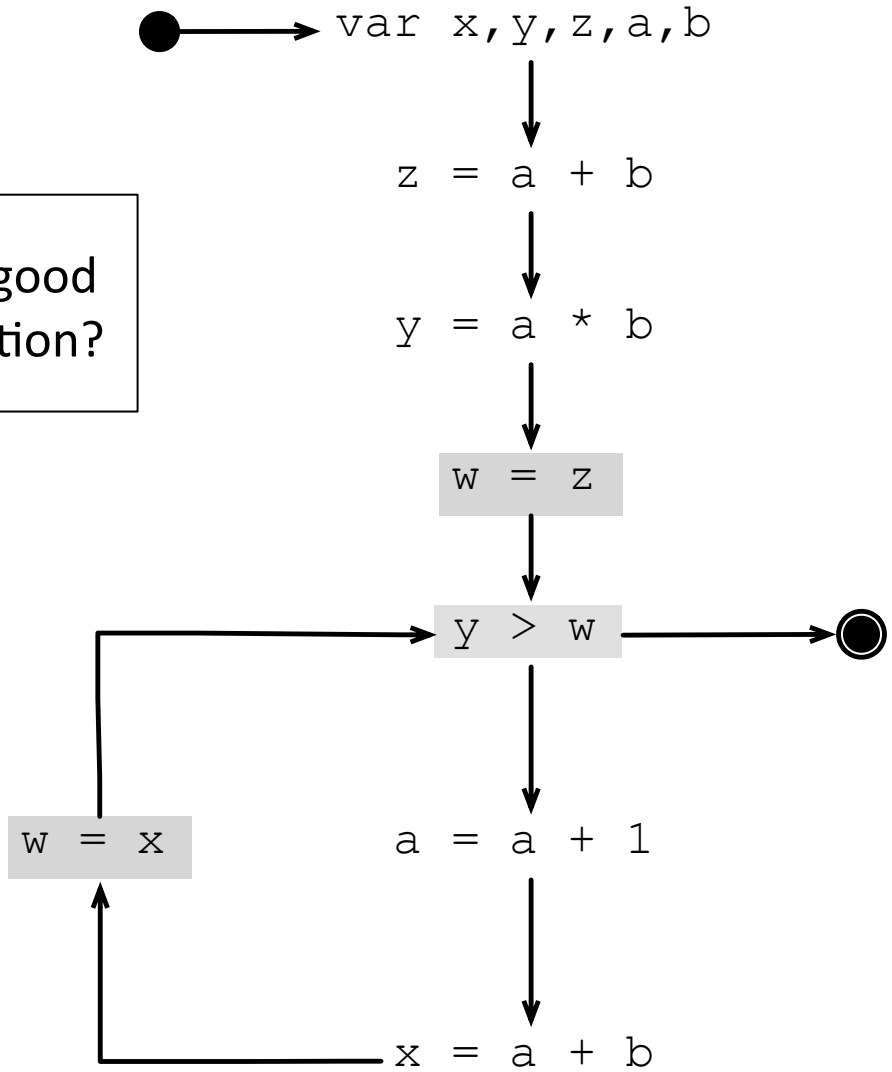
We know that the expression **a + b** is available at **these** two program points

- How could we improve this code?

Available Expressions

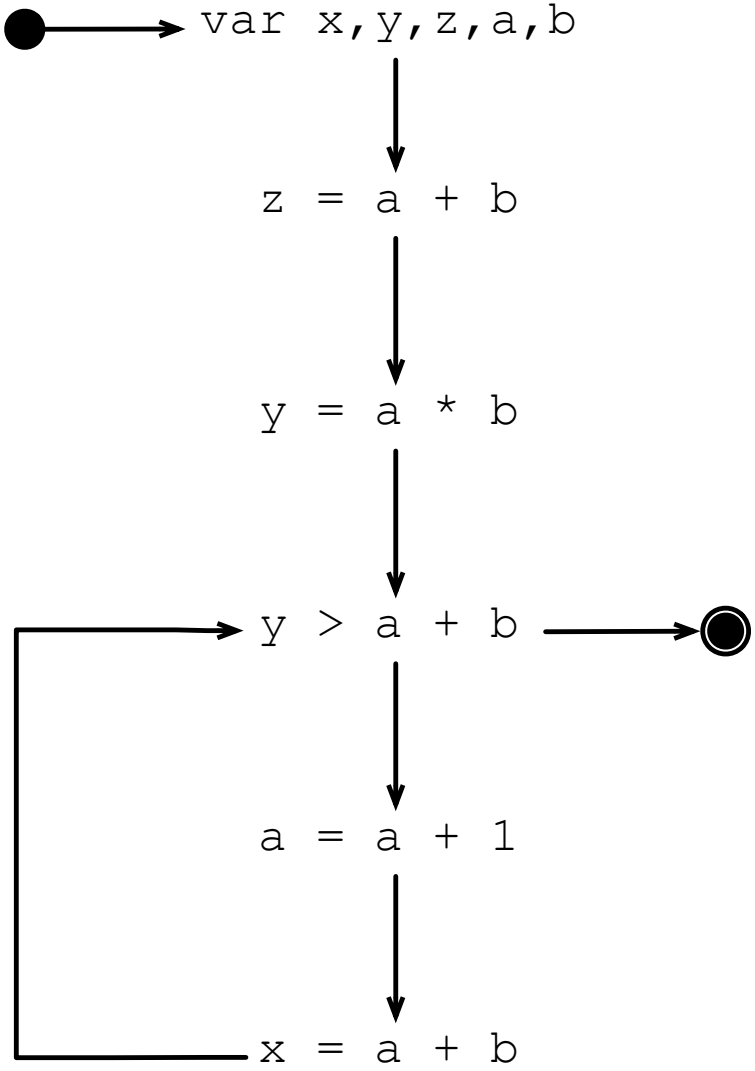


Is this a good optimization?



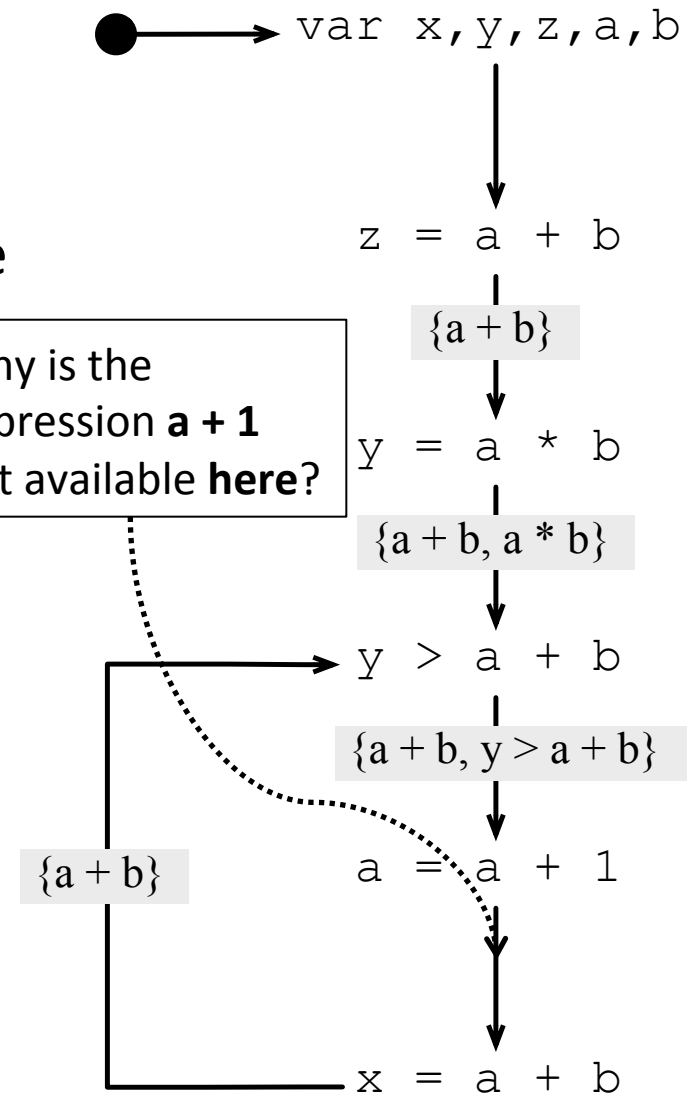
Available Expressions

- In order to apply the previous optimization, we had to know which expressions were available at the places where we removed expressions by variables.
- An expression is available at a program point if its <u>current</u> value has already been computed earlier in the execution.
- Which expressions are available in our example?



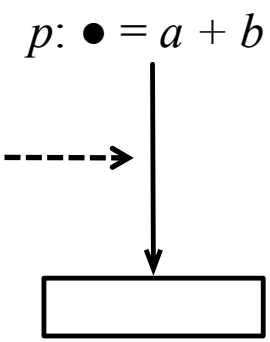
Available Expressions

- In order to apply the previous optimization, we had to know which expressions were available at the places where we removed expressions by variables.
- An expression is available at a program point if its <u>current</u> value has already been computed earlier in the execution.
- Which expressions are available in our example?
- We can approximate sets of available expressions with dataflow analysis



The Origin of Information

If an expression is used at a point *p*, then it is **available** <u>immediately</u> after *p*, as long as *p* does not redefine any of the variables that the expression uses.

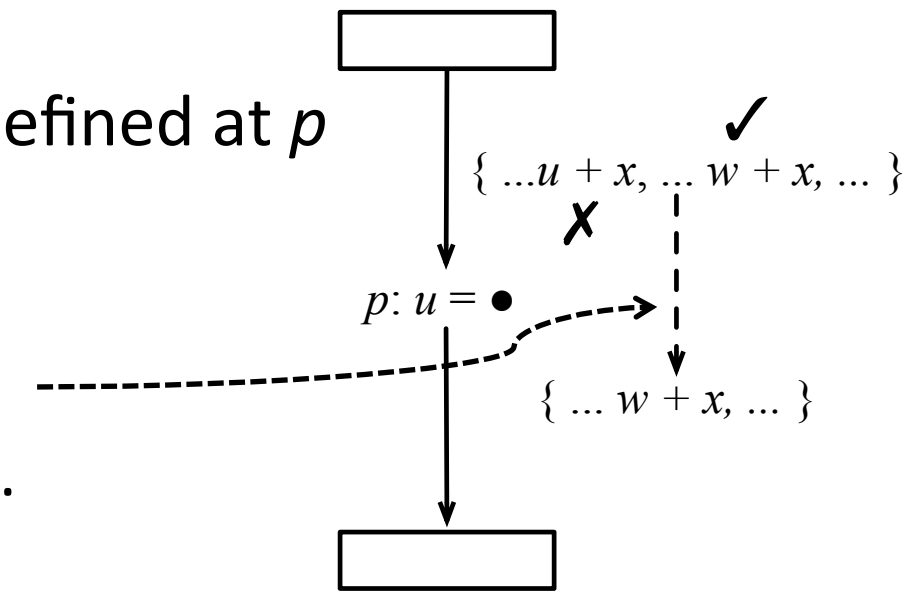


The Propagation of Information

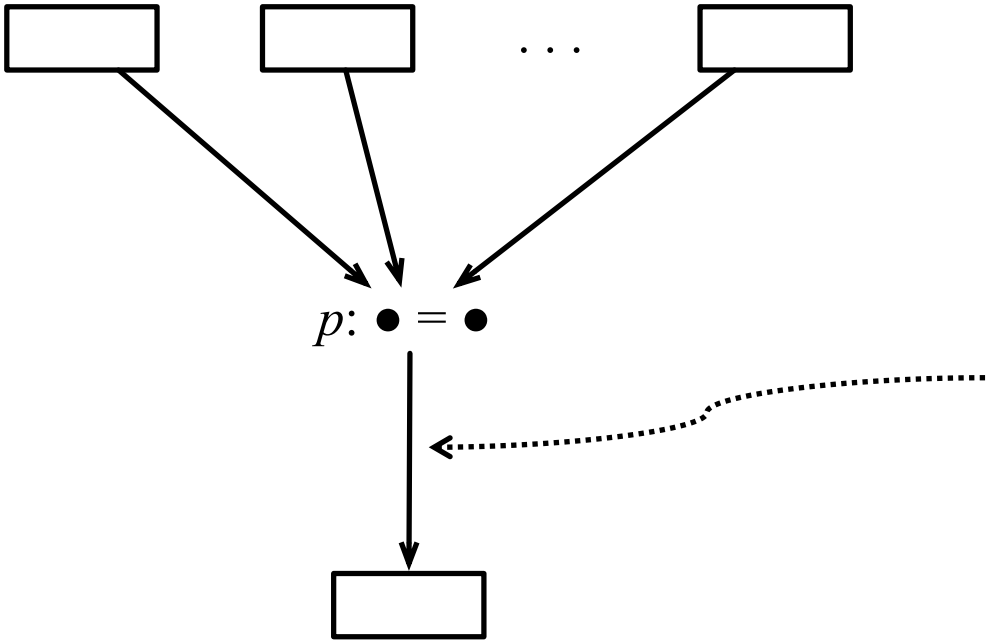
- An expression *E* is **available** immediately after a program point *p* if, and only if:
  1. It is available immediately before *p*
  2. No variable of *E* is redefined at *p*
- or
  1. It is used at *p*
  2. No variable of *E* is redefined at *p*

What if a program point has multiple predecessors?

This is the direction through which information propagates.



Joining Information

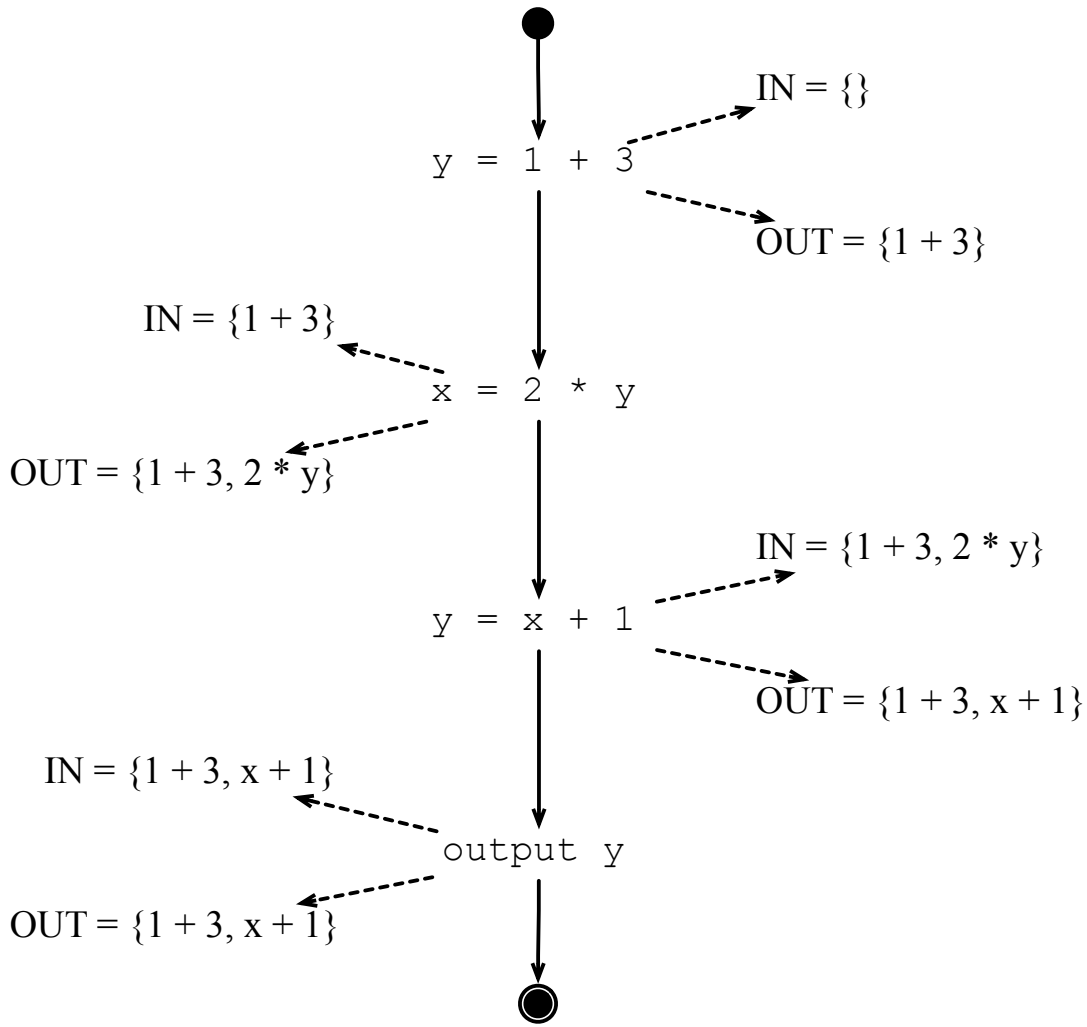


How do we find the expressions that are available at this program point?

If an expression E is available immediately after every predecessor of *p*, then it must be available immediately before *p*.

IN and OUT Sets for Availability

- To solve the available expression analysis, we associate with each program point *p* two sets, IN and OUT.
  - IN is the set of expressions available immediately before *p*.
  - OUT is the set of expressions available immediately after *p*.



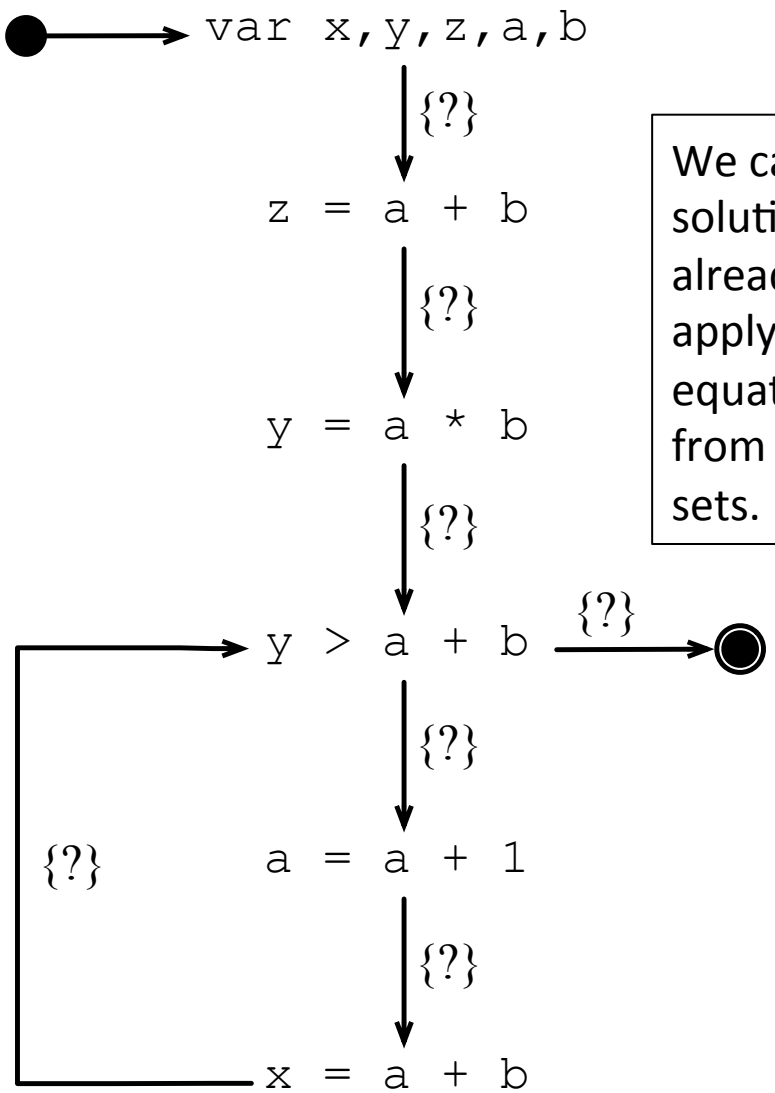
Dataflow Equations for Availability

$$p : v = E$$

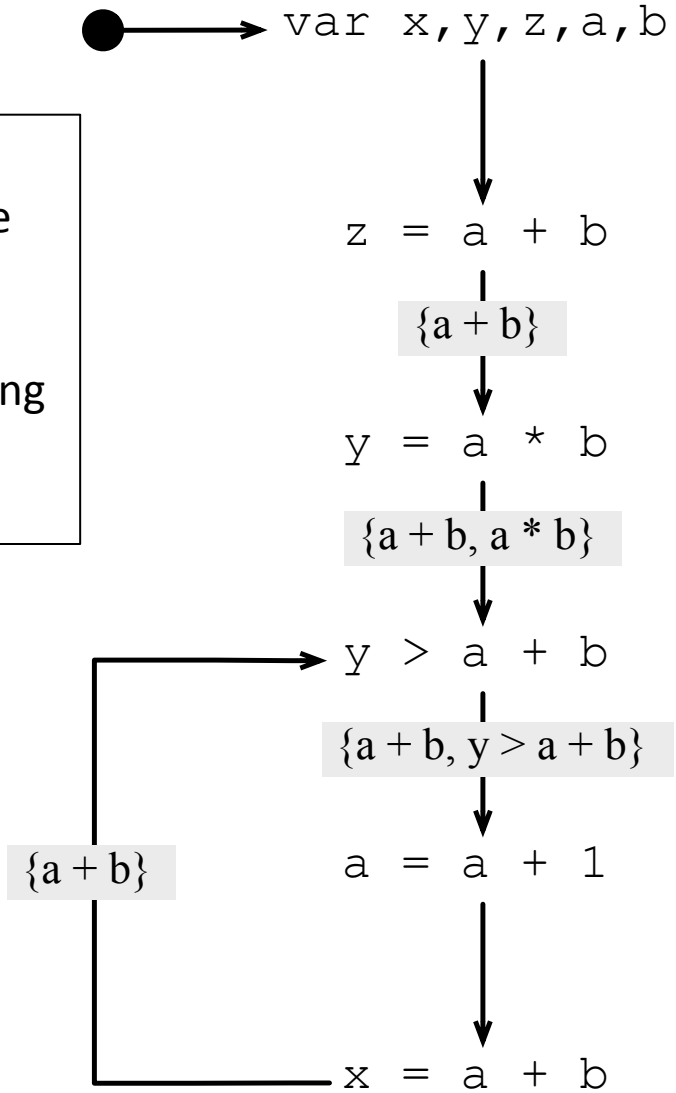
$$\begin{aligned} IN(p) &= \bigcap OUT(p_s), p_s \in pred(p) \\ OUT(p) &= (IN(p) \cup \{E\}) \setminus \{Expr(v)\} \end{aligned}$$

- *IN(p)* = the set of expressions available immediately before p
- *OUT(p)* = the set of expressions available immediately after p
- *pred(p)* = the set of control flow nodes that are predecessors of p
- *Expr(v)* = the set of expressions that use variable v.

Example of Available Expressions



We can get the solution we have already seen by applying the equations, starting from the empty sets.

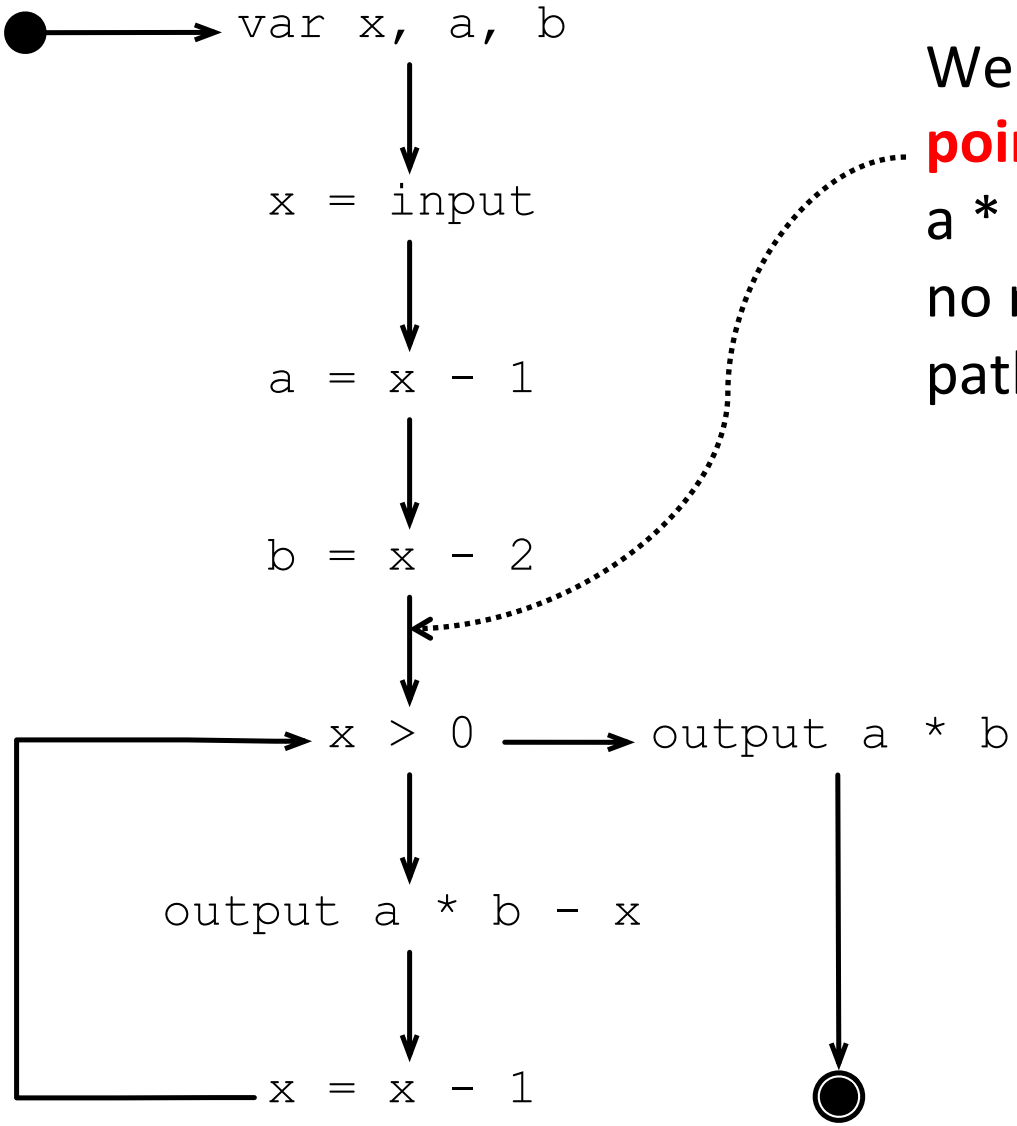


Very Busy Expressions

```
var x, a, b

x = input

a = x - 1

b = x - 2

while (x > 0) {

    output a * b - x

    x = x - 1

}

output a * b
```

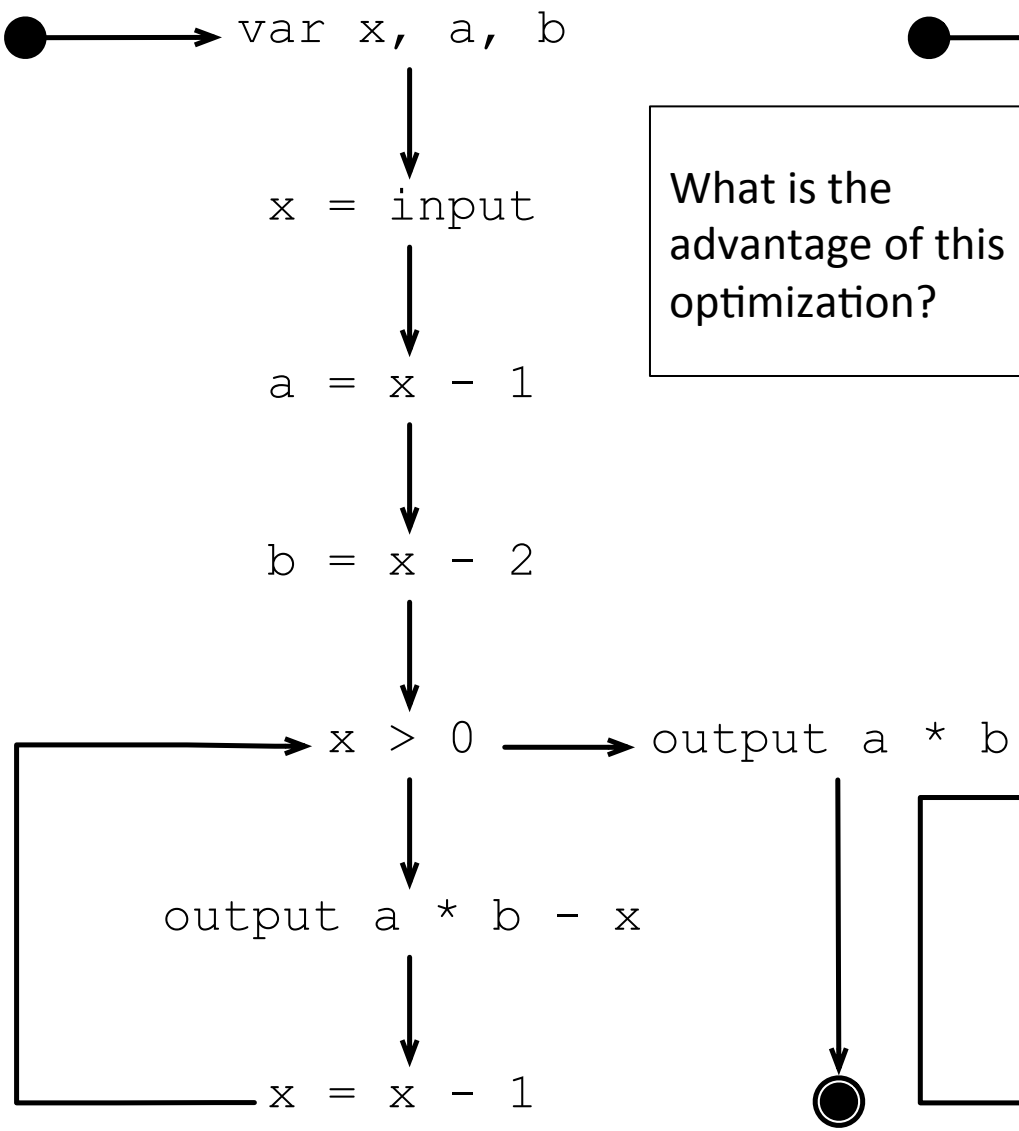
- Consider the program on the left. How could we optimize it?
- Which information does this optimization require?
- Consider the expression a * b
  - Does it change inside the loop?
- How is the control flow graph of this program?

Very Busy Expressions

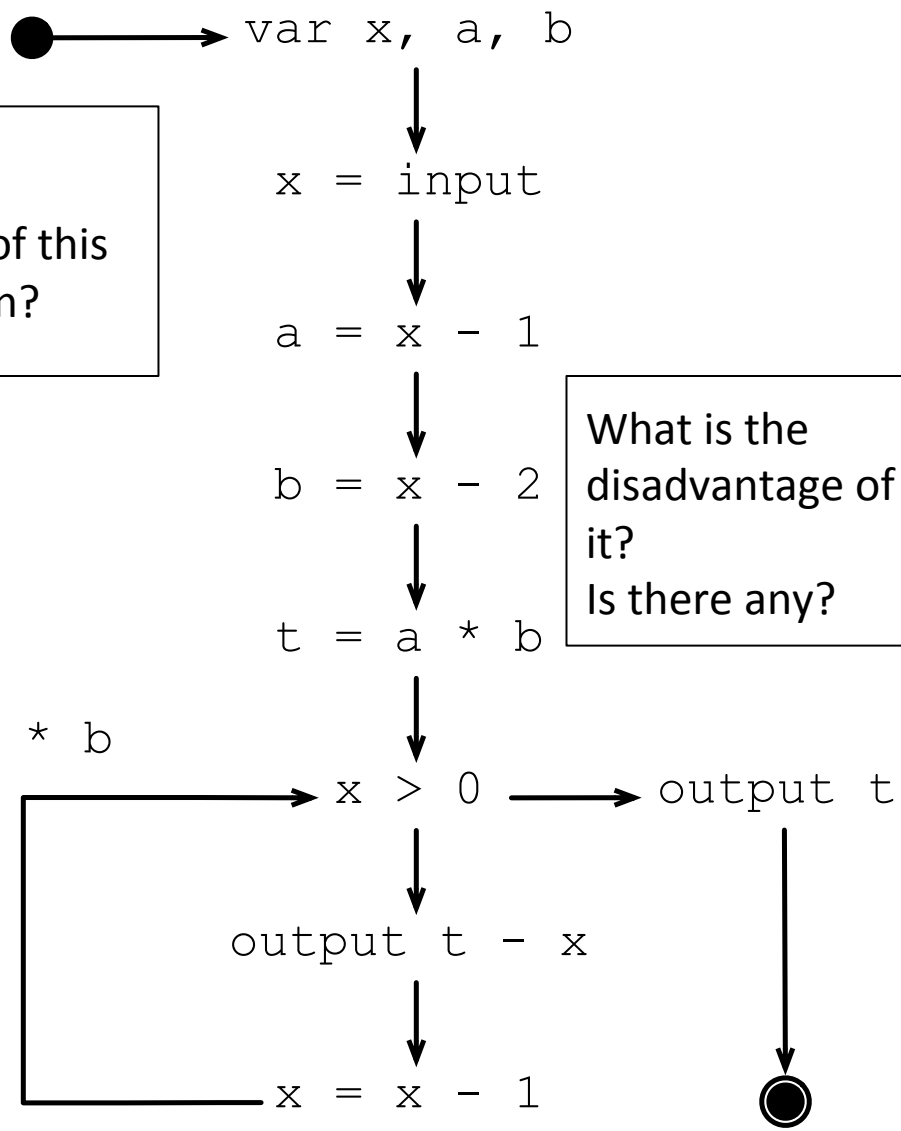


We know that at this **point**, the expression a * b will be computed no matter which program path is taken.

Very Busy Expressions



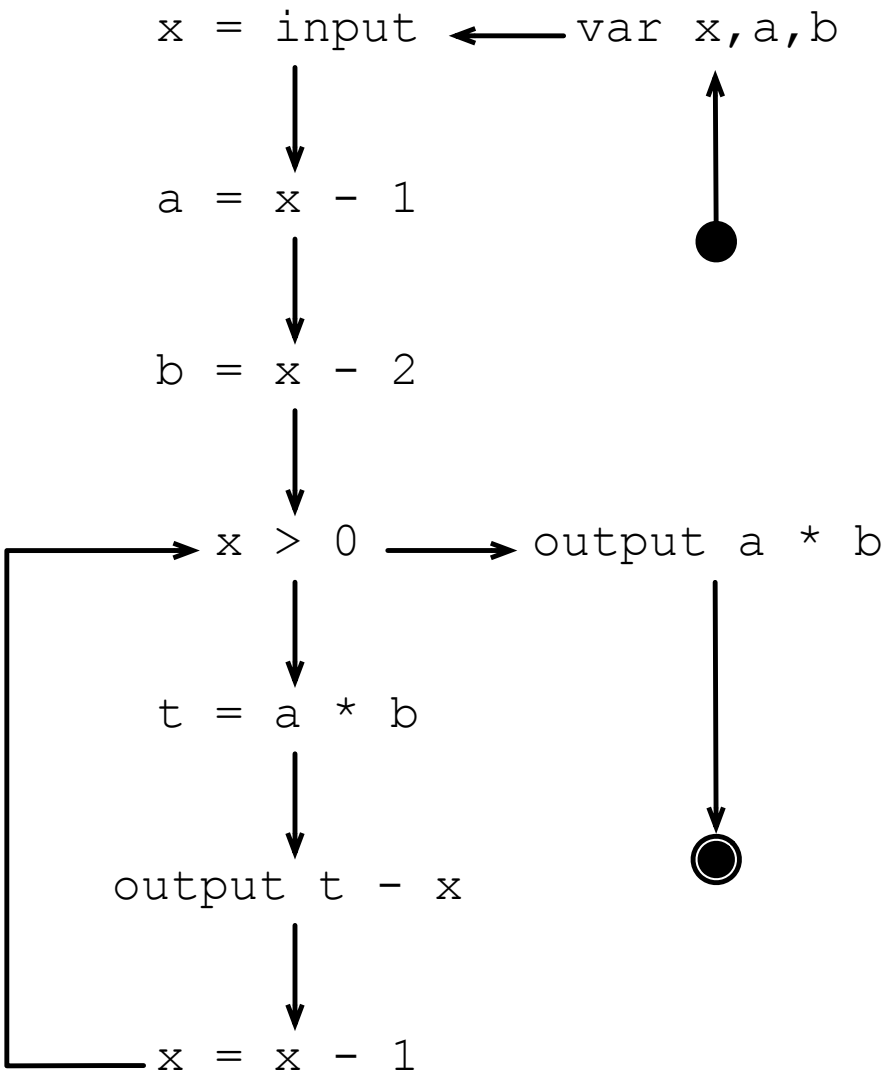
What is the advantage of this optimization?



What is the disadvantage of it?
Is there any?

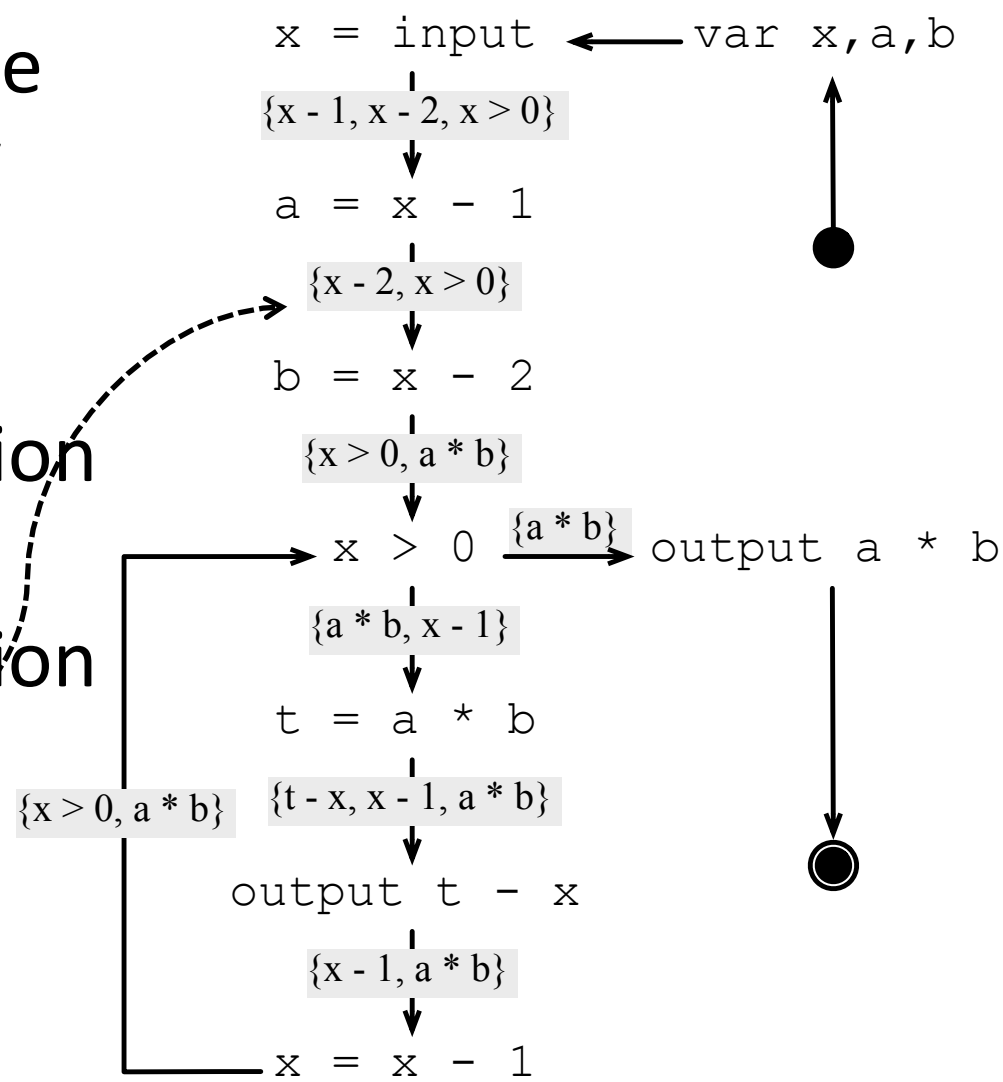
Very Busy Expressions

- In order to apply the previous optimization, we had to know that **a * b** was a *very busy expression* before the loop.
- An expression is *very busy* at a program point if it will be computed before the program terminates along any path that goes from that point to the end of the program.



Very Busy Expressions

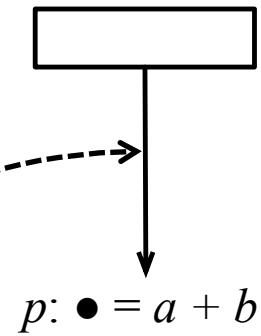
- We can approximate the set of very busy expressions by a dataflow analysis.
- How does information originate?
- How does information propagate?



Why is the expression **a * b** not very busy **here**?

The Origin of Information

If an expression is used at a point *p*, then it is very busy <u>immediately</u> before *p*.



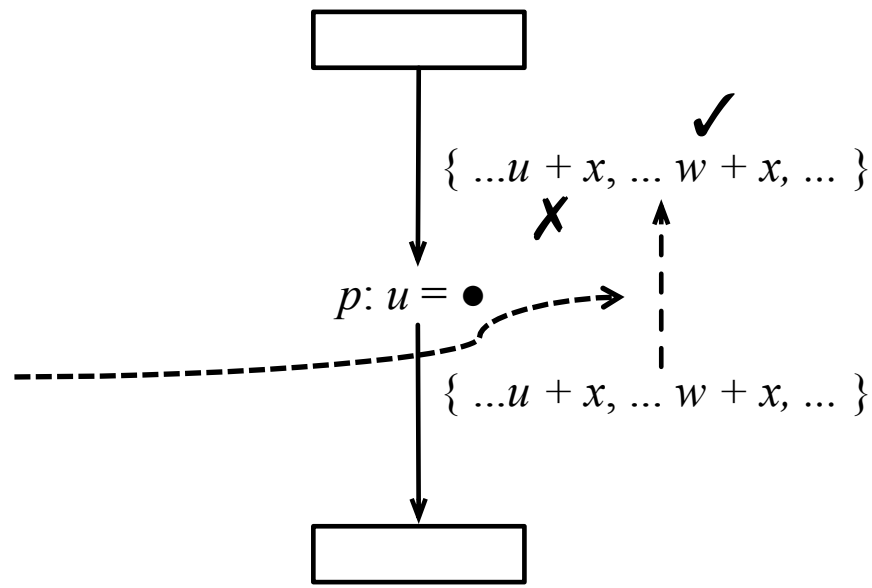
The Propagation of Information

- An expression *E* is very busy immediately before a program point *p* if, and only if:
  1. It is very busy immediately after *p*.
  2. No variable of E is redefined at *p*.

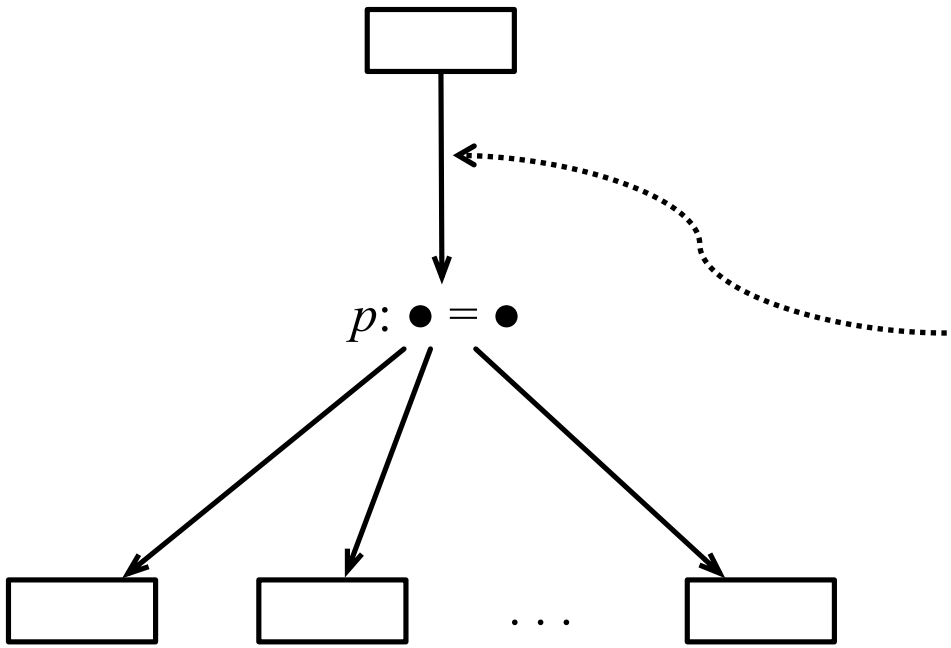
1. or
   1. It is used at *p*.

What if a program point has multiple successors?

This is the direction through which information propagates.



Joining Information

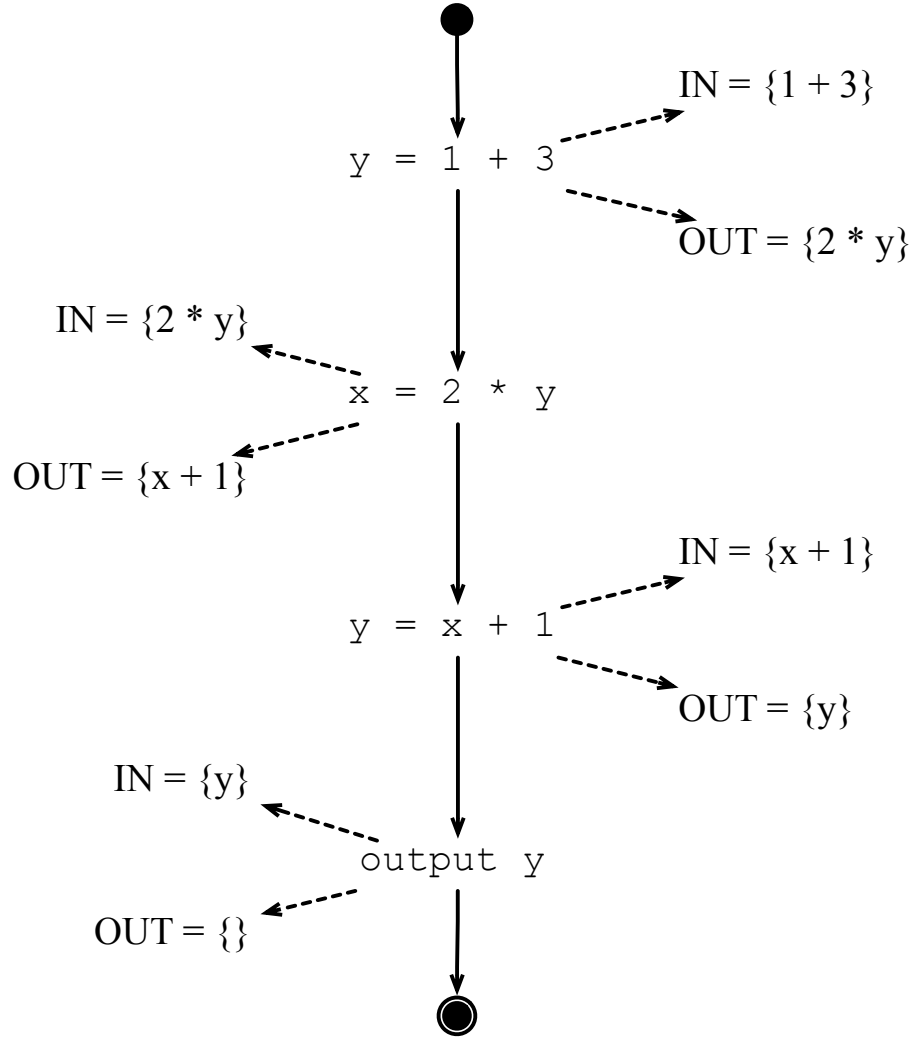


How do we find the expressions that are very busy at this program point?

If an expression *E* is very busy immediately before every successor of *p*, then it must be very busy immediately after *p*.

IN and OUT Sets for Very Busy Expressions

- To solve the very busy expression analysis, we associate with each program point *p* two sets, IN and OUT.
  - IN is the set of very busy expressions immediately before *p*.
  - OUT is the set of very busy expressions immediately after *p*.



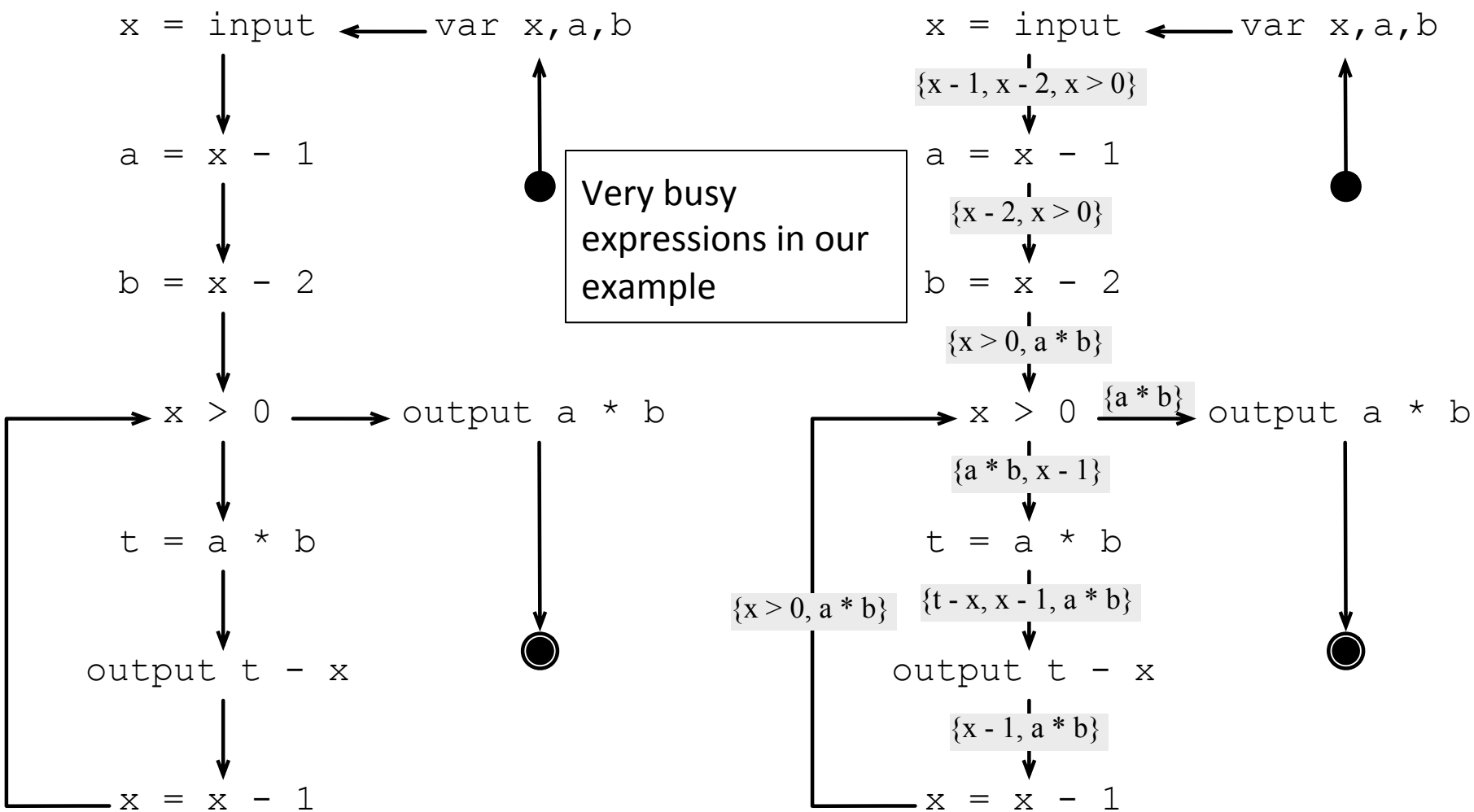
Dataflow Equations for Very Busy Expressions

$$p : v = E$$

$$\begin{aligned} IN(p) &= (OUT(p) \setminus \{v\}) \cup \{E\} \\ OUT(p) &= \bigcap IN(p_s), p_s \in succ(p) \end{aligned}$$

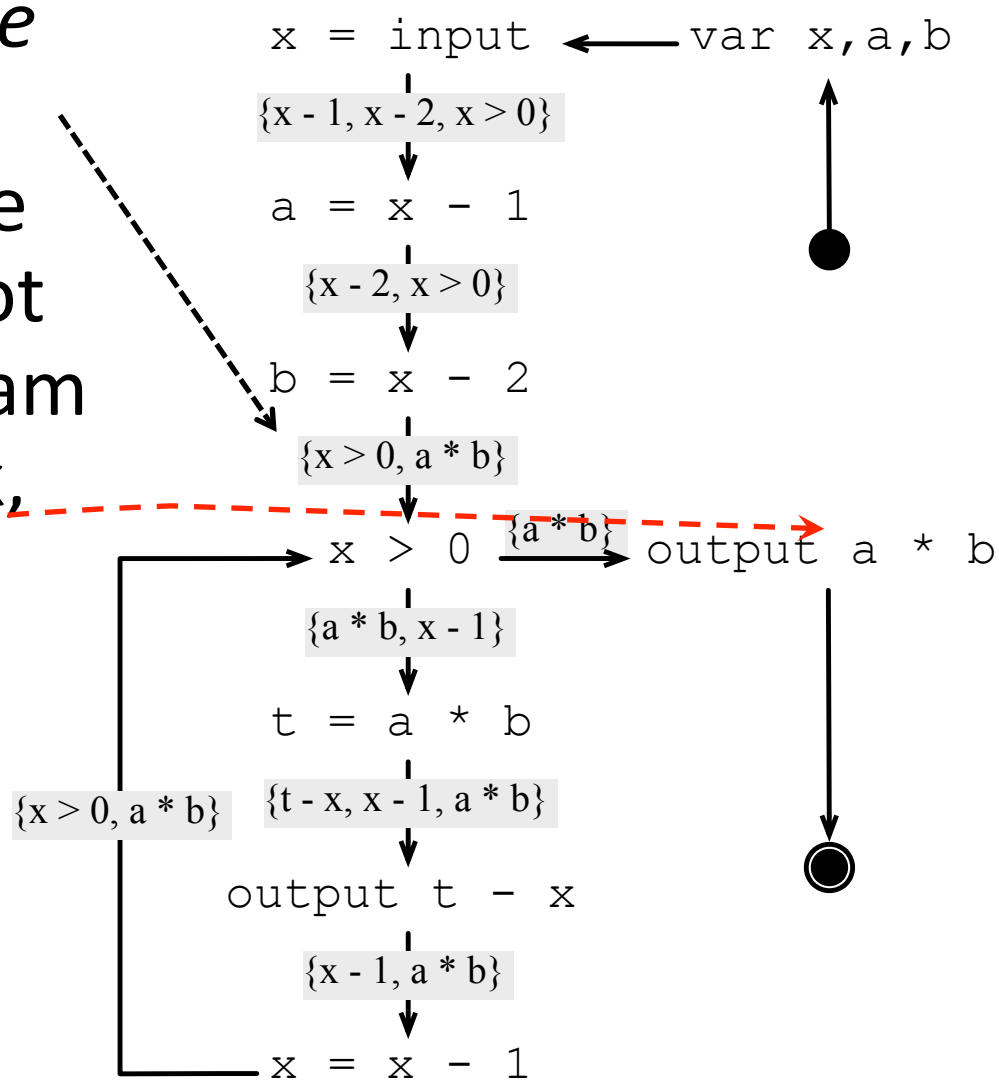
- *IN(p)* = the set of very busy expressions immediately before *p*
- *OUT(p)* = the set of very busy expressions immediately after *p*
- *succ(p)* = the set of control flow nodes that are successors of p

Example of Very Busy Expressions



Safe Code Hoisting

- It is *performance safe* to move a * b to this <u>program point</u>, in the sense that we will not be forcing the program to do any extra work, in any circumstance.



If we did not have this **a * b here**, the transformation would not be performance safe

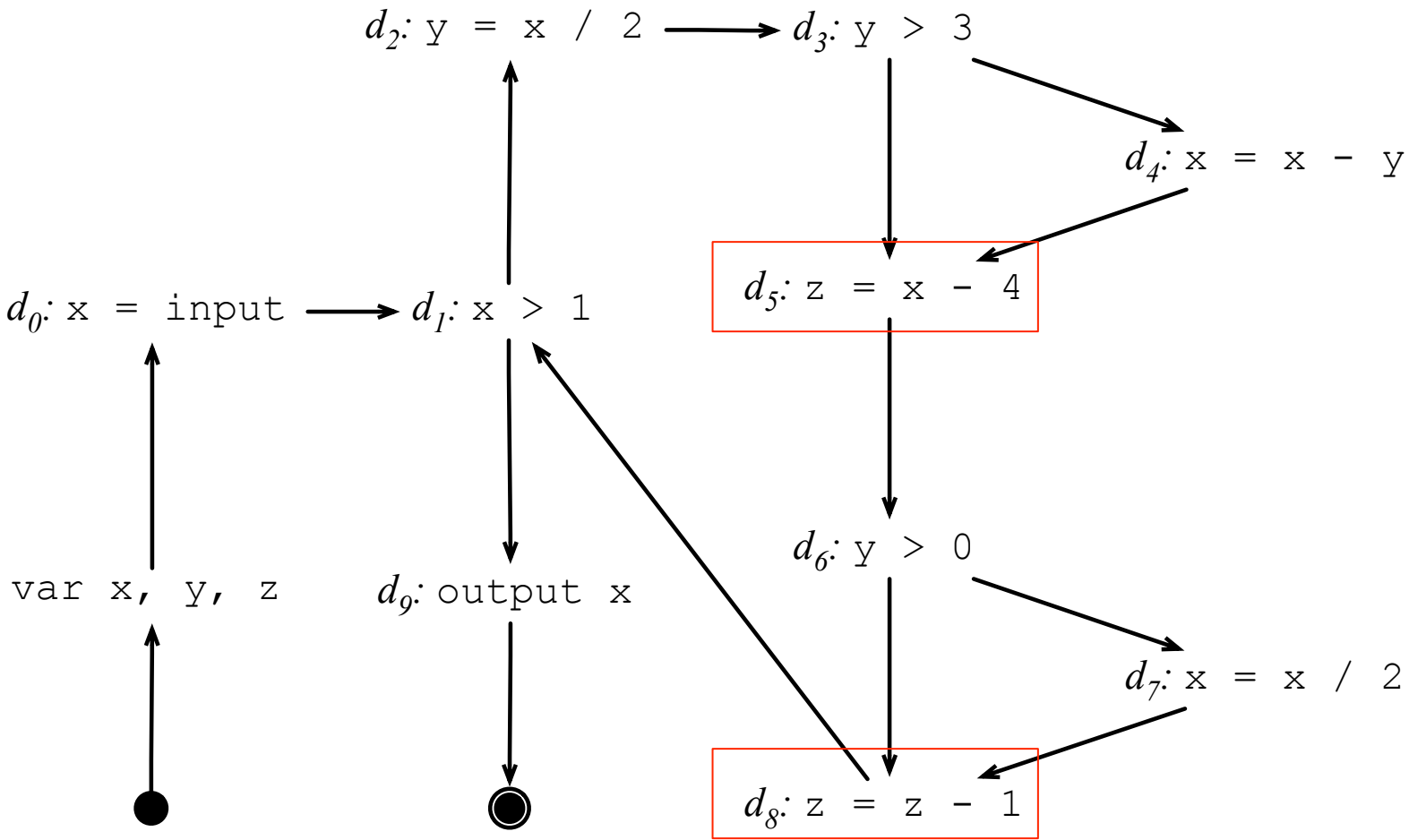
Reaching Definitions

```
var x,y,z;

x = input;

while (x > 1) {

    y = x / 2;

    if (y > 3)

        x = x - y;

    z = x - 4;

    if (y > 0)

        x = x / 2;

    z = z - 1;

}

output x;
```

- Consider the program on the left. How could we optimize it?
- The assignment z = z – 1 is *dead*.
  - What is a dead assignment?
  - How can we find them?
- Which information does this optimization require?
- How is the control flow graph of this program?

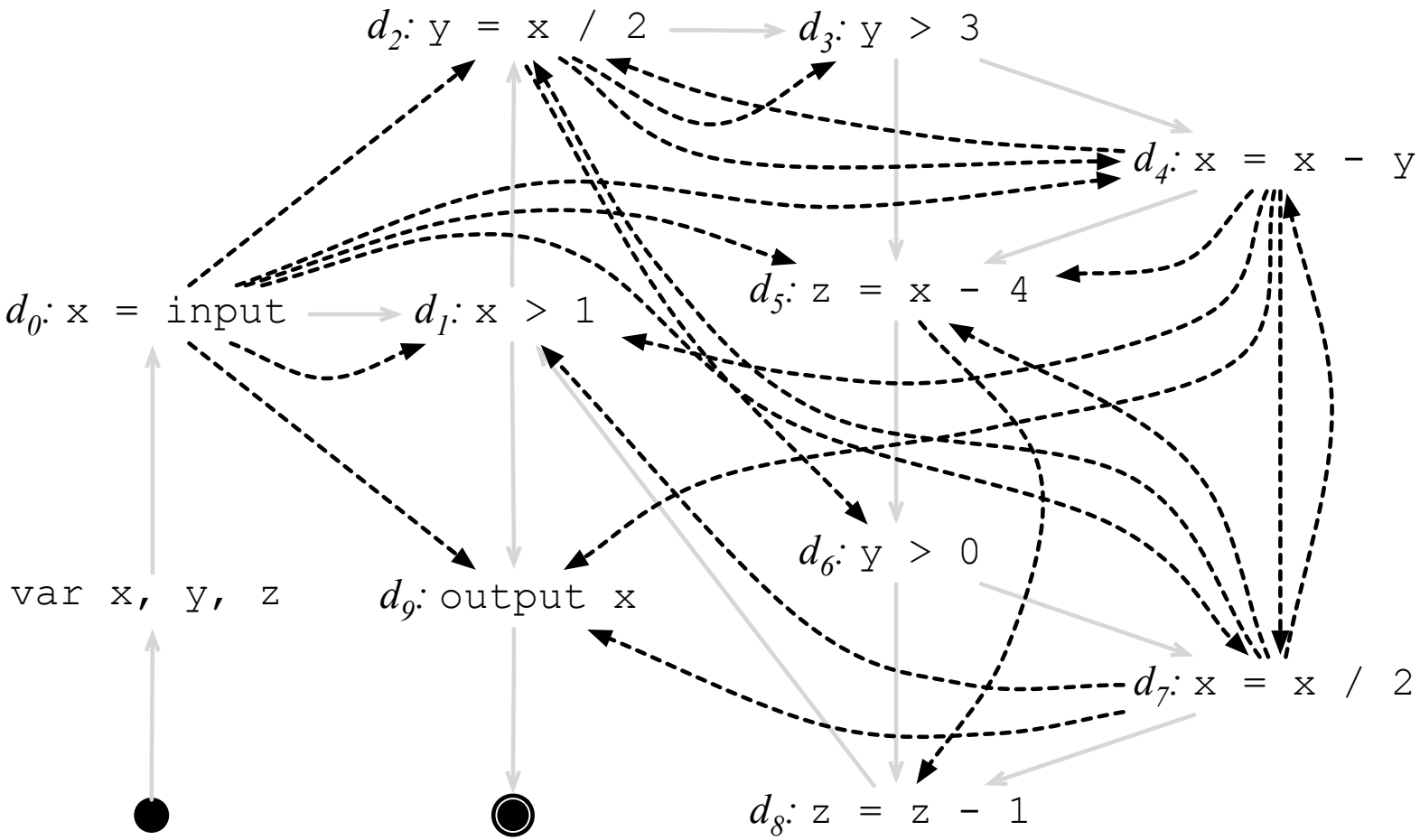
Reaching Definitions

- We know that the assignment z = z – 1 is dead because this definition of z does not reach any use of this variable.



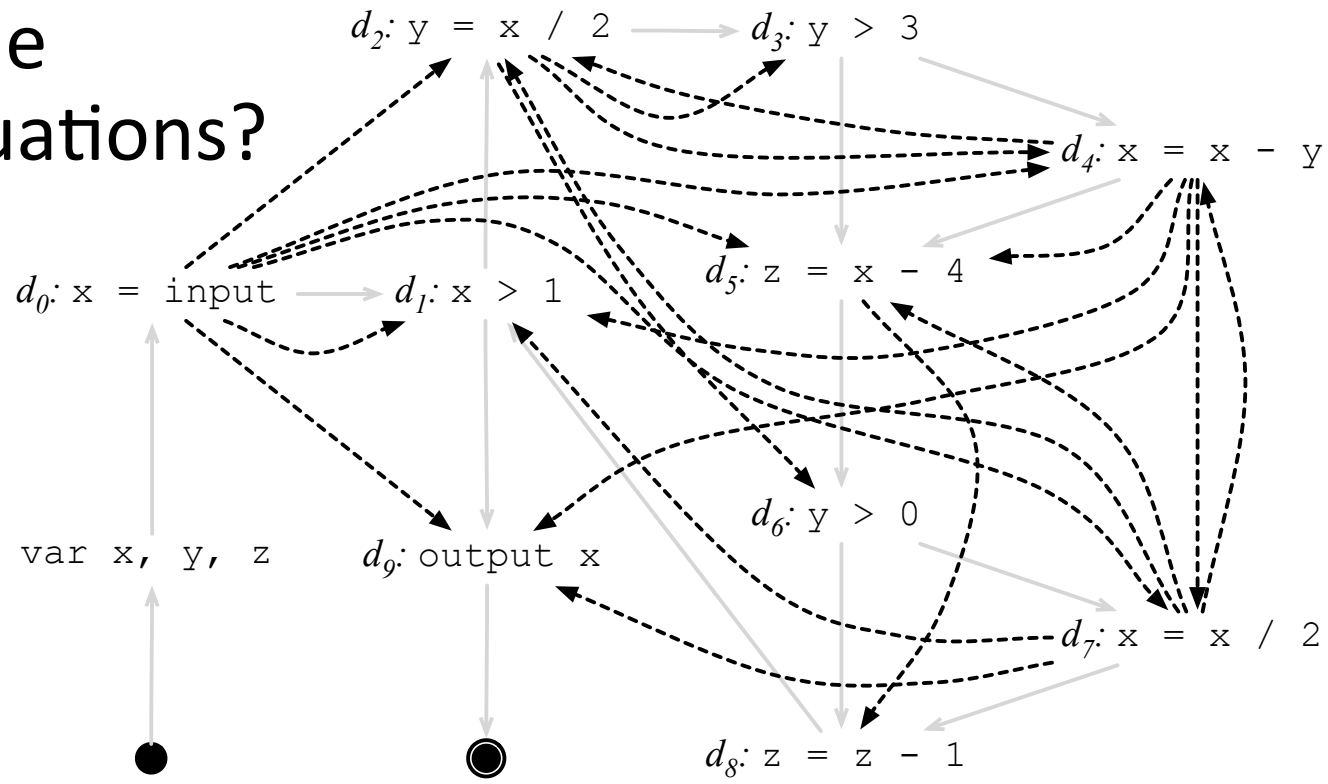
Reaching Definitions

- We say that a definition of a variable *v*, at a program point *p*, **reaches** a program point *p'*, if there is a path from *p* to *p'*, and this path does not cross any redefinition of *v*.



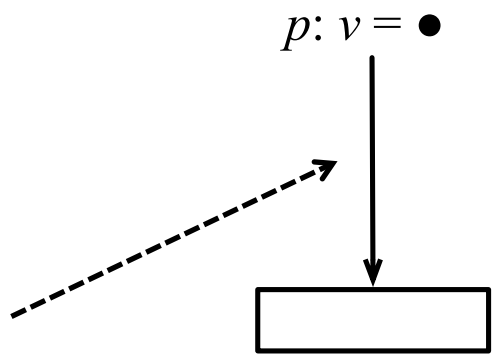
Reaching Definitions

- How does reaching def information originate?
- How does this information propagate?
- How can we join information?
- Which are the dataflow equations?



The Origin of Information

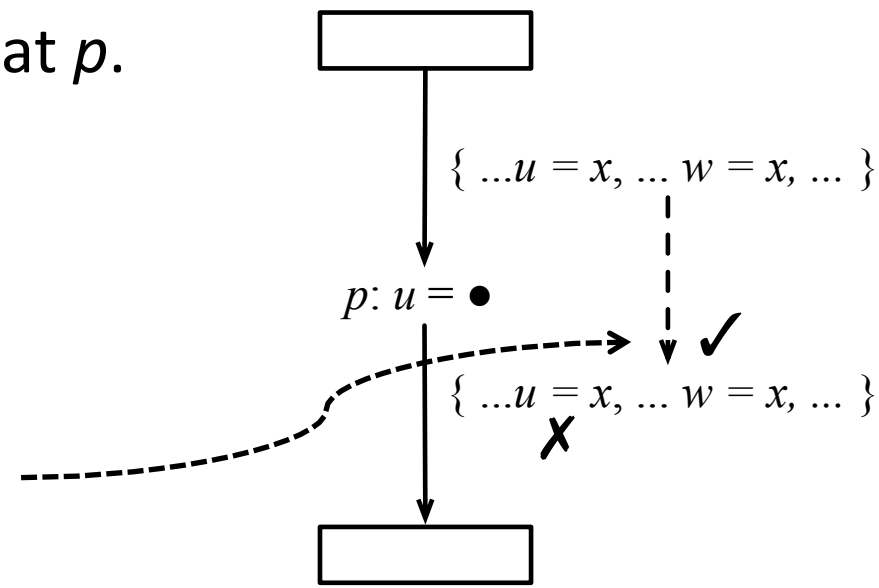
If a program point *p* defines a variable *v*, then *v* reaches the point immediately after *p*.



The Propagation of Information

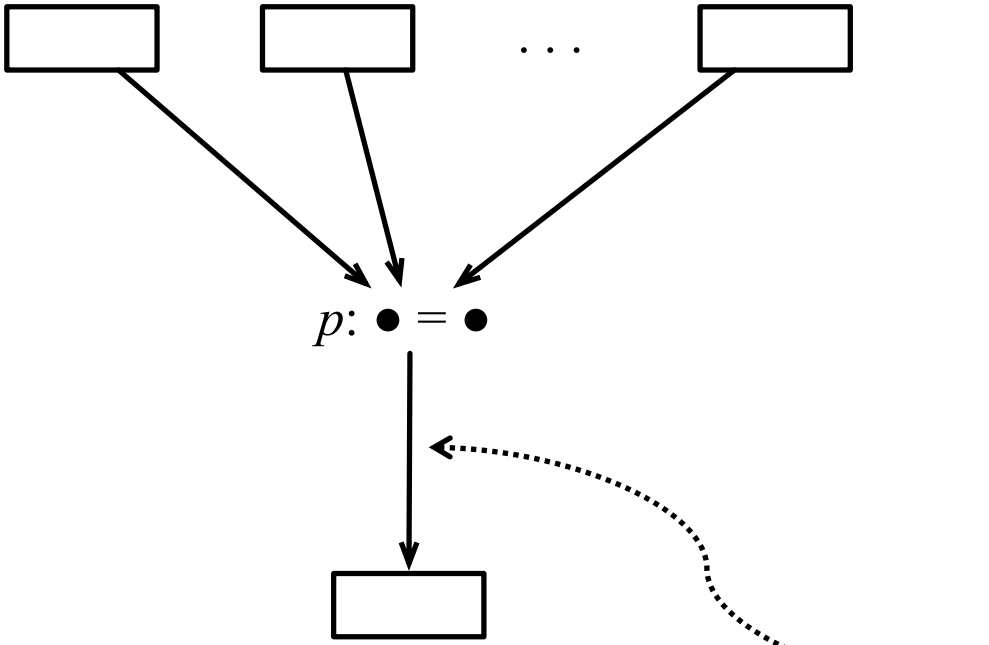
- A definition of a variable *v* **reaches** the program point immediately after *p* if, and only if:
  1. the definition reaches the point immediately before *p*.
  2. variable v is not redefined at *p*.

1. or
   1. Variable v is defined at *p*.



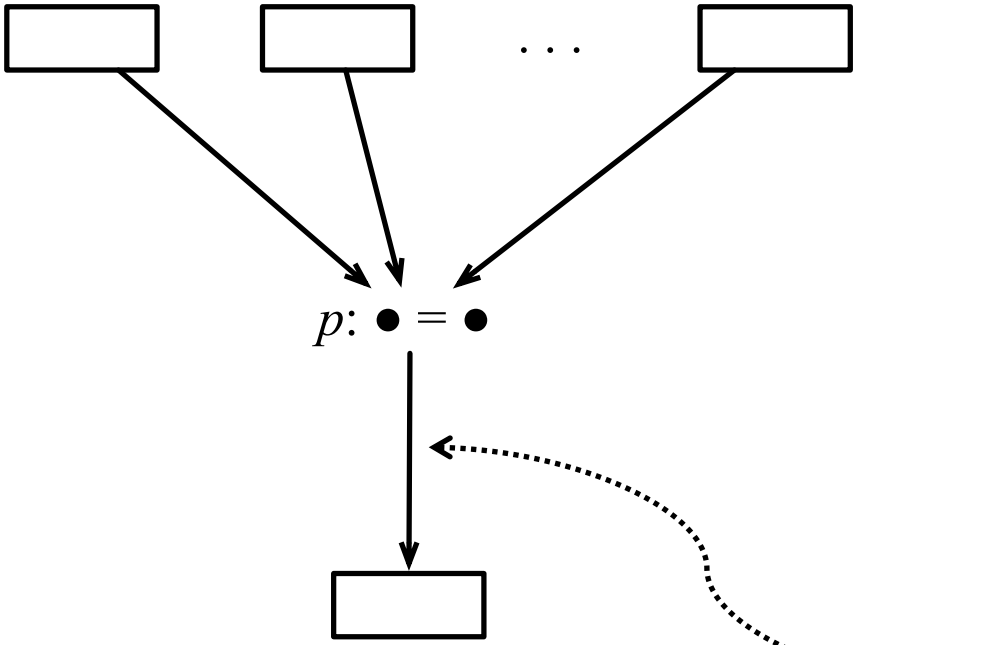
This is the direction through which information propagates.

Joining Information



How do we find the definitions that reach this program point?

Joining Information

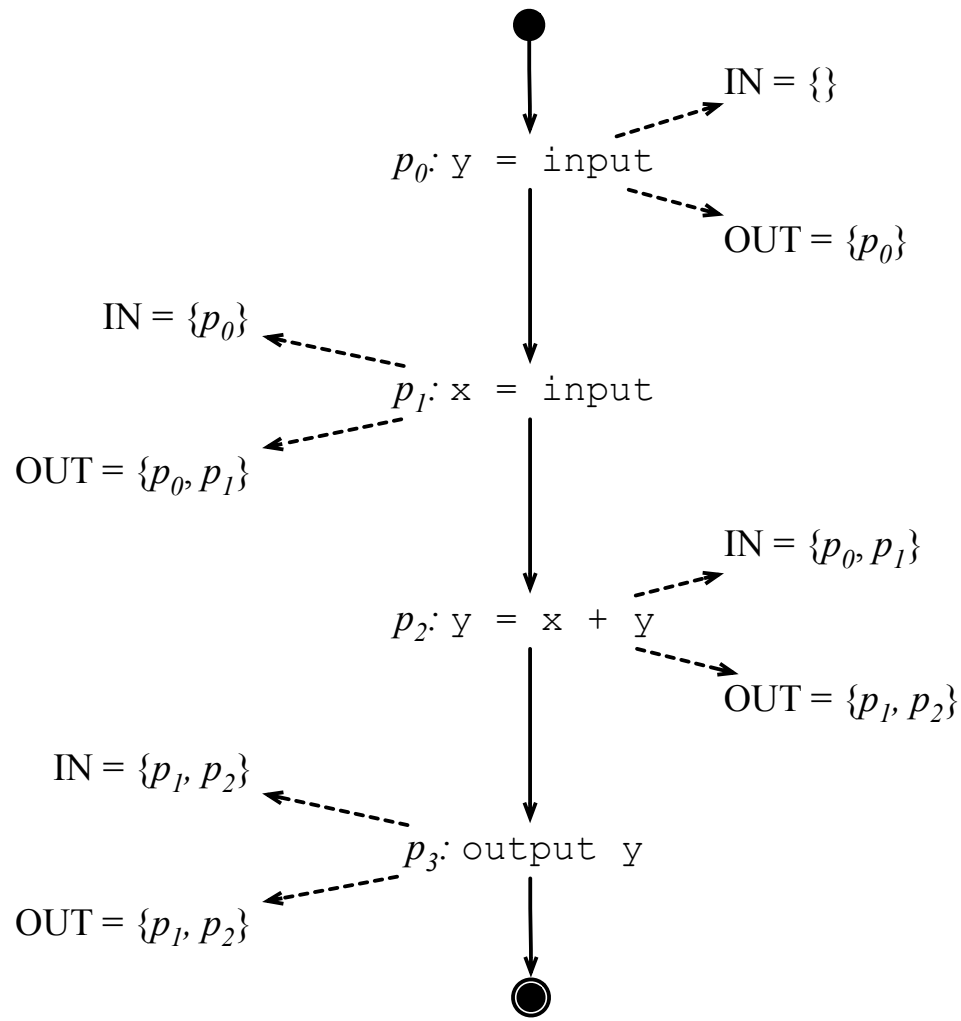


How do we find the definitions that reach this program point?

If a definition of a variable *v* reaches the point immediately after at least one predecessor of *p*, then it reaches the point immediately before *p*.

IN and OUT Sets for Reaching Definitions

- To solve the reaching definitions analysis, we associate with each program point *p* two sets, IN and OUT.
  - IN is the set of definitions that reach the point immediately before *p*.
  - OUT is the set of definitions that reach the point immediately after *p*.



Dataflow Equations for Reaching Definitions

$p : v = E$

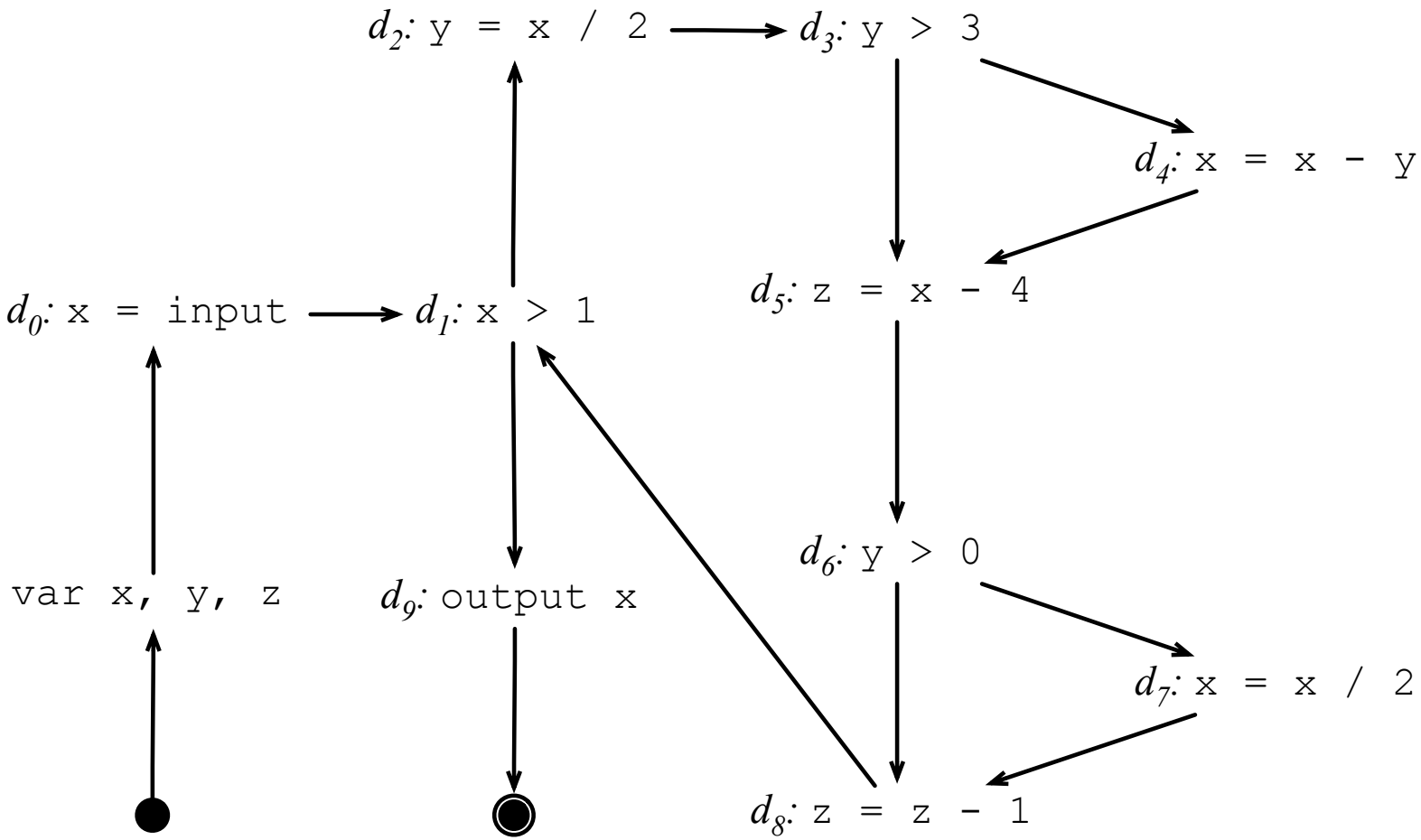
$$IN(p) = \bigcup OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \setminus \{defs(v)\}) \cup \{p\}$$

- *IN(p)* = the set of reaching definitions immediately before *p*
- *OUT(p)* = the set of reaching definitions immediately after *p*
- *pred(p)* = the set of control flow nodes that are predecessors of p
- *defs(v)* = the set of definitions of v in the program.

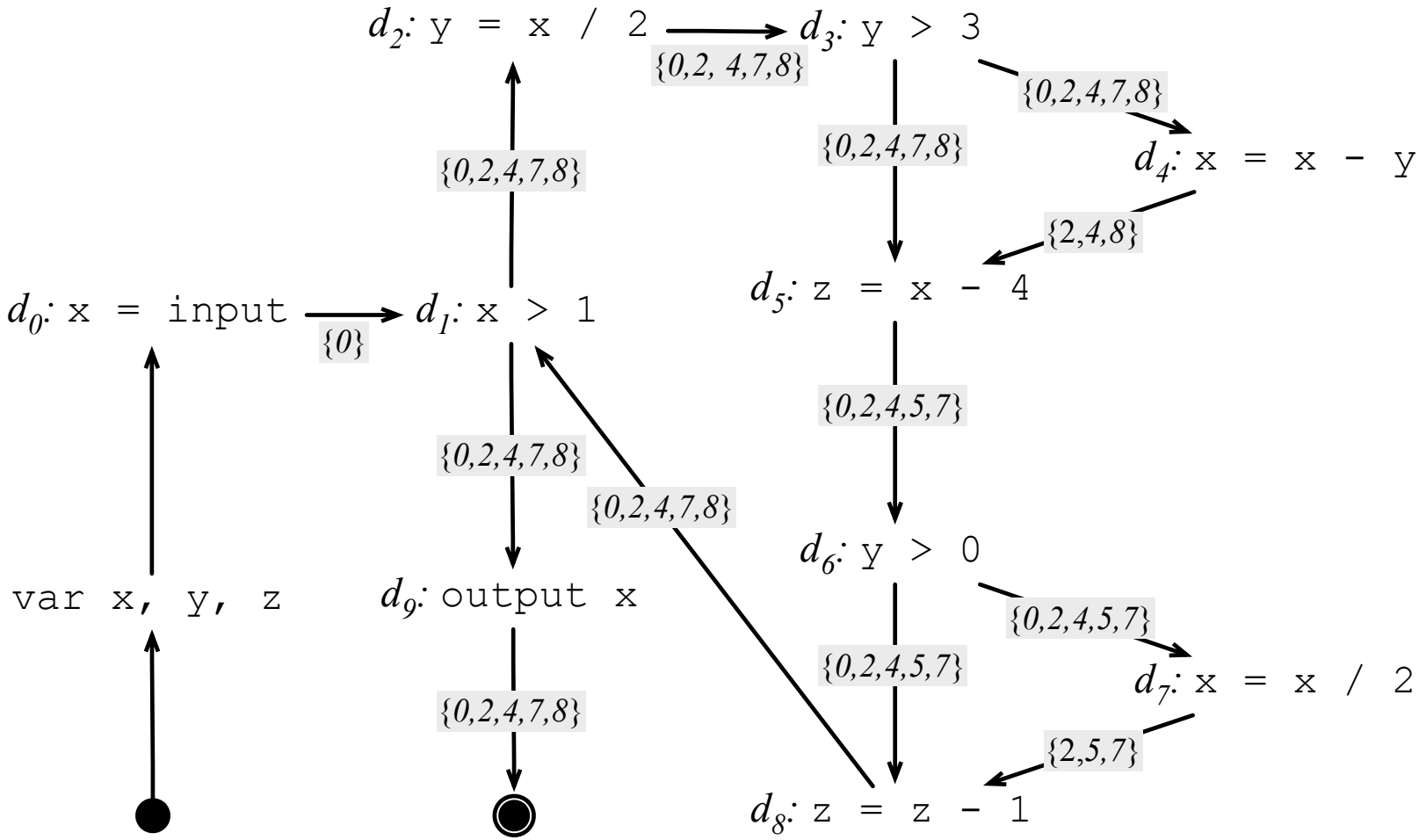
Example of Reaching Definitions Revisited

- Can you compute the set of reaching definitions for our original example?



Example of Reaching Definitions Revisited

- Can you compute the set of reaching definitions for our original example?



Finding Commonalities

$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$ $OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$ Liveness	$IN(p) = \bigcup OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \setminus \{defs(v)\}) \cup \{p\}$ Reaching Defs
$IN(p) = (OUT(p) \setminus \{v\}) \cup \{E\}$ $OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$ Very Busy Expressions	$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$ Available Expressions

What these two analyses have in common?

Find Commonalities

	$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$ $OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$ Liveness	$IN(p) = \bigcup OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \setminus \{defs(v)\}) \cup \{p\}$ Reaching Defs
	$IN(p) = (OUT(p) \setminus \{v\}) \cup \{E\}$ $OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$ Very Busy Expressions	$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$ Available Expressions

What about these two analysis?

Can you categorize the lines and columns?

The Dataflow Framework

	Backward	Forward
May	$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$ $OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$ Liveness	$IN(p) = \bigcup OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \setminus \{defs(v)\}) \cup \{p\}$ Reaching Defs
Must	$IN(p) = (OUT(p) \setminus \{v\}) \cup \{E\}$ $OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$ Very Busy Expressions	$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$ Available Expressions

The Dataflow Framework

- A <u>may analysis</u> keeps tracks of facts that *may* happen during the execution of the program.
  - A definition *may* reach a certain point.
- A <u>must analysis</u> tracks facts that *will* – for sure – happen during the execution of the program.
  - This expression *will* be used after certain program point.
- A <u>backward analysis</u> propagates information in the opposite direction in which the program flows.
- A <u>forward analysis</u> propagates information in the same direction in which the program flows.

Transfer Functions

- A data-flow analysis does some *interpretation* of the program, in order to obtain information.
- But, we do not interpret the concrete semantics of the program, or else our analysis could not terminate.
- Instead, we do some *abstract interpretation*.
- The abstract semantics of a statement is given by a *transfer function*.
- Transfer functions differ if the analysis is forward or backward:

$$\mathrm{OUT}[s] = f_s(\mathrm{IN}[s])$$ → Forward analysis

$$\mathrm{IN}[s] = f_s(\mathrm{OUT}[s])$$ → Backward analysis

Transfer Functions

	Backward	Forward
May	$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$ $OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$ Liveness	$IN(p) = \bigcup OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \setminus \{defs(v)\}) \cup \{p\}$ Reaching Defs
Must	$IN(p) = (OUT(p) \setminus \{v\}) \cup \{E\}$ $OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$ Very Busy Expressions	$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$ Available Expressions

Can you recognize the transfer functions of each analysis?

Transfer Functions

	Backward	Forward
May	$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$ $OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$ Liveness	$IN(p) = \bigcup OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \setminus \{defs(v)\}) \cup \{p\}$ Reaching Defs
Must	$IN(p) = (OUT(p) \setminus \{v\}) \cup \{E\}$ $OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$ Very Busy Expressions	$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$ Available Expressions

The transfer functions provides us with a new "interpretation" of the program. We can implement a machine that traverses the program, always fetching a given instruction, and applying the transfer function onto that instruction. This process goes on until the results produced by these transfer functions stop changing.

Transfer Functions

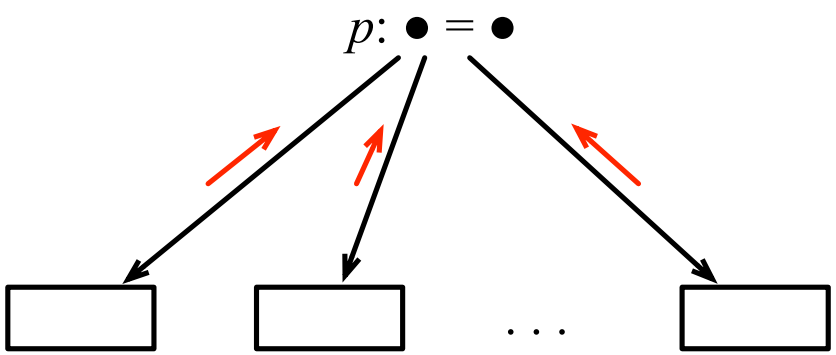
- The transfer functions do not have to be always the same, for every statement.
- In the concrete semantics of an assembly language, each statement does something different.
  - The same can be true for the abstract semantics of the programming language.

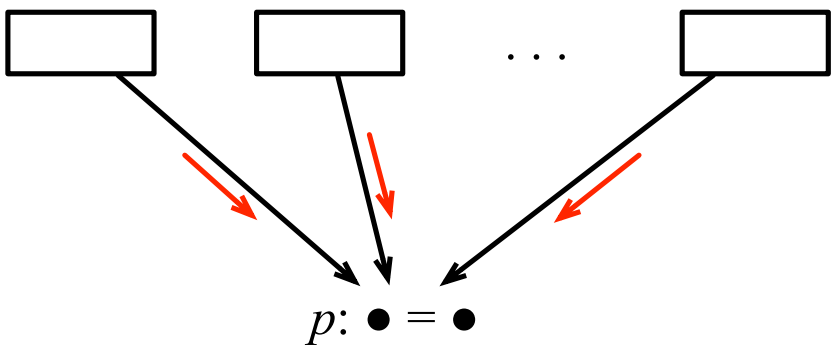
Statement	Transfer Function
exit	IN[p] = OUT[p]
output v	IN[p] = OUT[p] $\cup$ {v}
bgz v L	IN[p] = OUT[p] $\cup$ {v}
v = x + y	IN[p] = (OUT[p] \ {v}) $\cup$ {x, y}
v = u	IN[p] = (OUT[p] \ {v}) $\cup$ {u}

Which analysis is this one on the right?

The Merging Function

- The merging function (*meet* or *join*) specifies what happens once information collides.





Merging Functions

	Backward	Forward
May	$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$	$IN(p) = \bigcup OUT(p_s), p_s \in pred(p)$
	$OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$	$OUT(p) = (IN(p) \setminus \{defs(v)\}) \cup \{p\}$
	Liveness	Reaching Defs
Must	$IN(p) = (OUT(p) \setminus \{v\}) \cup \{E\}$	$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$
	$OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$	$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$
	Very Busy Expressions	Available Expressions

The combination of transfer functions, merging functions and – to a certain extent – the way that we initialize the IN and OUT gives us guarantees that the abstract interpretation terminates.

The Dataflow Framework

- Many other program analyses fit into the dataflow framework.
- It helps a lot if we can phrase our analysis into this framework:
  - We gain efficient resolution algorithms (implementation).
  - We can prove termination (theory).
- Compiler writers have been doing this since the late 60's.