PROGRAMMING LANGUAGES LABORATORY
Universidade Federal de Minas Gerais - Department of Computer Science

DATA FLOW ANALYSIS

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira
fernando@dcc.ufmg.br

The material in these slides have been taken from "Lecture Notes in Static Analysis", by Michael I. Schwartzbach.

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.
- Dataflow analyses are flow sensitive.
  - What are flow sensitive analysis?
- In this class we will go over these four analyses, and will eventually derive a common pattern for them.

A Simple Example to Warm up

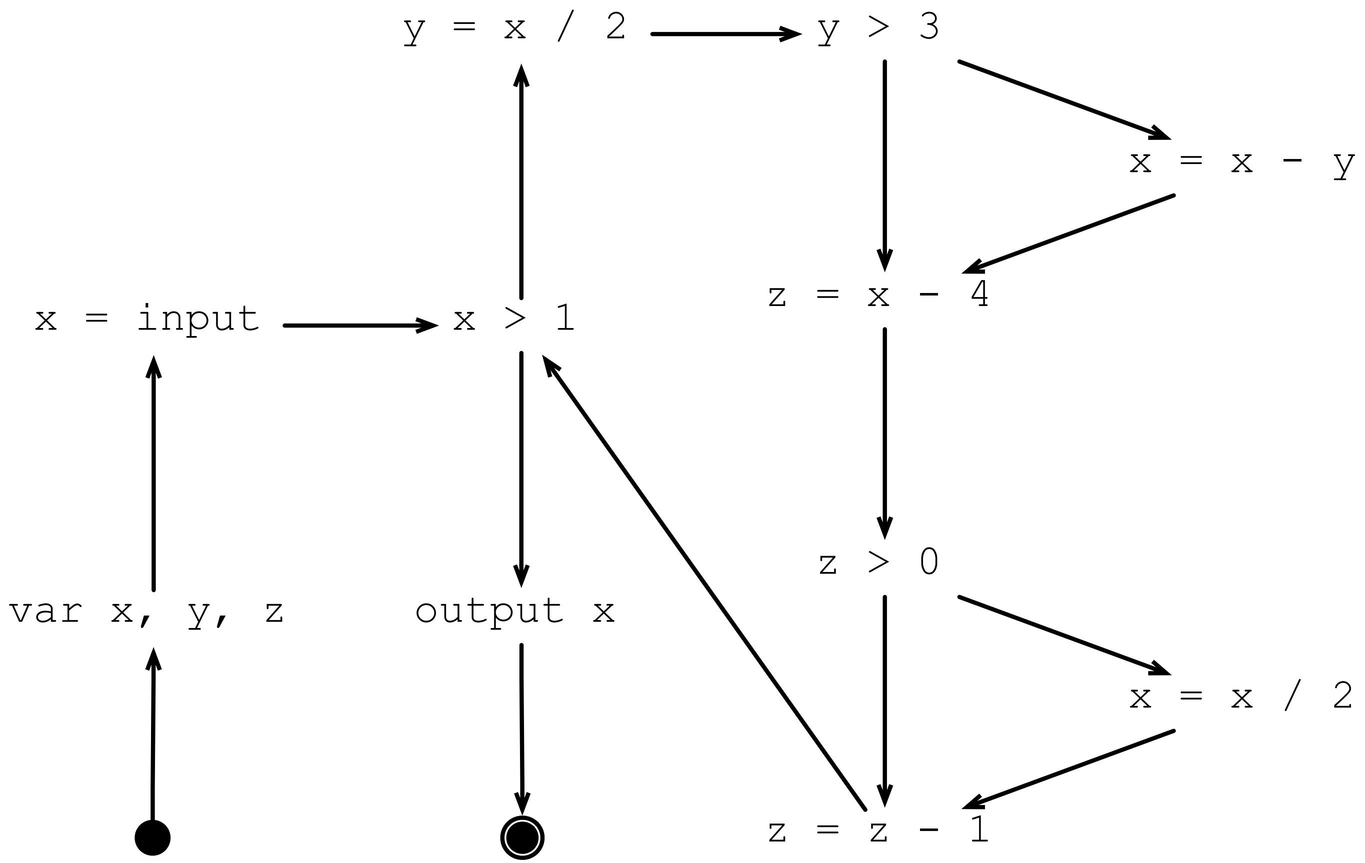
```
var x,y,z;

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?

- How many nodes?
- How many edges?

The 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;
```



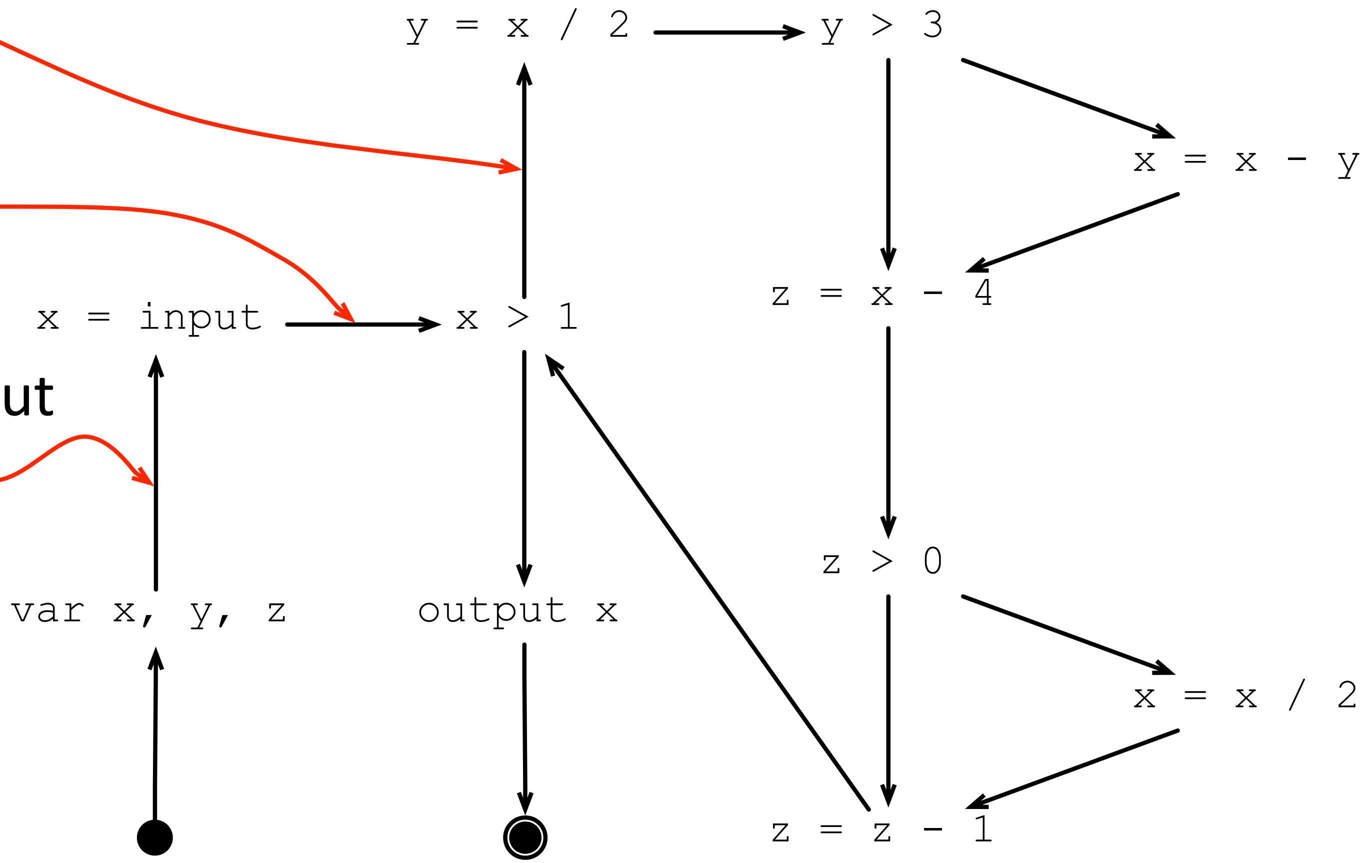
How many registers do I need to compile this program?

The Control Flow Graph

How many registers do I need **here**?

And here?

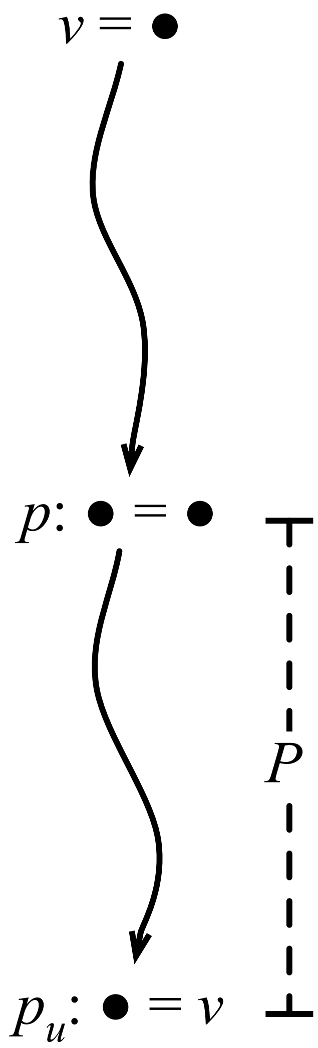
What about here?



Can you devise a general way to count this number?

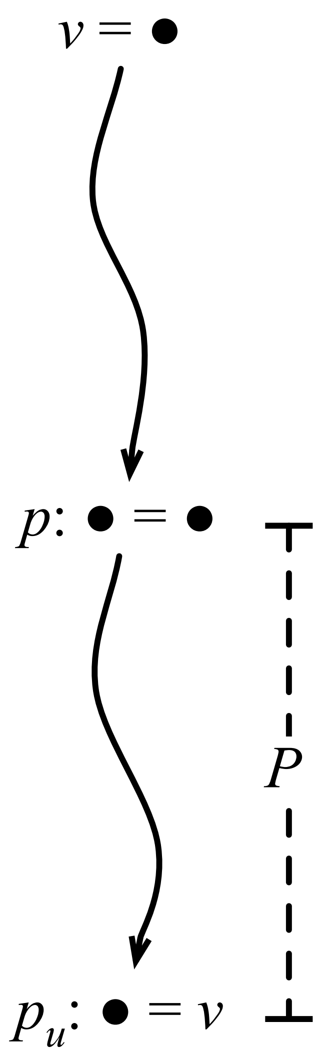
Liveness

- If we assume an infinite supply 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 go across any definition of *v*.
- Conditions 1 and 2 determine when a variable *v* is <u>alive</u> at a program point *p*.



Liveness

- If we assume an infinite supply 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 go across any definition of *v*.



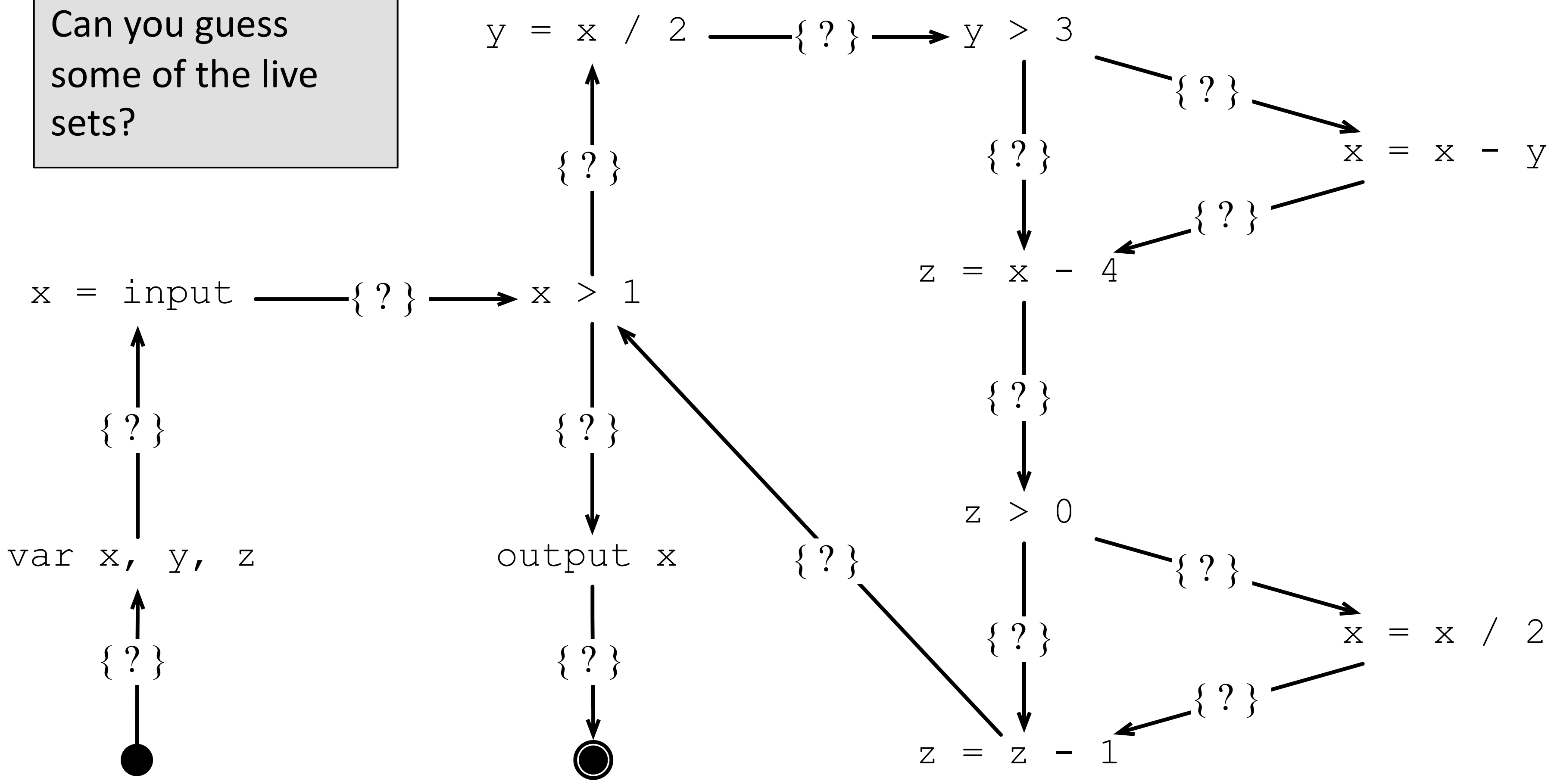
Why is the second condition really necessary?

Can you define the liveness problem more formally?

Liveness

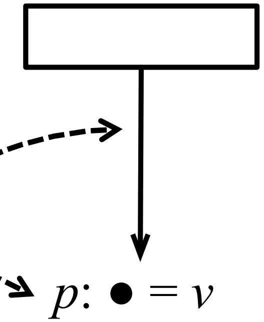
- Given a control flow graph G, find, for each edge E in G, the set of variables alive at E.

Can you guess some of the live sets?

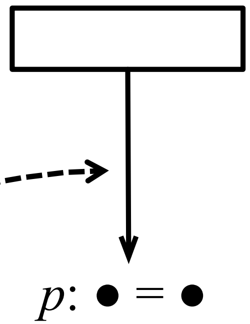


The Origin of Information

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



Ok, but what if the variable is not used at *p*, when is it going to be alive immediately before *p*?

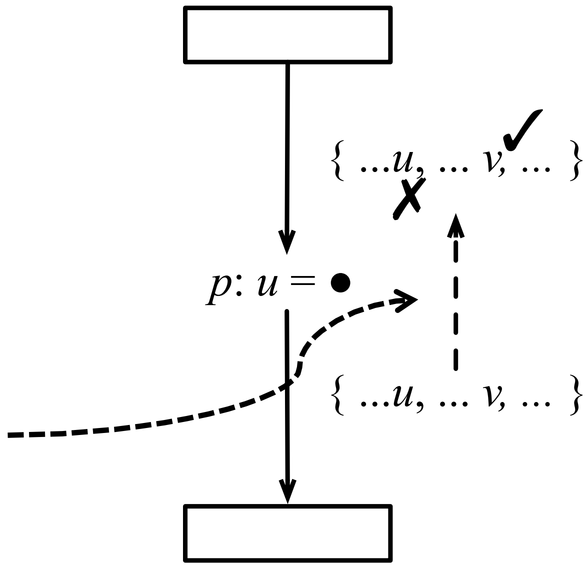


The Propagation of Information

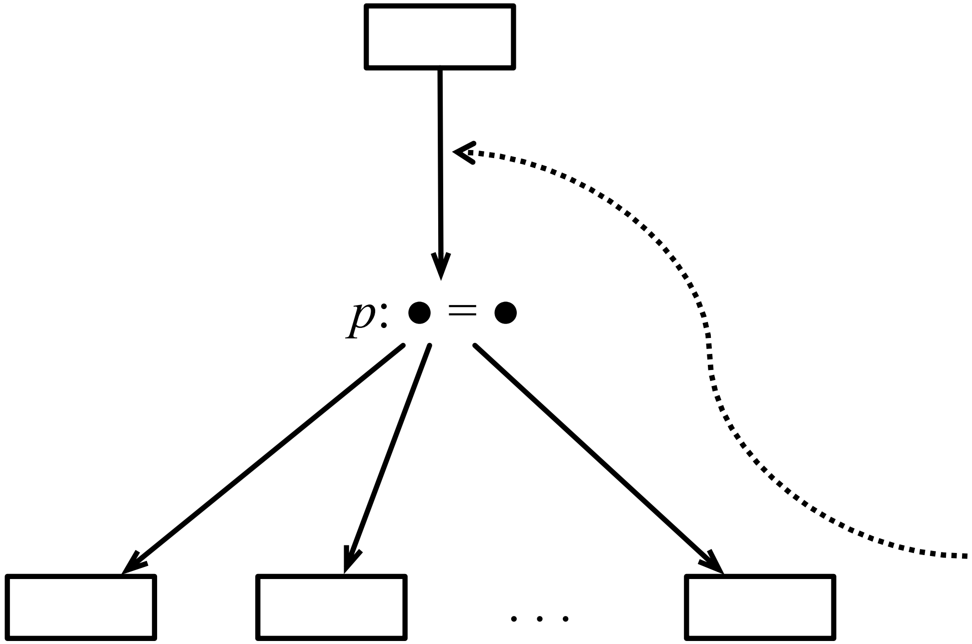
- A variable is alive immediately before a program point *p* if, and only if:
  1. It is alive immediate after *p*.
  2. It is not redefined at *p*.
- or
  1. It is used at *p*.

But, what if a program point has multiple successors?

This is the direction through which information propagates.



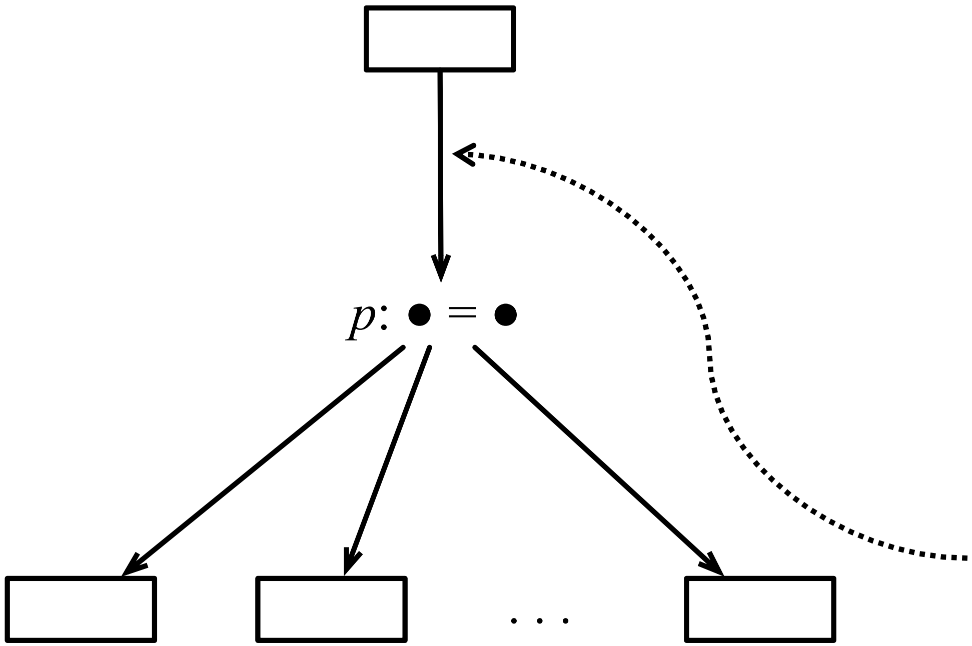
Joining Information



But, what if a program point has multiple successors?

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

Joining Information



But, what if a program point has multiple successors?

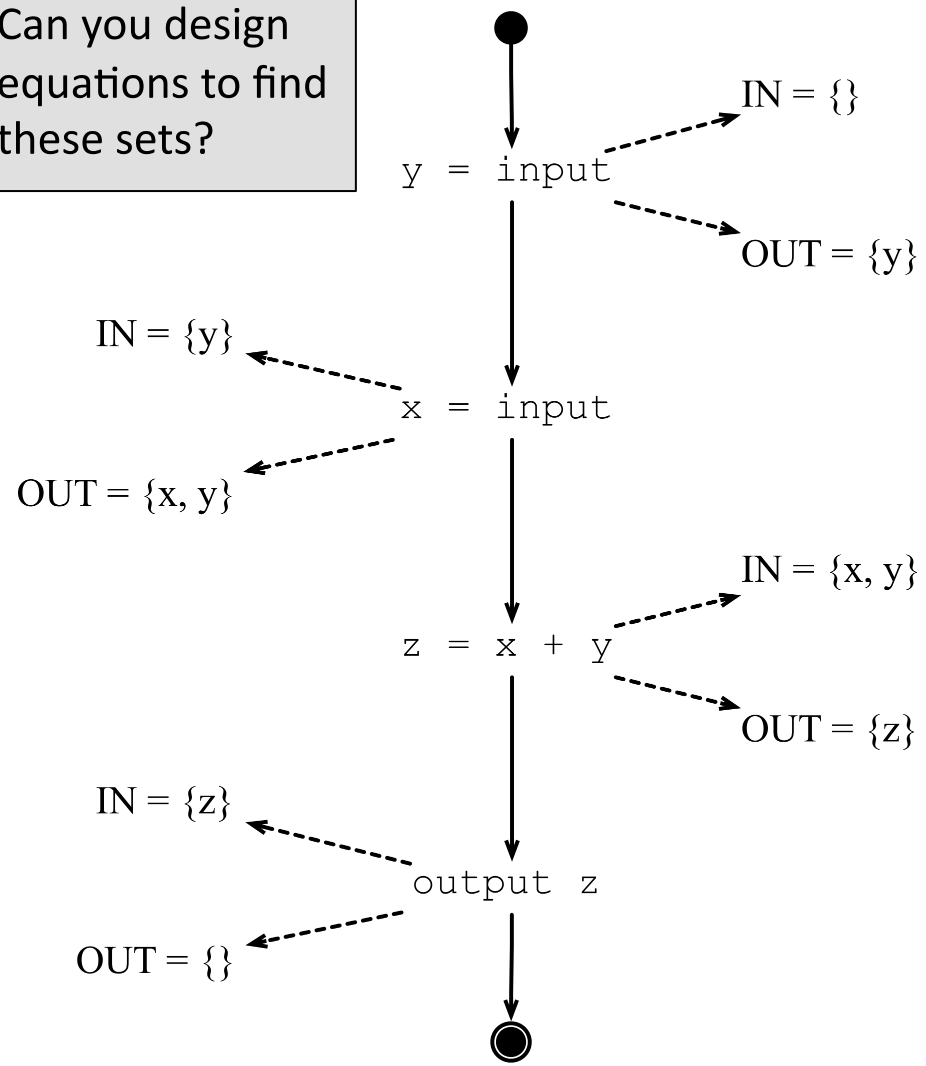
How do we find the variables that are alive at this program point?

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*.

Can you design equations to find these sets?



Dataflow Equations

$$p : v = E$$

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

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

How do we solve these equations?

Solving Liveness

$$p : v = E$$

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

- Each program point in the target CFG gives us two equations.
- We can iterate these equations until all the IN and OUT sets stop changing.

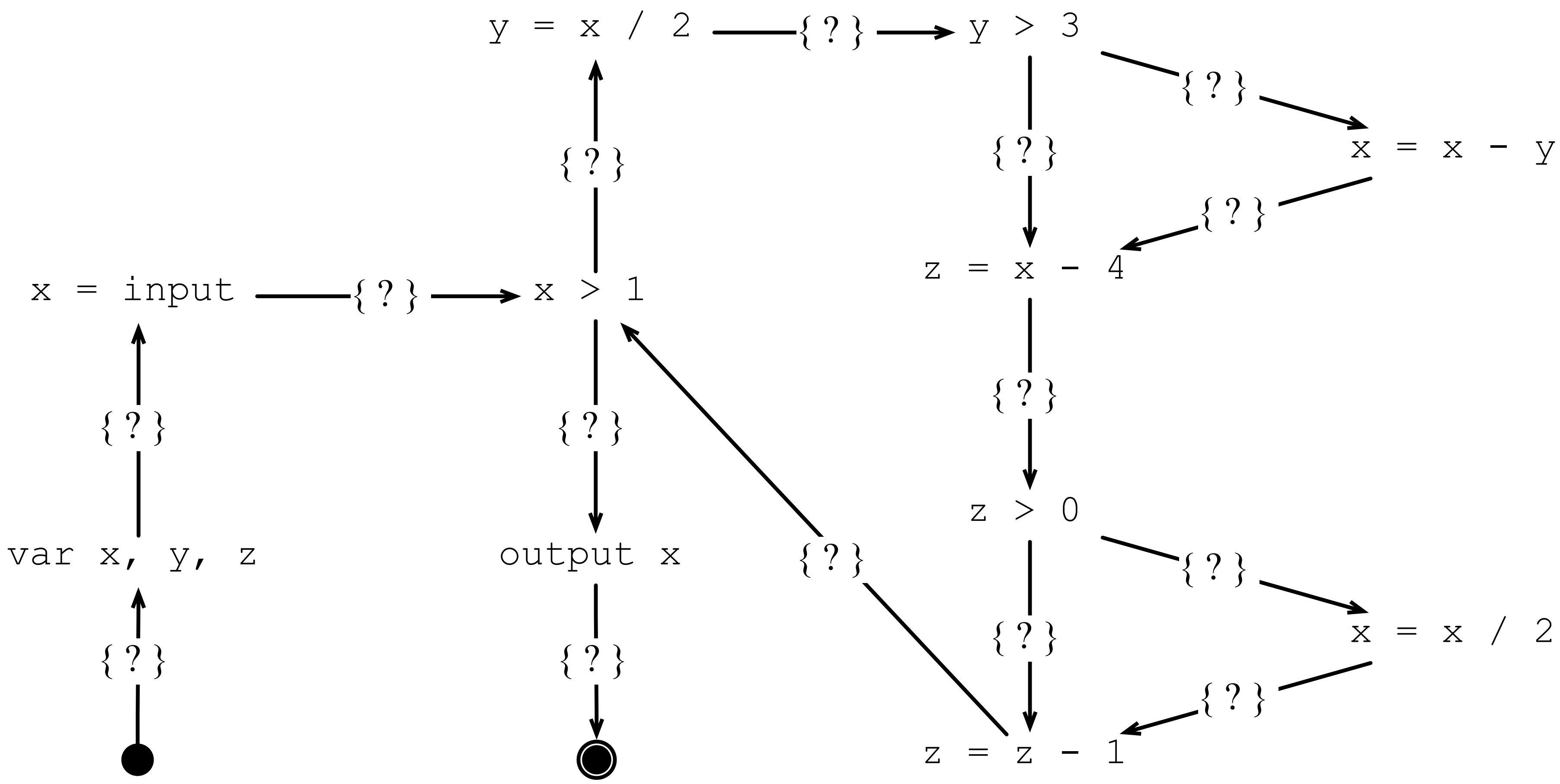
1) How are the IN and OUT sets initialized?

2) Are these sets guaranteed to stop changing?

3) What is the complexity of this algorithm?

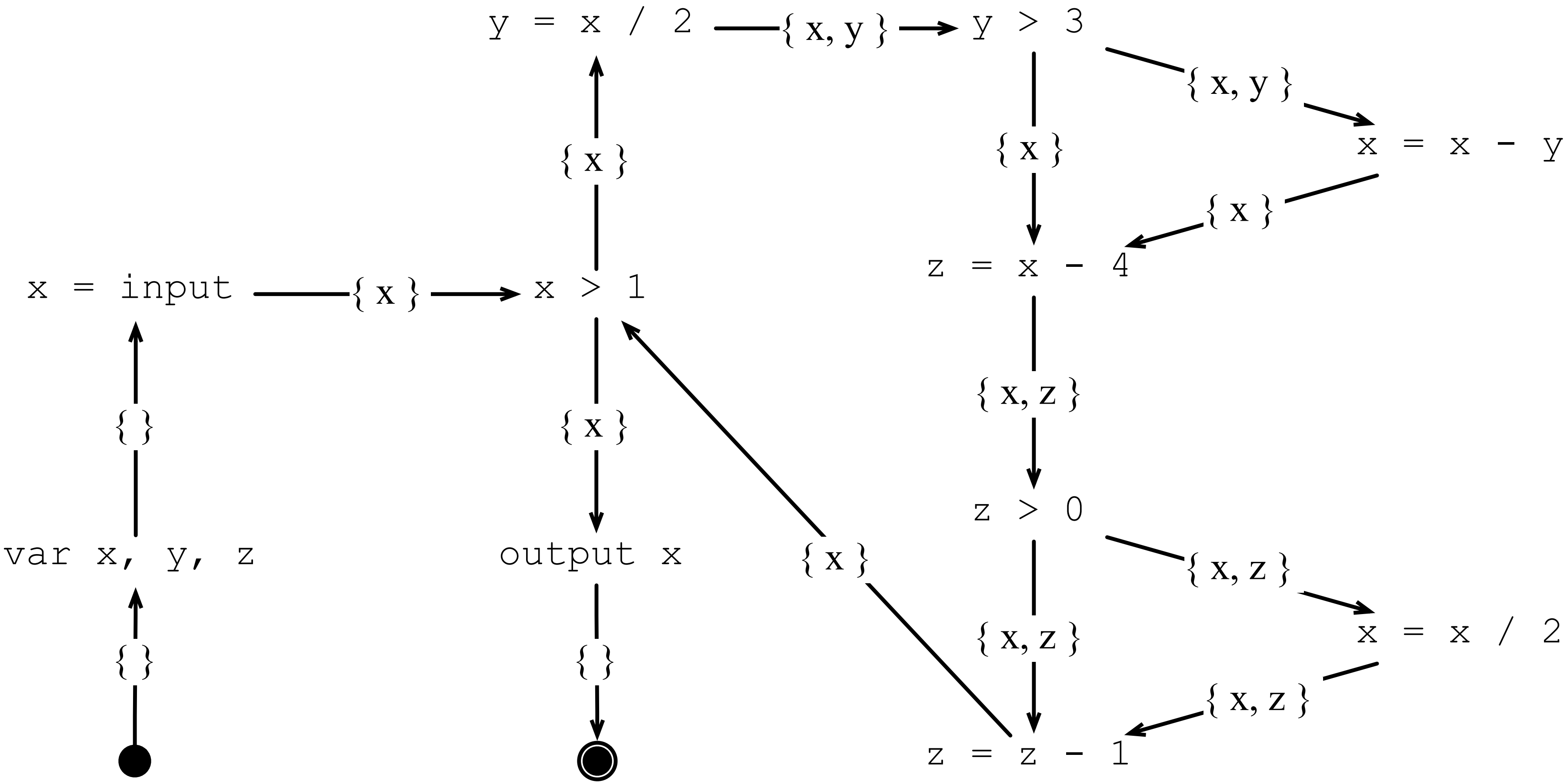
Example of Liveness Problem

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



Example of Liveness Problem

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



Available Expressions

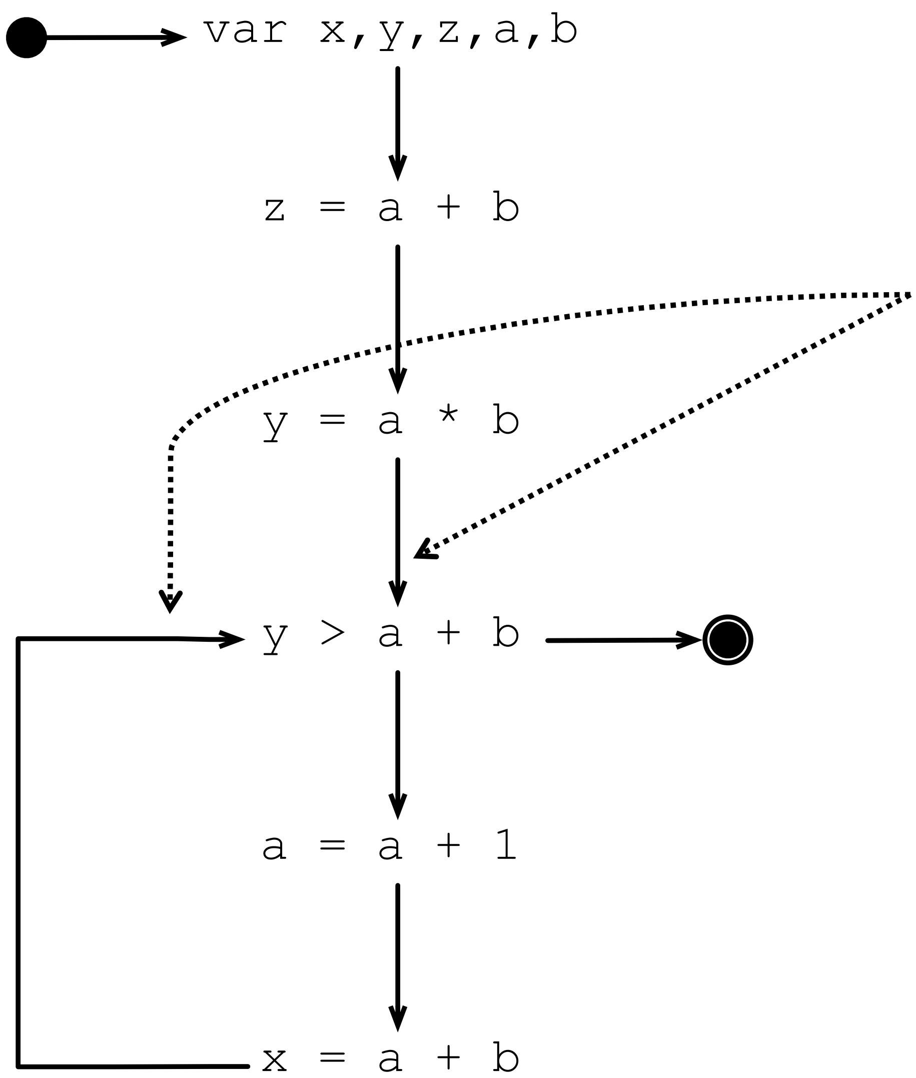
```
var x, y, z, a, b;

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 demand?

How is the control flow graph of this program?

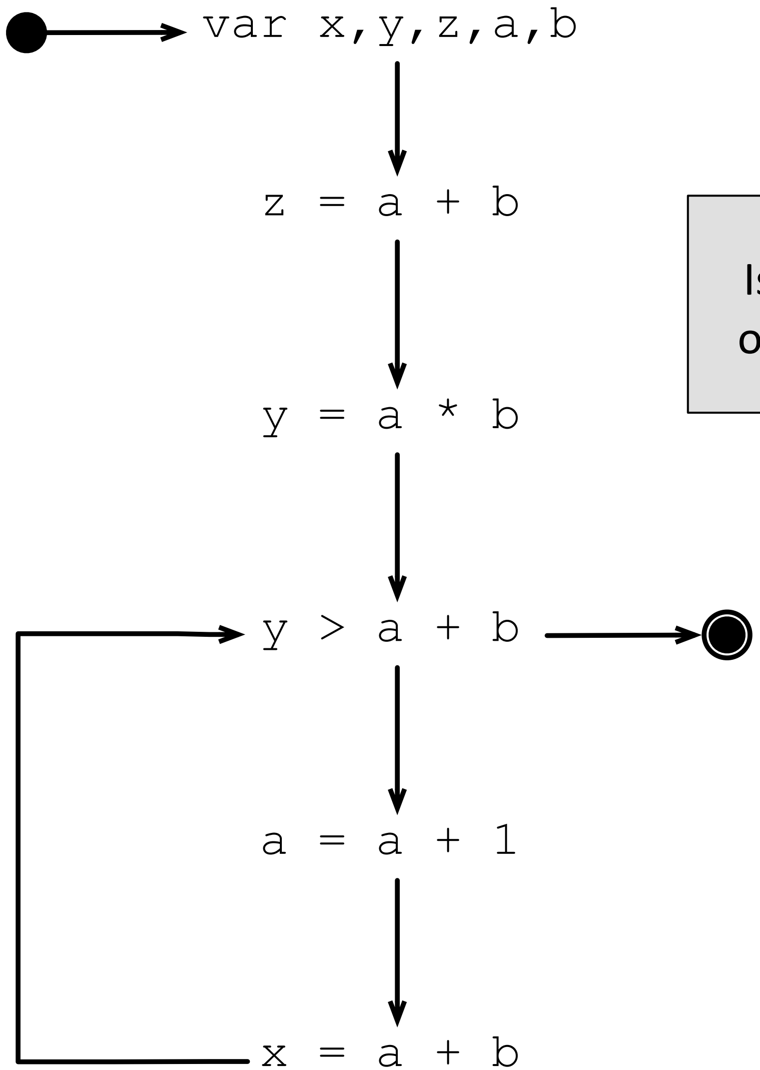
Available Expressions



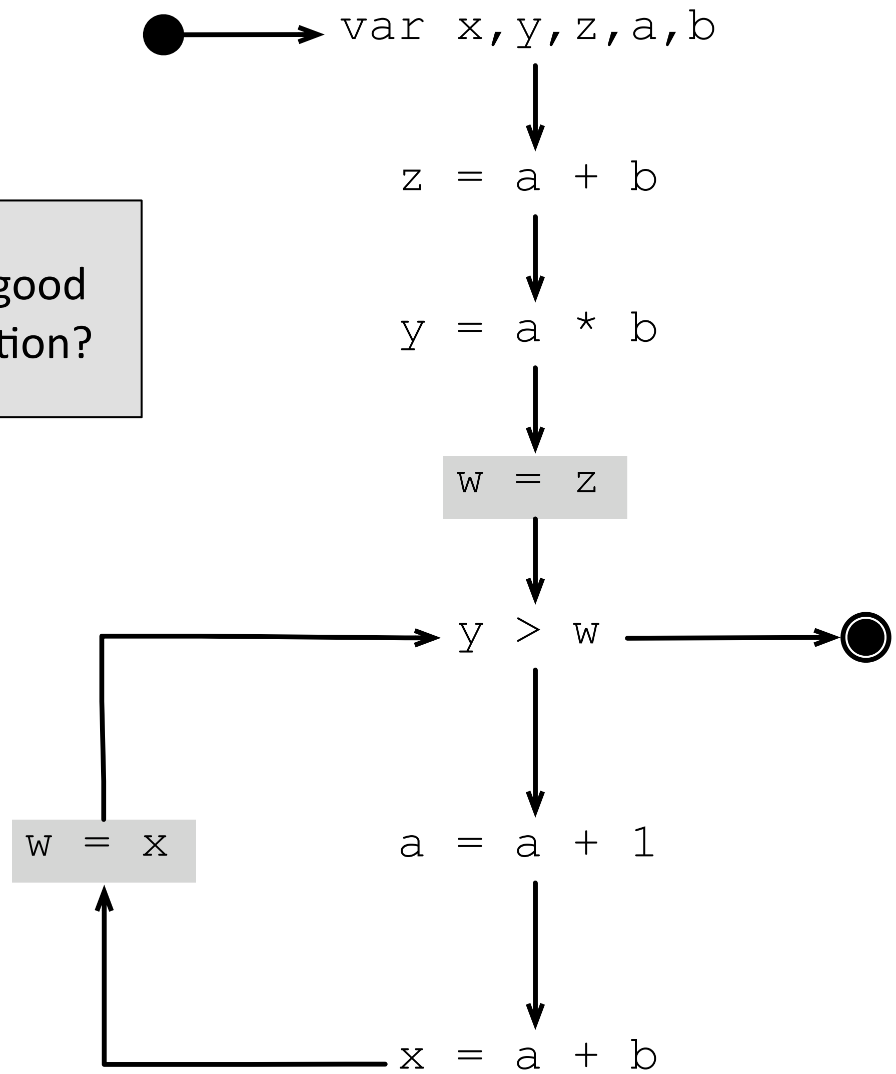
We know that the expression a + b is available at **these** two program points

So, how could we improve this code?

Available Expressions

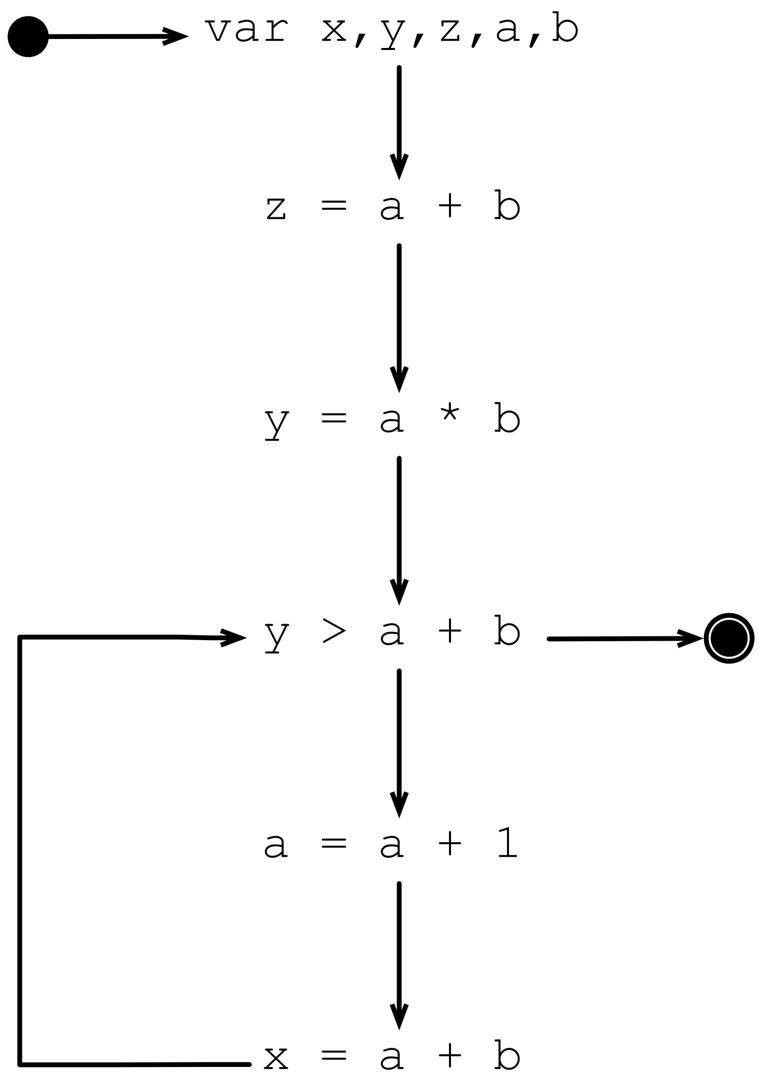


Is this a good optimization?



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 current value has already been computed earlier in the execution.

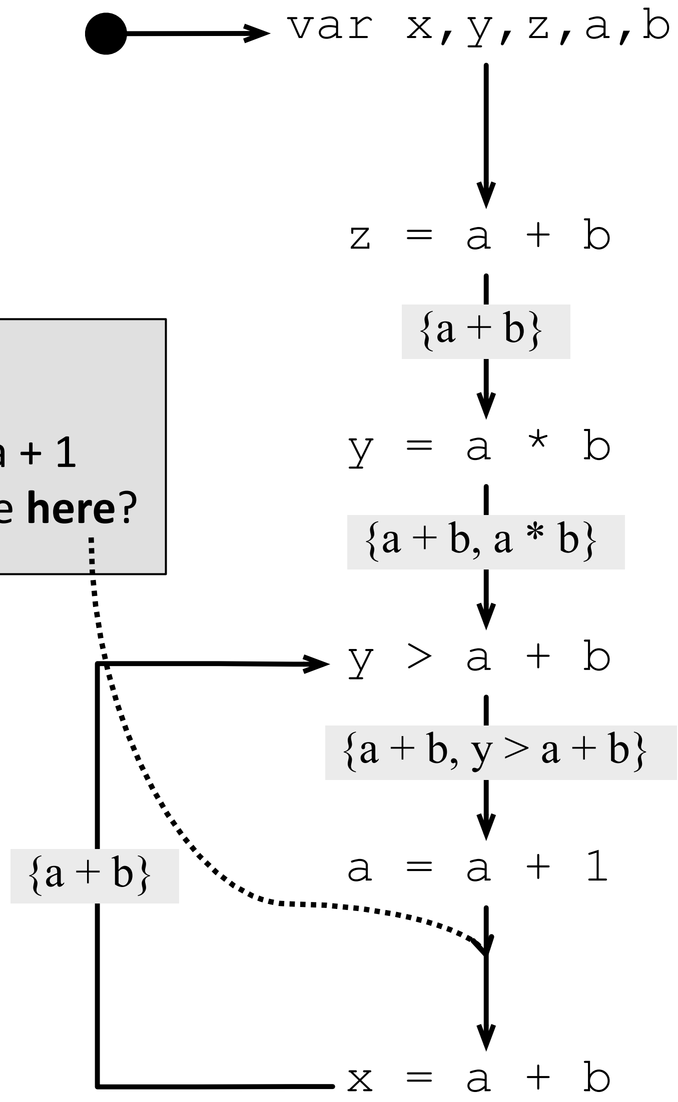


Which expressions are available in our example?

Available Expressions

- We can approximate the set of available expressions by a dataflow analysis.

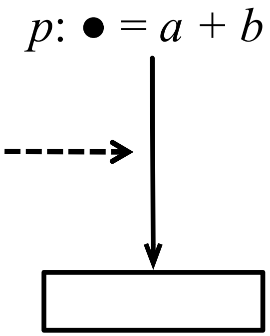
Why is the expression a + 1 not available **here**?



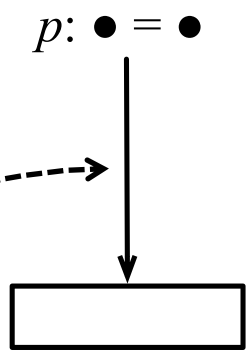
- How does information originate?
- How does information propagate?

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.



Ok, but what if p does not define expressions? Which expressions will be available after it?

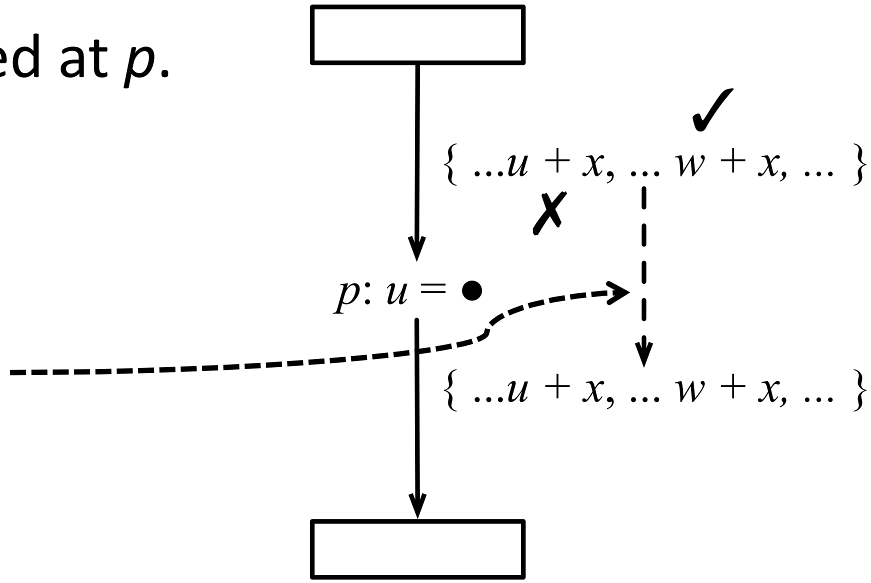


The Propagation of Information

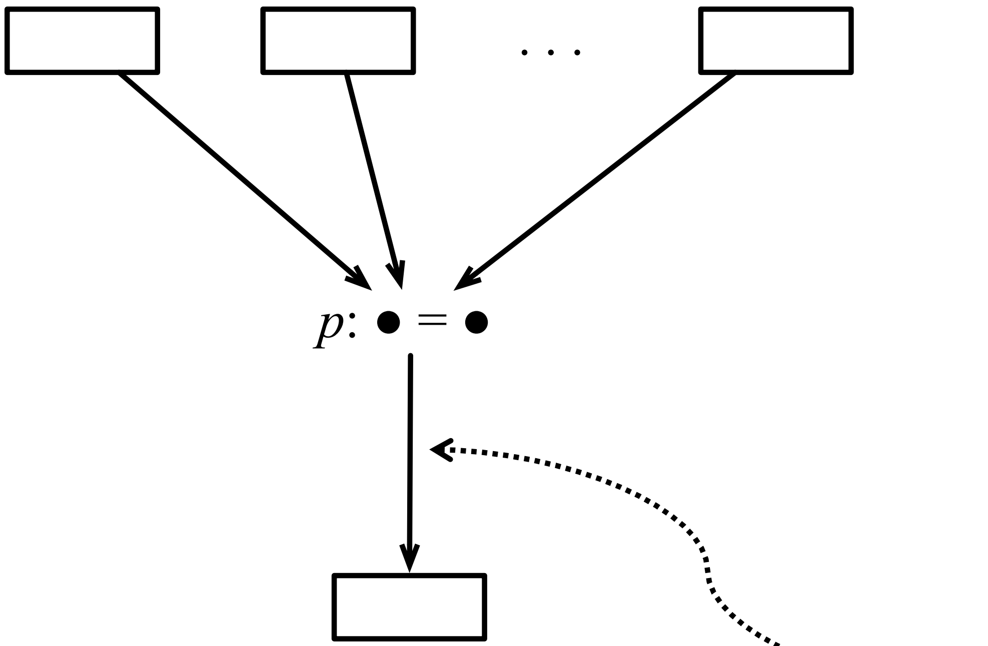
- An expression *E* is available immediately after a program point *p* if, and only if:
  1. It is available immediately before *p*.
  2. No variable of E is redefined at *p*.
- or
  1. It is used at p.
  2. No variable of E is redefined at *p*.

But, what if a program point has multiple predecessors?

This is the direction through which information propagates.



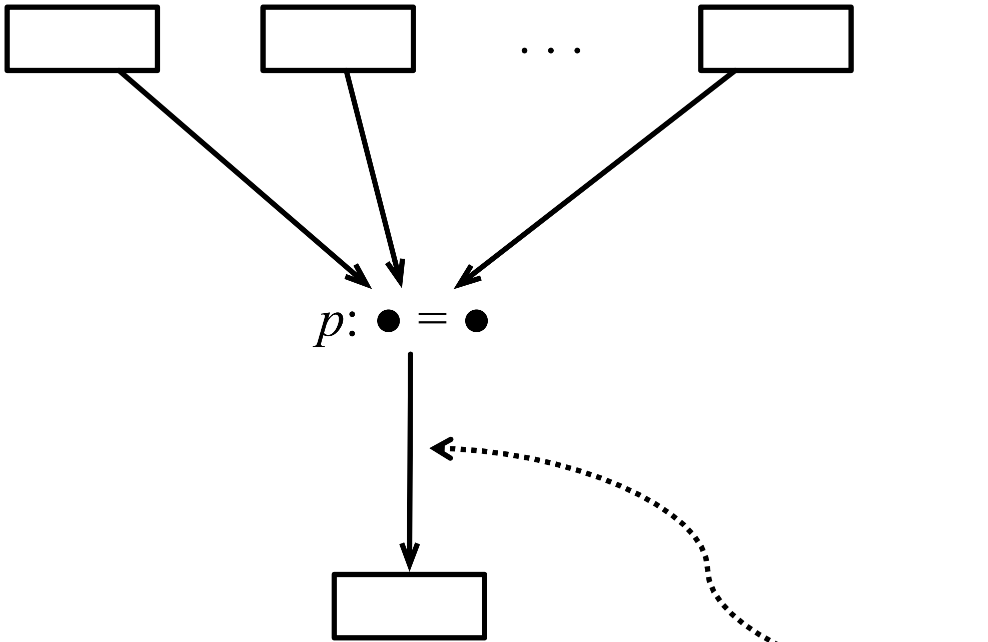
Joining Information



But, what if a program point has multiple predecessors?

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

Joining Information



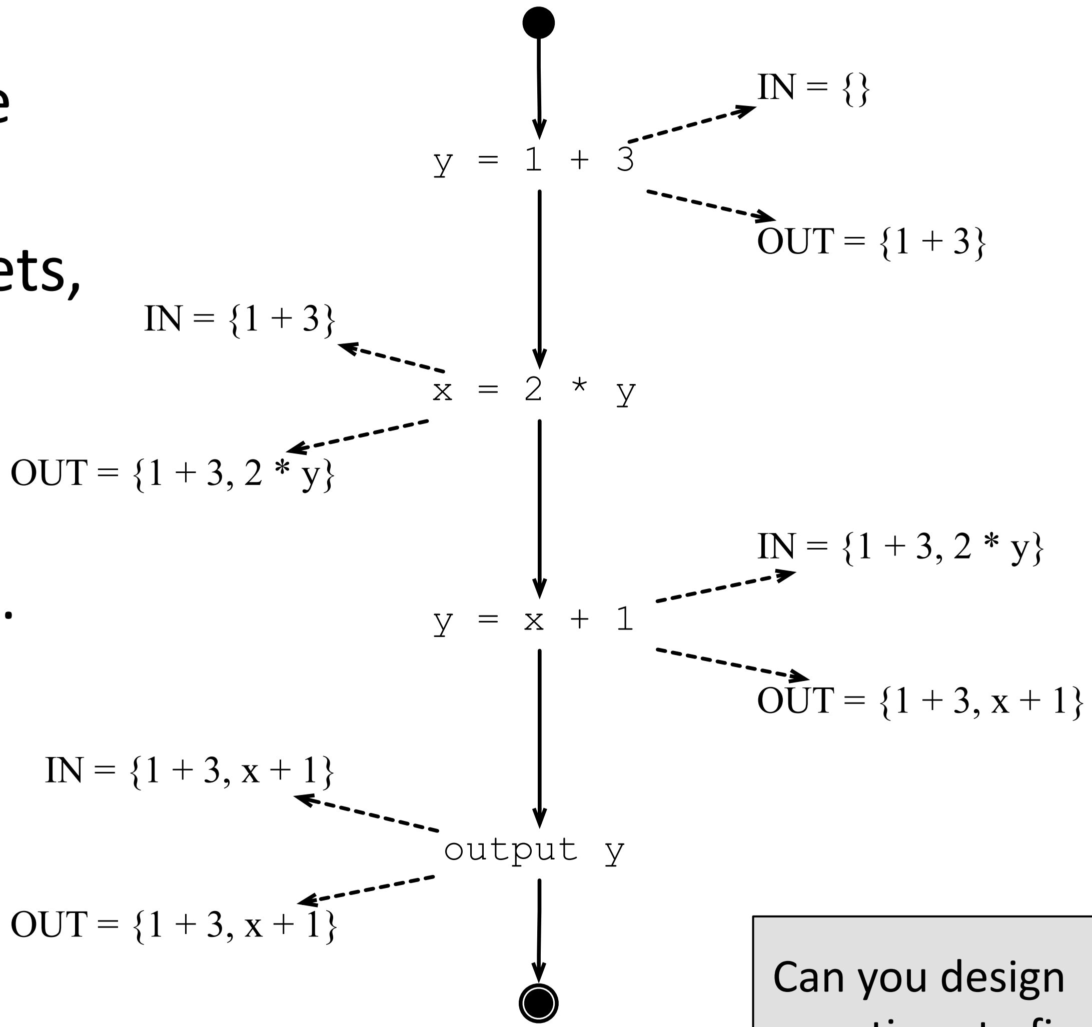
But, what if a program point has multiple predecessors?

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*.



Can you design equations to find these sets?

Dataflow Equations for Availability

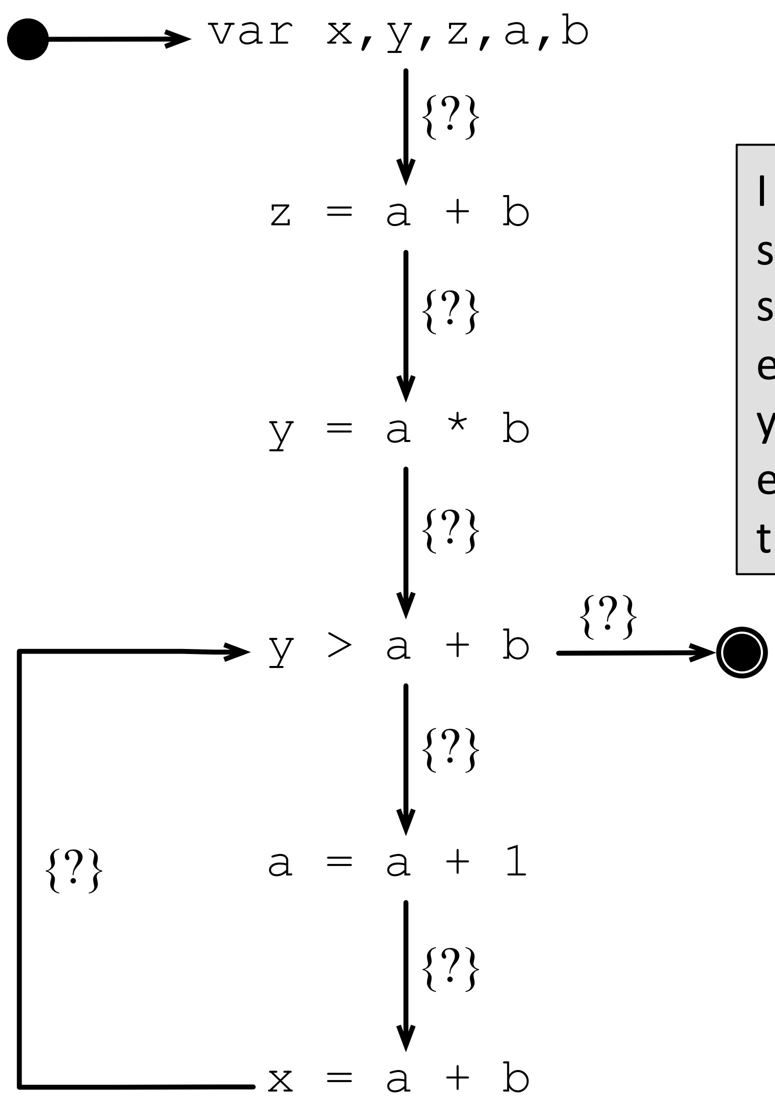
$$p : v = E$$

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

- *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.

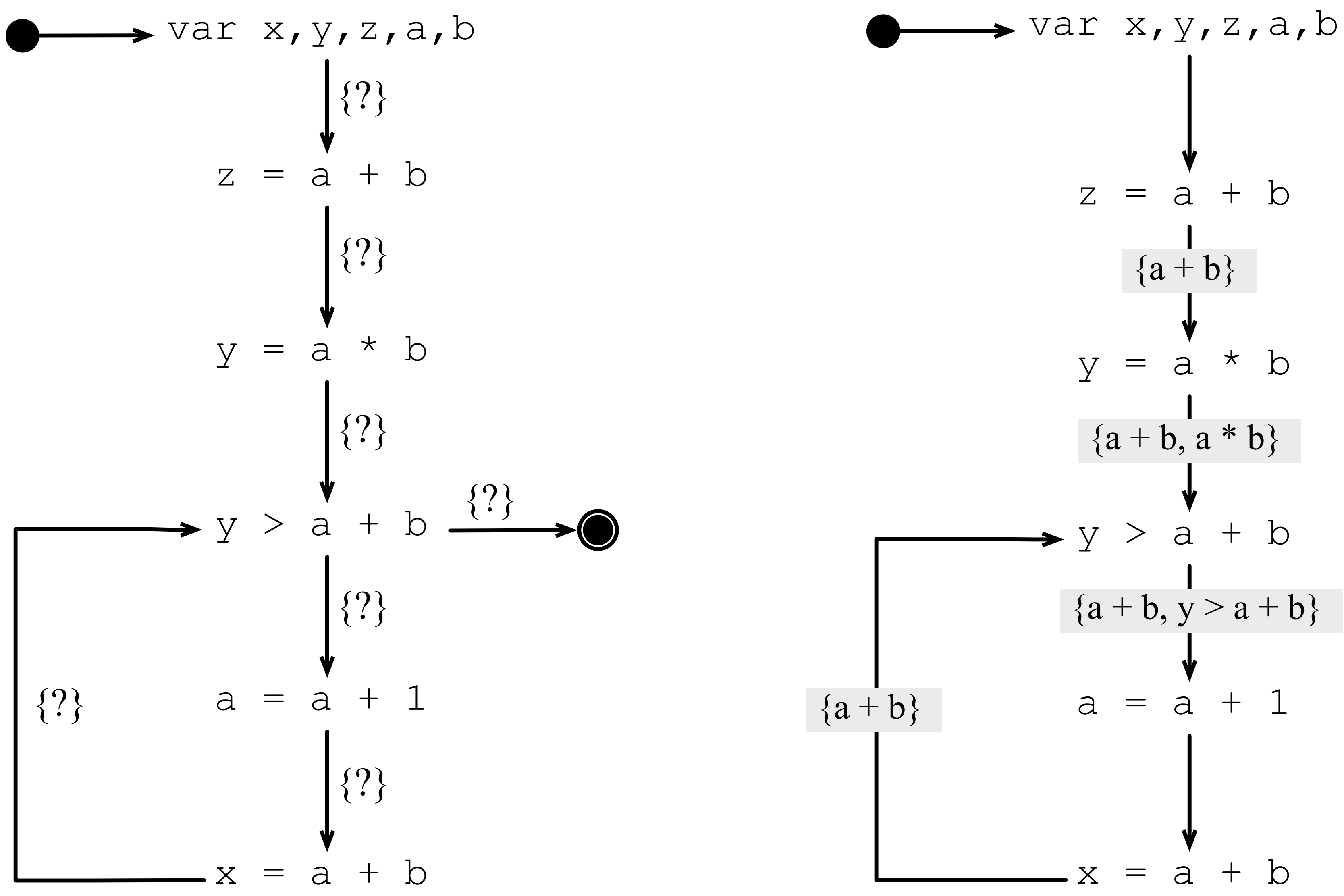
How do these equations differ from those used in liveness analysis?

Example of Available Expressions



I know you have seen already the solution of this example, but could you apply the equations to get them?

Example of Available Expressions



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 demands?

Very Busy Expressions

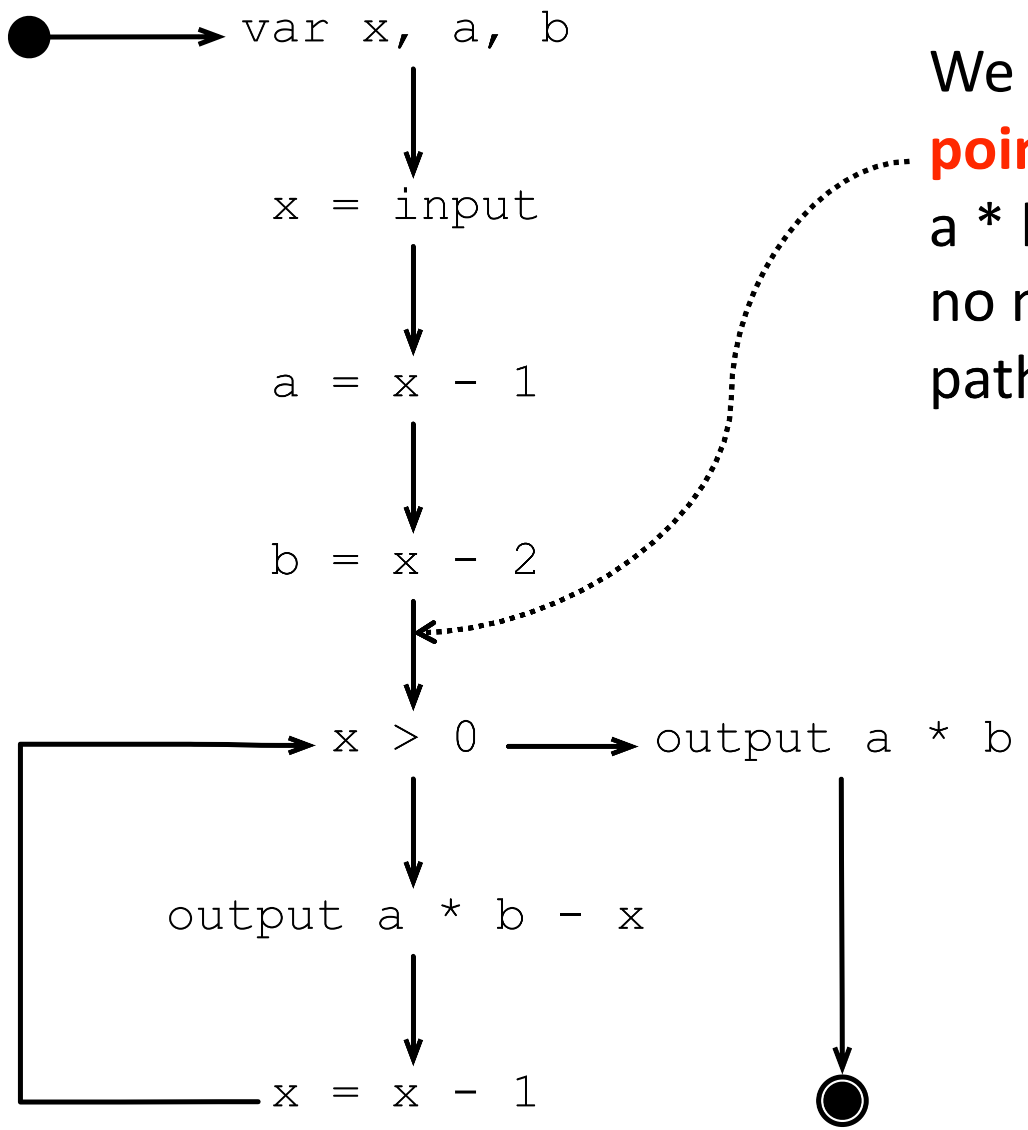
```
var x, a, b
x = input
a = x - 1
b = x - 2
while (x > 0) {
    output a * b - x
    x = x - 1
}
output a * b
```

- Consider the expression a * b
  - Does it change inside the loop?
- So, again, how could we optimize this program?

> Again: which information does this optimization demands?

- Generally it is easier to see opportunities for optimizations in the CFG.
  - How is the CFG of this program?

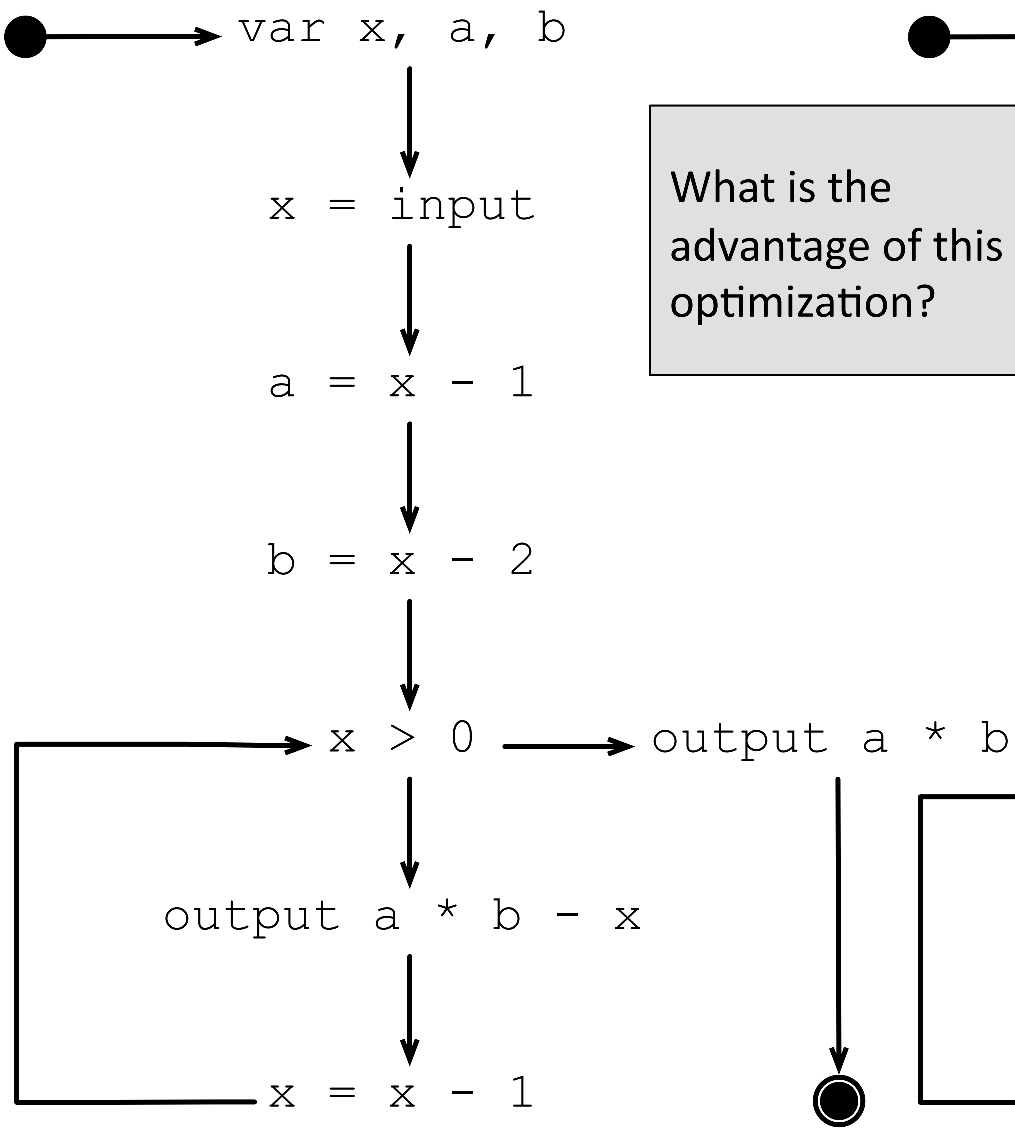
Very Busy Expressions



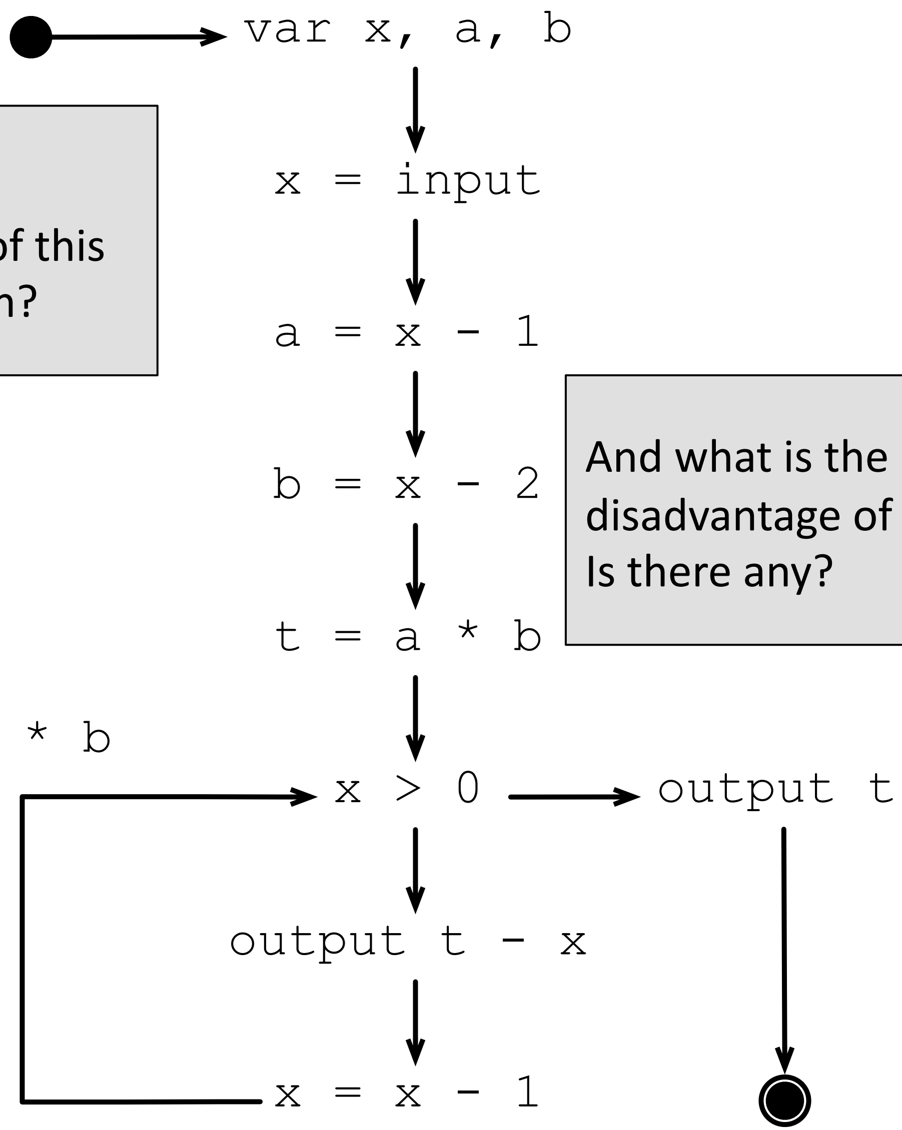
We know that at this **point**, the expression a * b will be computed no matter which program path is taken.

So, how could we improve this code?

Very Busy Expressions



What is the advantage of this optimization?

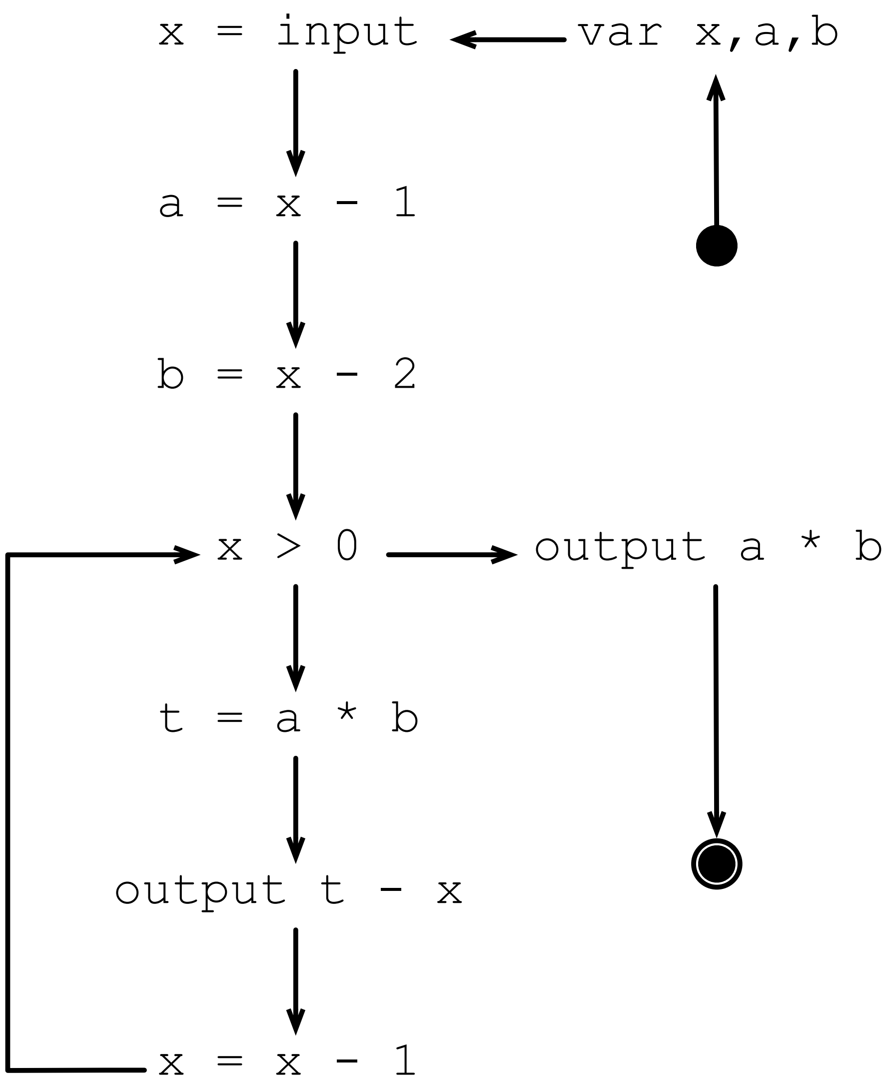


And what is the disadvantage of it? Is there any?

Very Busy Expressions

Which expressions are very busy in our example?

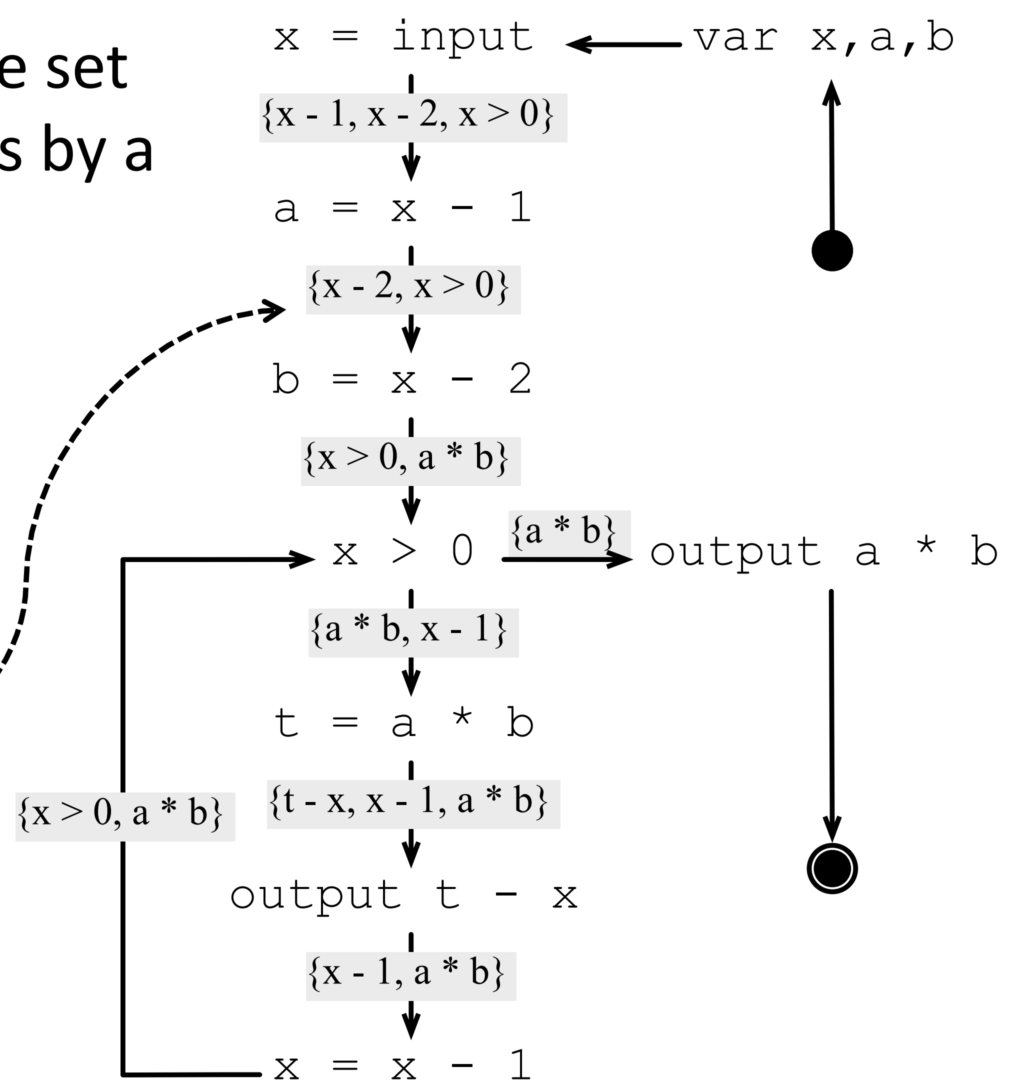
- In order to apply the previous optimization, we had to know that a * b was a very busy expression before the loop.
- An expression is very busy at a program point if it will be computed before the program terminates along any path that goes from that point to the end of the program.



Very Busy Expressions

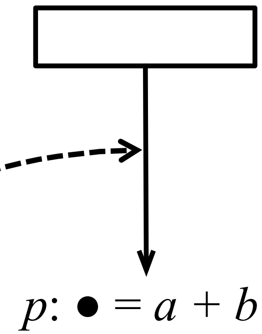
- We can approximate the set of very busy expressions by a dataflow analysis.
- How does information originate?
- How does information propagate?

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*.



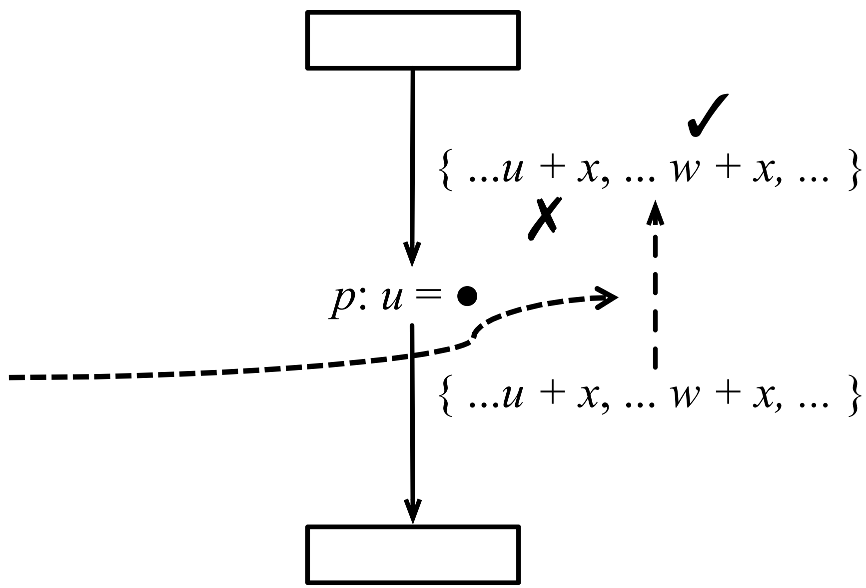
The Propagation of Information

- 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*.

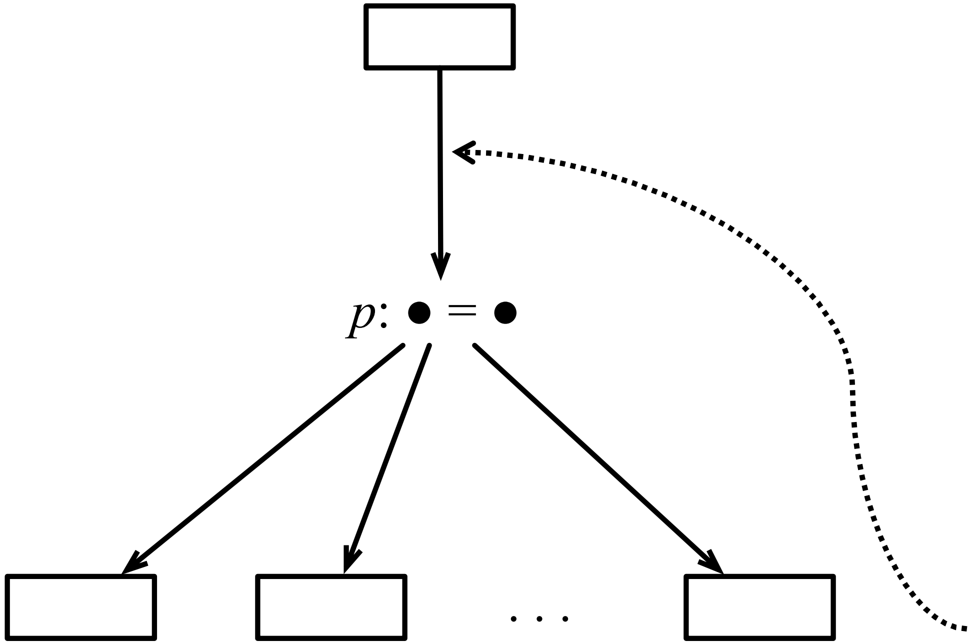
1. or
   1. It is used at *p*.

But, what if a program point has multiple successors?

This is the direction through which information propagates.



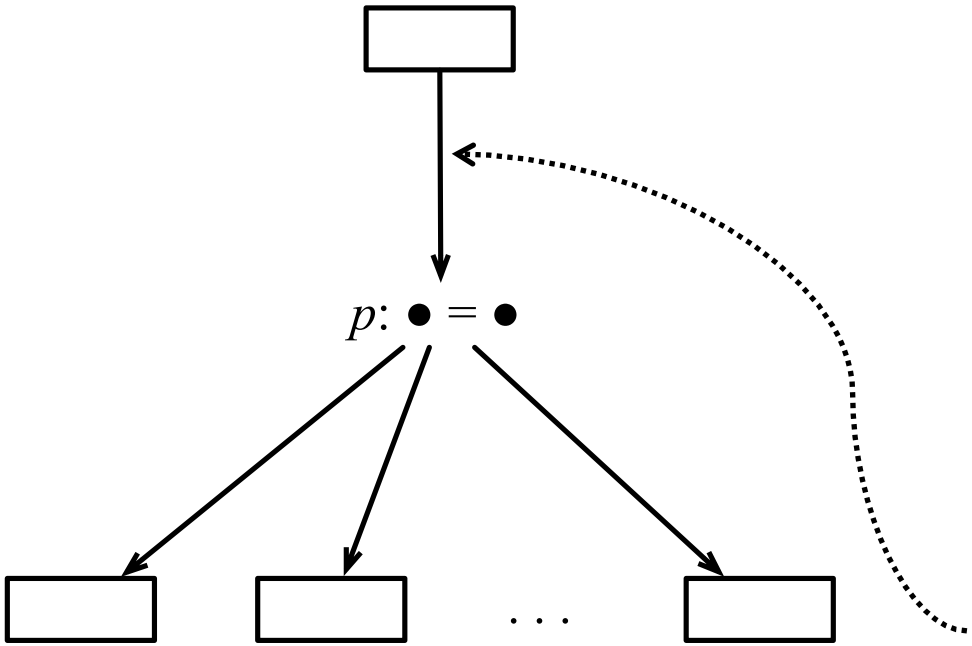
Joining Information



But, what if a program point has multiple successors?

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

Joining Information



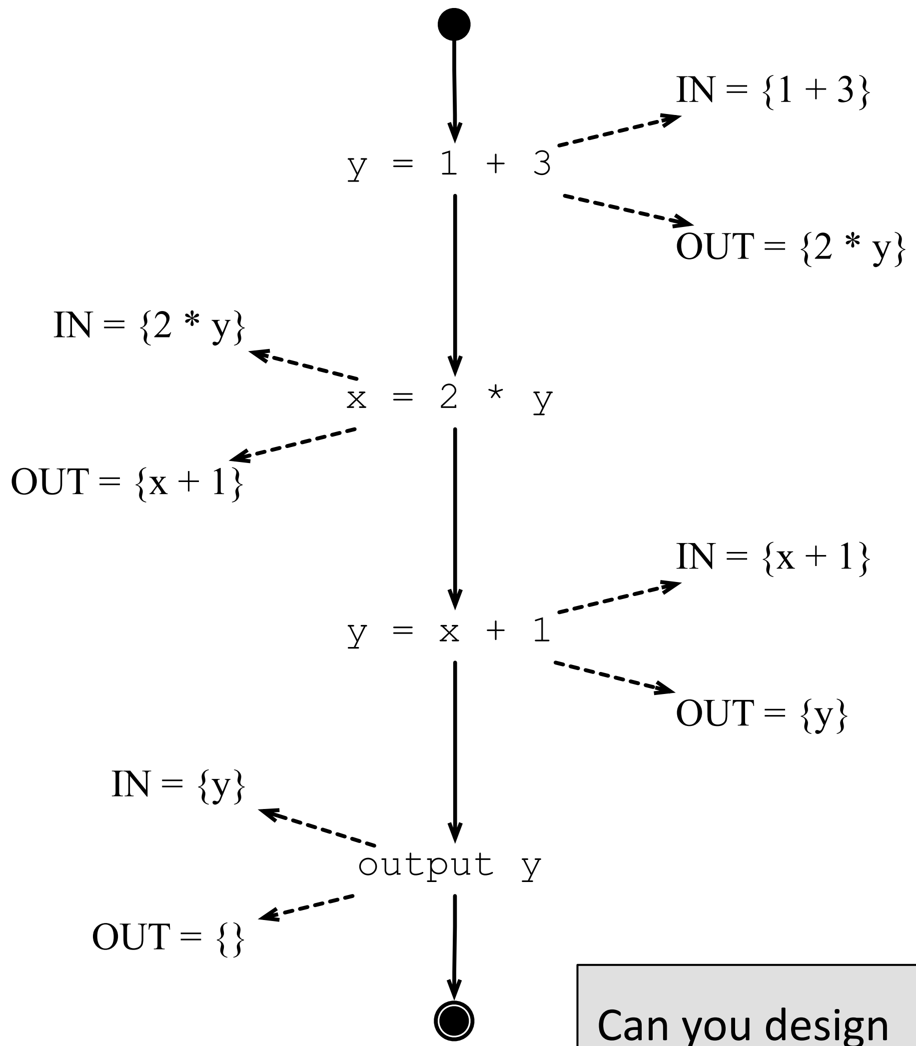
But, what if a program point has multiple successors?

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*.



Can you design equations to find these sets?

Dataflow Equations for Very Busy Expressions

$$p : v = E$$

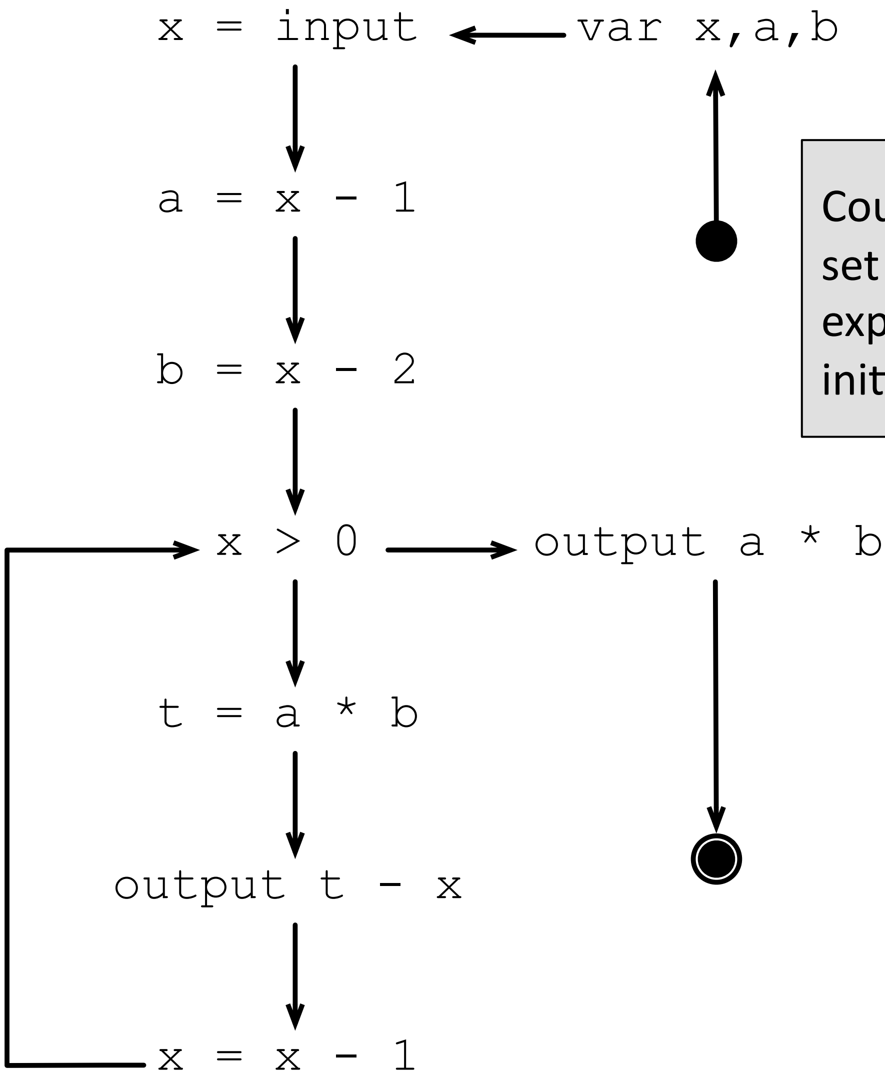
$$\begin{aligned} IN(p) &= (OUT(p) \setminus \{v\}) \cup \{E\} \\ OUT(p) &= \bigcap IN(p_s), p_s \in succ(p) \end{aligned}$$

- *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

How do these equations differ from those used in liveness analysis?

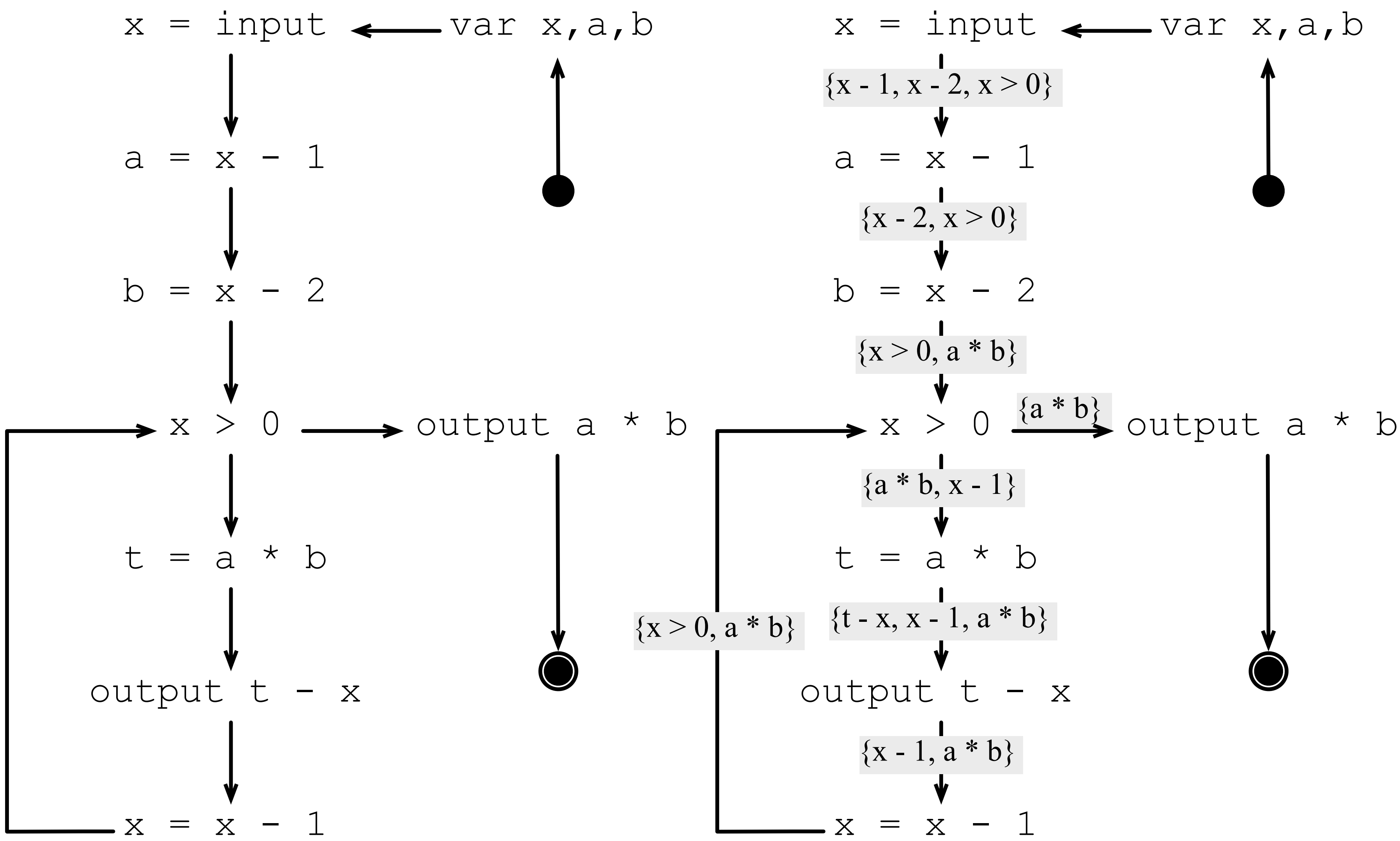
And what do they have in common?

Example of Very Busy Expressions



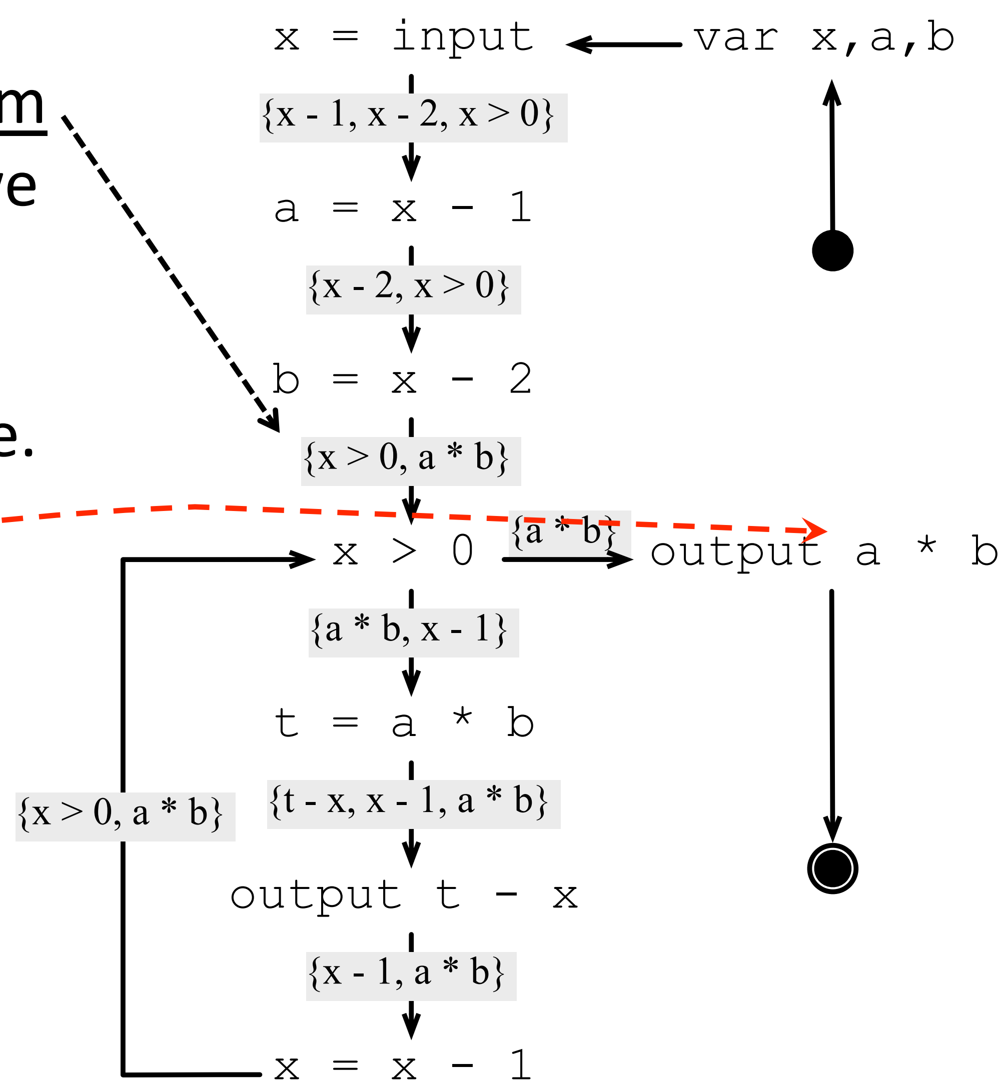
Could we find the set of very busy expressions in our initial example?

Example of Very Busy Expressions



Safe Code Hoisting

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



What if we did not have this a * b **here**. Would our transformation still 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?

Which information does this optimization demand?

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;
```

- The assignment z = z – 1 is *dead*.
- What is a dead assignment?
- How can we find them?

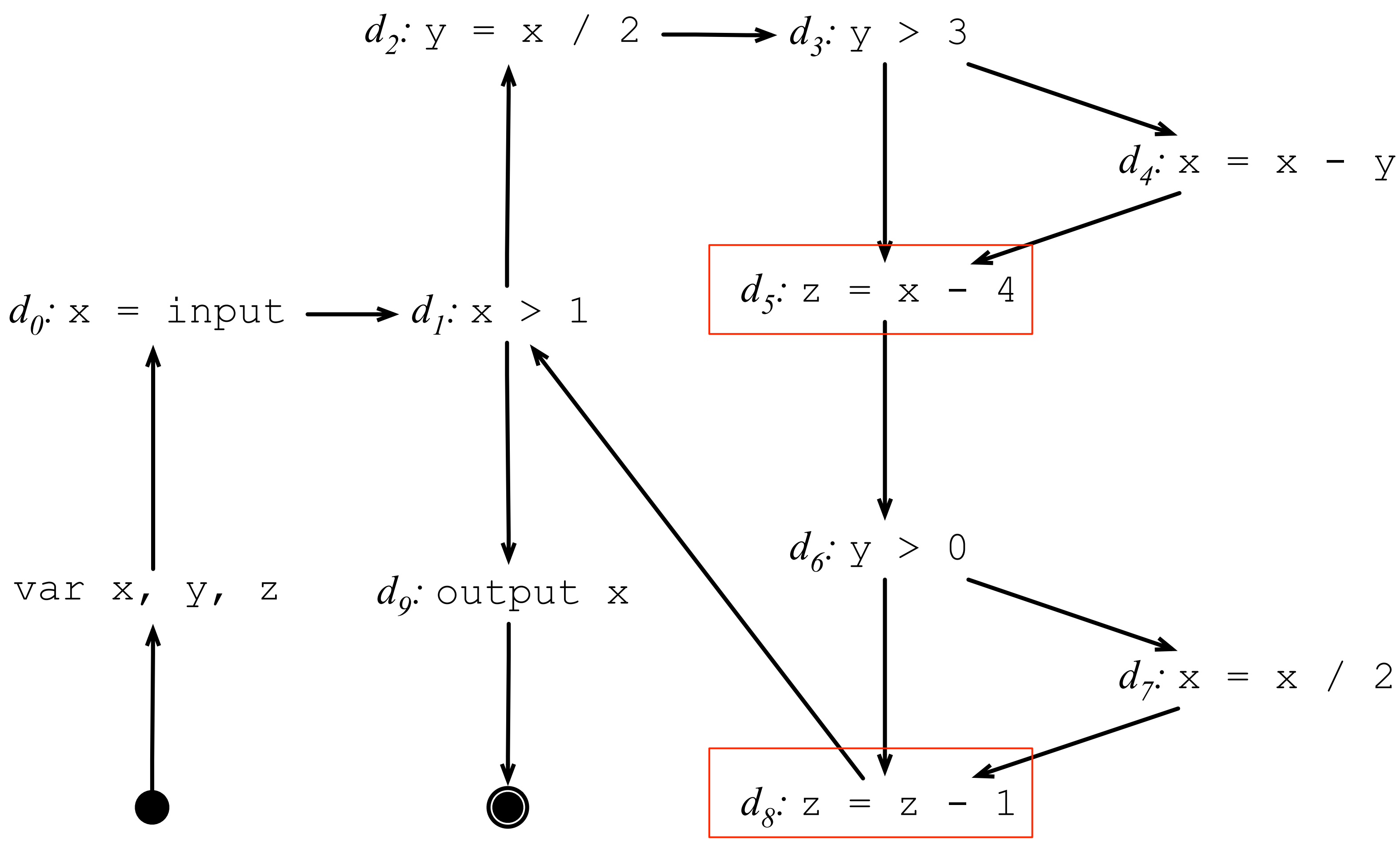
Again: which information does this optimization demand?

- Generally it is easier to see opportunities for optimizations in the CFG.
    - How is the CFG of this program?

How to compute the set of reaching definitions in a program?

Reaching Definitions

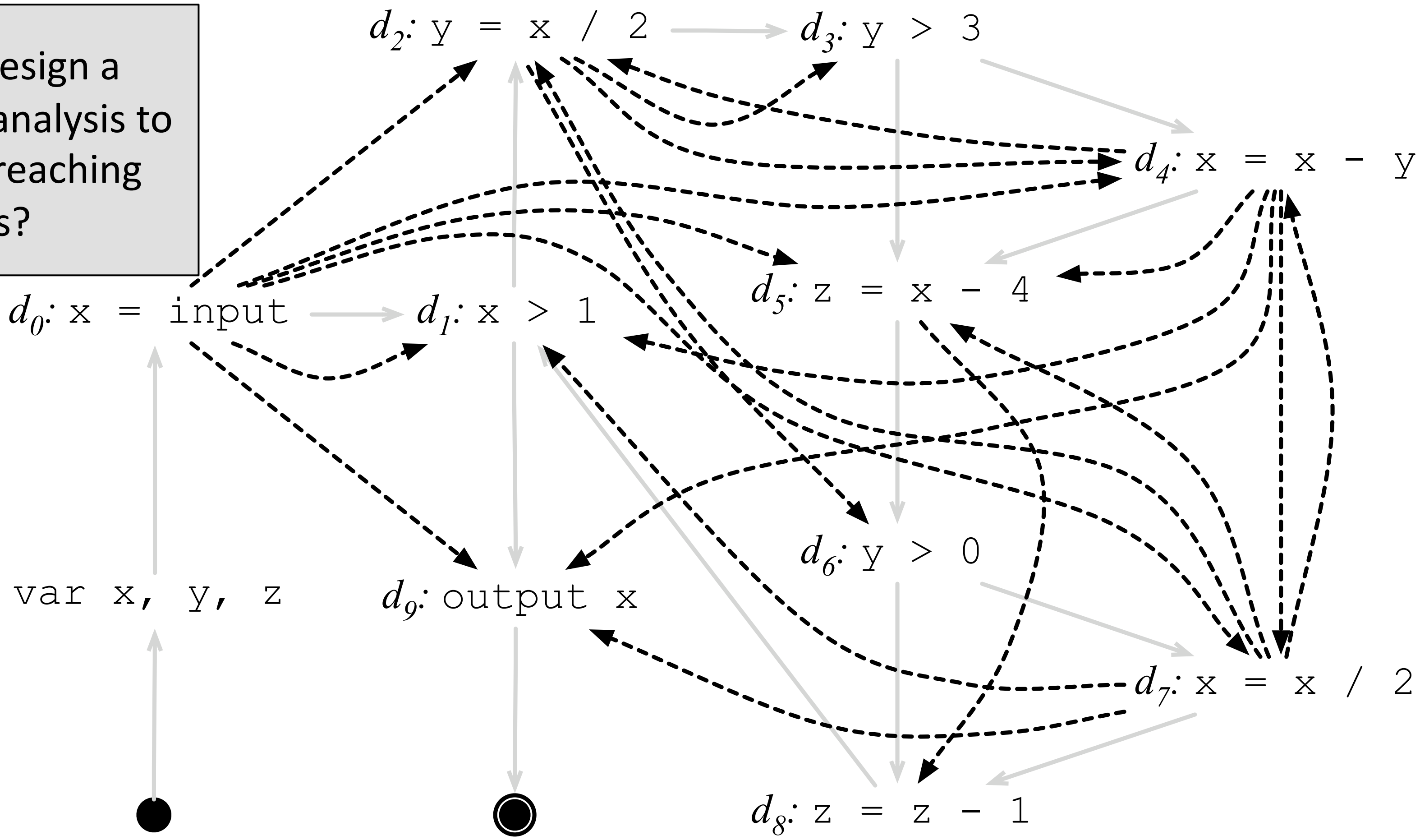
- We know that the assignment z = z – 1 is dead because this definition of z does not reach any use of this variable.



Reaching Definitions

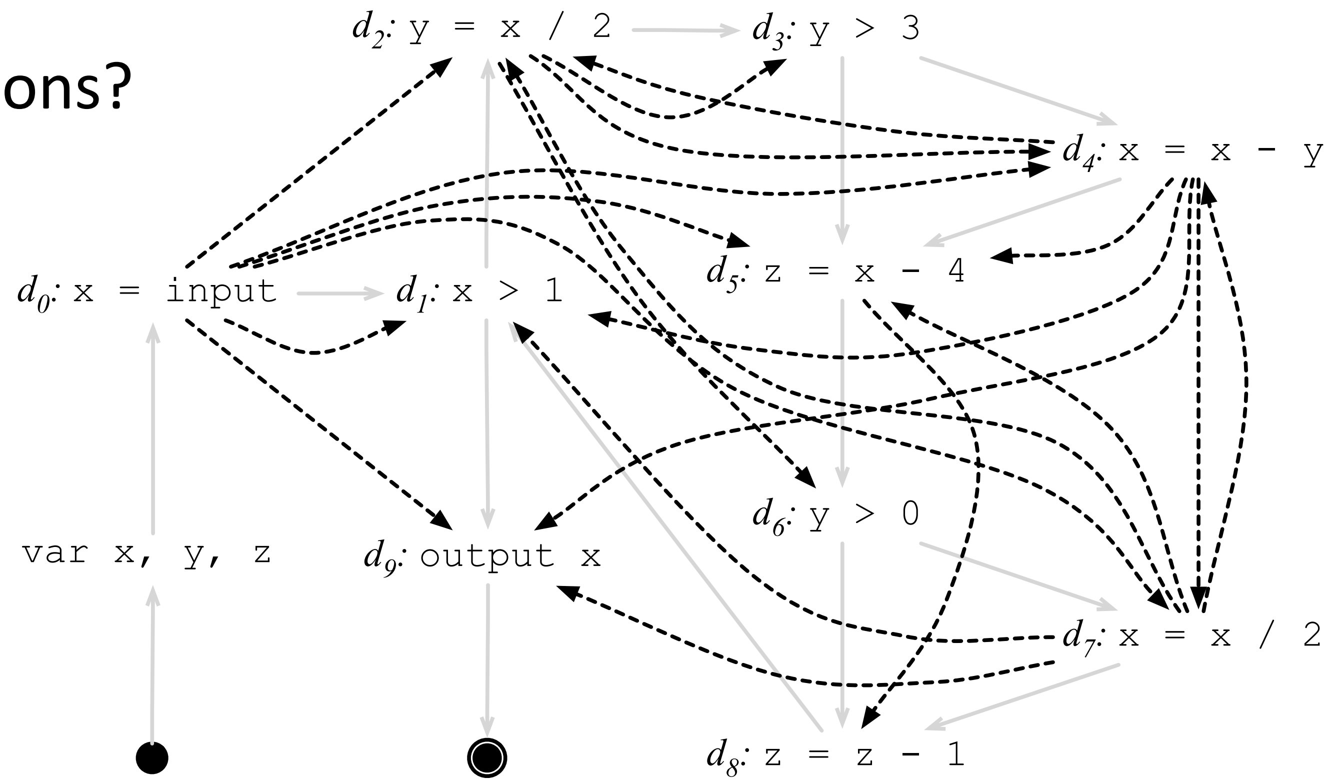
- We say that a definition of a variable *v*, at a program point *p*, reaches a program point *p'*, if there is a path from *p* to *p'*, and this path does not cross any redefinition of *v*.

Can you design a dataflow analysis to compute reaching definitions?

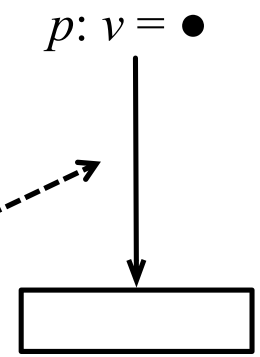


Reaching Definitions

- 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*.

And how do we propagate information?

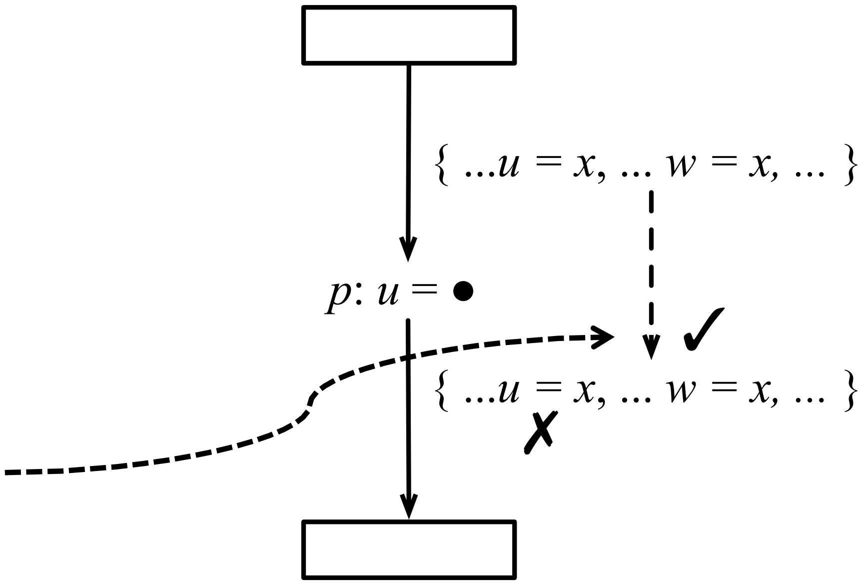
The Propagation of Information

- 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*.

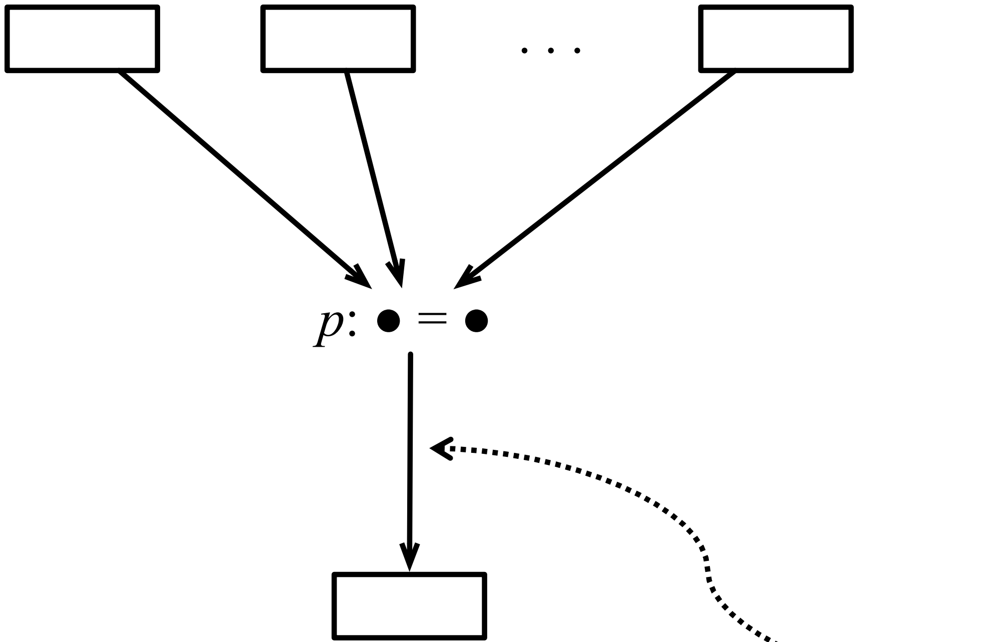
1. or
    1. Variable v is defined at *p*.

But, what if a program point has multiple predecessors?

This is the direction through which information propagates.

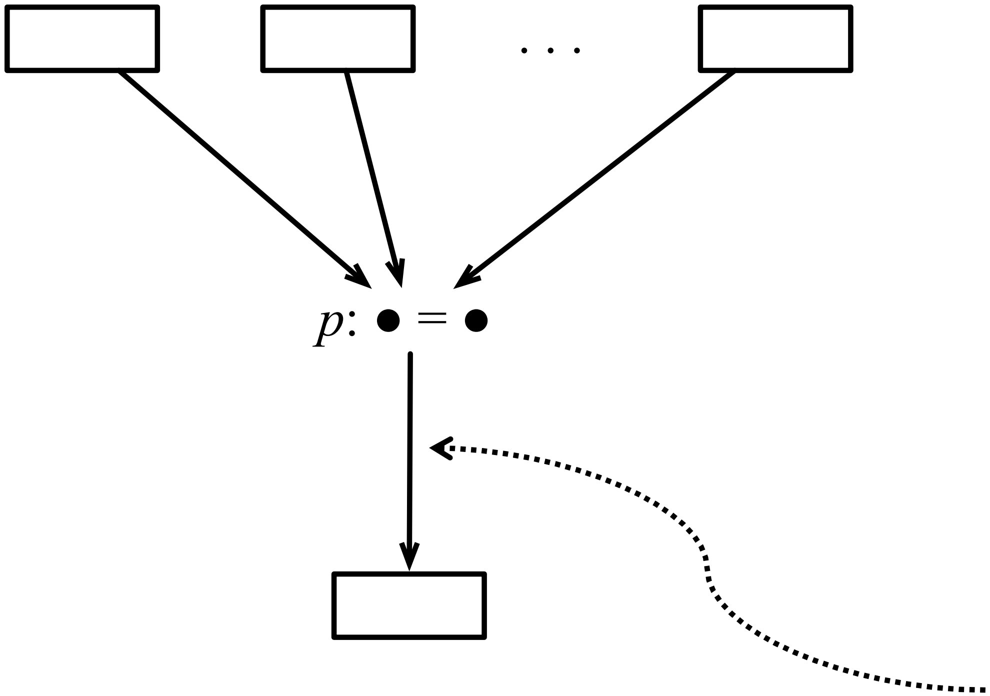


Joining Information



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

Joining Information

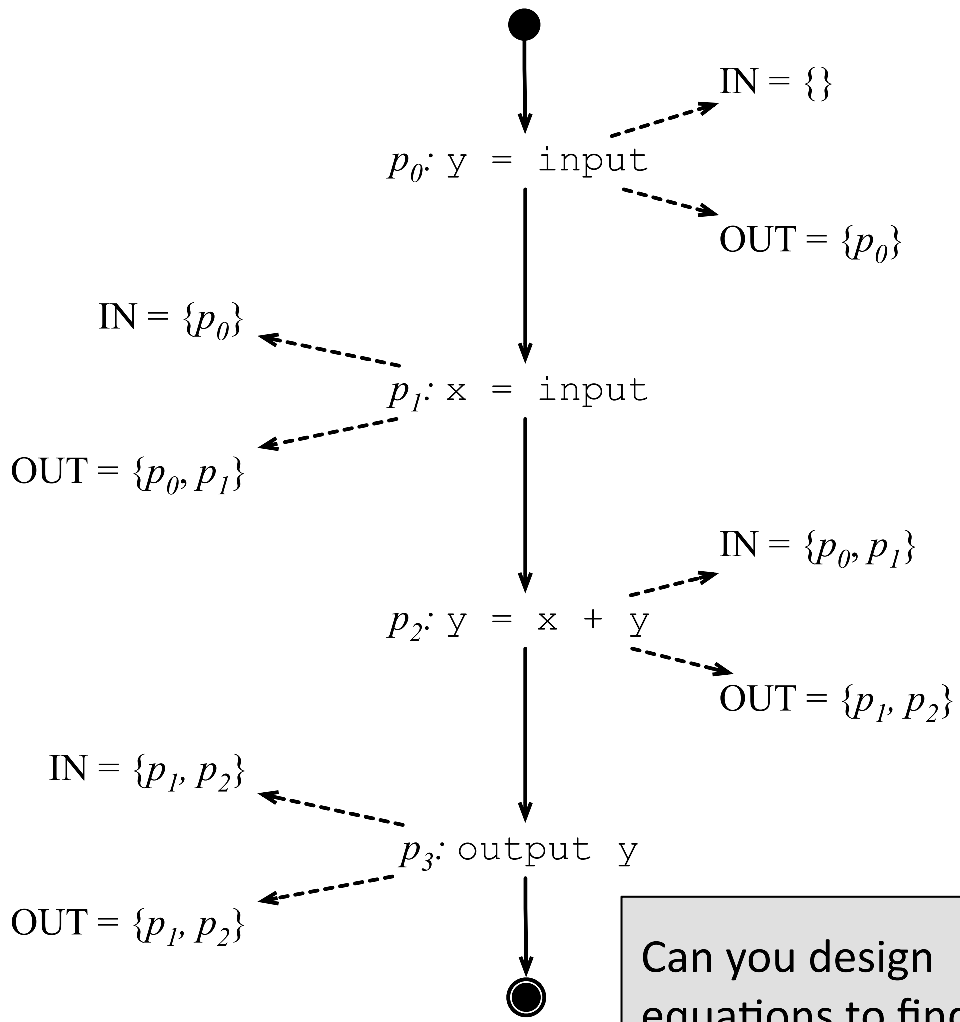


How do we find the definitions that reach this program point?

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*.



Can you design equations to find these sets?

Dataflow Equations for Reaching Definitions

$$p : v = E$$

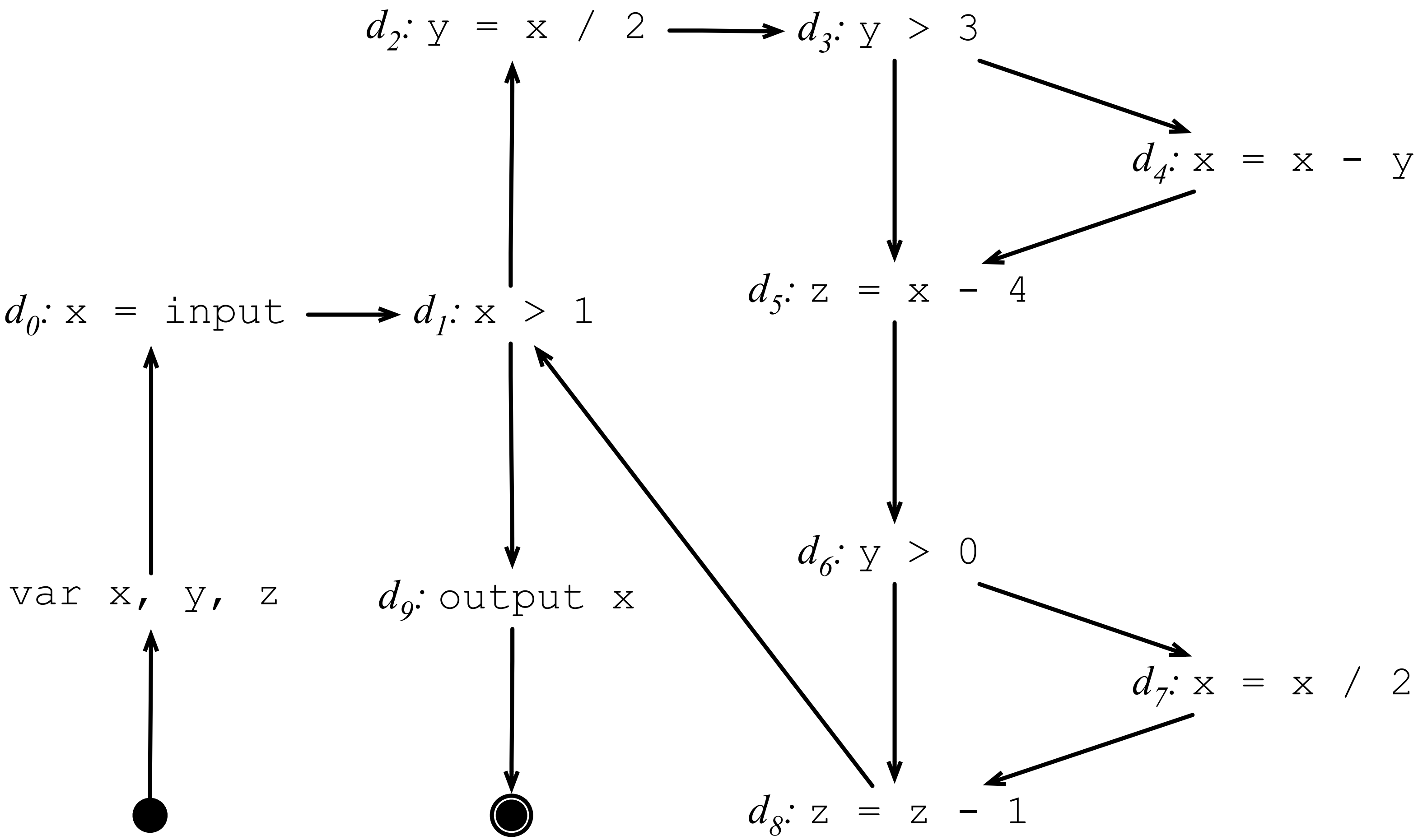
Why 'p' and not 'v'? What is stored in each set?

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

- *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.

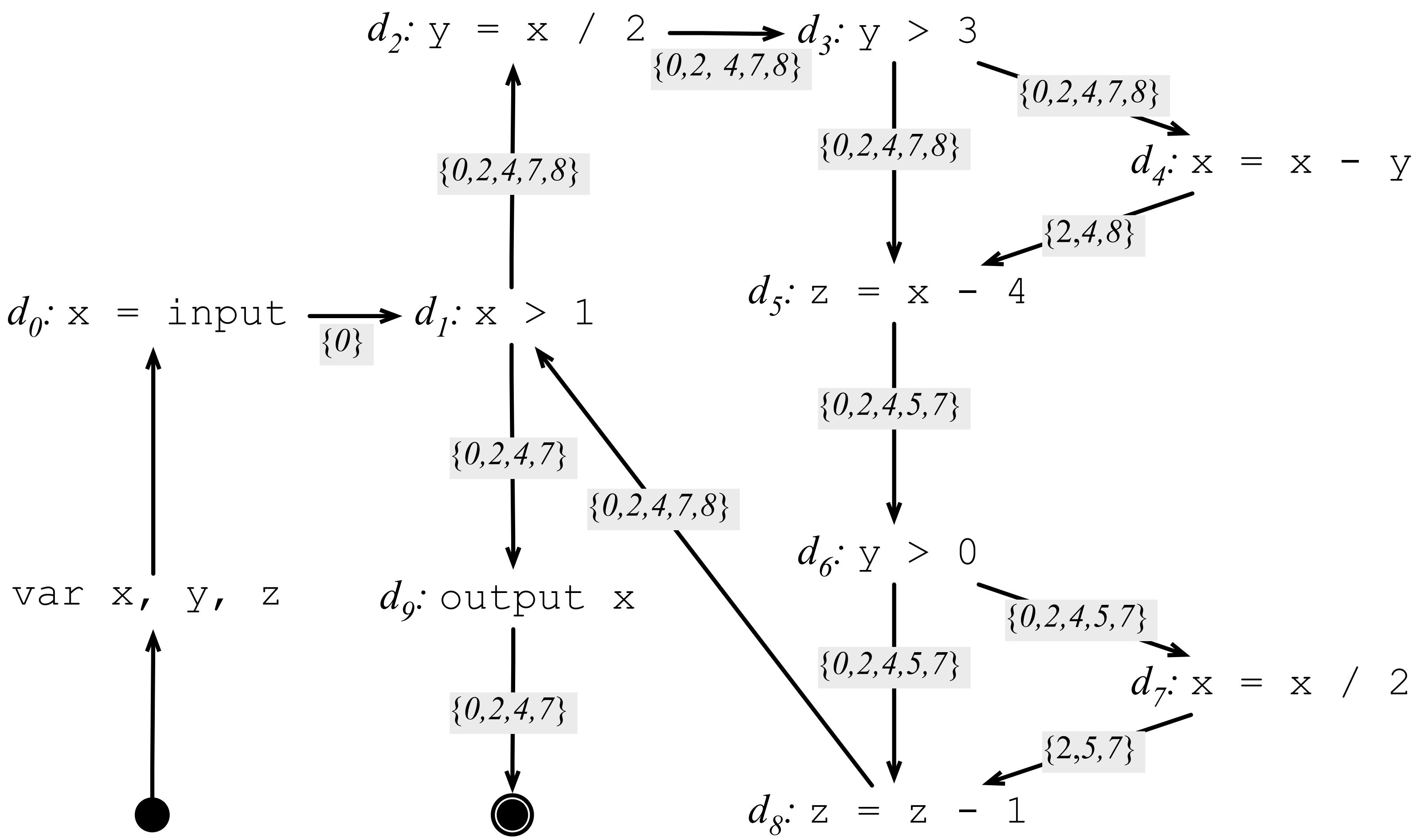
Example of Reaching Definitions Revisited

- 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?



THE MONOTONE FRAMEWORK

Finding Commonalities

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

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:

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

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

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	$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \cup \{E\}) \setminus \{v\}$ Available Expressions

Can you recognize the transfer functions of each analysis?

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] ∪ {v}
bgz v L	IN[p] = OUT[p] ∪ {v}
v = x + y	IN[p] = (OUT[p] \ {v}) ∪ {x, y}
v = u	IN[p] = (OUT[p] \ {v}) ∪ {u}

Which analysis is this one on the right?

The Dataflow Framework

- Many other program analyses fit into the dataflow framework.
  - Can you think of a few others?
- 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.

A Bit of History

- Dataflow analyses are some of the oldest allies of compiler writers.
- Frances Allen got the Turing Award of 2007. Some of her contributions touch dataflow analyses.

- Allen, F. E., "Program Optimizations", Annual Review in Automatic Programming 5 (1969), pp. 239-307
- Allen, F. E., "Control Flow Analysis", ACM Sigplan Notices 5:7 (1970), pp. 1-19
- Kam, J. B. and J. D. Ullman, "Monotone Data Flow Analysis Frameworks", Actal Informatica 7:3 (1977), pp. 305-318
- Kildall, G. "A Unified Approach to Global Program Optimizations", ACM Symposium on Principles of Programming Languages (1973), pp. 194-206