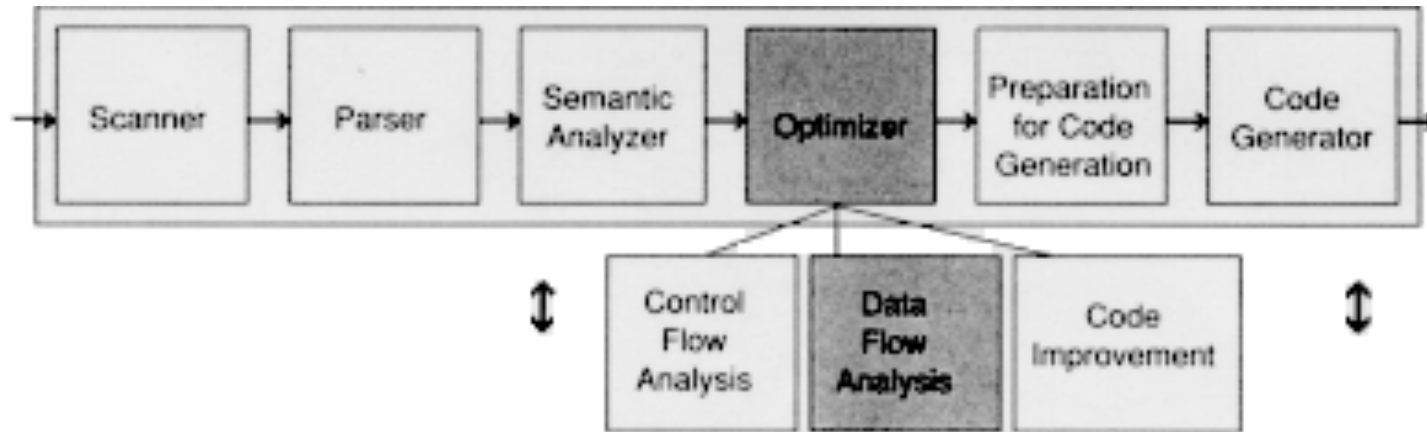


# Dataflow Analyses

# Code Optimization in Compilers

---



## Correctness Above All!

If may seem obvious, but it bears repeating that optimization should not change the correctness of the generated code. Transforming the code to something that runs faster but incorrectly is of little value. It is expected that the unoptimized and optimized variants give the same output for all inputs. This may not hold for an incorrectly written program (e.g., one that uses an uninitialized variable).

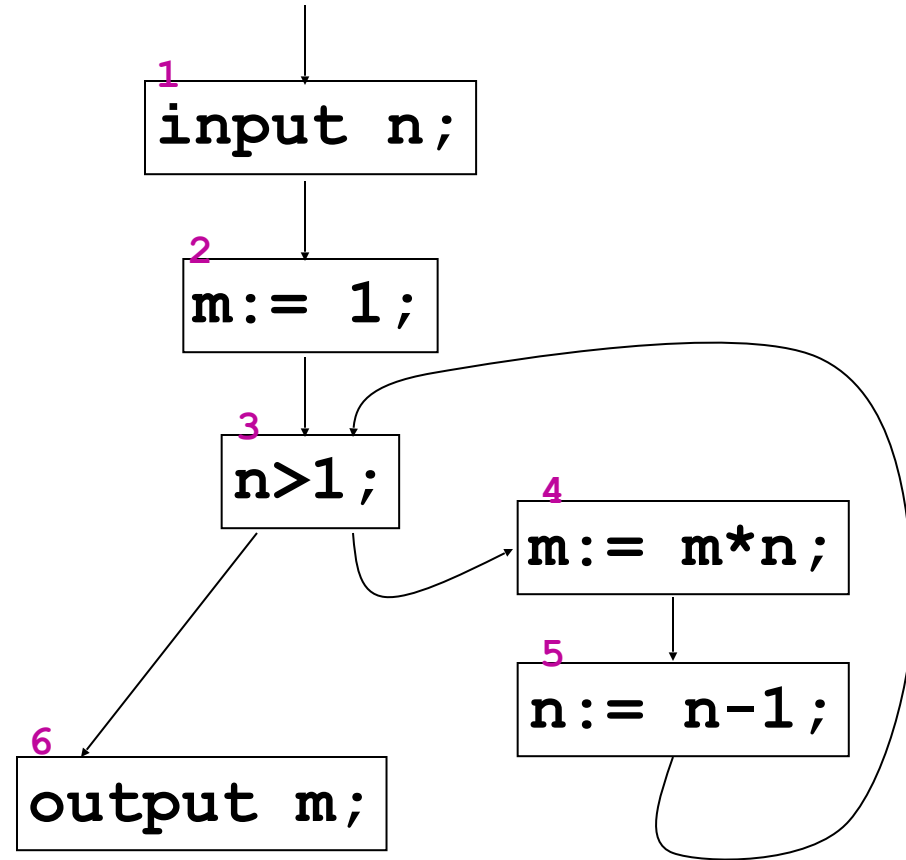
# Control flow graph

---

- Program commands are encoded by nodes in a control flow graph
- If a command  $S$  **may be directly followed** by a command  $T$  then the control flow graph must include a direct arc from the node encoding  $S$  to the node encoding  $T$

# Example

```
[ input n; ]1  
[ m := 1; ]2  
  [ while n > 1 do ]3  
    [ m := m * n; ]4  
      [ n := n - 1; ]5  
    [ output m; ]6
```





# Data-Flow analyses

---

We will see data-flow analyses:

- **Liveness** analysis
- **Reaching definitions** analysis
- **Available Expressions** analysis

# Liveness or Live Variables Analysis

- We need to translate the source program in the intermediate representation IR that can use a **large** (potentially unbounded) **number of registers**.
- but the program will be executed by a processor with a (finite and) **small number of registers**
- Two variables **a** and **b** can be stored in the same register when it turns out that **a** and **b** are **never simultaneously “used”**

## IR: Three Address Code

---

Three-address instruction has at most three operands and is typically a combination of an assignment and a binary operator.

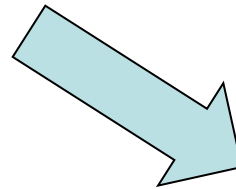
For example:  $t1 := t2 + t3$ .

The name derives from the use of three operands in these statements even though instructions with fewer operands may occur.

# IR: Three Address Code example

---

```
# Calculate one solution to the  
[[quadratic equation]].  
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

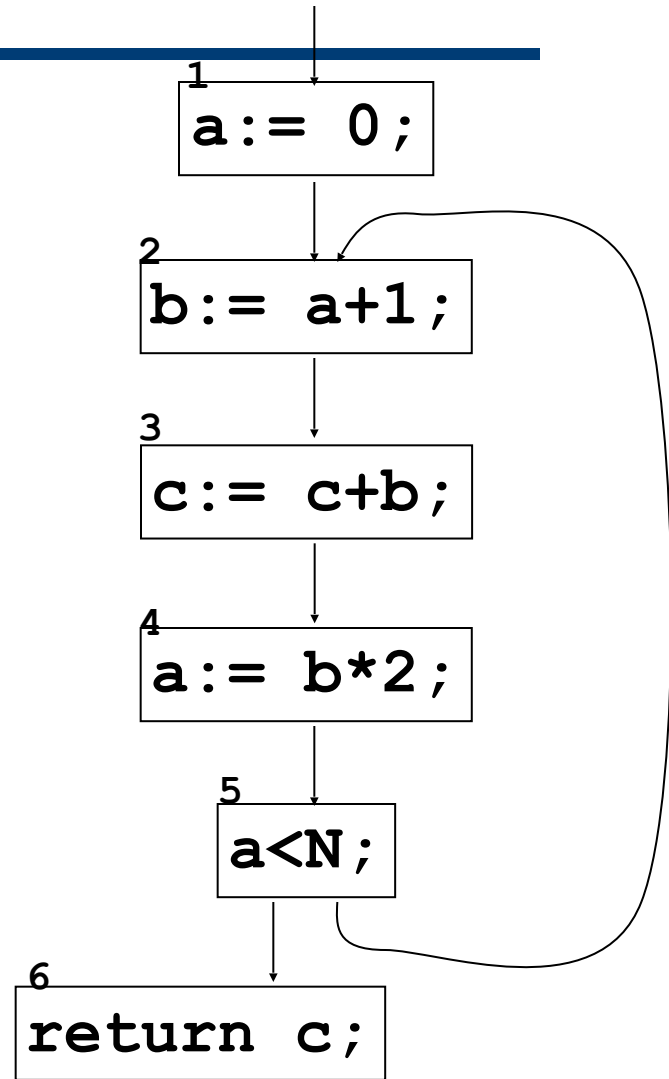


```
t1 := b * b  
t2 := 4 * a  
t3 := t2 * c  
t4 := t1 - t3  
t5 := sqrt(t4)  
t6 := 0 - b  
t7 := t5 + t6  
t8 := 2 * a  
t9 := t7 / t8  
x := t9
```

# Example

```
a = 0;
do {
  b = a+1;
  c += b;
  a = b*2;
}
while (a<N);
return c;
```

We want to know if **a** and **b** are simultaneously used.



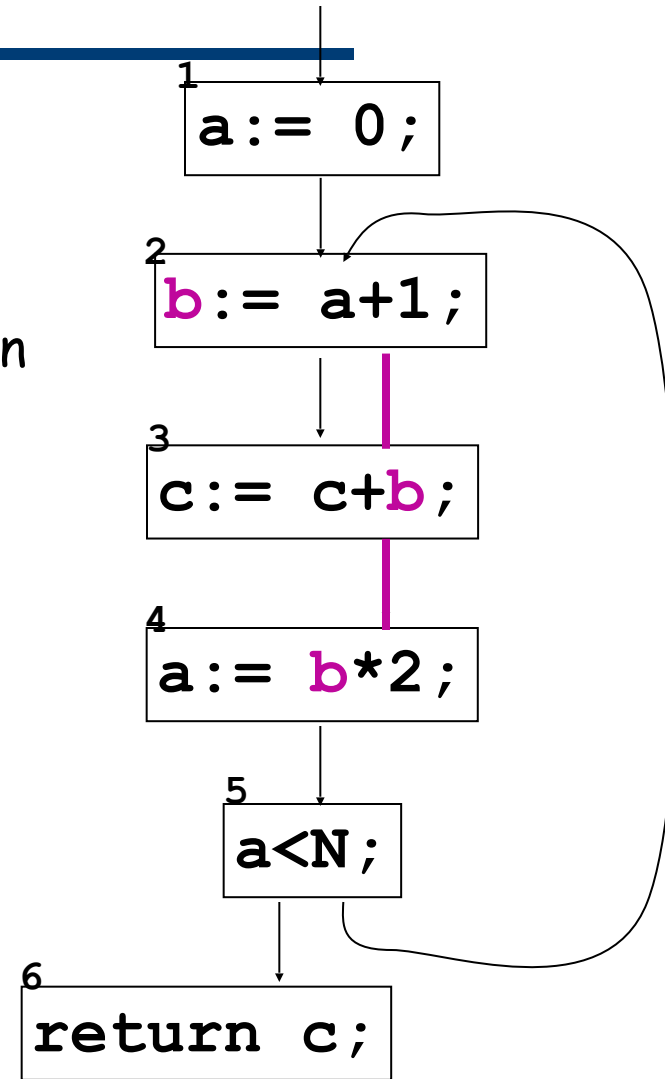
## Live Variables Analysis

---

- A compiler needs to analyze programs in IR in order to find out which variables are simultaneously used
- A variable  $X$  is **live** at the exit of a command  $C$  if  $X$  stores a value which will be **actually used** in the future, that is,  $X$  will be used as R-value with no previous use as L-value
- A variable  $X$  which is not live at the exit of  $C$  is also called **dead** (this information can be used for dead code elimination)
- This is an undecidable property

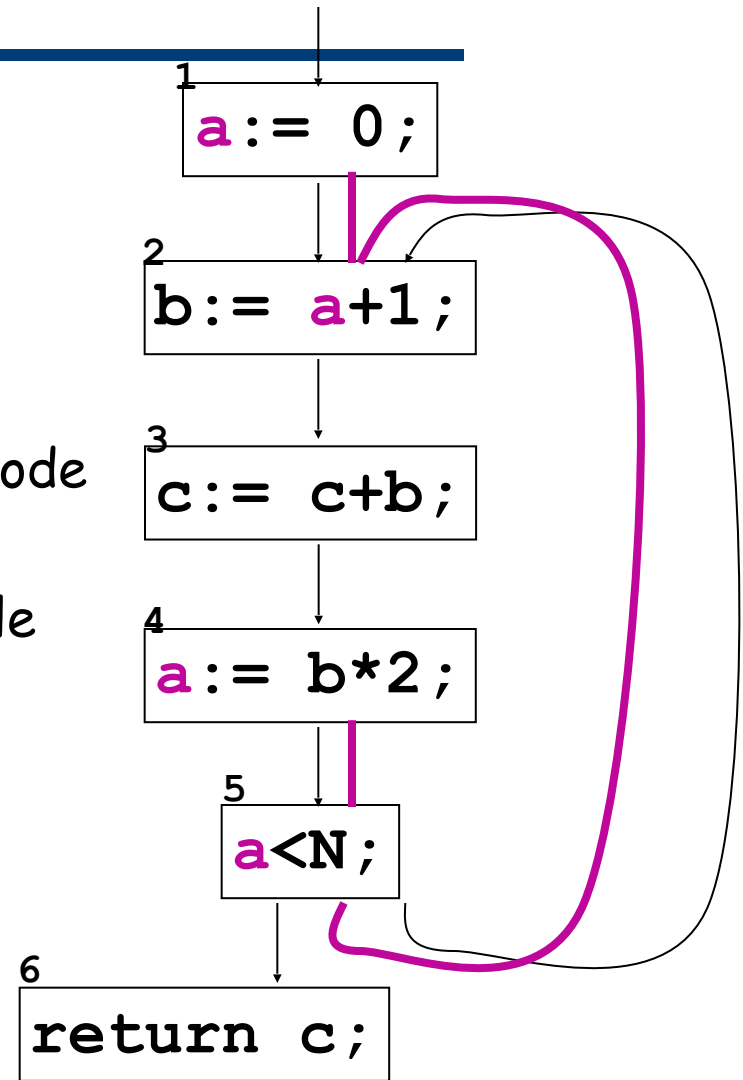
# Back to the example

- A variable  $X$  is live when it stores a value which will be later used with no prior assignment to  $X$
- The "last" use of the variable  $b$  as r-value is in command 4
- The variable  $b$  is used in command 4: it is therefore live along the arc  $3 \rightarrow 4$
- Command 3 does not assign  $b$ , hence  $b$  is live along  $2 \rightarrow 3$
- Command 2 assigns  $b$ . This means that the value of  $b$  along  $1 \rightarrow 2$  will not be used later
- Thus, the "live range" of  $b$  turns out to be:  $\{2 \rightarrow 3, 3 \rightarrow 4\}$



# Live variables

- a is live along  $4 \rightarrow 5$  and  $5 \rightarrow 2$
- a is live along  $1 \rightarrow 2$
- a is not live along  $2 \rightarrow 3$  and  $3 \rightarrow 4$
- Even if the variable a stores a value in node 3, this value will not be later used, since node 4 assigns a new value to the variable a.

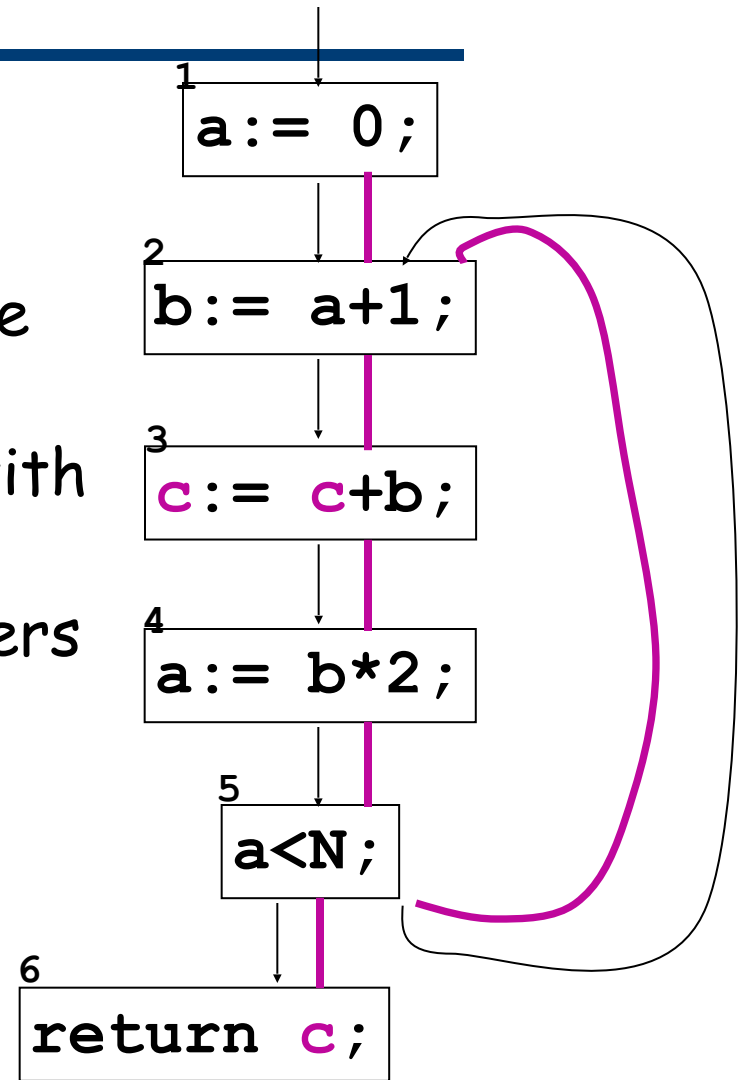


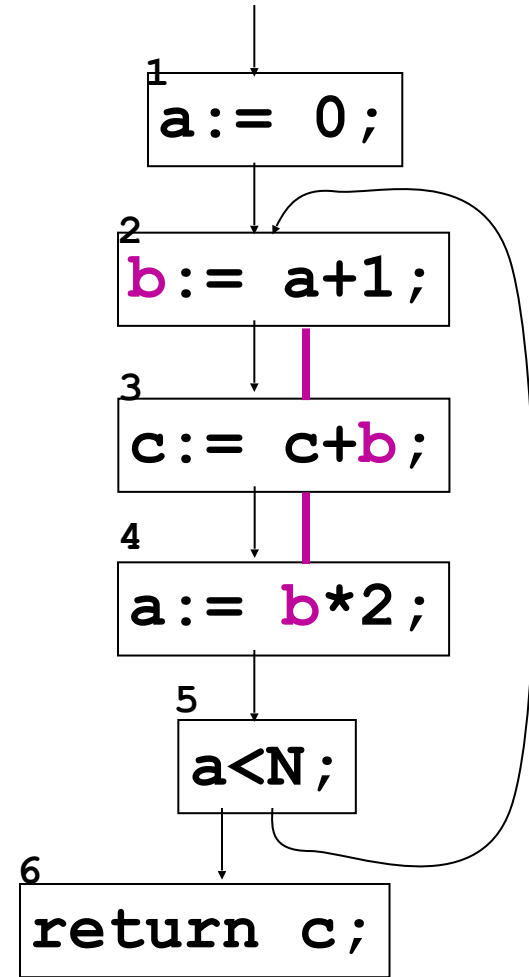
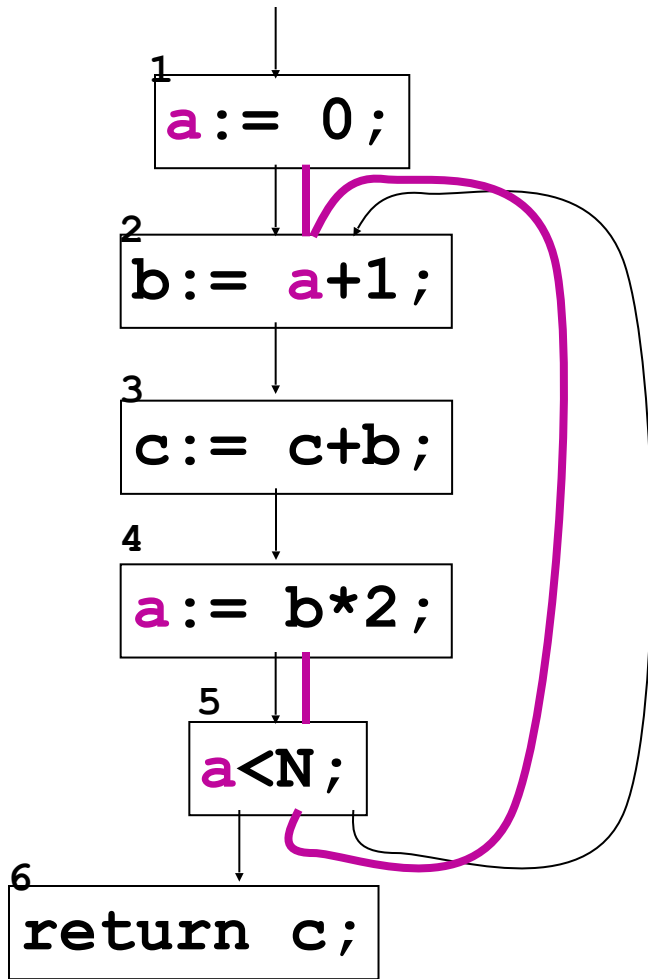


## More on live variables

---

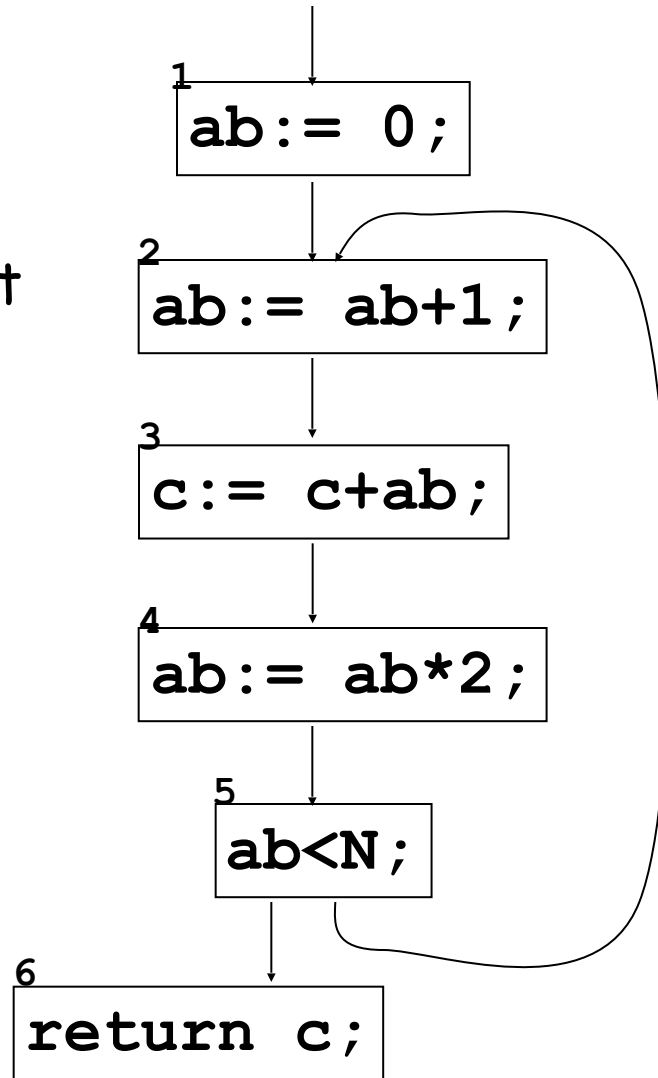
- **c** is live along all the arcs
- By the way: liveness analysis can be exploited to deduce that if **c** is a local variable then **c** will be used with no prior initialization (this information can be used by compilers to raise a warning message)





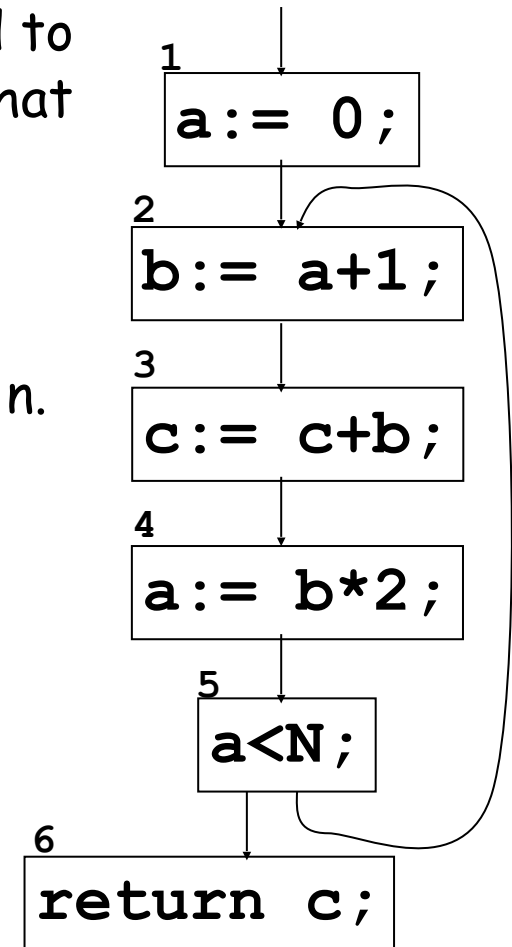
→ Two registers are enough: variables `a` and `b` will be never simultaneously live along the same arc

Variables **a** and **b** will be never simultaneously live along the same arc. Hence, instead of using two distinct variables **a** and **b** we can correctly employ a single variable **ab**



## We need a way to compute live variables

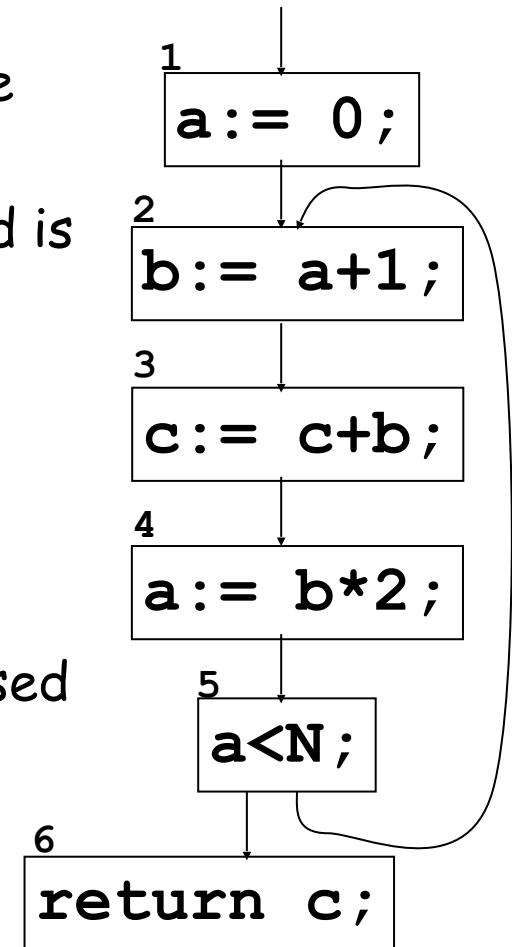
- A CFG has outgoing edges (**out-edges**) that lead to successor nodes, and ingoing edges (**in-edges**) that originate from predecessor nodes.
- **pre[n]** and **post[n]** denote, respectively, the predecessor and successor nodes of some node n.
- As an example, in this CFG:
  - 2 and 6 are successors of node 5 because  
5 → 6 and 5 → 2 are the out-edges of 5
  - 1 and 5 predecessor 2 since  
5 → 2 and 1 → 2 are the in-edges of 2
  - $\text{pre}[2]=\{1,5\}$ ;  $\text{post}[5]=\{2,6\}$ .



# Notation

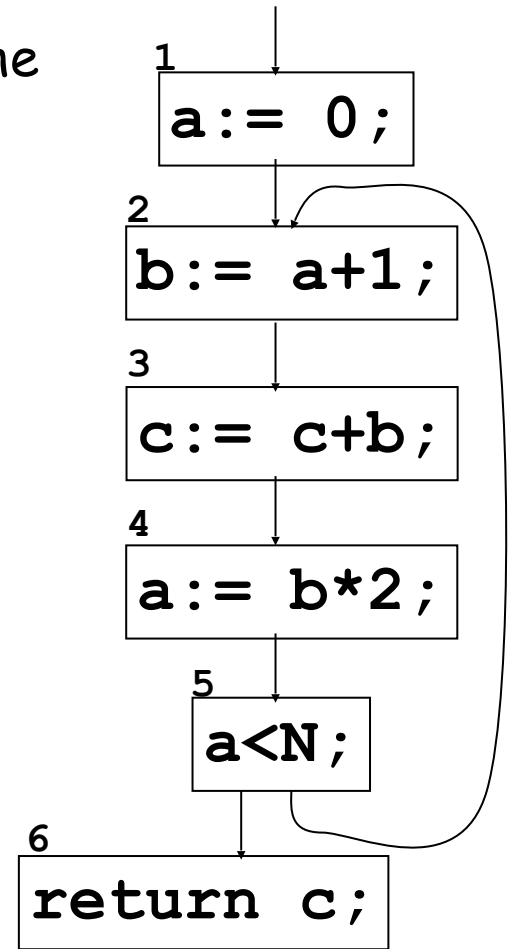
---

- An assignment to some variable (a use of the variable as L-value) is called a **definition** of the variable
- A use of some variable as R-value in a command is called a **use** of this variable
- **def**[n] denotes the set of variables that are defined in the node n
- **use**[n] denotes the set of variables that are used in the node n
- As an example, in this CFG:
  - $\text{def}[3]=\{c\}$ ,  $\text{def}[5]=\emptyset$
  - $\text{use}[3]=\{b,c\}$ ,  $\text{use}[5]=\{a\}$



# Formalization

- Of the property: A variable  $x$  is **live** along an arc  $e \rightarrow f$  if there exists a **real execution path**  $P$  from the node  $e$  to some node  $n$  such that:
  - $e \rightarrow f$  is the first arc of such path  $P$
  - $x \in \text{use}[n]$
  - for any node  $n' \neq e$  and  $n' \neq n$  in the path  $P$ ,  $x \notin \text{def}[n']$
- A variable  $x$  is **live-out** in some node  $n$  if  $x$  is live along **some** (i.e., at least one) out-edge of  $n$
- A variable  $x$  is **live-in** in some node  $n$  if  $x$  is live along **any** in-edge of  $n$



# Example

---

As an example, in this CFG:

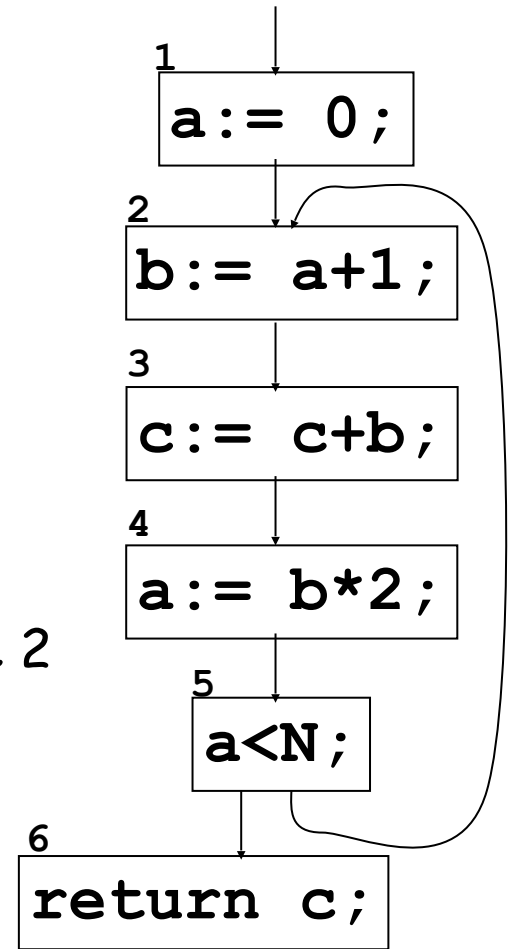
**a** is live along  $1 \rightarrow 2, 4 \rightarrow 5$  and  $5 \rightarrow 2$

**b** is live along  $2 \rightarrow 3, 3 \rightarrow 4$

**c** is live along any arc

**a** is live-in in node 2, while it is not live-out in node 2

**a** is live-out in node 5



## Computing Liveness

---

Let us define the following notation:

$in[n]$  is the set of variables that the static analysis determines to be **live-in** at node  $n$

$out[n]$  is the set of variables that the static analysis determines to be **live-out** at node  $n$

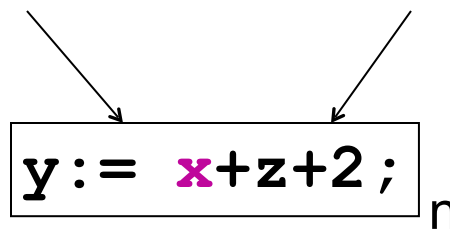


# Computing Liveness

n node of the CFG

Liveness information: the sets  $in[n]$  and  $out[n]$  is computed as an over-approximation in the following way

1. If a variable  $x \in use[n]$  then  $x$  is live-in in node  $n$ .  
In other terms, if a node  $n$  uses a variable  $x$  as R-value then this variable  $x$  is live along each arc that enters into  $n$ .

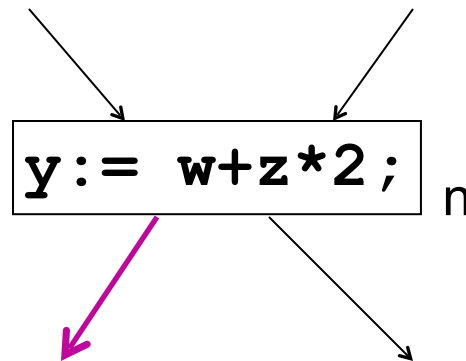


$$in[n] \supseteq use[n]$$

## Computing Liveness

---

2. If a variable  $x$  is live-out in a node  $n$  and  $x \notin \text{def}[n]$  then the variable  $x$  is also live-in in this node  $n$ .
- If a variable  $x$  is live for some arc that leaves a node  $n$  and  $x$  is not assigned in  $n$  then  $x$  is live for all the arcs that enter in  $n$



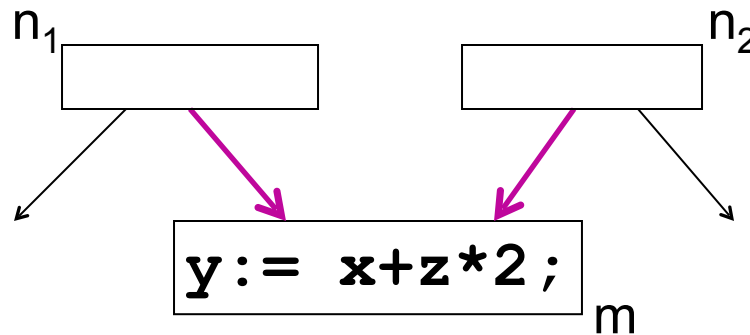
$$\text{in}[n] \supseteq \text{out}[n] - \text{def}[n]$$

## Computing Liveness

---

3. If a variable  $x$  is live-in in a node  $m$  then  $x$  is live-out for all the nodes  $n$  such that  $m \in \text{post}[n]$ .

This is clearly correct by definition.



$$\text{out}[n_1] \supseteq \cup \{ \text{in}[m] \mid m \in \text{post}[n_1] \}$$

$$\text{out}[n_2] \supseteq \cup \{ \text{in}[m] \mid m \in \text{post}[n_2] \}$$

# Dataflow Equations

---

The previous three rules of liveness analysis can be thus formalized by two equations for each node  $n$ :

1.  $in[n] = use[n] \cup (out[n] - def[n])$  (rules 1 and 2)

2.  $out[n] = \cup\{in[m] \mid m \in post[n]\}$  (rule 3)

## Correctness of Liveness

---

This definition of liveness analysis  $in[n]$  and  $out[n]$  is **correct**:  
If  $x$  is concretely live-in (live-out) in some node  $n$  then the static analysis will detect that  $x \in in[n]$  ( $x \in out[n]$ ):

$$in[n] \supseteq \text{live-in}[n]$$
$$out[n] \supseteq \text{live-out}[n]$$

In other terms, no actually live variable is neglected by liveness analysis.

# Correctness in Dragon Book

---

## Why the Available-Expressions Algorithm Works

We need to explain why starting all OUT's except that for the entry block with  $U$ , the set of all expressions, leads to a conservative solution to the data-flow equations; that is, all expressions found to be available really *are* available. First, because intersection is the meet operation in this data-flow schema, any reason that an expression  $x + y$  is found not to be available at a point will propagate forward in the flow graph, along all possible paths, until  $x + y$  is recomputed and becomes available again. Second, there are only two reasons  $x + y$  could be unavailable:

1.  $x + y$  is killed in block  $B$  because  $x$  or  $y$  is defined without a subsequent computation of  $x + y$ . In this case, the first time we apply the transfer function  $f_B$ ,  $x + y$  will be removed from  $\text{OUT}[B]$ .
2.  $x + y$  is never computed along some path. Since  $x + y$  is never in  $\text{OUT}[\text{ENTRY}]$ , and it is never generated along the path in question, we can show by induction on the length of the path that  $x + y$  is eventually removed from IN's and OUT's along that path.

Thus, after changes subside, the solution provided by the iterative algorithm of Fig. 9.20 will include only truly available expressions.

# Computing Liveness

---

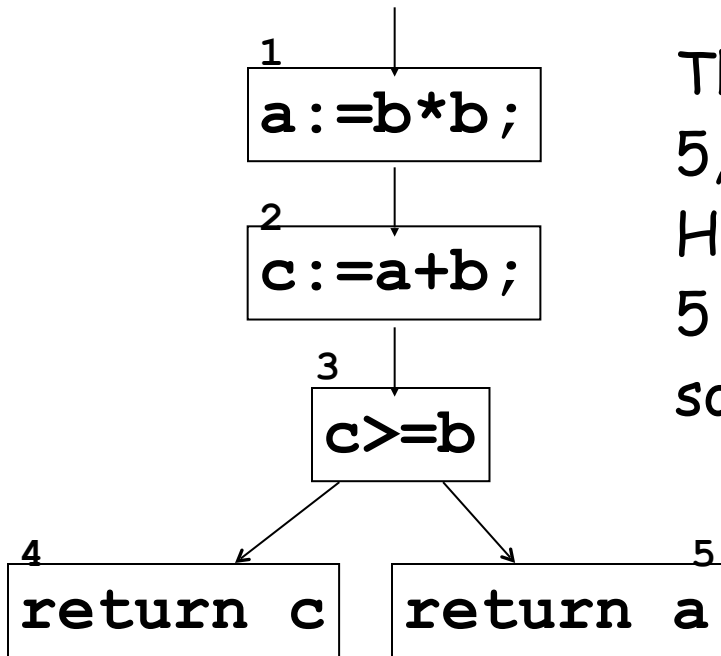
Liveness analysis is **approximate**:

it assumes that each path of the CFG is a **feasible path**  
while this hypothesis is obviously not true

## Computing Liveness

---

Liveness analysis is **approximate**: it assumes that each path of the CFG actually is a **feasible path** while this hypothesis is obviously not true.



The analysis determines that `a` is live-in in 5, and therefore `a` is live-out in 3.

However, no real execution path from 3 to 5 exists (because `b + b * b < b` is always false) so that `a` is not really live when exiting 3!



How can we compute a solution to 1 and 2?

1.  $in[n] = use[n] \cup (out[n] - def[n])$

2.  $out[n] = \cup \{in[m] \mid m \in post[n]\}$

Correctness tells us that  $in[n] \supseteq live-in[n]$  and  $out[n] \supseteq live-out[n]$

But we need a way to compute Live variable analysis

## How can we compute a solution to 1 and 2?

1.  $in[n] = use[n] \cup (out[n] - def[n])$
2.  $out[n] = \cup \{in[m] \mid m \in post[n]\}$

We need to compute a least fix point

- but how can we be sure that such fix-points exist?

It depends on the domain and on the function!

## The fix point in this case

---

- Let **Vars** be the finite set of variables that occur in the program P to analyze. Let N be the number of nodes of the CFG of P.

Thus, the map Live:

$$(\mathcal{P}(\mathbf{Vars}) \times \mathcal{P}(\mathbf{Vars}))^N \rightarrow (\mathcal{P}(\mathbf{Vars}) \times \mathcal{P}(\mathbf{Vars}))^N \text{ defined by}$$

$$\text{Live}(\langle in_1, out_1, \dots, in_N, out_N \rangle) =$$

$$\langle \text{use}[1] \cup (out_1 - \text{def}[1]), \bigcup_{m \in \text{post}[1]} in_m, \dots, \text{use}[N] \cup (out_N - \text{def}[N]), \bigcup_{m \in \text{post}[N]} in_m \rangle$$

is a monotonic (and therefore continuous) function on the finite lattice

$$\langle (\mathcal{P}(\mathbf{Vars}) \times \mathcal{P}(\mathbf{Vars}))^N, \subseteq^{2N} \rangle \text{ and therefore Live has}$$

a **least fixpoint**

# The fix point THEORY

# POSET (Partially ordered set, PO)

---

$$(P, \sqsubseteq) \quad \sqsubseteq \subseteq P \times P$$

reflexive  $\forall p \in P. \quad p \sqsubseteq p$

antisymmetry  $\forall p, q \in P. \quad p \sqsubseteq q \wedge q \sqsubseteq p \Rightarrow p = q$

transitive  $\forall p, q, r \in P. \quad p \sqsubseteq q \wedge q \sqsubseteq r \Rightarrow p \sqsubseteq r$

$p \sqsubseteq q$  means that p is less than (or equal to) q

$p \sqsubset q$  means  $p \sqsubseteq q \wedge p \neq q$

# Total Order

---

A PO  $(P, \sqsubseteq)$  is **total** iff

$$\forall p, q \in P. p \sqsubseteq q \vee q \sqsubseteq p$$

A PO where every two elements are **comparable**

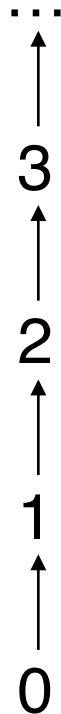
# Examples

---

$(\mathbb{N}, \leq)$

PO?  
Yes

Total?  
Yes



Hasse diagram notation  
(omit: reflexive arcs,  
transitive arcs)

# Examples

---

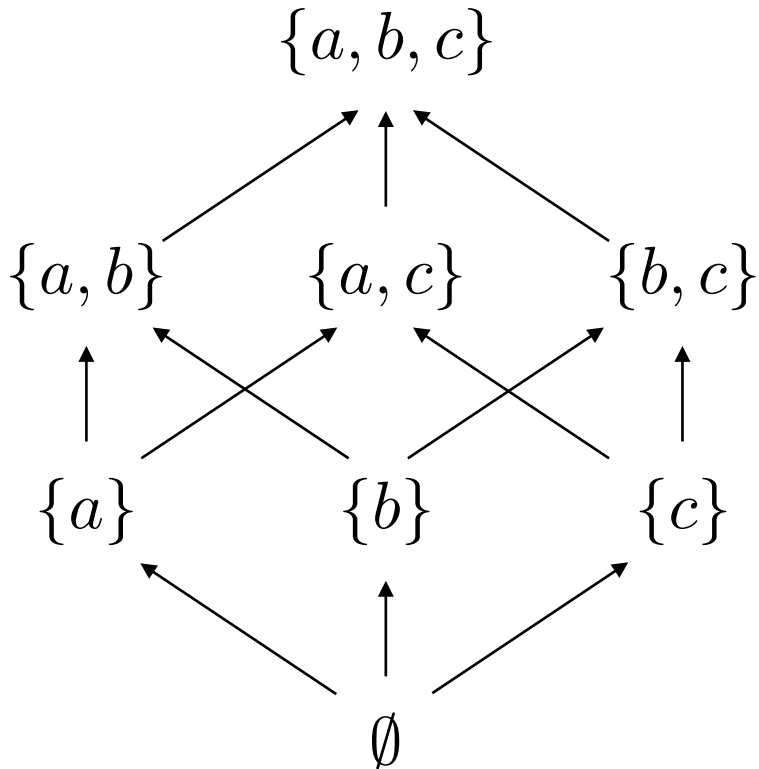
$(\wp(S), \subseteq)$

PO?

Total?

Yes

$|S| < 2$





# Examples

---

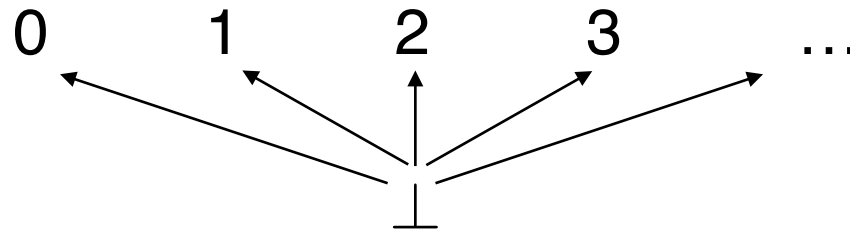
	PO?	Total?				
$(\mathbb{N}, =)$	Yes	No				
	0	1	2	3	...	

# Examples

---

$(\mathbb{N} \cup \{\perp\}, \{(\perp, n) \mid n \in \mathbb{N}\})$

PO?	Total?
Yes	No



## PO with bottom

---

A PO  $(P, \sqsubseteq)$  that has a least element  $e$ , i.e.,

$$\forall p \in P. e \sqsubseteq p$$

$e$  is often indicates as  $\perp$

# Examples

---

$(\mathbb{N}, \leq)$

PO with  $\perp$  ?

Yes



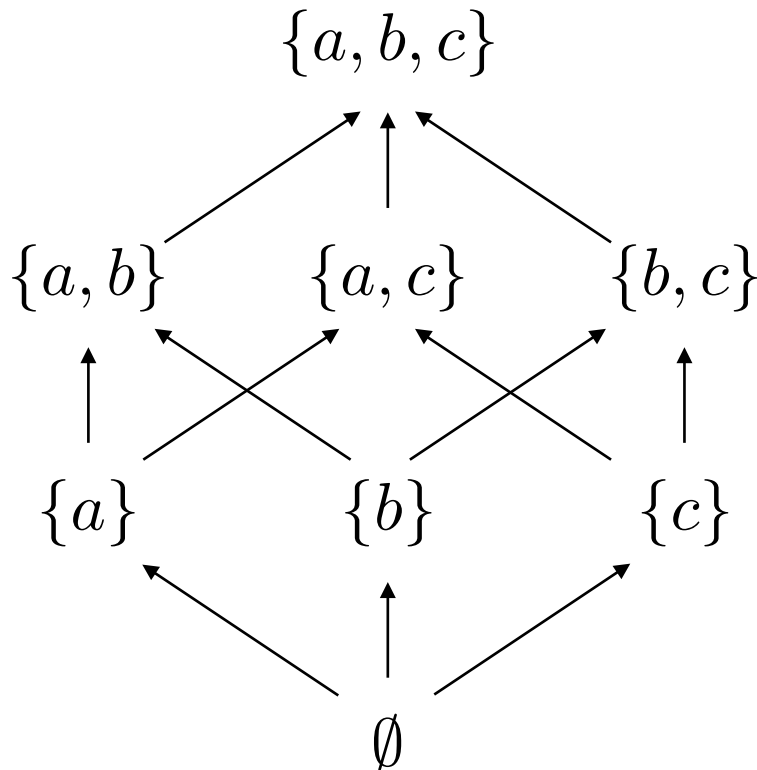
# Examples

---

$(\wp(S), \subseteq)$

PO with  $\perp$  ?

Yes



# Examples

---

$(\mathbb{N}, =)$

PO with  $\perp$  ?

No

0

1

2

3

...

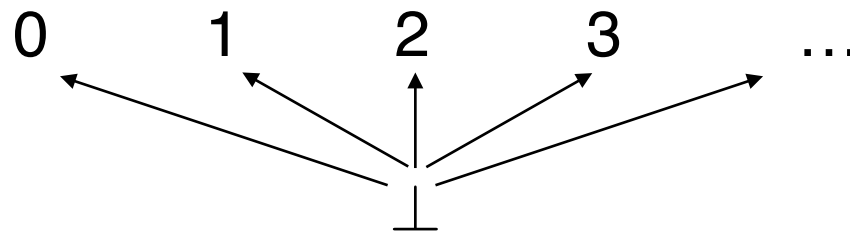
# Examples

---

$(\mathbb{N} \cup \{\perp\}, \{(\perp, n) \mid n \in \mathbb{N}\})$

PO with  $\perp$  ?

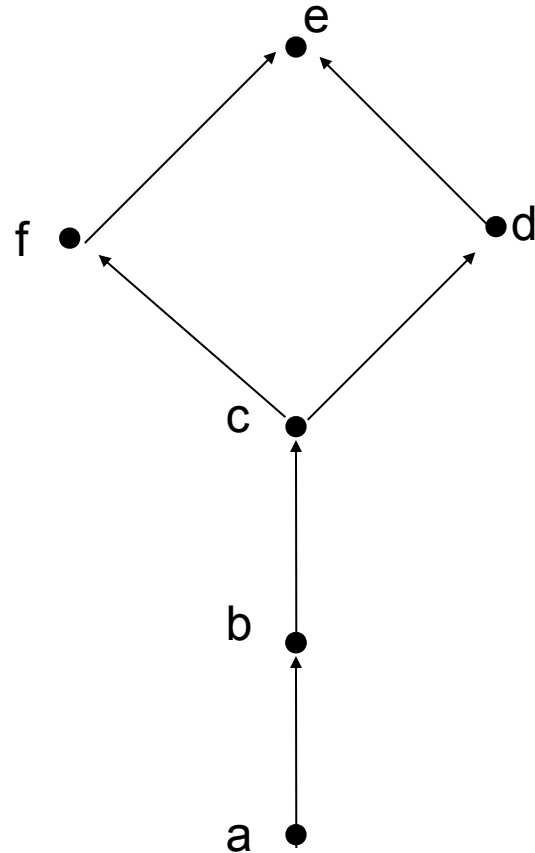
Yes



# Lattice

---

A special structure arises when every pair of elements in a poset has a least upper bound (lub) and a greatest lower (glb)

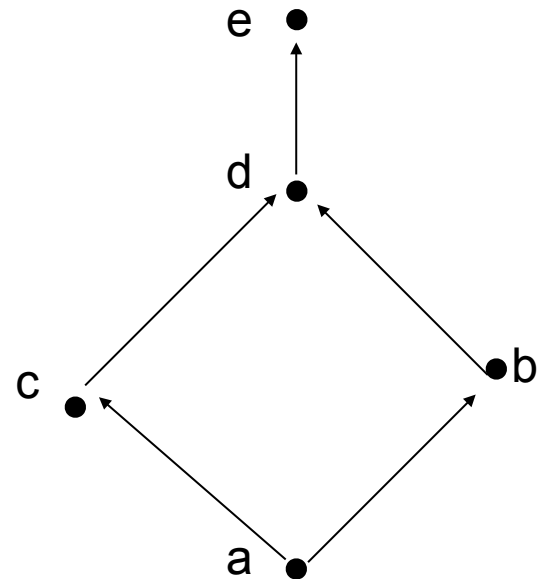




# Lattice Definition

---

A lattice is a PO in which every pair of elements has both a **lub** and a **glb**



# Examples

---

$(\mathbb{N}, \leq)$

Lattice ?

Yes



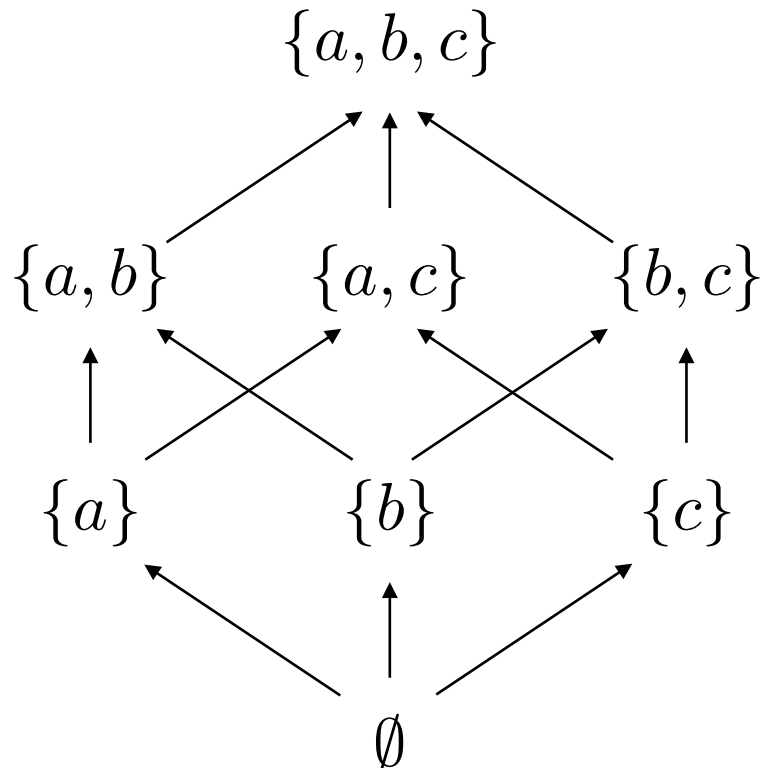
# Examples

---

$(\wp(S), \subseteq)$

Lattice ?

Yes



# Examples

---

$(\mathbb{N}, =)$

Lattice ?

No

0      1      2      3      ...

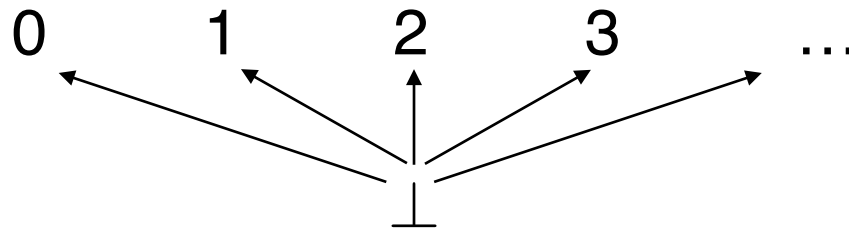
# Examples

---

$(\mathbb{N} \cup \{\perp\}, \{(\perp, n) \mid n \in \mathbb{N}\})$

Lattice ?

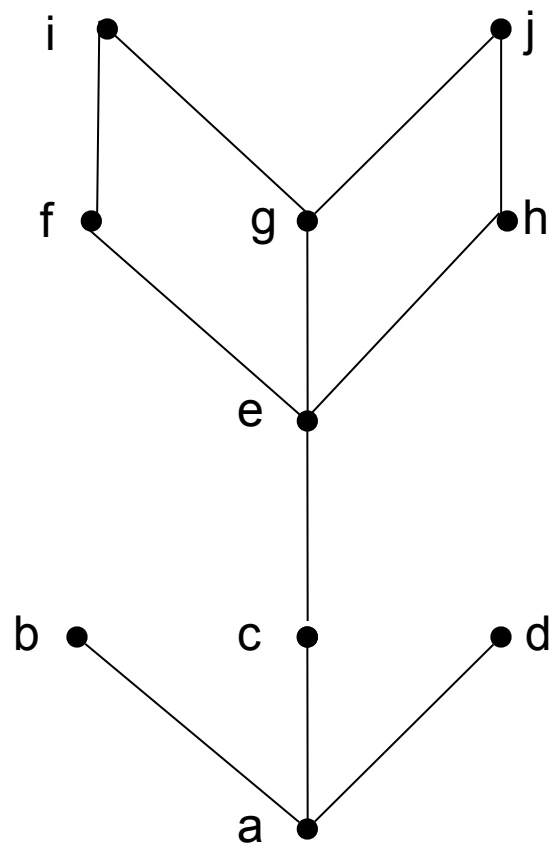
No



# Example

---

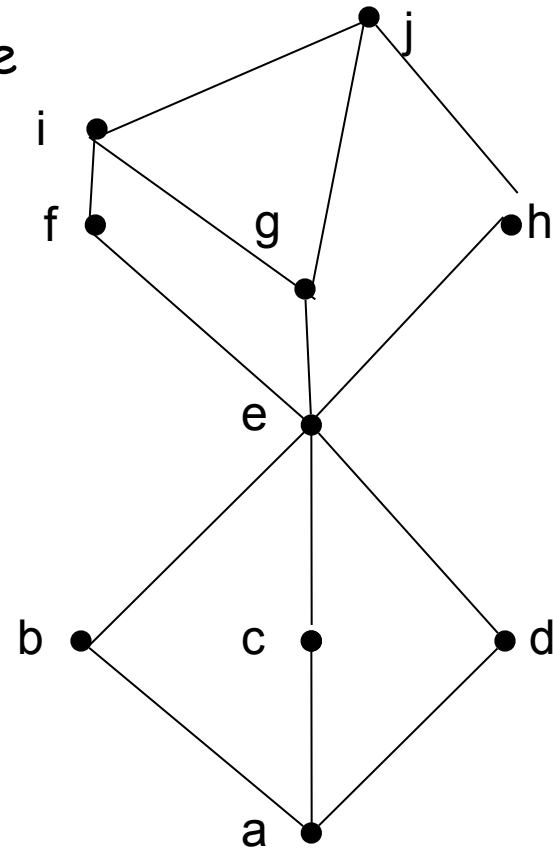
- Is this a lattice?
- **No, because the pair  $\{b,c\}$  does not have a least upper bound**



# Example

---

- What if we modified it as shown here



- Yes, because for any pair, there is a lub & a glb

# Ascending chains

---

- A sequence  $(l_n)_{n \in \mathbb{N}}$  of elements in a partial order  $L$  is an **ascending chain** if

$$n \leq m \Rightarrow l_n \leq l_m$$

- A sequence  $(l_n)_{n \in \mathbb{N}}$  **converges** if and only if

$$\exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : n_0 \leq n \Rightarrow l_{n_0} = l_n$$

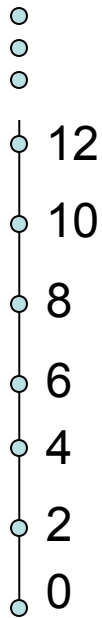
- A partial order  $(L, \leq)$  satisfies the **ascending chain condition (ACC)** iff each ascending chains converges.



# Example

---

- The PO  $(N, \sqsubseteq)$  does not satisfy the ascending chain condition,

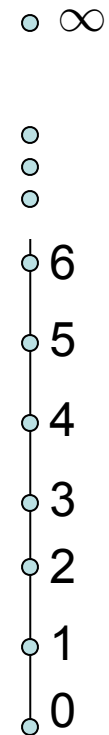


# Example

---

$$(N \cup \{\infty\}, \sqsubseteq)$$

satisfies the ACC condition



# Complete Partial Order

---

A poset  $(P, \sqsubseteq)$  is called a **complete partial order (CPO)** if and only if any of its chains has a lub

If  $(P, \sqsubseteq)$  has a bottom element and any of its chains has a lub then  $(P, \sqsubseteq)$  is called a **complete partial order (CPO) with bottom**

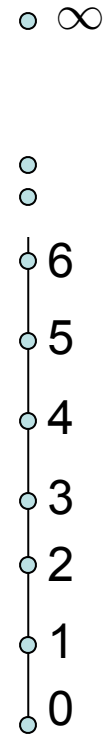
- $(\mathbb{N}, \leq)$  has bottom 0 but is not complete:  
the chain  $0 \leq 1 \leq 2 \leq \dots \leq n \leq \dots$  has no upper bound in  $\mathbb{N}$ .
- $(\mathbb{N}, \geq)$  is a CPO but has no bottom.

# Example

---

$$(N \cup \{\infty\}, \sqsubseteq)$$

is CPO with bottom



## Some complete PO

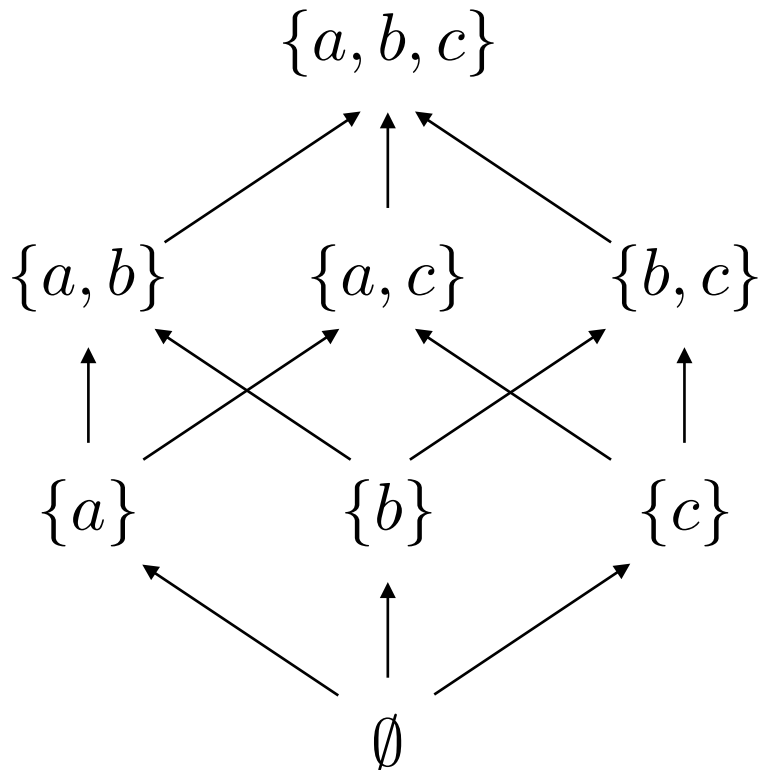
If  $(P, \sqsubseteq)$  has only finite chains it is complete

If  $(P, \sqsubseteq)$  is finite then it is complete

# CPO with bottom

---

$(\wp(S), \subseteq)$



# CPO without bottom

---

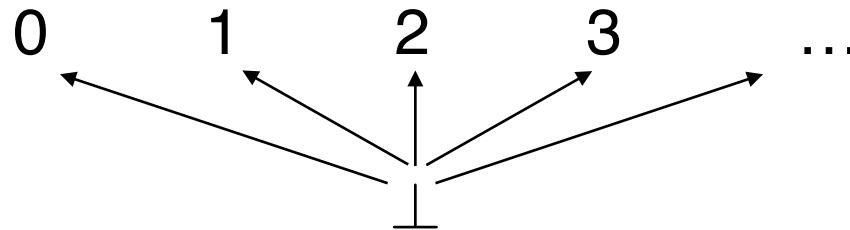
$(\mathbb{N}, =)$

0      1      2      3      ...

## CPO with bottom

---

$(\mathbb{N} \cup \{\perp\}, \{(\perp, n) \mid n \in \mathbb{N}\})$





## Equivalent Definitions of Complete Lattices

A lattice  $L$  is called a **complete** lattice if every subset  $S$  of  $L$  admits a lub in  $L$ .



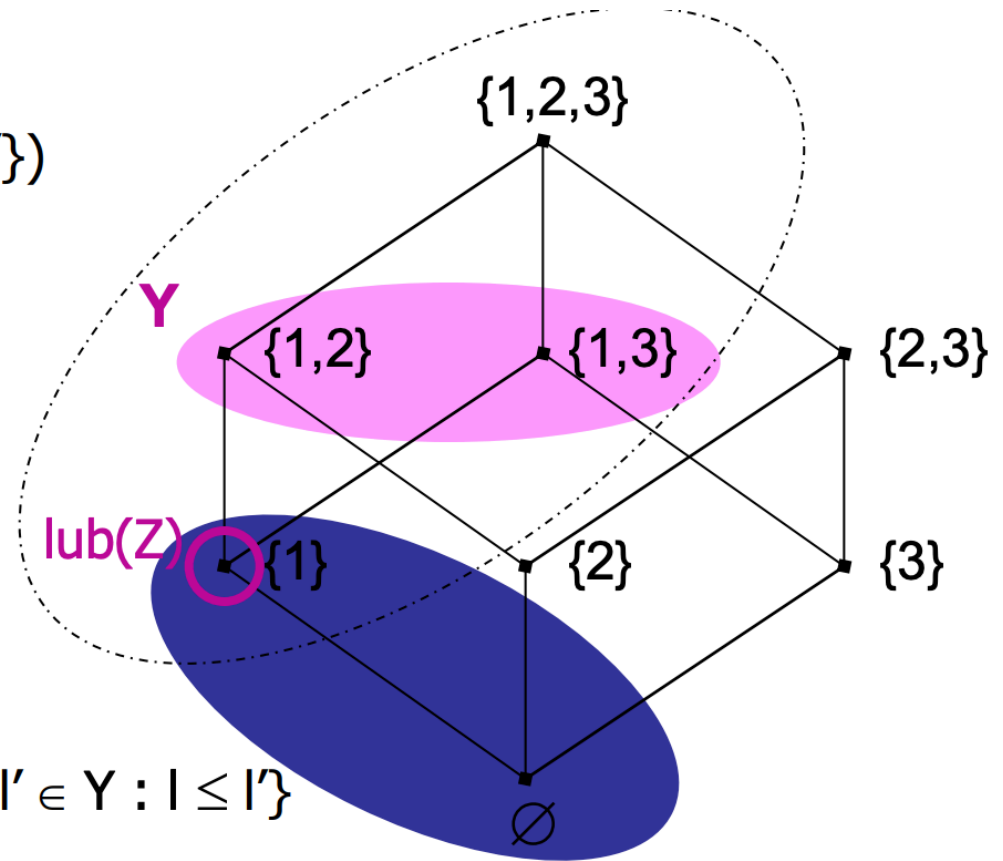
A lattice  $L$  is called a **complete** lattice if every subset  $S$  of  $L$  admits a glb in  $L$ .



A lattice  $L$  is called a **complete** lattice if every subset  $S$  of  $L$  admits a glb and a lub in  $L$ .

The idea: assume we just have  $\text{lub}$

$$\text{glb}(Y) = \text{lub}(\{l \in L \mid \forall l' \in Y : l \leq l'\})$$



$$Z = \{l \in L \mid \forall l' \in Y : l \leq l'\}$$

# Example

---

☞  $L = \mathbb{N} \cup \{\infty\}$

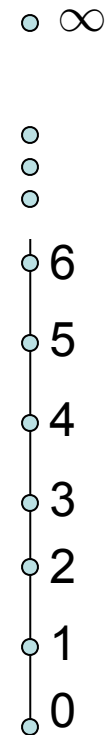
☞ total order on  $\mathbb{N} \cup \{\infty\}$

☞ lub = max

☞ glb = min

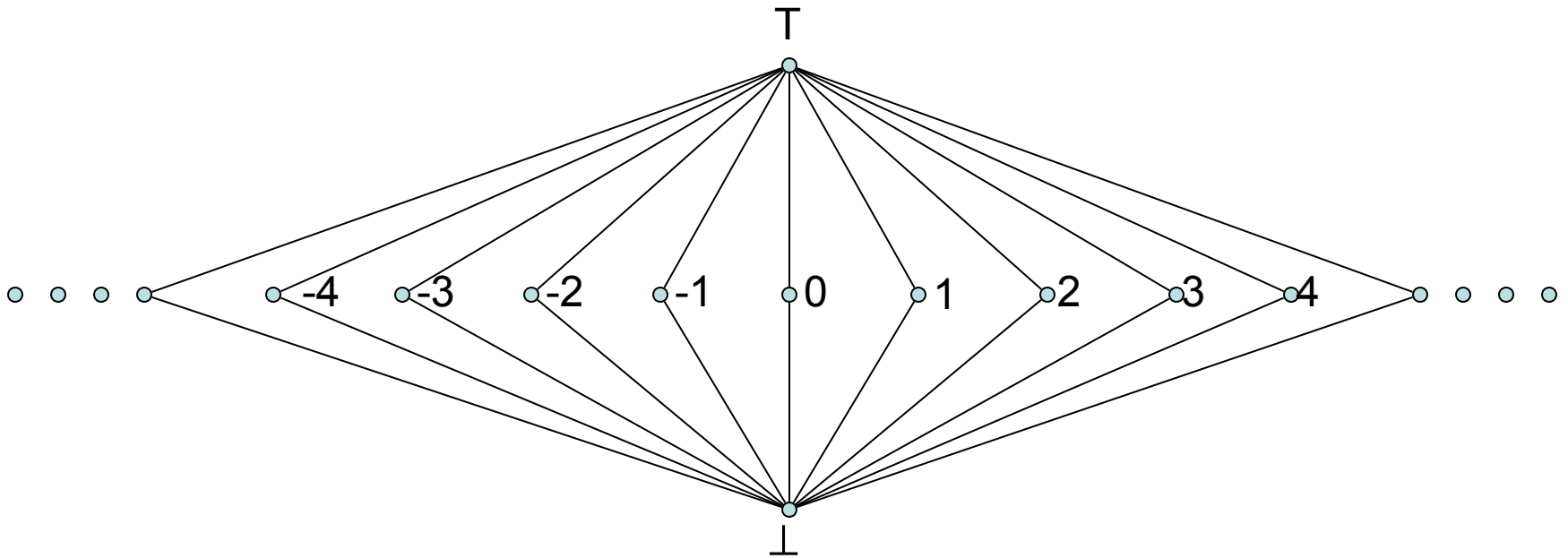
$\leq$

This is a complete lattice



# Example of complete lattice

---



☞  $L = \mathbb{N} \cup \{T, \perp\}$

☞  $\forall n \in \mathbb{N} : \perp < n < T$

☞ This is a complete lattice, with infinite elements

# Lattices and ACC

---

- If  $L$  is a lattice with a bottom element and ACC, then  $L$  is a complete lattice
- If  $L$  is a finite lattice then it satisfies the ACC and therefore it also is complete.

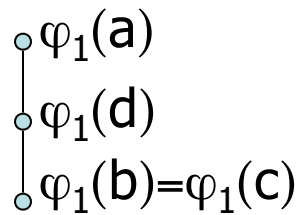
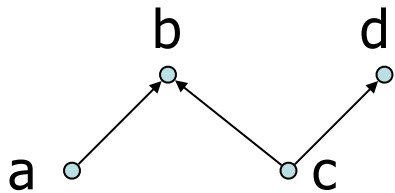
# Monotone functions on partial orders

---

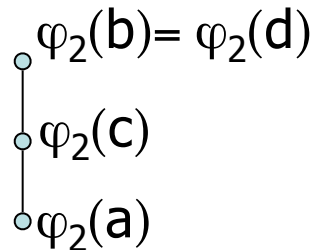
Let  $(P, \leq_P)$  and  $(Q, \leq_Q)$  be PO.

A function  $\varphi$  from  $P$  to  $Q$  is **monotone** iff

$$p_1 \leq_P p_2 \Rightarrow \varphi(p_1) \leq_Q \varphi(p_2)$$



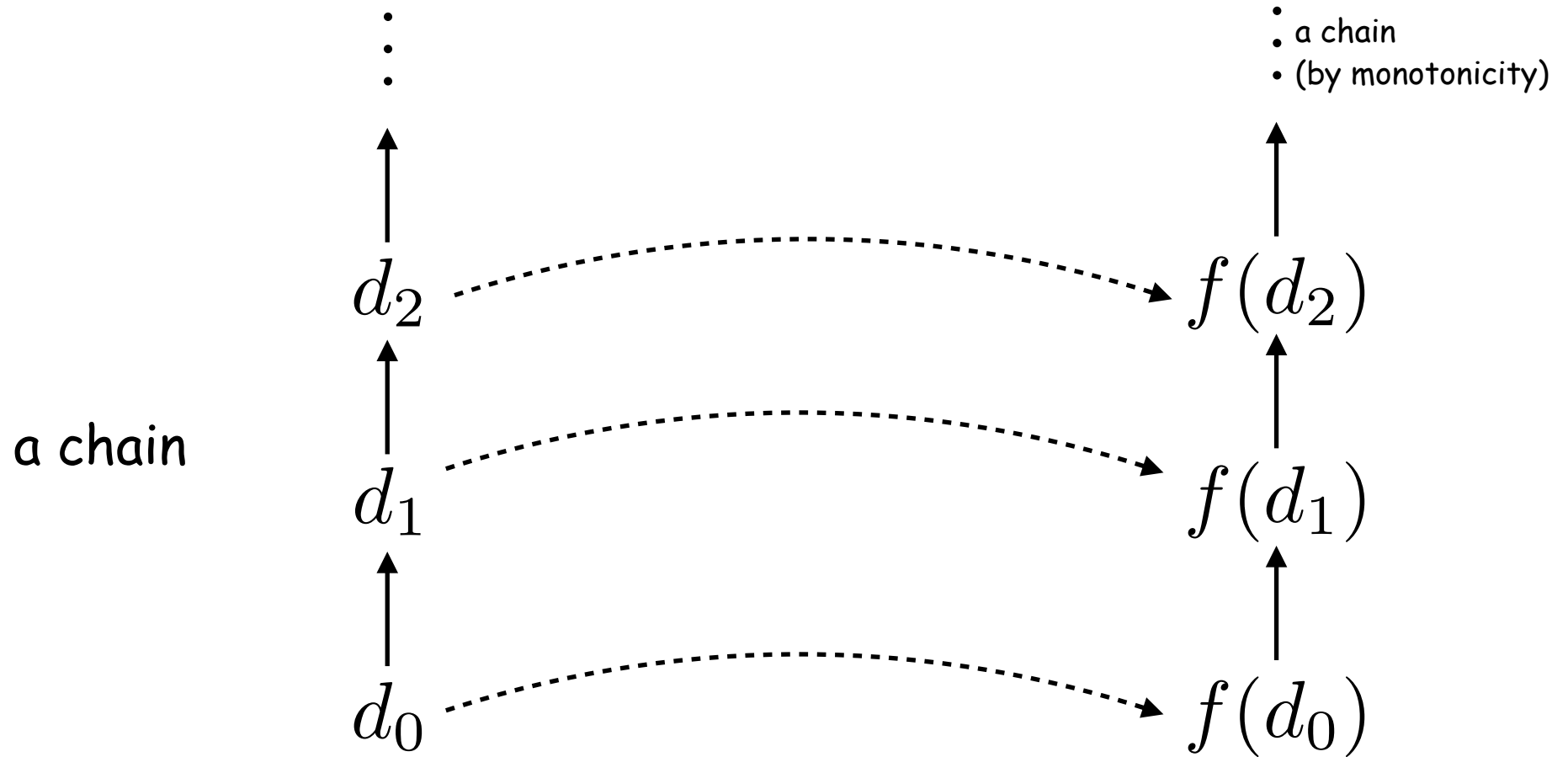
- $\varphi_1$  is not monotone



- $\varphi_2$  is monotone

# Monotonicity

---



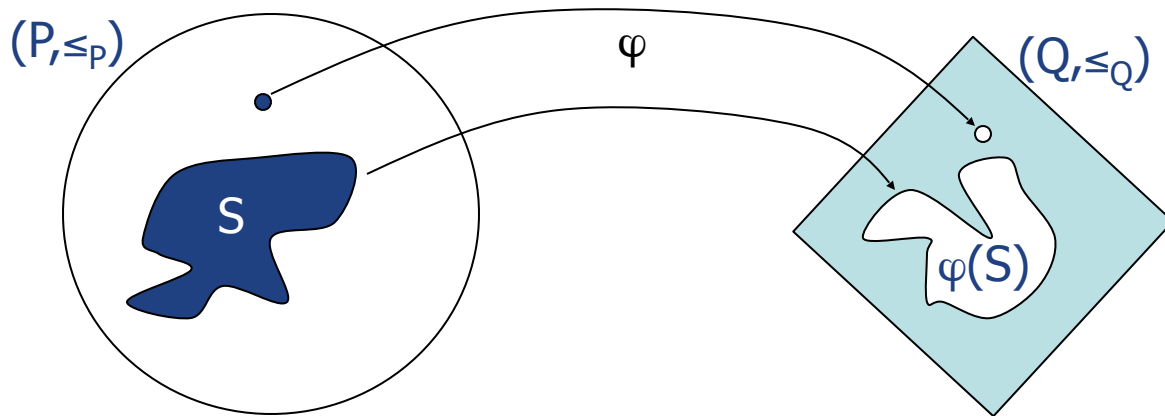
# Continuity

---

- Given two partial orders  $(P, \leq_P)$  and  $(Q, \leq_Q)$ , a function  $\varphi$  from  $P$  to  $Q$  is **continuous** if for every **chain**  $S$  in  $P$

$$\varphi(\text{lub}(S)) = \text{lub}\{ \varphi(x) \mid x \in S \}$$

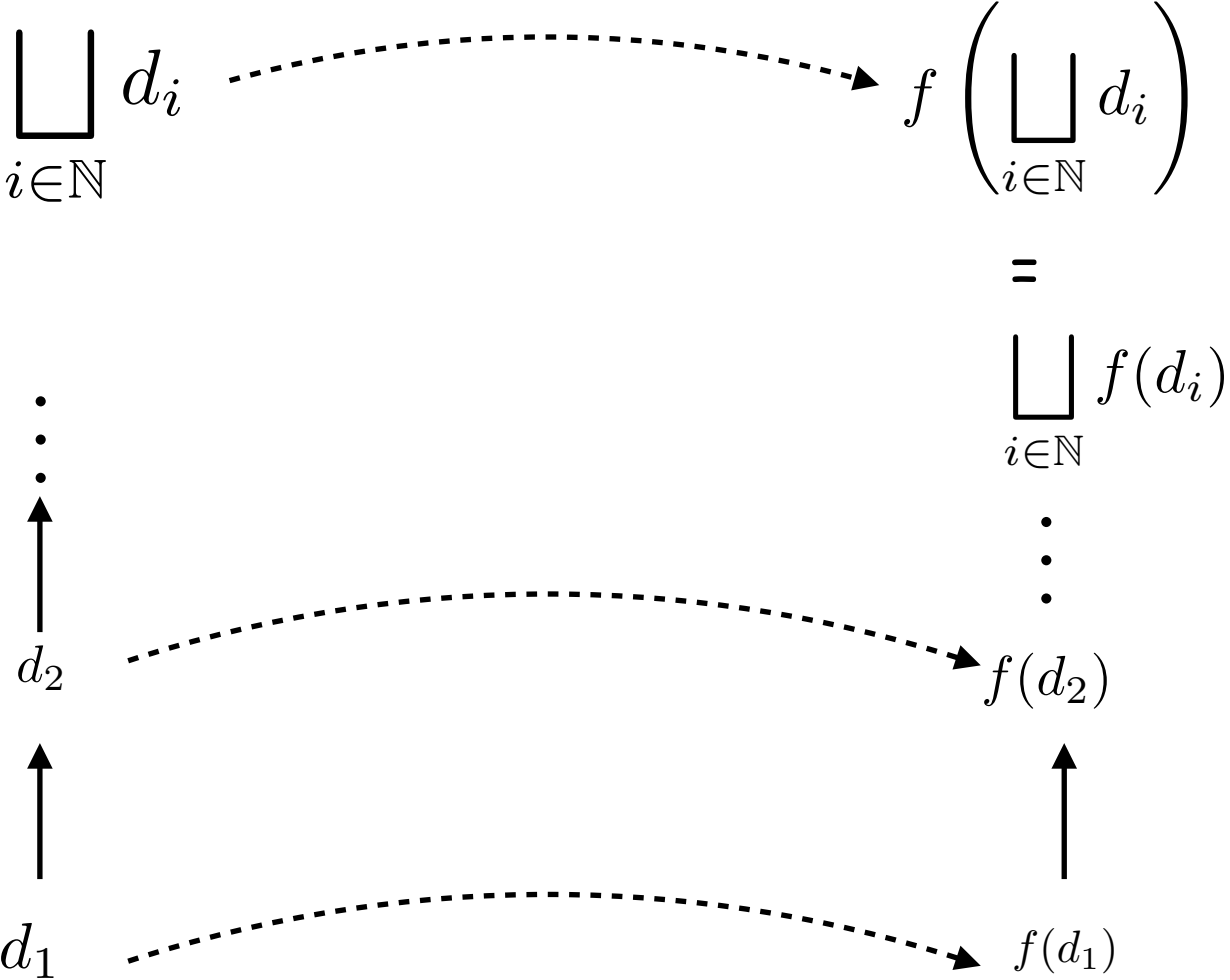
- if  $f$  is monotone on an ACC lattice then  $f$  is **continuous**





# Continuous function

---

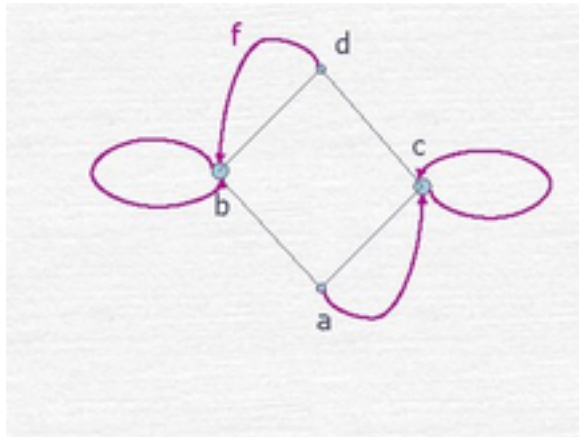


# Fixpoints

---

- Consider a monotone function  $f: (P, \leq_p) \rightarrow (P, \leq_p)$  on a partial order  $P$
- An element  $x$  of  $P$  is a **fixpoint** of  $f$  if  $f(x)=x$
- The set of fixpoints of  $f$  is a subset of  $P$  called  $\text{Fix}(f)$ :

$$\text{Fix}(f) = \{ x \in P \mid f(x)=x \}$$



$$\text{Fix}(f) = \{ b, c \}$$

# Least and greatest Fixpoints

---

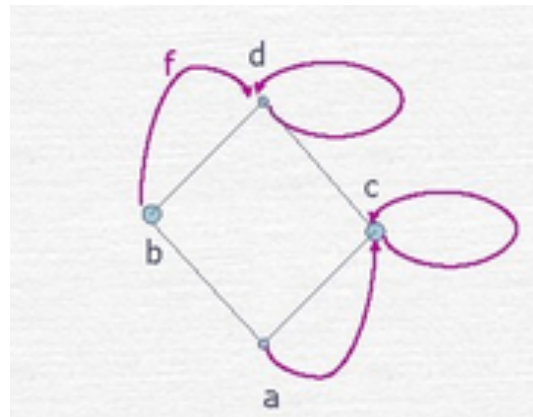
Consider a monotone function  $f: (P, \leq_p) \rightarrow (P, \leq_p)$  on a partial order  $P$

- a point  $x \in \text{Fix}(f)$  is a minimal fix point if for all  $y \in \text{Fix}(f)$ ,  $x \leq y$ .

If the minimal fix point exists and it is unique call it a **least fix point** and it is indicated as **lfp(f)**

- a point  $x \in \text{Fix}(f)$  is a maximal fix point if for all  $y \in \text{Fix}(f)$ ,  $y \leq x$ .

If the maximal fix point exists and it is unique call it a **greatest fix point** and it is indicated as **gfp(f)**



$$\begin{aligned} \text{lfp}(f) &= \{c\} \\ \text{gfp}(f) &= \{d\} \end{aligned}$$

# Example of Fixpoints

---

$$I \subseteq \mathbb{N}$$

$$f(I) = I \cap \{1, 2\} \quad \text{lfp? gfp?}$$

$$f(I) = \mathbb{N} \setminus I \quad \text{lfp? gfp?}$$

$$f(I) = I \cup \{1, 2\} \quad \text{lfp? gfp?}$$

# Fixpoint on Complete Lattices

---

- Consider a **monotone** function  $f:L \rightarrow L$  on a **complete lattice**  $L$ .

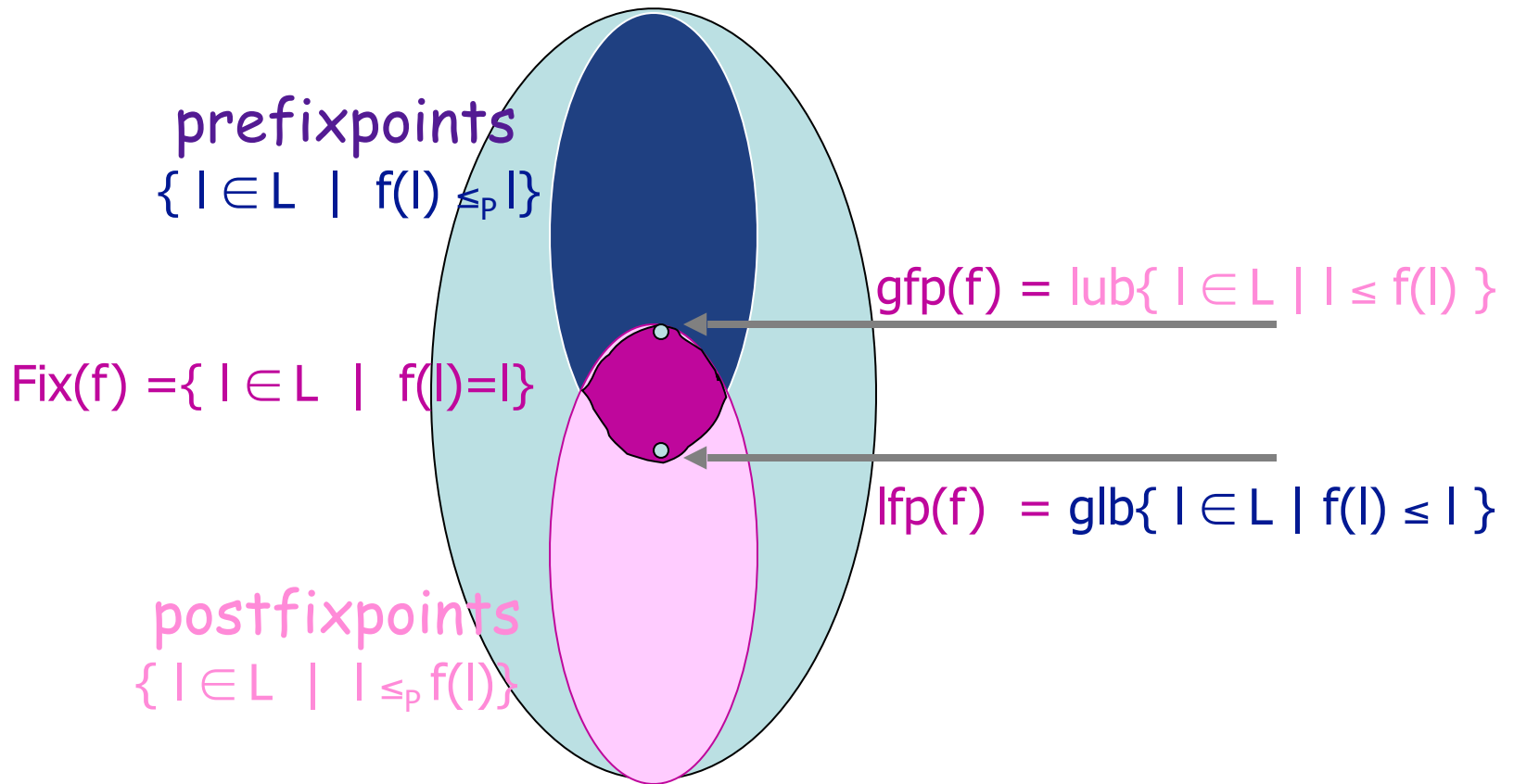
- **Tarski Theorem:**

Let  $L$  be a complete lattice. If  $f:L \rightarrow L$  is **monotone** then

$$\text{lfp}(f) = \text{glb}\{ l \in L \mid f(l) \leq l \}$$

$$\text{gfp}(f) = \text{lub}\{ l \in L \mid l \leq f(l) \}$$

# Fixpoints on Complete Lattices



## Function monotone on complete lattice

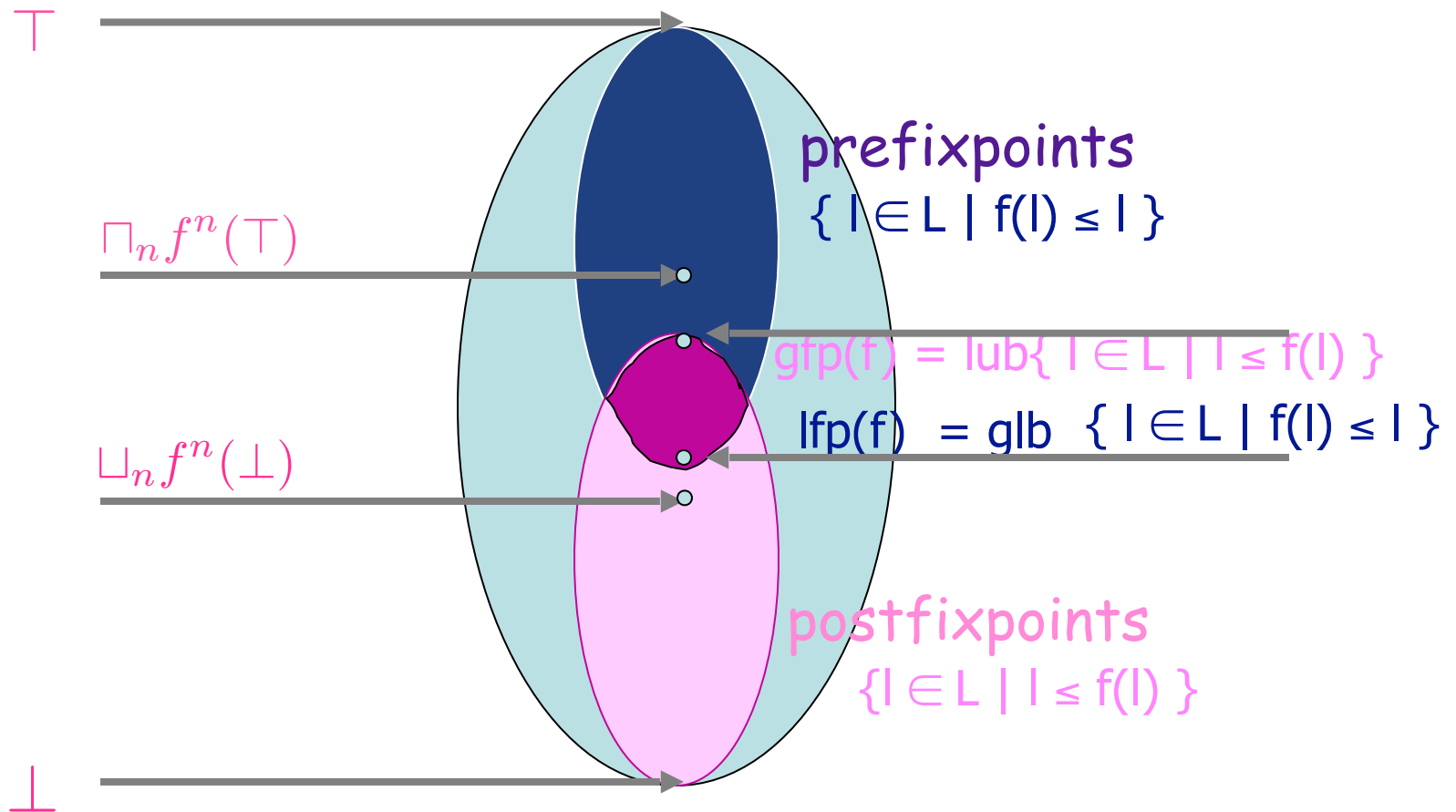
---

- Let  $f$  be a **monotone** function:  $(P, \leq_p) \rightarrow (P, \leq_p)$  on a **complete lattice**  $P$ .

$$\text{Let } \alpha = \bigsqcup_{n \geq 0} f^n(\perp)$$

- If  $\alpha \in \text{Fix}(f)$  then  $\alpha = \text{lfp}(f)$

# Fixpoints on Complete Lattices





# Kleene Theorem

---

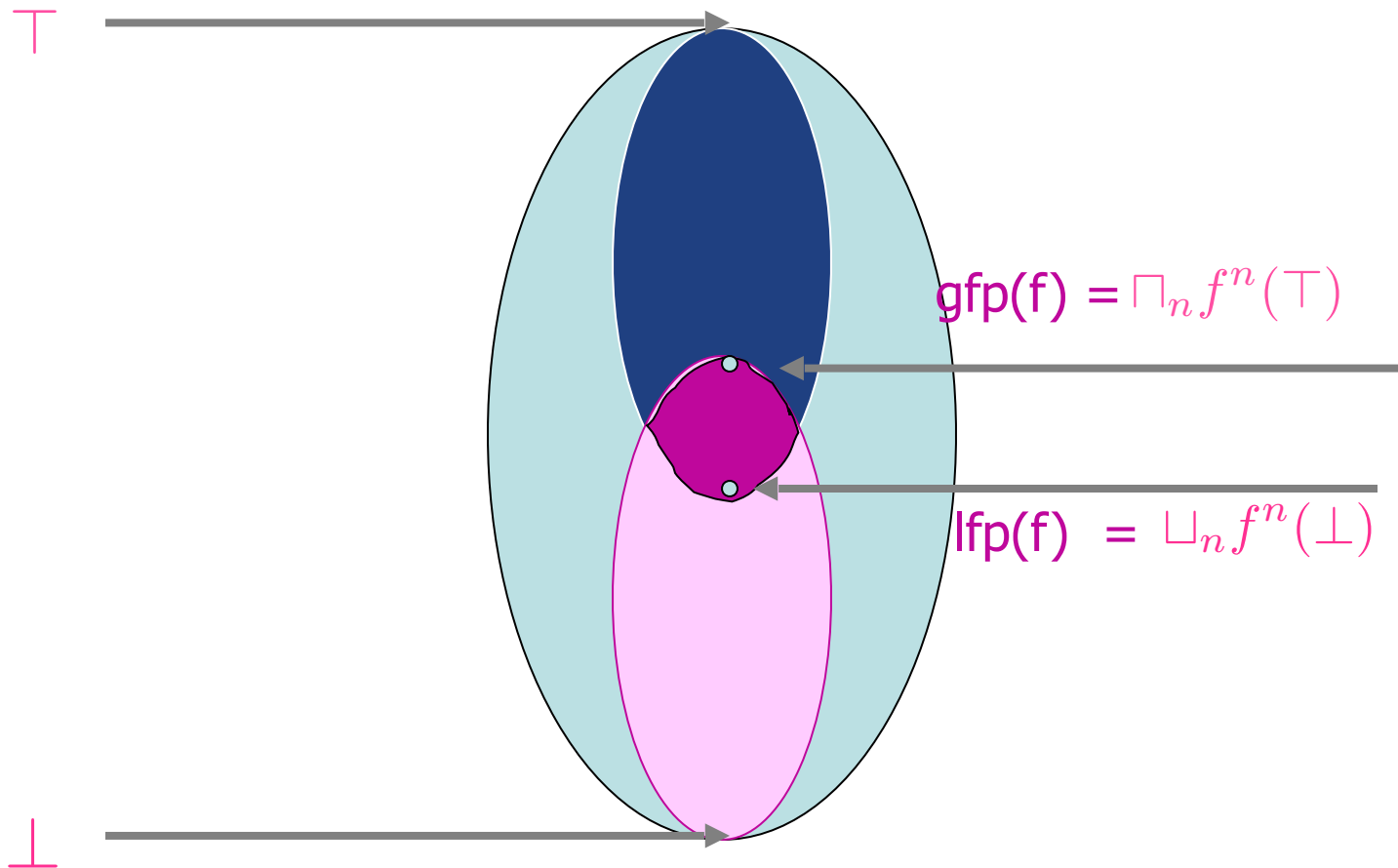
## Kleene Theorem

If  $f$  is **continuous** on a complete CPO with bottom then the least fixpoint of  $f$  **exists** and it is equal to  $\alpha$

recall that

- if  $f$  is monotone on an ACC lattice then  $f$  is **continuous**

# Fixpoints on CPO with bottom when $f$ is continuous



## Back to our example

---

1.  $in[n] = use[n] \cup (out[n] - def[n])$
2.  $out[n] = \cup \{in[m] \mid m \in post[n]\}$

We need to compute a fix point

- but how can we be sure that such fix-points exist?

We can apply the fix point theory results !

We need to check that we have

- a) a continuous function on
- b) a CPO with bottom

Kleene's Theorem

# Point b first

**Vars** is the (finite) set of variables occurring in the program P.

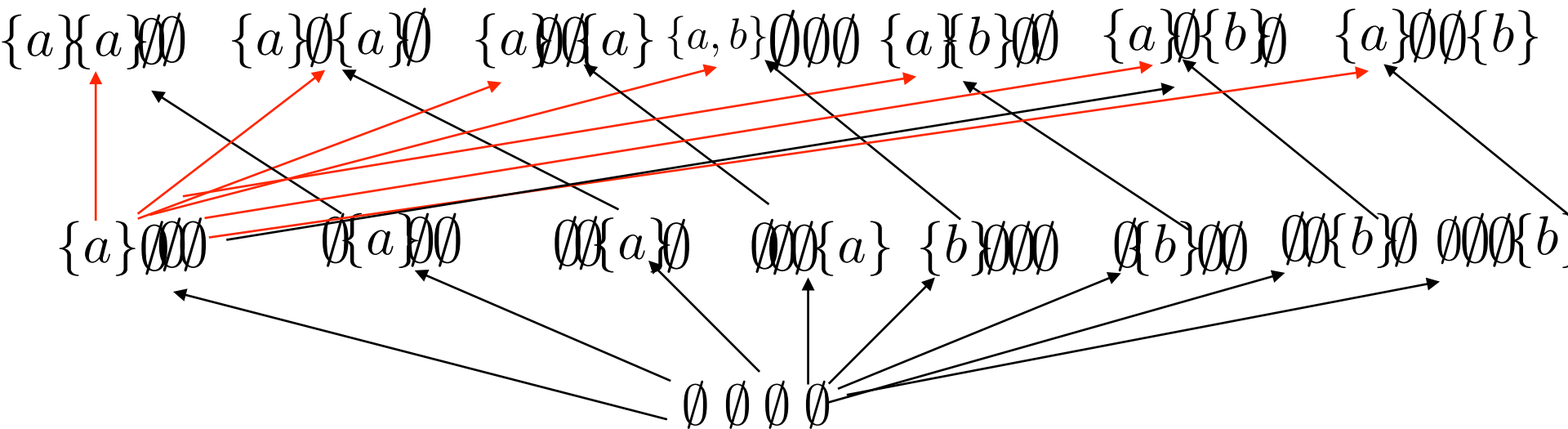
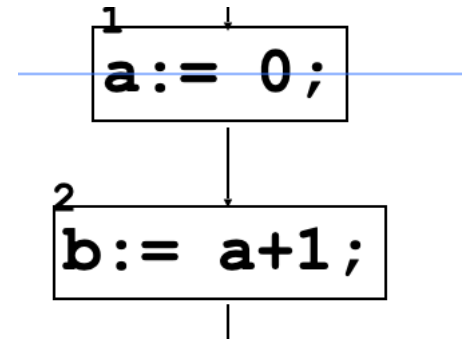
Let **N** be the number of nodes of the CFG of P.

$\langle (\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}))^N, \subseteq^{2N} \rangle$  is a finite domain.

- Example Vars={a,b} e N=2

$\{a, b\} \{a, b\} \{a, b\} \{a, b\}$

...



## Point b

---

$$\langle (\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}))^{\mathbb{N}}, \subseteq^{2N} \rangle$$

CPO with bottom?

Yes! Because it is finite

## Point a

---

The map Live:

$( (\text{Vars}) \times \mathcal{P}(\text{Vars}) )^N \rightarrow ( \mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}) )^N$  defined by

$\text{Live}(\langle \text{in}_1, \text{out}_1, \dots, \text{in}_N, \text{out}_N \rangle) =$

$\langle \text{use}[1] \cup (\text{out}_1 - \text{def}[1]), \bigcup \text{in}_m, \dots, \text{use}[N] \cup (\text{out}_N - \text{def}[N]), \bigcup \text{in}_m \rangle$

## Point a

---

The map Live:

$((\text{Vars}) \times \mathcal{P}(\text{Vars}))^N \rightarrow (\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}))^N$  defined by

$\text{Live}(\langle in_1, out_1, \dots, in_N, out_N \rangle) =$

$\langle \text{use}[1] \cup (\text{out}_1 - \text{def}[1]), \bigcup_{m \in \text{post}[1]} in_m, \dots, \text{use}[N] \cup (\text{out}_N - \text{def}[N]), \bigcup_{m \in \text{post}[N]} in_m \rangle$

is continuous?

Yes! because it is monotone on a finite domain

## Why a **least** fixpoint

---

- Live is a **possible** analysis,

$$\mathit{in}[n] \supseteq \mathit{live-in}[n] \text{ and } \mathit{out}[n] \supseteq \mathit{live-out}[n]$$

i.e., if a variable  $x$  will be really live in a node  $n$  during some program execution then  $x$  belongs to  $\mathit{in}[n]$  of all the fixpoints of the function Live

All fixpoints of the equation system is an over-approximation of really live variables.

We want the least fixpoint ( more precise over approximations)



# Conservative Approximation

- How to interpret the output of this static analysis?
- Correctness tells us that:

$$\text{in}[n] \supseteq \text{live-in}[n] \text{ and } \text{out}[n] \supseteq \text{live-out}[n]$$

If the variable  $x$  will be really live in some node  $n$  during some program execution then  $x$  belongs to  $\text{in}[n]$  of all the fixpoints of the function  $\text{Live}$  (least fixpoint)

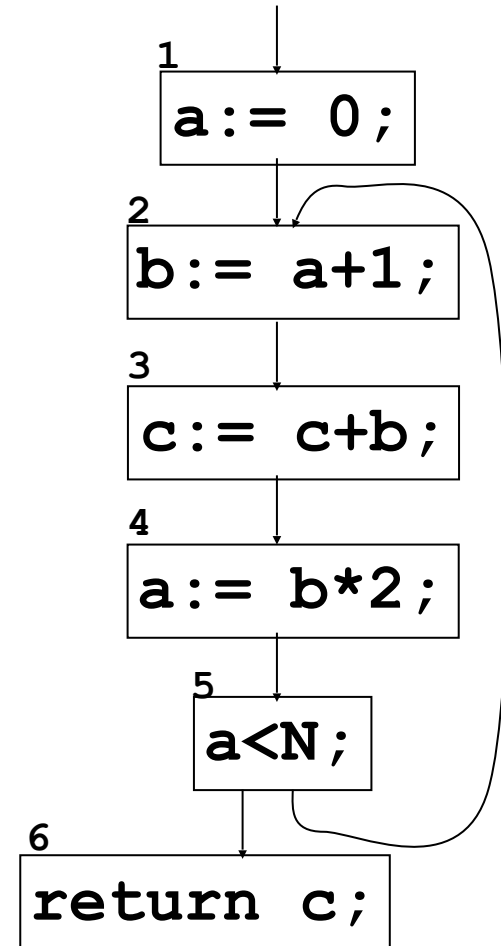
- The converse does not hold: the analysis can tell us that  $x$  is in the computed set  $\text{out}[n]$ , but this does not imply that  $x$  will be necessarily live in  $n$  during some program execution
- In liveness analysis “conservative approximation” means that the analysis may erroneously derive that a variable is live, while the analysis is not allowed to erroneously derive that a variable is “dead” (i.e., not live).
  - ★if  $x \in \text{in}[n]$  then  $x$  **could be live** at program point  $n$ .
  - ★if  $x \notin \text{in}[n]$  then  $x$  is **definitely** dead at program point  $n$ .

```

for all n
  in[n] := {} out[n] := {};
repeat
  for all n (1 to 6)
    in'[n] := in[n]; out'[n] := out[n];
    in[n] := use[n] U (out[n] - def[n]);
    out[n] := U { in[m] | m ∈ post[n] };
until (for all n: in'[n]=in[n] && out'[n]=out[n])

```

		Live <sup>1</sup>		Live <sup>2</sup>		Live <sup>3</sup>		
	use	def	in	out	in	out	in	out
1		a				a		a
2	a	b	a		a	b c	a c	b c
3	b c	c	b c		b c	b	b c	b
4	b	a	b		b	a	b	a
5	a		a	a	a	a c	a c	a c
6	c		c		c		c	

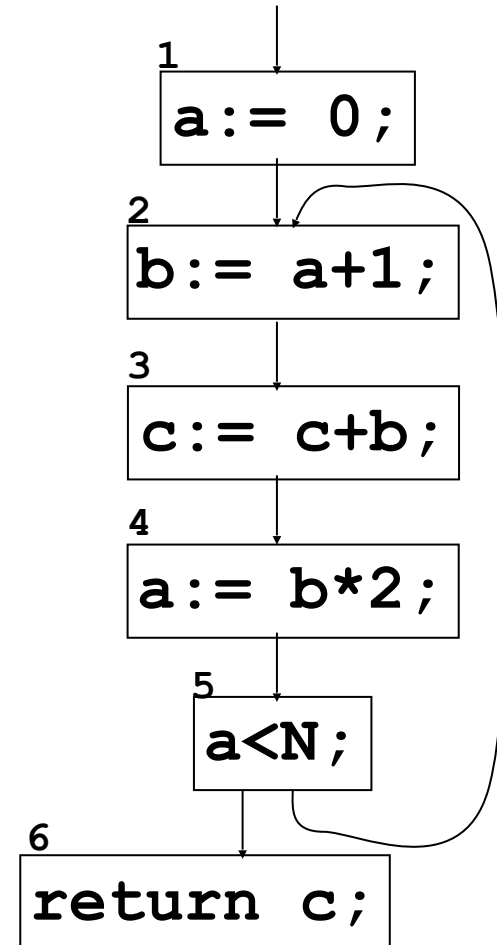


```

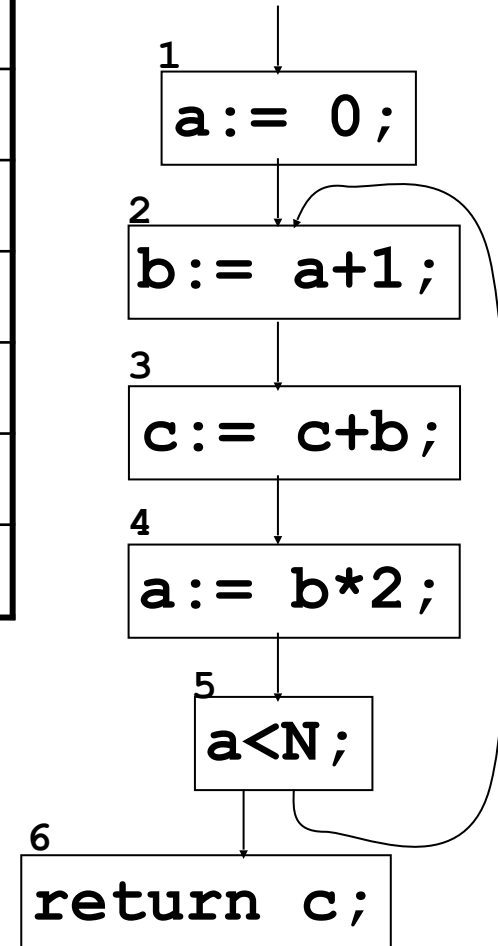
for all n
  in[n]:=?; out[n]:=?;
repeat
  for all n (1 to 6)
    in'[n]:=in[n]; out'[n]:=out[n];
    in[n]:= use[n] U (out[n] - def[n]);
    out[n]:= U { in[m] | m ∈ post[n]};
until (for all n: in'[n]=in[n] && out'[n]=out[n])

```

		Live <sup>3</sup>	Live <sup>4</sup>	Live <sup>5</sup>
	use def	in out	in out	in out
1	a	a	a c	c a c
2	a b	a c b c	a c b c	a c b c
3	b c c	b c b	b c b	b c b
4	b a	b a	b a c	b c a c
5	a	a c a c	a c a c	a c a c
6	c	c	c	c



		Live <sup>5</sup>		Live <sup>6</sup>		Live <sup>7</sup>		
	use	def	in	out	in	out	in	out
1		a	c	a c	c	a c	c	a c
2	a	b	a c	b c	a c	b c	a c	b c
3	b c	c	b c	b	b c	b c	b c	b c
4	b	a	b c	a c	b c	a c	b c	a c
5	a		a c	a c	a c	a c	a c	a c
6	c		c		c		c	



The algorithm thus gives the following output:

out[1]={a,c}, out[2]={b,c}, out[3]={b,c}, out[4]={a,c},  
out[5]={a,c}

In this case, the output of the analysis is precise

# Improvement

---

In this iterative computation, observe that we have to wait for the next iteration in order to exploit the new information computed for in and out on the nodes.

By a suitable reordering of the nodes and by first computing out[n] and then in[n], we are able to converge to the fixpoint in just 3 iteration steps.

```
for all n
  in[n]:=?; out[n]:=?;
repeat
  for all n (6 to 1)
    in'[n]:=in[n]; out'[n]:=out[n];
    out[n]:= U { in[m] | m ∈ post[n] };
    in[n]:= use[n] U (out[n] - def[n]);
until (for all n: in'[n]=in[n] && out'[n]=out[n])
```

```

for all n
  in[n]:=?; out[n]:=?;
repeat
  for all n (6 to 1)
    in'[n]:=in[n]; out'[n]:=out[n];
    out[n]:= U { in[m] | m ∈ post[n] };
    in[n]:= use[n] U (out[n] - def[n]);
until (for all n: in'[n]=in[n] && out'[n]=out[n])

```

		Live <sup>1</sup>	Live <sup>2</sup>	Live <sup>3</sup>
	use def	out in	out in	out in
<b>6</b>	c	c	c	c
<b>5</b>	a	c a c	a c a c	a c a c
<b>4</b>	b a	a c b c	a c b c	a c b c
<b>3</b>	b c c	b c b c	b c b c	b c b c
<b>2</b>	a b	b c a c	b c a c	b c a c
<b>1</b>	a	ac c	ac c	ac c

## Backward Analysis

---

As shown by the previous example, Live Variable Analysis is a “backward” analysis. This means that information propagates “backward” from terminal nodes to initial nodes:

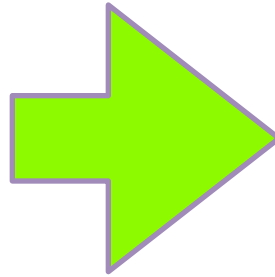
1.  $in[n]$  can be computed from  $out[n]$ ;
2.  $out[n]$  can be computed from  $in[m]$  for all the nodes  $m$  that are successors of  $n$ .

# Application: Dead Code Elimination

---

```
i := 0;
t3 := 0;
while i <= n do
  j := 0;
  t2 := t3;
  while j <= m do
    t1 := t3 + j;
    temp := Base(A) + t1;
    Cont(temp) := Cont(Base(B) + t1)
                + Cont(Base(C) + t1);
    j := j+1
  od;
  i := i+1;
  t3 := t3 + (m+1)
od
```

dead variable



```
i := 0;
t3 := 0;
while i <= n do
  j := 0;
  while j <= m do
    t1 := t3 + j;
    temp := Base(A) + t1;
    Cont(temp) := Cont(Base(B) + t1)
                + Cont(Base(C) + t1);
    j := j+1
  od;
  i := i+1;
  t3 := t3 + (m+1)
od
```



# Reaching Definitions (Reaching Assignment) Analysis

One of the more useful data-flow analysis

```
d1 : y := 3  
d2 : x := y
```

d1 is a reaching definition for d2

```
d1 : y := 3  
d2 : y := 4  
d3 : x := y
```

d1 is no longer a reaching definition for d3, because d2 kills its reach: the value defined in d1 is no longer available and cannot reach d3

A definition d at point i reaches a point p if there is a path from the point i to p such that d is not killed (redefined) along that path

## Reaching definitions

---

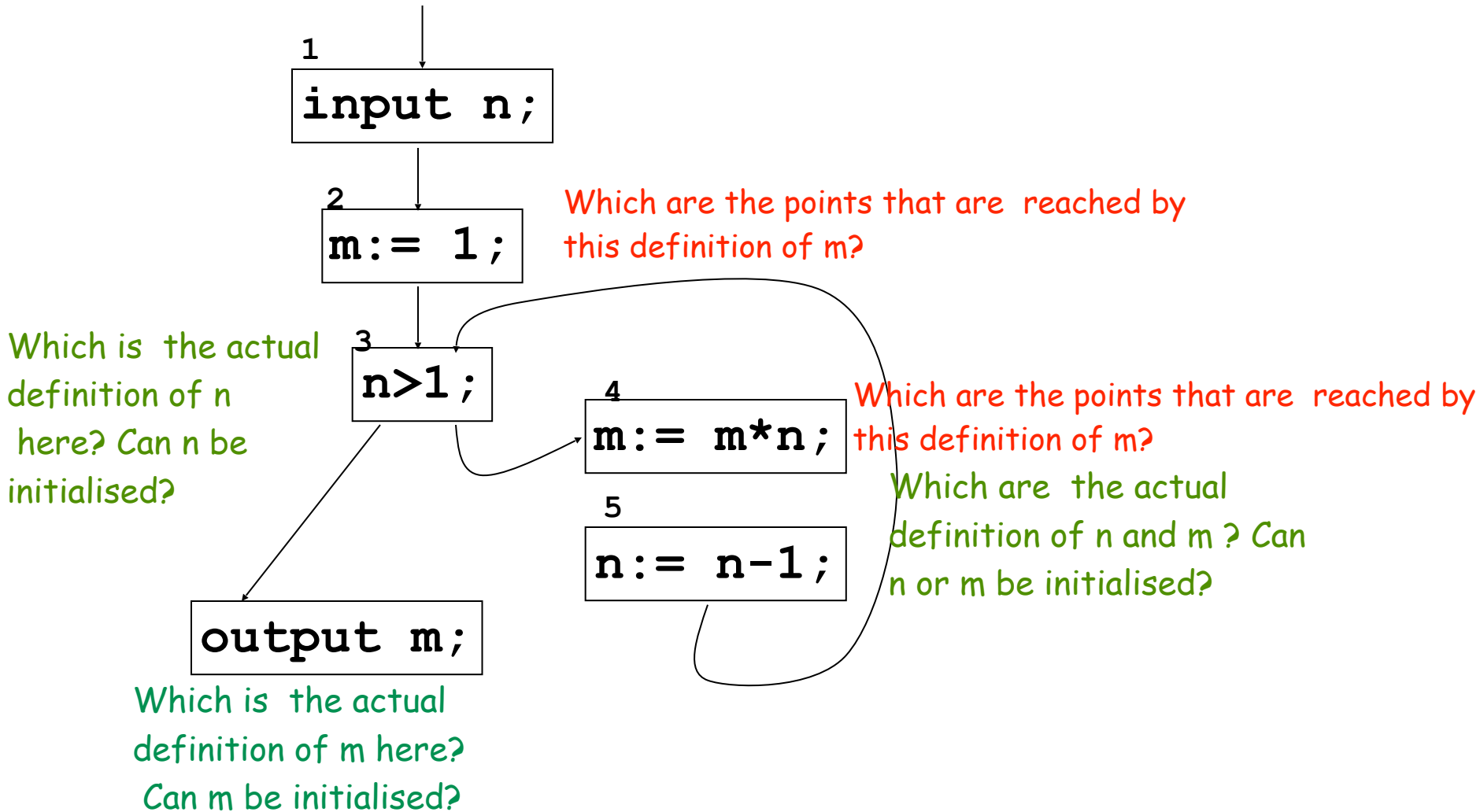
This information is very useful

- The compiler can know whether  $x$  is a constant at point  $p$
- The debugger can tell whether it is possible that  $x$  is an undefined variable at point  $p$

## Reaching definitions

- Given a program point  $n$ , which **definitions** are actual - not successively overwritten by a different assignment - when the execution reaches  $n$ ?  
And when the execution leaves  $n$ ?
- A program point may clearly "generate" new definitions
- A program point  $n$  may "kill" a definition:  
if  $n$  is an assignment  $x:=exp$  then  $n$  kills all the assignments to the variable  $x$  which are actual in input to  $n$
- We are thus interested in computing input and output reaching definitions for any program point

# The intuition: the factorial of n



## Formalization of the reaching definition property

- The property can be represented by sets of pairs:  
 $\{(x,p) \mid x \in \mathbf{Vars}, p \text{ is a program point}\} \in \mathcal{P}(\mathbf{Vars} \times \mathbf{Points})$   
where  $(x,p)$  means that the variable  $x$  is assigned at program point  $p$
- For each program point, this dataflow analysis computes a set of such pairs
- The meaning of a pair  $(x,p)$  in the set for a program point  $q$  is that the assignment of  $x$  at point  $p$  is actual at point  $q$
- $?$  is a special symbol that we add to  $\mathbf{Points}$  and we use to represent the fact that a variable  $x$  is not initialized.
- The set  $\iota = \{(x,?) \mid x \in \mathbf{Vars}\}$  therefore denotes that all the program variables are not initialized.

## The domain for Reaching Definitions Analysis

**Vars** is the (finite) set of variables occurring in the program  $P$ .

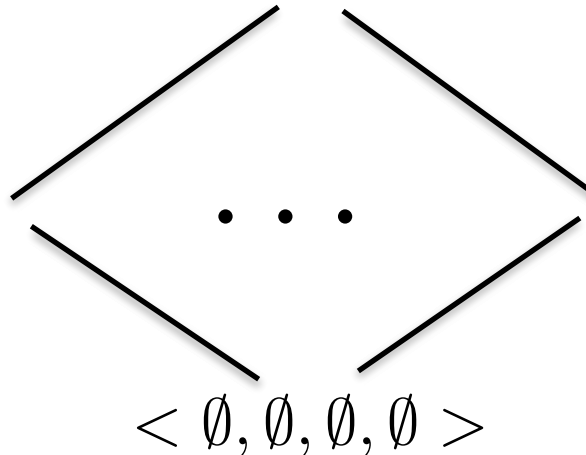
Let  $N$  be the number of nodes of the CFG of  $P$ .

Let **Points** =  $\{?, 1, \dots, N\}$ .

$$\langle \mathcal{P}(\text{Vars} \times \text{Points}) \times \mathcal{P}(\text{Vars} \times \text{Points}) \rangle^N, \subseteq^{2N} \rangle$$

- Example  $\text{Vars} = \{a, b\}$  e  $N = 1$

$$\langle S = \{(a, ?), (a, 1), (b, ?), (b, 1)\}, S, S, S \rangle$$



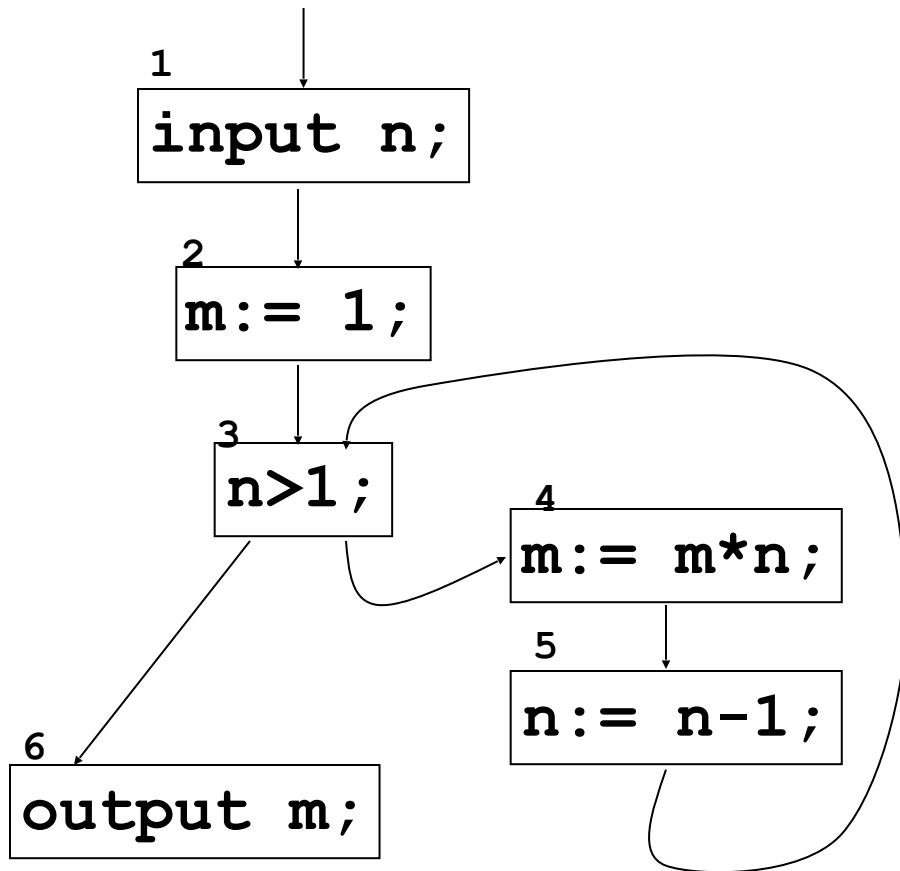
# Specification

---

- $\text{kill}_{\text{RD}}[p] = \begin{cases} \{(x,q) \mid q \in \mathbf{Points} \text{ and } \{x\} = \text{def}[q]\} & \text{if } \{x\} = \text{def}[p] \\ \emptyset & \text{if } \emptyset = \text{def}[p] \end{cases}$
- $\text{gen}_{\text{RD}}[p] = \begin{cases} \{(x,p)\} & \text{if } \{x\} = \text{def}[p] \\ \emptyset & \text{if } \emptyset = \text{def}[p] \end{cases}$

As usual,  $\text{def}[p] = \{x\}$  when the command in the point  $p$  is an assignment  $x := \text{exp}$

# Kill and Gen



	kill <sub>RD</sub>	gen <sub>RD</sub>
1		
2	(m,?)(m,2) (m,4)	(m,2)
3		
4	(m,?)(m,2) (m,4)	(m,4)
5	(n,?) (n,5)	(n,5)
6		

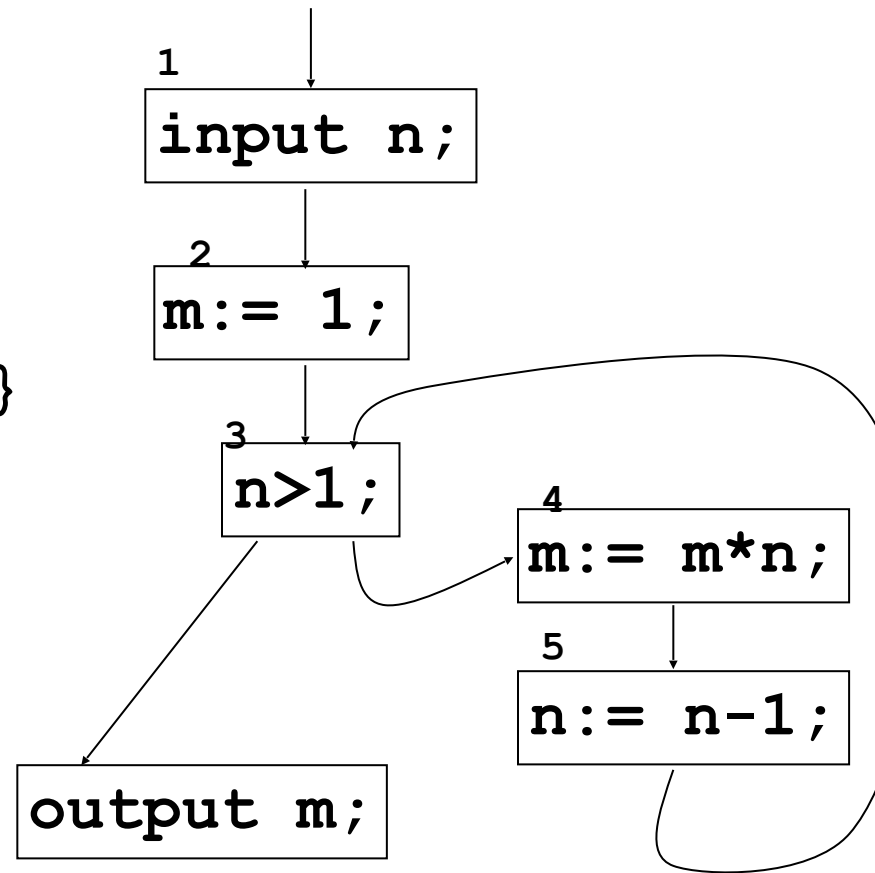


# Specification

- Reaching definitions analysis is specified by equations:

$$RD_{\text{entry}}(p) = \begin{cases} \{(x, ?) \mid x \in \text{VARS}\} & \text{if } p \text{ is initial} \\ \bigcup \{RD_{\text{exit}}(q) \mid q \in \text{pre}[p]\} & \text{if } p \text{ is not initial} \end{cases}$$

$$RD_{\text{exit}}(p) = (RD_{\text{entry}}(p) \setminus \text{kill}_{RD}[p]) \cup \text{gen}_{RD}[p]$$



## The solution of the previous system

Once again the solution for the equations in the previous system are require the existence of a fix point

We can apply the Kleene theorem if we have

- a) a continuous function on
- b) a CPO with bottom

## Point b

---

$$\langle (\mathcal{P}(\text{Vars} \times \text{Points}) \times \mathcal{P}(\text{Vars} \times \text{Points}))^{\mathbb{N}}, \subseteq^{2N} \rangle$$

is a CPO with bottom?

Yes! Because it is finite

## Point a: the map

---

The map Reach:

$$\langle \mathcal{P}(\text{Vars} \times \text{Points}) \times \mathcal{P}(\text{Vars} \times \text{Points})^N \rightarrow \langle \mathcal{P}(\text{Vars} \times \text{Points}) \times \mathcal{P}(\text{Vars} \times \text{Points})^N \rangle$$

defined by

(assuming 1 is the only initial node)

$$\text{Reach}(\langle \text{RD}_{\text{entry}_1}, \text{RD}_{\text{exit}_1}, \dots, \text{RD}_{\text{entry}_N}, \text{RD}_{\text{exit}_N} \rangle) =$$

$$\langle \{(x, ?) \mid x \in \text{VARS}\}, (\text{RD}_{\text{entry}_1} \setminus \text{kill}_{\text{RD}}[1]) \cup \text{gen}_{\text{RD}}[1],$$

$$\cup \{\text{RD}_{\text{exit}_2} \mid m \in \text{pre}[2]\}, (\text{RD}_{\text{entry}_2} \setminus \text{kill}_{\text{RD}}[2]) \cup \text{gen}_{\text{RD}}[2],$$

....,

$$\cup \{\text{RD}_{\text{exit}_m} \mid m \in \text{pre}[N]\}, (\text{RD}_{\text{entry}_N} \setminus \text{kill}_{\text{RD}}[N]) \cup \text{gen}_{\text{RD}}[N] \rangle$$

# Point a

$$\text{Reach}(\langle \text{RDentry}_1, \text{RDexit}_1, \dots, \text{RDentry}_N, \text{RDexit}_N \rangle) =$$

$$\langle \{(x, ?) \mid x \in \text{VARS}\}, (\text{RD}_{\text{entry}_1} \setminus \text{kill}_{\text{RD}}[1]) \cup \text{gen}_{\text{RD}}[1],$$

$$\cup \{\text{RD}_{\text{exit}_2} \mid m \in \text{pre}[2]\}, (\text{RD}_{\text{entry}_2} \setminus \text{kill}_{\text{RD}}[2]) \cup \text{gen}_{\text{RD}}[2]$$

$$\dots,$$

$$\cup \{\text{RD}_{\text{exit}_m} \mid m \in \text{pre}[N]\}, (\text{RD}_{\text{entry}_N} \setminus \text{kill}_{\text{RD}}[N]) \cup \text{gen}_{\text{RD}}[N] \rangle$$

- Example

$$\text{Reach}(\langle \{\}, \{\}, \{\}, \{\} \rangle) = \langle \{(a, ?)(b, ?)\}, \{(a, 1)(b, ?)\}, \{(a, 1)(b, ?)\}, \{(a, 1)(b, 2)\} \rangle$$

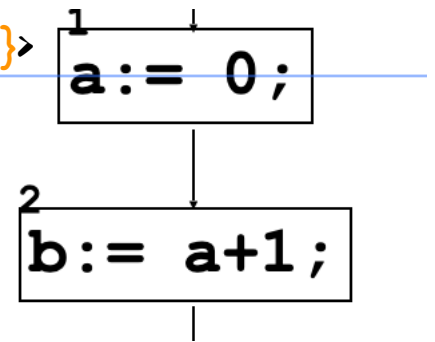
$$\text{Reach}(\langle \{(a, ?)(b, ?)\}, \{(a, 1)(b, ?)\}, \{(a, 1)(b, ?)\}, \{(a, 1)(b, 2)\} \rangle) =$$

$$\langle \{(a, ?)(b, ?)\}, \{(a, 1)(b, ?)\}, \{(a, 1)(b, ?)\}, \{(a, 1)(b, 2)\} \rangle$$

Note that Reach is monotone!

$$\text{kill}_{\text{RD}}(1) = \{(a, ?)\}, \text{gen}_{\text{RD}}(1) = \{(a, 1)\}$$

$$\text{kill}_{\text{RD}}(2) = \{(b, ?)\}, \text{gen}_{\text{RD}}(2) = \{(b, 2)\}$$



Since it is monotone on a finite domain then it is continuous

## Why a **least** fix point

---

RD analysis is **possible**,

if an assignment  $x:=a$  in some point  $q$  is really actual in entry to some point  $p$  then

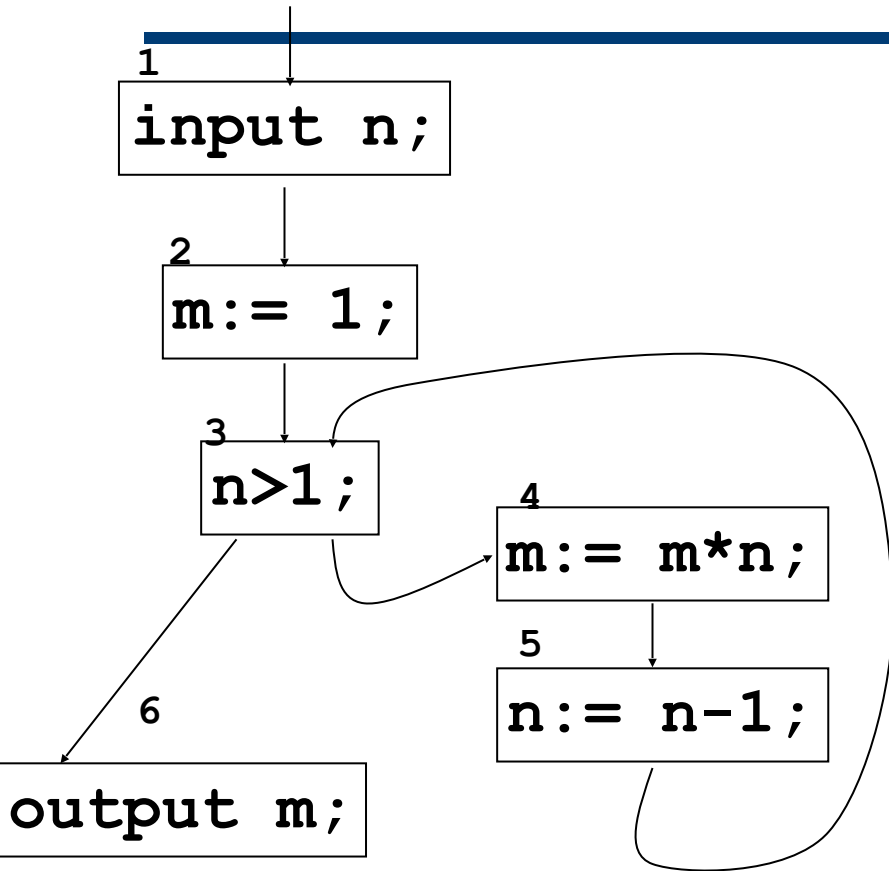
$$(x,q) \in \text{RD}_{\text{entry}}(p)$$

The vice versa does not hold

All fixpoints of the above equation system is an over-approximation of really reaching definitions.

Computing the least fixpoint gives a more precise over approximation

First iteration:



$$RD_{\text{entry}}(p) = \{(x, ?) \mid x \text{ in Vars}\}, \text{ if } p \text{ is initial}$$

$$RD_{\text{entry}}(p) = \cup \{RD_{\text{exit}}(q) \mid q \text{ in pre}[p]\}, \text{ otherwise}$$

$$RD_{\text{exit}}(p) = (RD_{\text{entry}}(p) \setminus \text{kill}_{RD}[p]) \cup \text{gen}_{RD}[p]$$

	kill	gen
2	$(m, ?)(m, 2)$ $(m, 4)$	$(m, 2)$
4	$(m, ?)(m, 2)$ $(m, 4)$	$(m, 4)$
5	$(n, ?)$ $(n, 5)$	$(n, 5)$

$$RD_{\text{entry}}(1) = \{(n, ?), (m, ?)\}$$

$$RD_{\text{exit}}(1) = \{(n, ?), (m, ?)\}$$

$$RD_{\text{entry}}(2) = \{(n, ?), (m, ?)\}$$

$$RD_{\text{exit}}(2) = \{(n, ?), (m, 2)\}$$

$$RD_{\text{entry}}(3) = \{(n, ?), (m, 2)\}$$

$$RD_{\text{exit}}(3) = \{(n, ?), (m, 2)\}$$

$$RD_{\text{entry}}(4) = \{(n, ?), (m, 2)\}$$

$$RD_{\text{exit}}(4) = \{(n, ?), (m, 4)\}$$

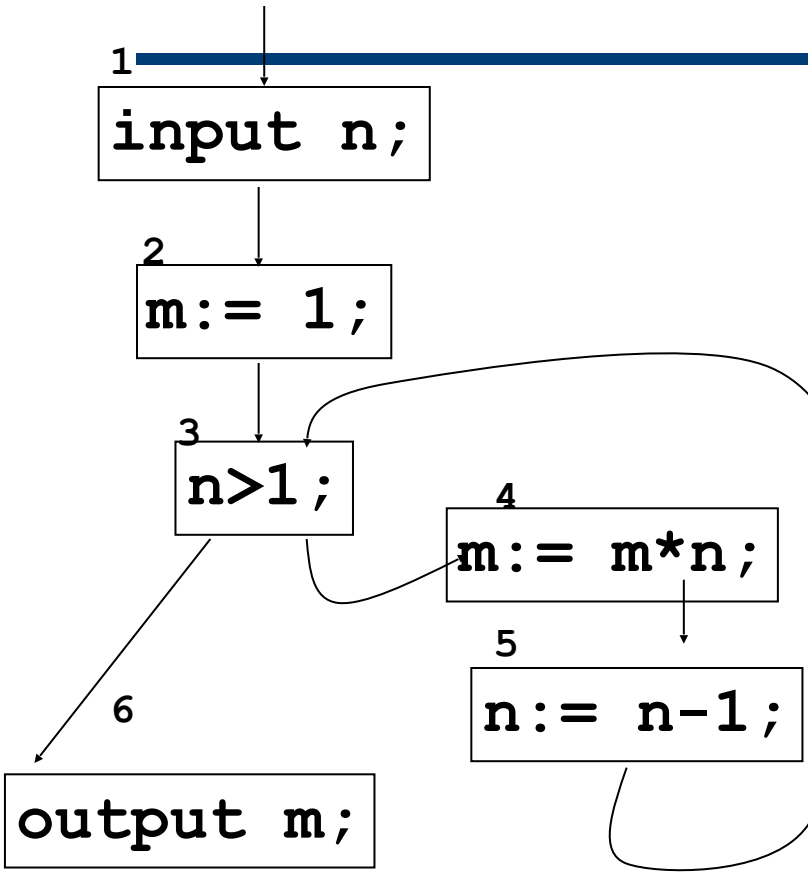
$$RD_{\text{entry}}(5) = \{(n, ?), (m, 4)\}$$

$$RD_{\text{exit}}(5) = \{(n, 5), (m, 4)\}$$

$$RD_{\text{entry}}(6) = \{(n, ?), (m, 2)\}$$

$$RD_{\text{exit}}(6) = \{(n, ?), (m, 2)\}$$

# Second iteration:



2	(m,?)(m,2) (m,4)	(m,2)
4	(m,?)(m,2) (m,4)	(m,4)
5	(n,?) (n,5)	(n,5)

$RD_{entry}(p) = \{(x,?) \mid x \text{ in Vars}\}$ , if p is initial  
 $RD_{entry}(p) = \cup \{RD_{exit}(q) \mid q \text{ in pre}[p]\}$ , otherwise

$RD_{exit}(p) = (RD_{entry}(p) \setminus kill_{RD}[p]) \cup gen_{RD}[p]$

$RD_{entry}(1) = \{(n,?), (m,?)\}$	$RD_{entry}(1) = \{(n,?), (m,?)\}$
$RD_{exit}(1) = \{(n,?), (m,?)\}$	$RD_{exit}(1) = \{(n,?), (m,?)\}$
$RD_{entry}(2) = \{(n,?), (m,?)\}$	$RD_{entry}(2) = \{(n,?), (m,?)\}$
$RD_{exit}(2) = \{(n,?), (m,2)\}$	$RD_{exit}(2) = \{(n,?), (m,2)\}$
$RD_{entry}(3) = \{(n,?), (m,2)\}$	$RD_{entry}(3) = \{(n,?), (m,2), (n,5)(m,4)\}$
$RD_{exit}(3) = \{(n,?), (m,2)\}$	$RD_{exit}(3) = \{(n,?), (m,2), (n,5)(m,4)\}$
$RD_{entry}(4) = \{(n,?), (m,2)\}$	$RD_{entry}(4) = \{(n,?), (m,2), (n,5)(m,4)\}$
$RD_{exit}(4) = \{(n,?), (m,4)\}$	$RD_{exit}(4) = \{(n,?), (n,5)(m,4)\}$
$RD_{entry}(5) = \{(n,?), (m,4)\}$	$RD_{entry}(5) = \{(n,?), (n,5)(m,4)\}$
$RD_{exit}(5) = \{(n,5), (m,4)\}$	$RD_{exit}(5) = \{(n,5), (m,4)\}$
$RD_{entry}(6) = \{(n,?), (m,2)\}$	$RD_{entry}(6) = \{(n,?), (m,2), (n,5)(m,4)\}$
$RD_{exit}(6) = \{(n,?), (m,2)\}$	$RD_{exit}(6) = \{(n,?), (m,2), (n,5)(m,4)\}$

**fix point!**



# RD analysis

---

- RD analysis is forward and **possible**,  
i.e., if an assignment  $x:=a$  in some point  $q$  is really actual in entry  
to some point  $p$  then  
 $(x,q) \in \text{RD}_{\text{entry}}(p)$  (while the vice versa does not hold).

How can we use this?

- If the analysis tells us that a variable is undefined then it is
- Loop invariant code motions

## Application: Loop invariant code motion

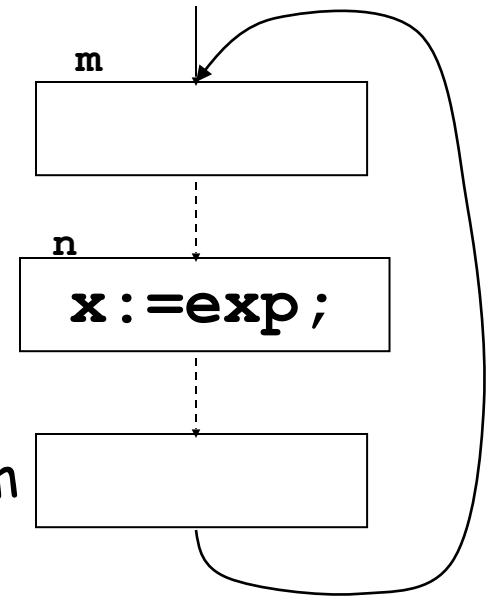
---

Consider a loop where:

1.  $m$  is the entry point
2. an inner point  $n$  contains an assignment  $x := \text{exp}$
3. if for any variable  $y$  occurring in  $\text{exp}$  (i.e.  $y \in \text{vars}(\text{exp})$ ) and for any program point  $p$ , we have that

$$(y, p) \in \text{RD}_{\text{entry}}(m) \iff (y, p) \in \text{RD}_{\text{entry}}(n)$$

then, the assignment  $x := \text{exp}$  can be correctly moved out as preceding the entry point of the loop



## Application: Loop invariant code motion

### Loop-invariant code motion

```
y:=3; z:=5;
for(int i=0; i<9; i++) {
    x = y + z;
    a[i] = 2*i + x;
}
```

```
y:=3; z:=5;
x = y + z;
for(int i=0; i<9; i++) {
    a[i] = 2*i + x;
}
```

## Available Expressions Analysis

Let  $p$  be a program point. For each execution path ending in  $p$ , we want track the expressions that have already been evaluated and then not modified.

These are called **available expressions**

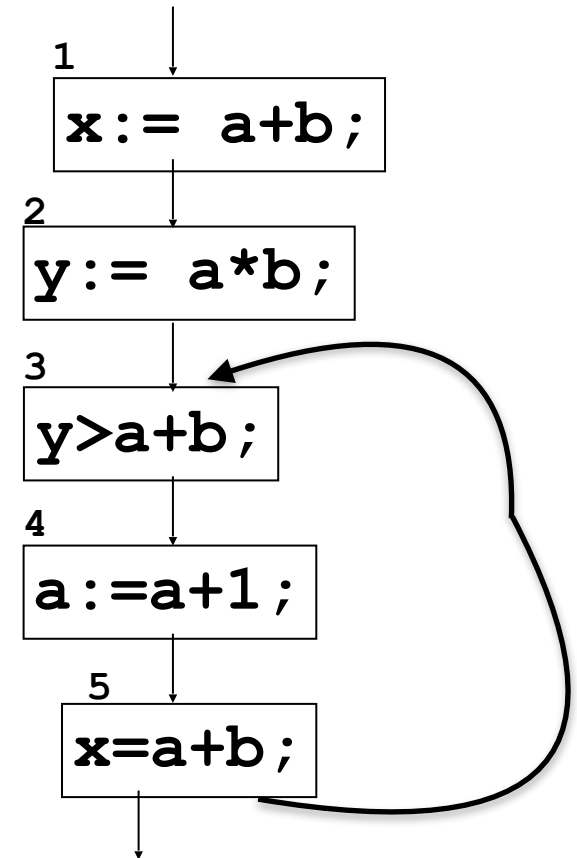
# Example

---

```
x:=a+b;  
y:=a*b;  
while y>a+b  
do (a:=a+1;  
    x:=a+b;)
```

when the execution reaches 3, the expression  $a+b$  is available, since it has been previously evaluated (in point 1 for the first iteration of the while-loop and in point 5 for the next iterations) and does not need to be evaluated again in 3

- This analysis can be therefore used to avoid re-evaluations of available expressions



## The domain

---

Let  $E = \{ e \mid e \text{ is a sub-expression/expression appearing in } P \}$

Let  $N$  be the number of nodes of the CFG of  $P$

$(\mathcal{P}(E) \times \mathcal{P}(E))^N, \subseteq^{2N}$  is a finite domain

## Kill<sub>AE</sub> and Gen<sub>AE</sub>

- An expression  $e$  in  $E$  is killed in a program point  $p$  ( $e$  is in  $\text{kill}_{AE}(p)$ ) if a variable occurring in  $e$  is modified (i.e., it is defined by some assignment) by the command in  $p$ .

$$\text{kill}_{AE}([x:=e']^p) = \{e \text{ in } E \mid x \in \text{vars}(e)\}$$

- An expression  $e$  is generated in a program point  $p$  ( $e$  is in  $\text{gen}_{AE}(p)$ ) if  $e$  is evaluated in  $p$  and no variable occurring in  $e$  is modified in  $p$ .

$$\text{gen}_{AE}([x:=e]^p) = \{e\} \quad \text{if } x \notin \text{vars}(e),$$

$$\text{gen}_{AE}([x:=e]^p) = \emptyset \quad \text{if } x \in \text{vars}(e);$$

$$\text{gen}_{AE}([e_1 \triangleright e_2]^p) = \text{expr}(\{e_1, e_2\}) \quad \text{where } \text{expr}(S) \text{ returns}$$

the subset of  $S$  that are expressions

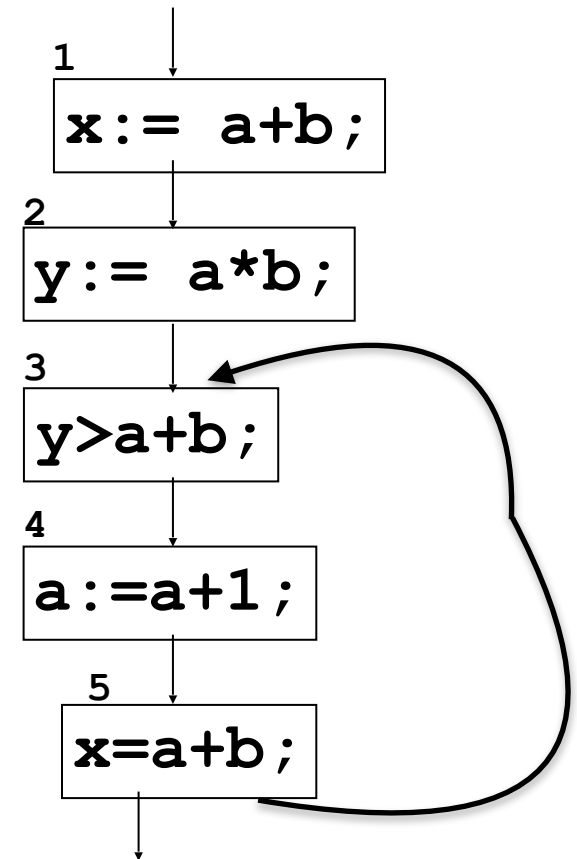
# Example

---

$x := a + b; y := a * b; \text{ while } y > a + b \text{ do } (a := a + 1; x := a + b)$

$E = \{a + b, a * b, a + 1\}$

n	$\text{kill}_{AE}(n)$	$\text{gen}_{AE}(n)$
1	$\emptyset$	$\{a + b\}$
2	$\emptyset$	$\{a * b\}$
3	$\emptyset$	$\{a + b\}$
4	$\{a + b, a * b, a + 1\}$	$\emptyset$
5	$\emptyset$	$\{a + b\}$





# Specification

---

- Available expressions analysis is specified by the following equations, for any program point  $p$ :

$$AE_{\text{entry}}(p) = \begin{cases} \emptyset & \text{if } p \text{ is initial} \\ \cap \{AE_{\text{exit}}(q) \mid q \in \text{pre}[p]\} & \text{otherwise} \end{cases}$$

$$AE_{\text{exit}}(p) = (AE_{\text{entry}}(p) \setminus \text{kill}_{AE}(p)) \cup \text{gen}_{AE}(p)$$

## Point a and b to apply Kleene Theorem

To find a solution to the previous equation system we need to apply Kleene Theorem

b)  $(\mathcal{P}(\mathbf{E}) \times \mathcal{P}(\mathbf{E}))^{\mathbb{N}}, \subseteq^{2^{\mathbb{N}}}$  is a finite domain therefore is a CPO, moreover, it has a bottom element

a) The map  $(\mathcal{P}(\mathbf{E}) \times \mathcal{P}(\mathbf{E}))^{\mathbb{N}} \rightarrow (\mathcal{P}(\mathbf{E}) \times \mathcal{P}(\mathbf{E}))^{\mathbb{N}}$  defined by  
(assuming 1 is the only initial node)

$$AE(\langle AE_{\text{entry}1}, AE_{\text{exit}1}, \dots, AE_{\text{entry}N}, AE_{\text{exit}N} \rangle) =$$

$$\langle \emptyset, (AE_{\text{entry}1} \setminus \text{kill}_{AE}(1)) \cup \text{gen}_{AE}(1),$$

$$\cap \{AE_{\text{exit}q} \mid q \text{ in pre}[2]\}, (AE_{\text{entry}2} \setminus \text{kill}_{AE}(2)) \cup \text{gen}_{AE}(2),$$

.....

$$\cap \{AE_{\text{exit}q} \mid q \text{ in pre}[N]\}, (AE_{\text{entry}N} \setminus \text{kill}_{AE}(N)) \cup \text{gen}_{AE}(N) \rangle$$

## Point a

a) The map

$$AE(\langle AE_{\text{entry}1}, AE_{\text{exit}1}, \dots, AE_{\text{entry}N}, AE_{\text{exit}N} \rangle) =$$

$$\langle \emptyset, (AE_{\text{entry}1} \setminus \text{kill}_{AE}(1)) \cup \text{gen}_{AE}(1),$$

$$\cap \{AE_{\text{exit}q} \mid q \text{ in } \text{pre}[2]\}, (AE_{\text{entry}2} \setminus \text{kill}_{AE}(2)) \cup \text{gen}_{AE}(2),$$

.....

$$\cap \{AE_{\text{exit}q} \mid q \text{ in } \text{pre}[N]\}, (AE_{\text{entry}N} \setminus \text{kill}_{AE}(N)) \cup \text{gen}_{AE}(N) \rangle$$

is monotone on the finite domain

$$(\mathcal{P}(\mathbf{E}) \times \mathcal{P}(\mathbf{E}))^N, \subseteq^{2N} \rangle$$

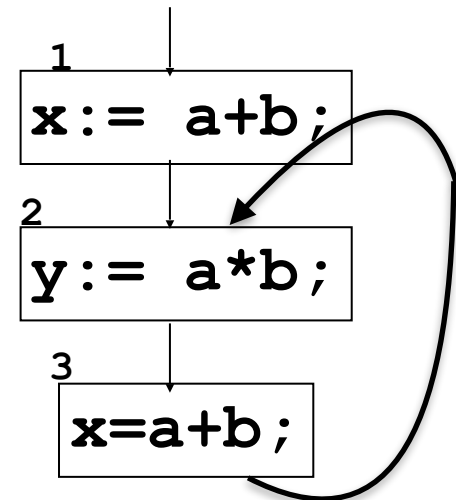
• Example

$$AE(\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle) =$$

$$\langle \emptyset, \{a+b\}, \{\}, \{a*b\}, \{a*b\}, \{a+b, a*b\} \rangle$$

$$AE(\langle \emptyset, \{a+b\}, \{\}, \{a*b\}, \{a*b\}, \{a+b, a*b\} \rangle) =$$

$$\langle \emptyset, \{a+b\}, \{a+b\}, \{a+b, a*b\}, \{a+b, a*b\}, \{a+b, a*b\} \rangle$$



## Which fix point?

---

AE is a definite analysis:

if  $e \in AE_{\text{entry}}(p)$  then  $e$  is really available in entry to  $p$

the converse does not hold

- Any fixpoint of the above equation system is an under-approximation of really available expressions.

Between all fix points, we are thus interested in computing the **greatest fixpoint** (the more precise approximation)

Also, observe that this is a **forward** analysis.

# Computing the greatest fix point

The starting point, for all  $n$   
 $AE_{\text{entry}}(n) = AE_{\text{exit}}(n) = \{a+b, a*b, a+1\}$

$x := a+b; y := a*b; \text{ while } y > a+b \text{ do } (a := a+1; x := a+b)$

$E = \{a+b, a*b, a+1\}$

$AE_{\text{entry}}(p) = \emptyset$  if  $p$  is initial

$AE_{\text{entry}}(p) = \bigcap \{AE_{\text{exit}}(q) \mid q \text{ in } \text{pre}[p]\}$

$AE_{\text{exit}}(p) = (AE_{\text{entry}}(p) \setminus \text{kill}_{AE}(p)) \cup \text{gen}_{AE}(p)$

$n$	$\text{kill}_{AE}(n)$	$\text{gen}_{AE}(n)$
1	$\emptyset$	$\{a+b\}$
2	$\emptyset$	$\{a*b\}$
3	$\emptyset$	$\{a+b\}$
4	$\{a+b, a*b, a+1\}$	$\emptyset$
5	$\emptyset$	$\{a+b\}$

$$AE_{\text{entry}}(1) = \emptyset$$

$$AE_{\text{exit}}(1) = \{a+b\}$$

$$AE_{\text{entry}}(2) = \{a+b\}$$

$$AE_{\text{exit}}(2) = \{a+b, a*b\}$$

$$AE_{\text{entry}}(3) = \{a+b, a*b\}$$

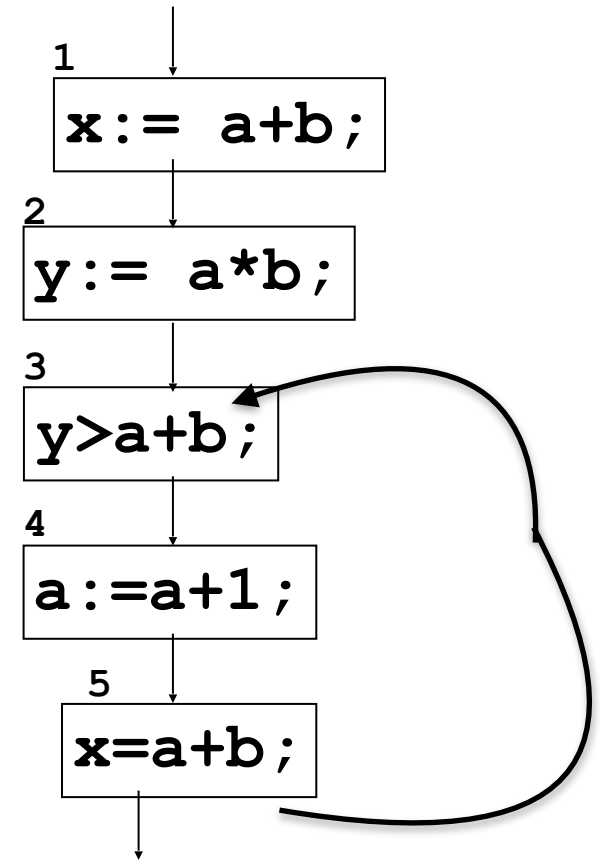
$$AE_{\text{exit}}(3) = \{a+b, a*b\}$$

$$AE_{\text{entry}}(4) = \{a+b, a*b\}$$

$$AE_{\text{exit}}(4) = \{\}$$

$$AE_{\text{entry}}(5) = \{\}$$

$$AE_{\text{exit}}(5) = \{a+b\}$$



# Second iteration

$$AE_{\text{entry}}(p) = \emptyset \text{ if } p \text{ is initial}$$

$$AE_{\text{entry}}(p) = \bigcap \{AE_{\text{exit}}(q) \mid q \text{ in } \text{pre}[p]\}$$

$$AE_{\text{exit}}(p) = (AE_{\text{entry}}(p) \setminus \text{kill}_{AE}(p)) \cup \text{gen}_{AE}(p)$$

## Previous iteration

n	$AE_{\text{entry}}(n)$	$AE_{\text{exit}}(n)$
1	$\emptyset$	$\{a+b\}$
2	$\{a+b\}$	$\{a+b, a*b\}$
3	$\{a+b, a*b\}$	$\{a+b, a*b\}$
4	$\{a+b, a*b\}$	$\emptyset$
5	$\emptyset$	$\{a+b\}$

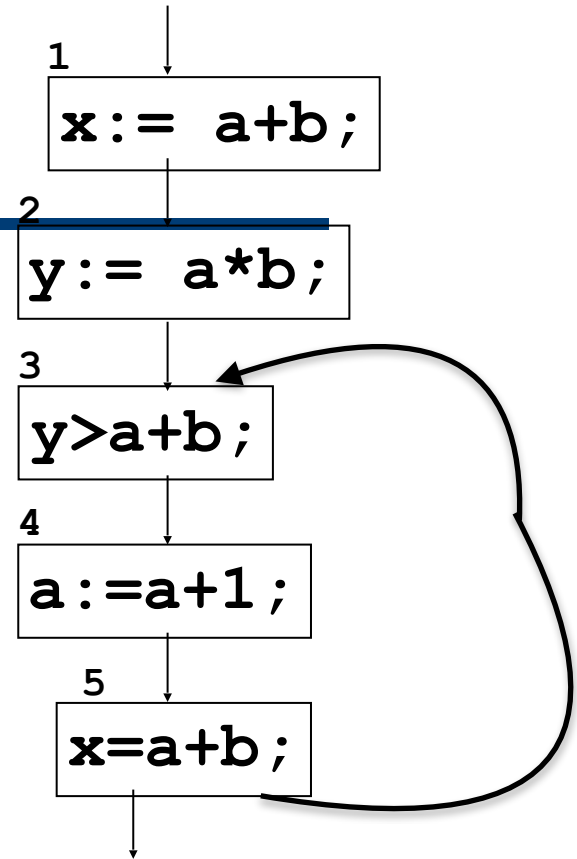
$$AE_{\text{exit}}(1) = AE_{\text{entry}}(1) \cup \{a+b\}$$

$$AE_{\text{exit}}(2) = AE_{\text{entry}}(2) \cup \{a*b\}$$

$$AE_{\text{exit}}(3) = AE_{\text{entry}}(3) \cup \{a+b\}$$

$$AE_{\text{exit}}(4) = AE_{\text{entry}}(4) - \{a+b, a*b, a+1\}$$

$$AE_{\text{exit}}(5) = AE_{\text{entry}}(5) \cup \{a+b\}$$



n	$AE_{\text{entry}}(n)$	$AE_{\text{exit}}(n)$
1	$\emptyset$	$\{a+b\}$
2	$\{a+b\}$	$\{a+b, a*b\}$
3	$\{a+b\}$	$\{a+b\}$
4	$\{a+b\}$	$\emptyset$
5	$\emptyset$	$\{a+b\}$

# Third iteration and Greatest Fixpoint

$AE_{\text{entry}}(p) = \emptyset$  if  $p$  is initial

$AE_{\text{entry}}(p) = \bigcap \{AE_{\text{exit}}(q) \mid q \text{ in } \text{pre}[p]\}$

$AE_{\text{exit}}(p) = (AE_{\text{entry}}(p) \setminus \text{kill}_{AE}(p)) \cup \text{gen}_{AE}(p)$

Previous iteration

n	$AE_{\text{entry}}(n)$	$AE_{\text{exit}}(n)$
1	$\emptyset$	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	$\emptyset$
5	$\emptyset$	{a+b}

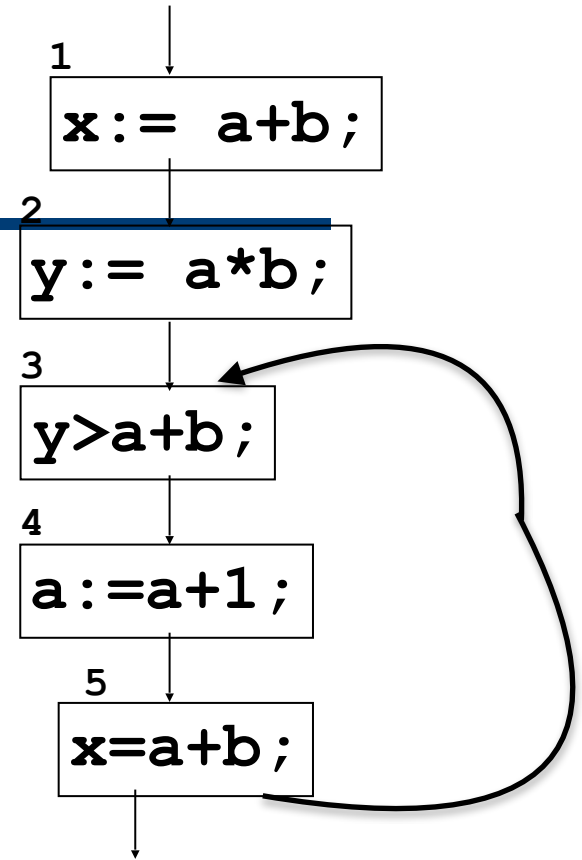
$$AE_{\text{exit}}(1) = AE_{\text{entry}}(1) \cup \{a+b\}$$

$$AE_{\text{exit}}(2) = AE_{\text{entry}}(2) \cup \{a*b\}$$

$$AE_{\text{exit}}(3) = AE_{\text{entry}}(3) \cup \{a+b\}$$

$$AE_{\text{exit}}(4) = AE_{\text{entry}}(4) - \{a+b, a*b, a+1\}$$

$$AE_{\text{exit}}(5) = AE_{\text{entry}}(5) \cup \{a+b\}$$

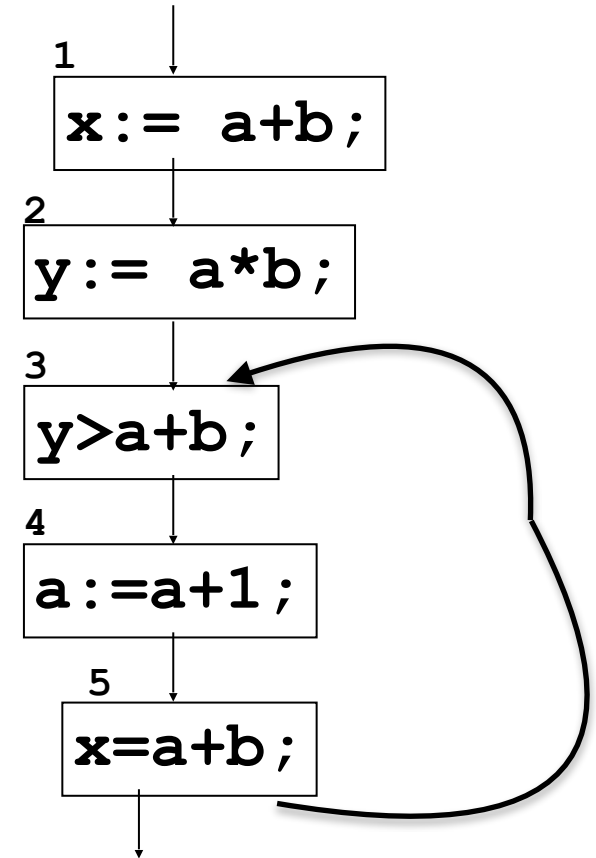


n	$AE_{\text{entry}}(n)$	$AE_{\text{exit}}(n)$
1	$\emptyset$	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	$\emptyset$
5	$\emptyset$	{a+b}

# Result

$x:=a+b; y:=a*b; \text{ while } y>a+b \text{ do } (a:=a+1; x:=a+b)$

n	$AE_{\text{entry}}(n)$	$AE_{\text{exit}}(n)$
1	$\emptyset$	$\{a+b\}$
2	$\{a+b\}$	$\{a+b, a*b\}$
3	$\{a+b\}$	$\{a+b\}$
4	$\{a+b\}$	$\emptyset$
5	$\emptyset$	$\{a+b\}$





# Application: Common Subexpression Elimination

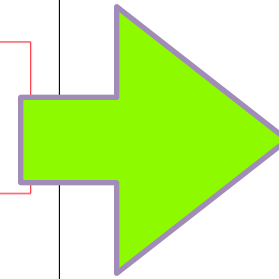
$$A[i,j]=B[i,j]+C[i,j]$$

```
i := 0;
while i <= n do
  j := 0;
  while j <= m do
    temp := Base(A) + i*(m+1) + j;
    Cont(temp) := Cont(Base(B) + i*(m+1) + j)
                + Cont(Base(C) + i*(m+1) + j);
    j := j+1
  od;
  i := i+1
od
```

first computation

```
temp := Base(A) + i*(m+1) + j;
Cont(temp) := Cont(Base(B) + i*(m+1) + j)
              + Cont(Base(C) + i*(m+1) + j);
```

re-computations







```
t1 := i * (m+1) + j;
temp := Base(A) + t1;
Cont(temp) := Cont(Base(B)+t1)
              + Cont(Base(C)+t1);
```

## A Dataflow Analysis Framework

- The above dataflow analyses (Reaching Definitions, Available Expressions, Live Variables) reveal many similarities.
- One major advantage of a unifying framework of dataflow analysis lies in the design of a generic analysis algorithm that can be instantiated in order to compute different dataflow analyses.

# Catalogue of Dataflow Analyses

---

	<i>Possible Analysis</i> <b>Semantics <math>\subseteq</math> Analysis</b>	<i>Definite Analysis</i> <b>Analysis <math>\subseteq</math> Semantics</b>
<i>Forward</i> <b>in[n]</b>  <b>out[n]</b> <b>pre</b>  <b>post</b>	Reaching definitions	Available expressions
<i>Backward</i> <b>out[n]</b>  <b>in[n]</b> <b>post</b>  <b>pre</b>	Live variables	Very busy expressions

# Static Analyses

# Summary of analyses and transformations

---

## Analysis

Available expressions analysis

Reaching Definition

Live variables analysis

Detection of induction variables

Equivalent expression analysis

## Transformation

Common subexpression elimination

Invariant code motion

Dead code elimination

Strength reduction

Copy propagation

# The essence of program analysis

Program analysis offers techniques for predicting **statically** at compile-time

safe & efficient **approximations**

to the set of configurations or behaviours arising **dynamically** at run-time

**Safe:** faithful to the **semantics**

**Efficient:** implementation with

- good time performance and
- low space consumption

# Why approximations?

---

read(x); (if  $x > 0$  then  $y = 1$  else  $(y = 2; S)$ );  $z = y$

Assume  $S$  does not contain any assignment to  $y$

Which values of  $y$  can reach the assignment  $z = y$ ??

It depends if  $S$  diverges

**Correct (Safe)** approximations :

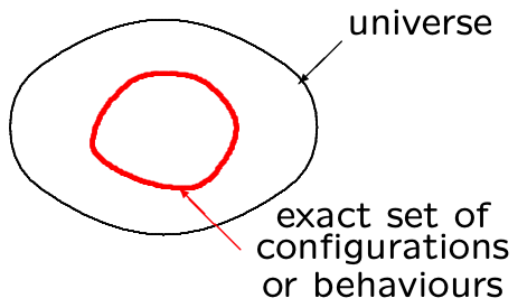
- $\{1, 17\}$  if  $S$  diverges
- $\{1, 2, 5, 27\}$  otherwise

**Best (Precise)** approximations :

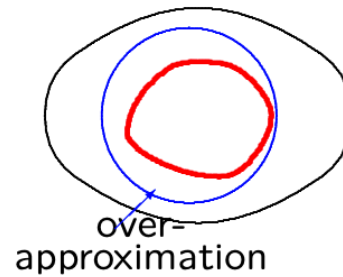
- $\{1\}$  in the first case
- $\{1, 2\}$  otherwise

# The nature of approximations

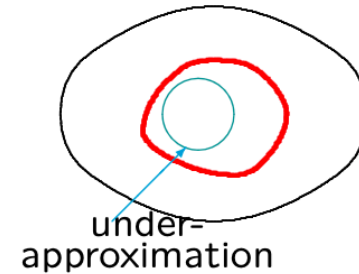
The exact world



Over-approximation



Under-approximation



Trade precision for efficiency!

but approximation have to be on the right side!



# Approaches to Program Analysis

A family of techniques . . .

- data flow analysis
- constraint based analysis
- type and effect systems
- abstract interpretation

that differs in their focus

- algorithmic methods
- semantic foundations
- language paradigms
  - imperative/procedural
  - object oriented
  - logical
  - functional
  - concurrent/distributive
  - mobile

# Dataflow Analysis

---

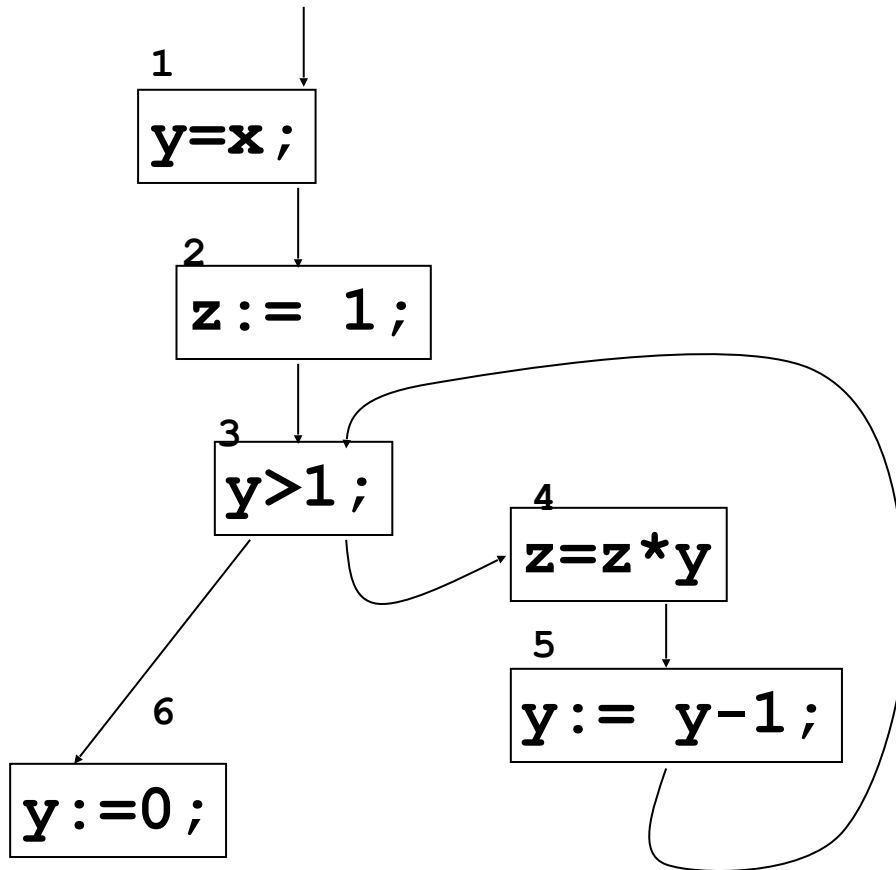
The ideas :

- Based on the CFG
- Constructs an equation system in terms of the property at the entry and exit of a node in the CFG
- Looks for solution of the system of equations as a fix point
- Compute either a least fixpoint or a greatest fix point depending on the direction of the approximation

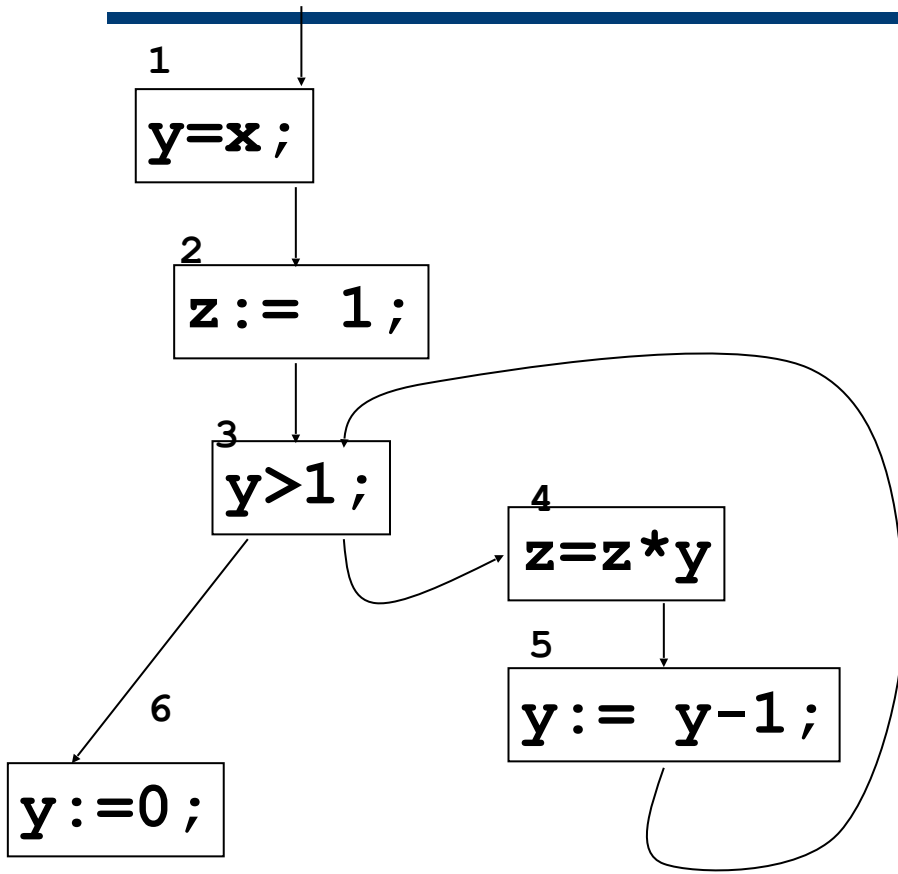
# An Example Reaching Definitions

---

$[y := x]^1; [z := 1]^2; \text{while}[y > 1]^3 \text{do}[z = z * y]^4; [y := y - 1]^5 \text{od}; [y := 0]^6$



# An example : RD



$$RD_{\text{entry}}(p) = \{(x,?) \mid x \text{ in Vars}\}, \text{ if } p \text{ is initial}$$

$$RD_{\text{entry}}(p) = \cup \{RD_{\text{exit}}(q) \mid q \text{ in pre}[p]\}, \text{ otherwise}$$

$$RD_{\text{exit}}(p) = (RD_{\text{entry}}(p) \setminus \text{kill}_{RD}[p]) \cup \text{gen}_{RD}[p]$$

	Kill	Gen
1	$(y,?)$ , $(y,1)$ , $(y,5)$ , $(y,6)$	$(y,1)$
2	$(z,?)$ , $(z,2)$ , $(z,4)$	$(z,2)$
4	$(z,?)$ , $(z,2)$ , $(z,4)$	$(z,4)$
5	$(y,?)$ , $(y,1)$ , $(y,5)$ , $(y,6)$	$(y,5)$
6	$(y,?)$ , $(y,1)$ , $(y,5)$ , $(y,6)$	$(y,6)$

Second iteration fix point!

$$RD_{\text{entry}}(1) = \{(x,?)$$

$$RD_{\text{exit}}(1) = \{(x,?)$$

$$RD_{\text{entry}}(2) = \{(x,?)$$

$$RD_{\text{exit}}(2) = \{(x,?)$$

$$RD_{\text{entry}}(3) = \{(x,?)$$

$$RD_{\text{exit}}(3) = \{(x,?)$$

$$RD_{\text{entry}}(4) = \{(x,?)$$

$$RD_{\text{exit}}(4) = \{(x,?)$$

$$RD_{\text{entry}}(5) = \{(x,?)$$

$$RD_{\text{exit}}(5) = \{(x,?)$$

$$RD_{\text{entry}}(6) = \{(x,?)$$

$$RD_{\text{exit}}(6) = \{(x,?)$$

# Control Flow Analysis

---

- Constructs a constraint system
- Looks for a solution of the constraint system as a fix point

# Control Flow Analysis

---

- An alternative to equational approach is the constraint based approach.
- The idea is to extract a number of inclusions (equation or constraints) out of the program
- Once again we consider the property at the entry and exit of a node
- We encode with constraints the information of the flow of the CFG

## Constraints for effects of elementary blocks

$[y = 1]^1; [z = 1]^2; \text{ while } [y > 1]^3 \text{ do } ([z = z * y]^4; [y = y - 1]^5); [y = 0]^6$

- We have constraints that express the effects for elementary blocks

$$RD_{exit}(1) \supseteq RD_{entry}(1) / \{(y, l) \text{ such that } l \in \mathbf{Lab}\}$$

$$RD_{exit}(1) \supseteq \{(y, 1)\}$$

$$RD_{exit}(2) \supseteq RD_{entry}(2) / \{(z, l) \text{ such that } l \in \mathbf{Lab}\}$$

$$RD_{exit}(2) \supseteq \{(z, 2)\}$$

$$RD_{exit}(3) \supseteq RD_{entry}(3)$$

$$RD_{exit}(4) \supseteq RD_{entry}(4) / \{(z, l) \text{ such that } l \in \mathbf{Lab}\}$$

$$RD_{exit}(4) \supseteq \{(z, 4)\}$$

$$RD_{exit}(5) \supseteq RD_{entry}(5) / \{(y, l) \text{ such that } l \in \mathbf{Lab}\}$$

$$RD_{exit}(5) \supseteq \{(y, 5)\}$$

$$RD_{exit}(6) \supseteq RD_{entry}(6) / \{(y, l) \text{ such that } l \in \mathbf{Lab}\}$$

$$RD_{exit}(6) \supseteq \{(y, 6)\}$$

# More constraints

- We have constraints that expression control may flow through the program

$[y = 1]^1; [z = 1]^2; \text{while } [y > 1]^3 \text{ do } ([z = z * y]^4; [y = y - 1]^5); [y = 0]^6$

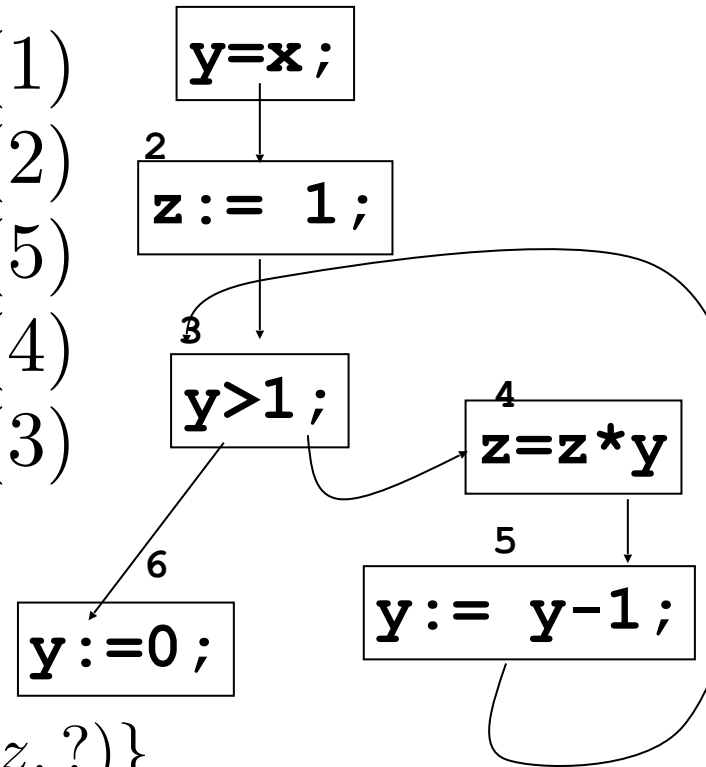
$$RD_{entry}(2) \supseteq RD_{exit}(1)$$

$$RD_{entry}(3) \supseteq RD_{exit}(2)$$

$$RD_{entry}(3) \supseteq RD_{exit}(5)$$

$$RD_{entry}(5) \supseteq RD_{exit}(4)$$

$$RD_{entry}(6) \supseteq RD_{exit}(3)$$



Finally the constraint

$$RD_{entry}(1) \supseteq \{(x, ?), (y, ?), (z, ?)\}$$



# Summary of the constraints system

---

$$RD_{exit}(1) \supseteq RD_{entry}(1) / \{(y, l) \text{ such that } l \in \mathbf{Lab}\}$$

$$RD_{exit}(1) \supseteq \{(y, 1)\}$$

$$RD_{exit}(2) \supseteq RD_{entry}(2) / \{(z, l) \text{ such that } l \in \mathbf{Lab}\}$$

$$RD_{exit}(2) \supseteq \{(z, 2)\}$$

$$RD_{exit}(3) \supseteq RD_{entry}(3)$$

$$RD_{exit}(4) \supseteq RD_{entry}(4) / \{(z, l) \text{ such that } l \in \mathbf{Lab}\}$$

$$RD_{exit}(4) \supseteq \{(z, 4)\}$$

$$RD_{exit}(5) \supseteq RD_{entry}(5) / \{(y, l) \text{ such that } l \in \mathbf{Lab}\}$$

$$RD_{exit}(5) \supseteq \{(y, 5)\}$$

$$RD_{exit}(6) \supseteq RD_{entry}(6) / \{(y, l) \text{ such that } l \in \mathbf{Lab}\}$$

$$RD_{exit}(6) \supseteq \{(y, 6)\}$$

$$RD_{entry}(2) \supseteq RD_{exit}(1)$$

$$RD_{entry}(3) \supseteq RD_{exit}(2)$$

$$RD_{entry}(3) \supseteq RD_{exit}(5)$$

$$RD_{entry}(5) \supseteq RD_{exit}(4)$$

$$RD_{entry}(6) \supseteq RD_{exit}(3)$$

$$RD_{entry}(1) \supseteq \{(x, ?), (y, ?), (z, ?)\}$$

12 sets

17 constraints

We look for  $\overleftarrow{RD}$

$$\overleftarrow{RD} \sqsubseteq F(\overleftarrow{RD})$$

the solution is a prefix point

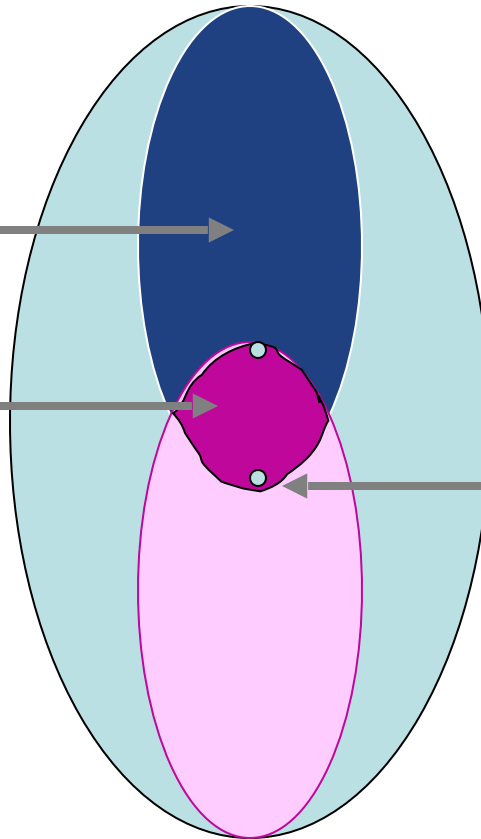
# Remember...Fixpoints on Complete Lattices

Prefixpoints

$$\{ l \in L \mid f(l) \leq_p l \}$$

$$\text{Fix}(f) = \{ l \in L \mid f(l) = l \}$$

$$\text{lfp}(f) = \text{glb} \{ l \in L \mid f(l) \leq l \}$$



The solution can be computed as a least fix point!

# The same solution!

---

$$RD_{\text{entry}}(1) = \{(x,?)(y,?)(z,?)\}$$

$$RD_{\text{exit}}(1) = \{(x,?)(y,1)(z,?)\}$$

$$RD_{\text{entry}}(2) = \{(x,?)(y,1)(z,?)\}$$

$$RD_{\text{exit}}(2) = \{(x,?)(y,1)(z,2)\}$$

$$RD_{\text{entry}}(3) = \{(x,?)(y,1)(z,2) (z,4)(y,5)\}$$

$$RD_{\text{exit}}(3) = \{(x,?)(y,1)(z,2) (z,4)(y,5)\}$$

$$RD_{\text{entry}}(4) = \{(x,?),(y,1)(z,2)(z,4) (y,5)\}$$

$$RD_{\text{exit}}(4) = \{(x,?),(y,1)(z,4)(y,5)\}$$

$$RD_{\text{entry}}(5) = \{(x,?),(y,1)(z,4)(y,5)\}$$

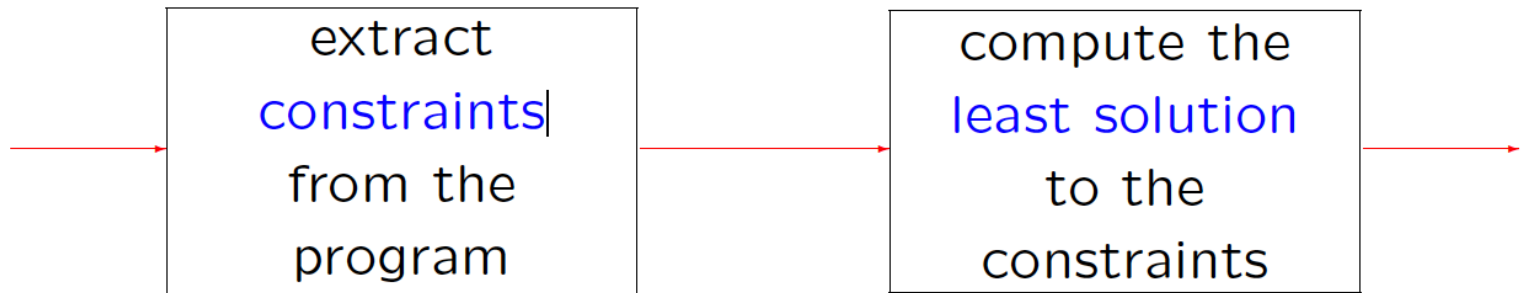
$$RD_{\text{exit}}(5) = \{(x,?),(y,5)(z,4)\}$$

$$RD_{\text{entry}}(6) = \{(x,?),(y,1)(z,2)(z,4)(y,5)\}$$

$$RD_{\text{exit}}(6) = \{(x,?),(y,6)(z,2) (z,4)\}$$

# Constraint based analysis

How to automate the analysis



# Type and Effect Systems

---

idea:

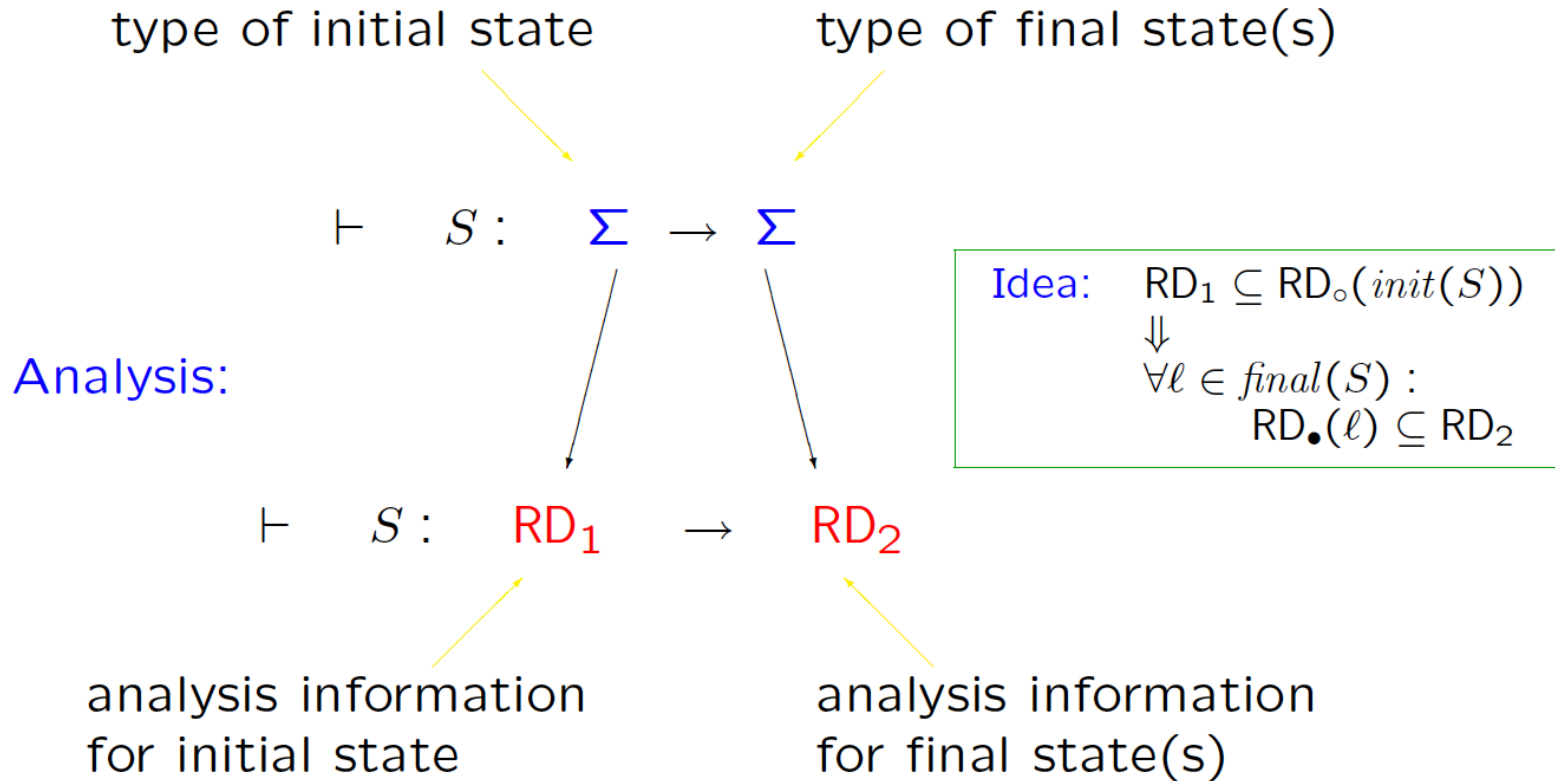
- annotated base types
- annotated type constructors

• types:

$\Sigma$  is the type of states;

all statements  $S$  have type  $\Sigma \rightarrow \Sigma$  written  $\vdash S : \Sigma \rightarrow \Sigma$

# Annotated data types



# Annotated type system I

---

$$\vdash [x := a]^\ell : \underbrace{\text{RD}}_{\text{before}} \rightarrow \underbrace{(\text{RD} \setminus \{(x, \ell') \mid \ell' \in \text{Lab}\}) \cup \{(x, \ell)\}}_{\text{after}}$$

$$\text{seq} \quad \frac{\vdash S_1 : \text{RD}_1 \rightarrow \text{RD}_2 \quad \vdash S_2 : \text{RD}_2 \mid \rightarrow \text{RD}_3}{\vdash S_1; S_2 : \underbrace{\text{RD}_1}_{\text{before}} \rightarrow \underbrace{\text{RD}_3}_{\text{after}}} \quad \frac{\text{assumptions}}{\text{conclusion}}$$

**Implicit:** the analysis information at the **exit** of  $S_1$   
equals the analysis information at the **entry** of  $S_2$

# Annotated type system II

---

$$\text{if} \frac{\vdash S_1 : \text{RD}_1 \rightarrow \text{RD}_2 \quad \vdash S_2 : \text{RD}_1 \rightarrow \text{RD}_2}{\vdash \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \text{ fi} : \text{RD}_1 \rightarrow \text{RD}_2}$$

**Implicit:** the two branches have the same analysis information at their respective **entry** and **exit** points

$$\text{while} \frac{\vdash S : \text{RD} \rightarrow \text{RD}}{\vdash \text{while } [b]^\ell \text{ do } S \text{ od} : \text{RD} \rightarrow \text{RD}}$$

**Implicit:** the occurrences of **RD** express an invariance i.e. a fixed point property!



# Annotated type system III

The subsumption rule:

$$\text{sub} \frac{\vdash S : RD'_1 \rightarrow RD'_2}{\vdash S : RD_1 \rightarrow RD_2} \quad \text{if } RD_1 \subseteq RD'_1 \text{ and } RD'_2 \subseteq RD_2$$

The rule is essential for the rules for conditional and iteration to work

- $RD_1 \subseteq RD'_1$ : strengthen the analysis information for the initial state
- $RD'_2 \subseteq RD_2$ : weaken the analysis information for the final states

We want to prove that

$$\text{seq} \frac{\vdash S_1 : RD_1 \rightarrow RD_2 \quad \vdash S_2 : RD_2 \rightarrow RD_3}{\vdash S_1; S_2 : RD_1 \rightarrow RD_3}$$

$$\{(x, ?), (y, ?), (z, ?)\} \rightarrow \{(x, ?), (y, 6), (z, 2), (z, 4)\}$$

$$\vdash [z := 1]^2; \text{ while } [y > 1]^3 \text{ do } [z := z * y]^4; [y := y - 1]^5 \text{ od}; [y := 0]^6$$

$$\{(x, ?), (y, 1), (z, ?)\} \rightarrow \{(x, ?), (y, 6), (z, 2), (z, 4)\}$$

$$\vdash [y := x]^1; \{(x, ?), (y, ?), (z, ?)\} \rightarrow \{(x, ?), (y, 1), (z, ?)\}$$

---


$$\vdash [y := x]^1; [z := 1]^2; \text{ while } [y > 1]^3 \text{ do } [z := z * y]^4; [y := y - 1]^5 \text{ od}; [y := 0]^6:$$

$$\{(x, ?), (y, ?), (z, ?)\} \rightarrow \{(x, ?), (y, 6), (z, 2), (z, 4)\}$$

We now need to prove that

$$\text{seq} \frac{\vdash S_1 : RD_1 \rightarrow RD_2 \quad \vdash S_2 : RD_2 \mid \rightarrow RD_3}{\vdash S_1; S_2 : RD_1 \rightarrow RD_3}$$

$$[z := 1]^2; \text{while}[y := 1]^3 \text{do}[z = z * y]^4; [y := y - 1]^5 \text{od}; [y := 0]^6 : \\ \{(x, ?), (y, 1), (z, ?)\} \rightarrow \{(x, ?), (y, 6), (z, 2), (z, 4)\}$$

$$\vdash \text{while}[y > 1]^3 \text{do}[z = z * y]^4; [y := y - 1]^5 \text{od}; [y := 0]^6 : \\ \{(x, ?), (y, 1), (z, 2)\} \rightarrow \{(x, ?), (y, 6), (z, 2), (z, 4)\}$$

$$\vdash [z := 1]^2 : \{(x, ?), (y, 1), (z, ?)\} \rightarrow \{(x, ?), (y, 1), (z, 2)\}$$

SEQ

$$\vdash [z := 1]^2; \text{while}[y > 1]^3 \text{do}[z = z * y]^4; [y := y - 1]^5 \text{od}; [y := 0]^6 :$$

$$\{(x, ?), (y, 1), (z, ?)\} \rightarrow \{(x, ?), (y, 6), (z, 2), (z, 4)\}$$



We now need to prove that

$while[y := 1]^3 do[z = z * y]^4; [y := y - 1]^5 od; [y := 0]^6 :$

$\{(x, ?), (y, 1), (z, 2)\} \rightarrow \{(x, ?), (y, 6), (z, 2), (z, 4)\}$

sub  $\frac{\vdash S : RD'_1 \rightarrow RD'_2}{\vdash S : RD_1 \rightarrow RD_2}$  if  $RD_1 \subseteq RD'_1$  and  $RD'_2 \subseteq RD_2$

seq  $\frac{\vdash S_1 : RD_1 \rightarrow RD_2 \quad \vdash S_2 : RD_2 \rightarrow RD_3}{\vdash S_1; S_2 : RD_1 \rightarrow RD_3}$

$\vdash [y := 0]^6$   
 $RD \rightarrow \{(x, ?), (y, 6), (z, 2), (z, 4)\}$

$\vdash while [y > 1]^3 do [z = z * y]^4; [y := y - 1]^5 od; [y := 0]^6 : RD \rightarrow RD$

SUB

$while [y > 1]^3 do [z = z * y]^4; [y := y - 1]^5 od; [y := 0]^6 :$   
 $\{(x, ?), (y, 1), (z, 2)\} \rightarrow \{(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)\} = RD$

$while [y > 1]^3 do [z = z * y]^4; [y := y - 1]^5 od; [y := 0]^6 :$

$\{(x, ?), (y, 1), (z, 2)\} \rightarrow \{(x, 2), (y, 6), (z, 2), (z, 4)\}$

seq

$$\text{seq} \frac{\frac{\vdash S_1 : RD_1 \rightarrow RD_2 \quad \vdash S_2 : RD_2 \rightarrow RD_3}{\vdash S_1; S_2 : \underline{RD_1} \rightarrow \underline{RD_3}}}{\vdash S : RD \rightarrow RD} \quad \text{sub} \frac{\vdash S : RD'_1 \rightarrow RD'_2}{\vdash S : RD_1 \rightarrow RD_2} \text{ if } RD_1 \subseteq RD'_1 \text{ and } RD'_2 \subseteq RD_2$$

$$\text{while} \frac{\vdash S : RD \rightarrow RD}{\vdash \text{while } [b]^\ell \text{ do } S \text{ od} : RD \rightarrow RD}$$

We are left to prove that

$$\text{while}[y := 1]^3 \text{ do } [z = z * y]^4; [y := y - 1]^5 \text{ od} : RD \rightarrow RD$$

Abbreviation:  $RD = \{(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)\}$

$$\frac{\frac{\vdash [z := z * y]^4 : RD \rightarrow \{(x, ?), (y, 1), (y, 5), (z, 4)\}}{\vdash [y := y - 1]^5 : \{(x, ?), (y, 1), (y, 5), (z, 4)\} \rightarrow \{(x, ?), (y, 5), (z, 4)\}}}{\vdash [z := z * y]^4; [y := y - 1]^5 : RD \rightarrow \{(x, ?), (y, 5), (z, 4)\}} \text{ seq}$$

$$\frac{\vdash [z := z * y]^4; [y := y - 1]^5 : RD \rightarrow RD \text{ using } \{(x, ?), (y, 5), (z, 4)\} \subseteq RD}{\vdash \text{while } [y > 1]^3 \text{ do } [z := z * y]^4; [y := y - 1]^5 \text{ od} : RD \rightarrow RD} \text{ while}$$

# How to automate the analysis

