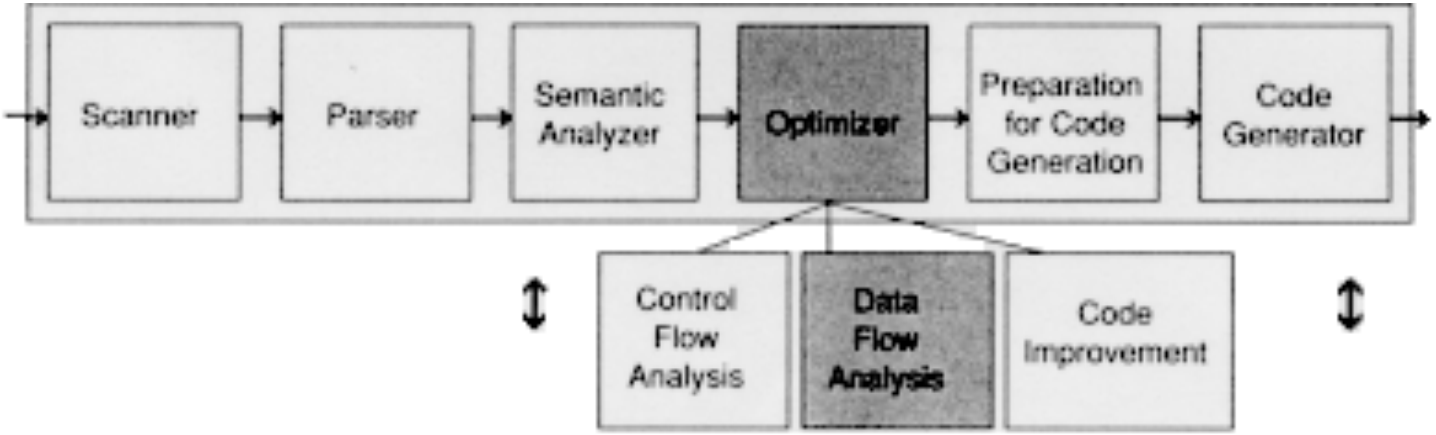
Dataflow Analyses

Code Optimization in Compilers



Correctness Above All!

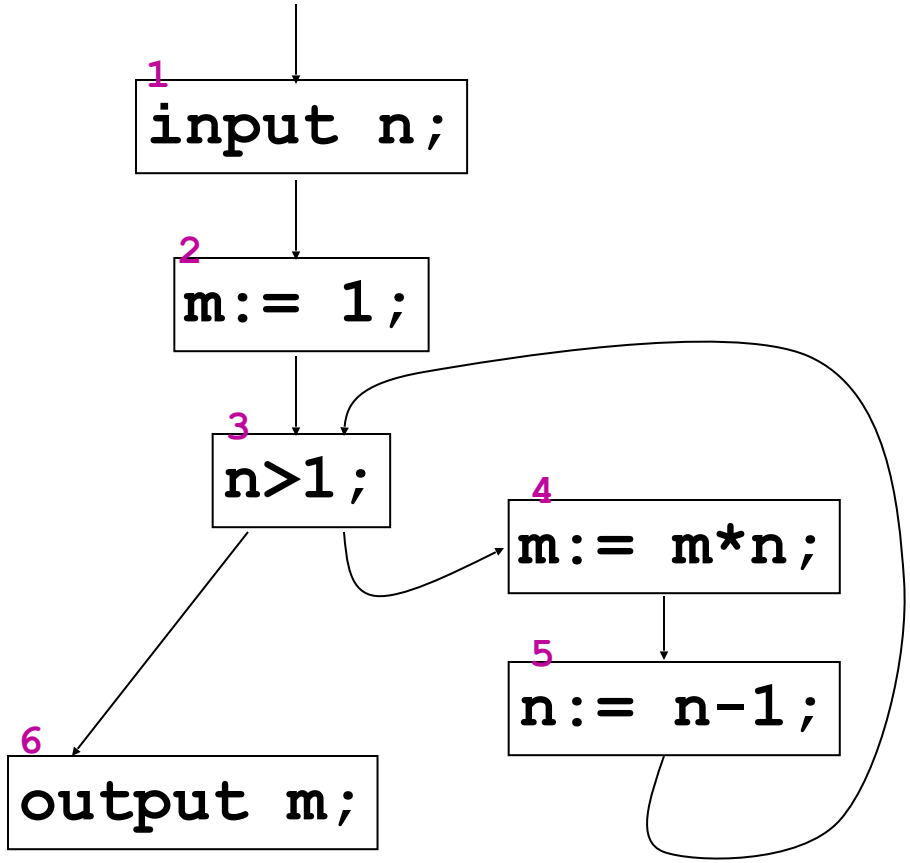
If may seem obvious, but it bears repeating that optimization should not change the correctness of the generated code. Transforming the code to something that runs faster but incorrectly is of little value. It is expected that the unoptimized and optimized variants give the same output for all inputs. 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.

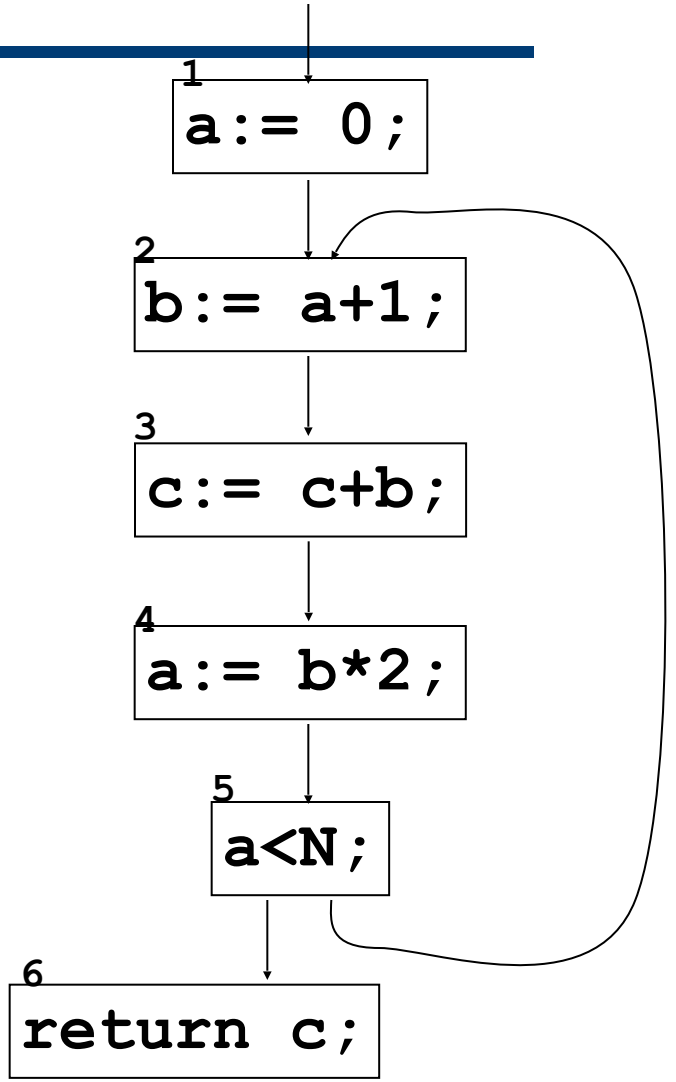
For example: t1 := t2 + t3.

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

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

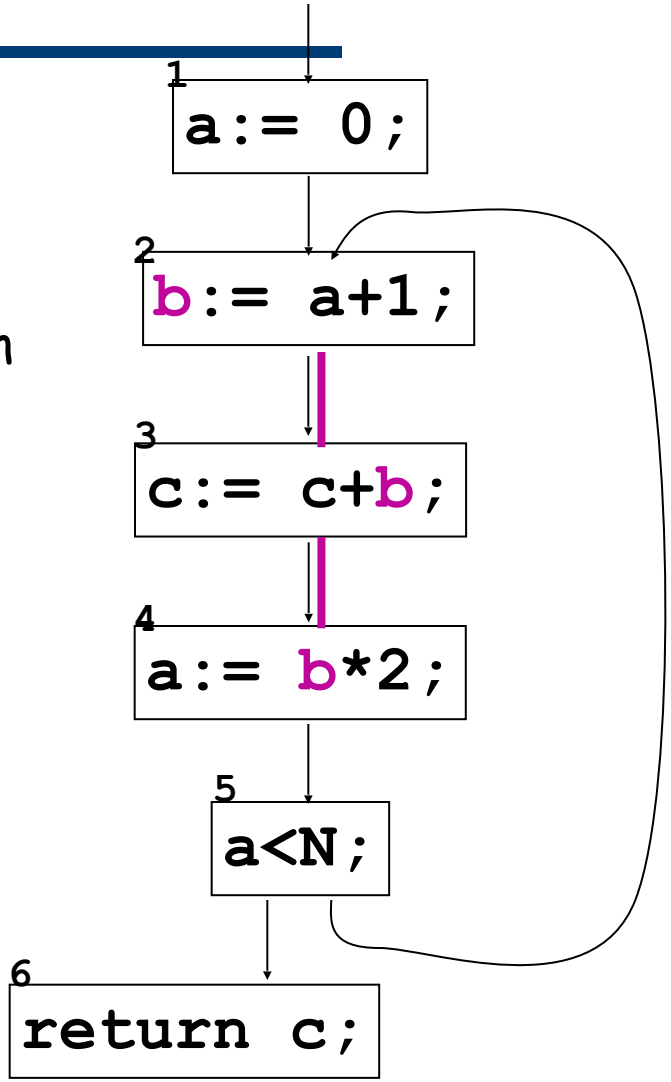
- A variable X is live at the exit of a command C if X stores a value which will be actually used in the future, that is, X will be used as R-value with no previous use as L-value

- 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

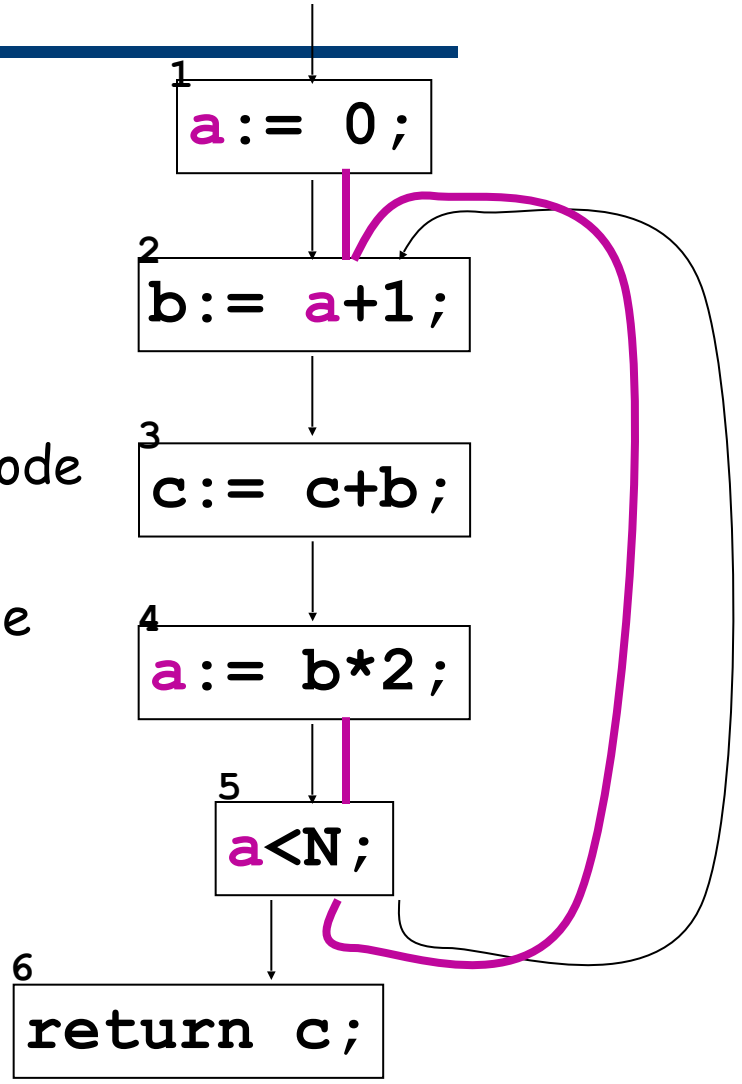
Back to the example

- 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 → 4
- Command 3 does not assign **b**, hence **b** is live along 2 → 3
- Command 2 assigns **b**. This means that the value of **b** along 1 → 2 will not be used later
- Thus, the "live range" of **b** turns out to be: {2 → 3, 3 → 4}



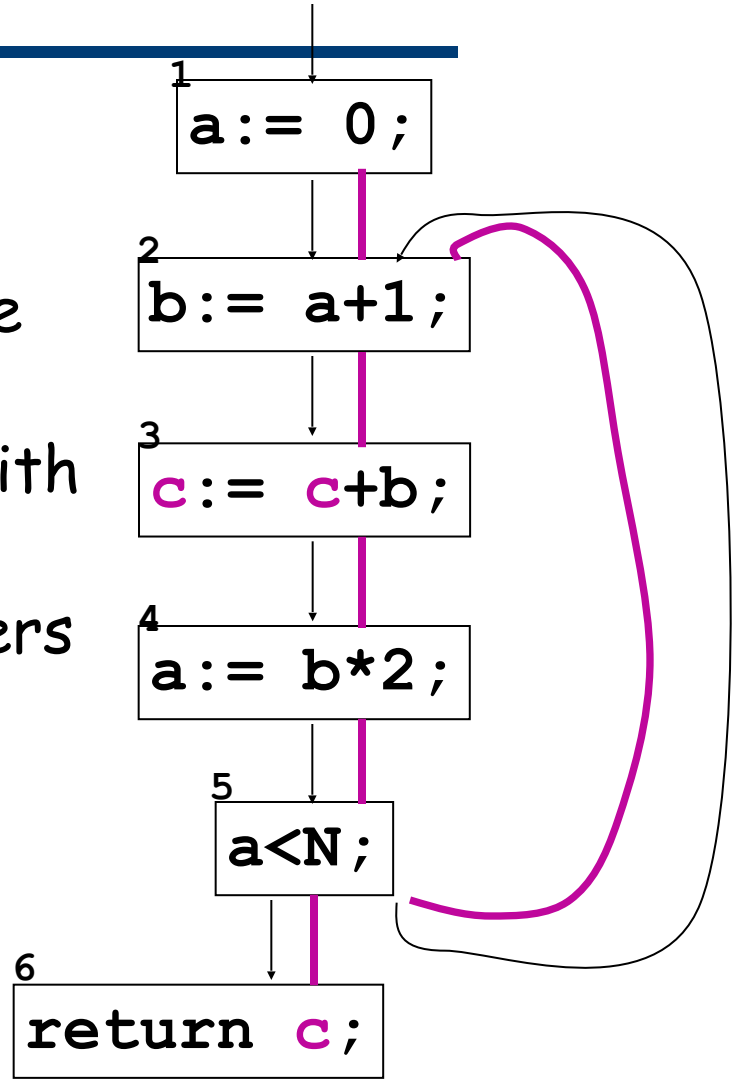
Live variables

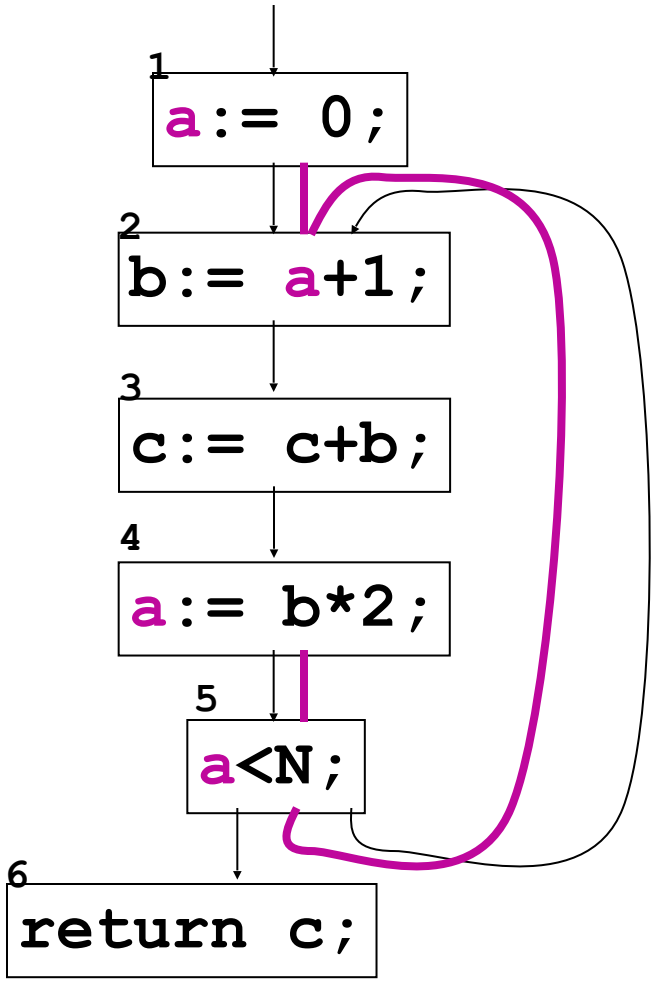
- **a** is live along 4 → 5 and 5 → 2
- **a** is live along 1 → 2
- **a** is not live along 2 → 3 and 3 → 4
- Even if the variable **a** stores a value in node 3, this value will not be later used, since node 4 assigns a new value to the variable **a**.

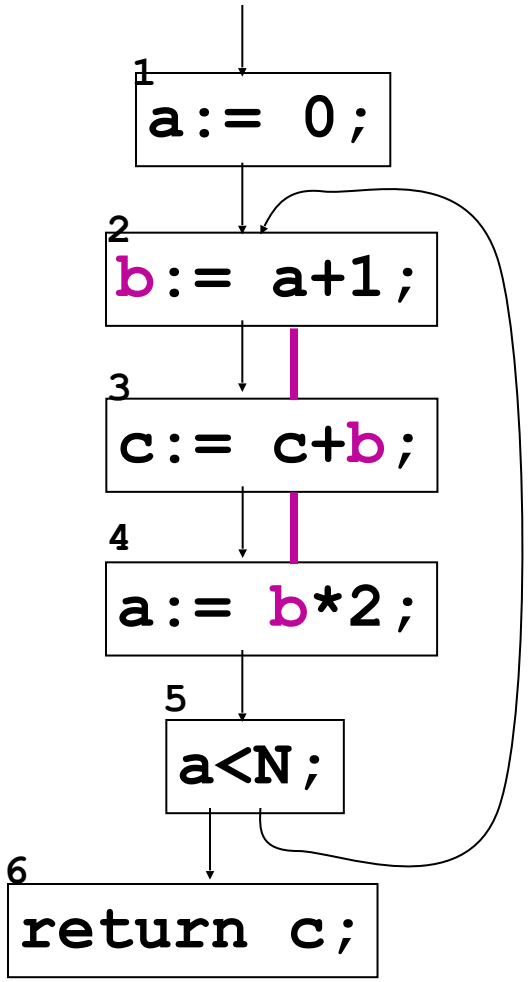


More on live variables

- **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)

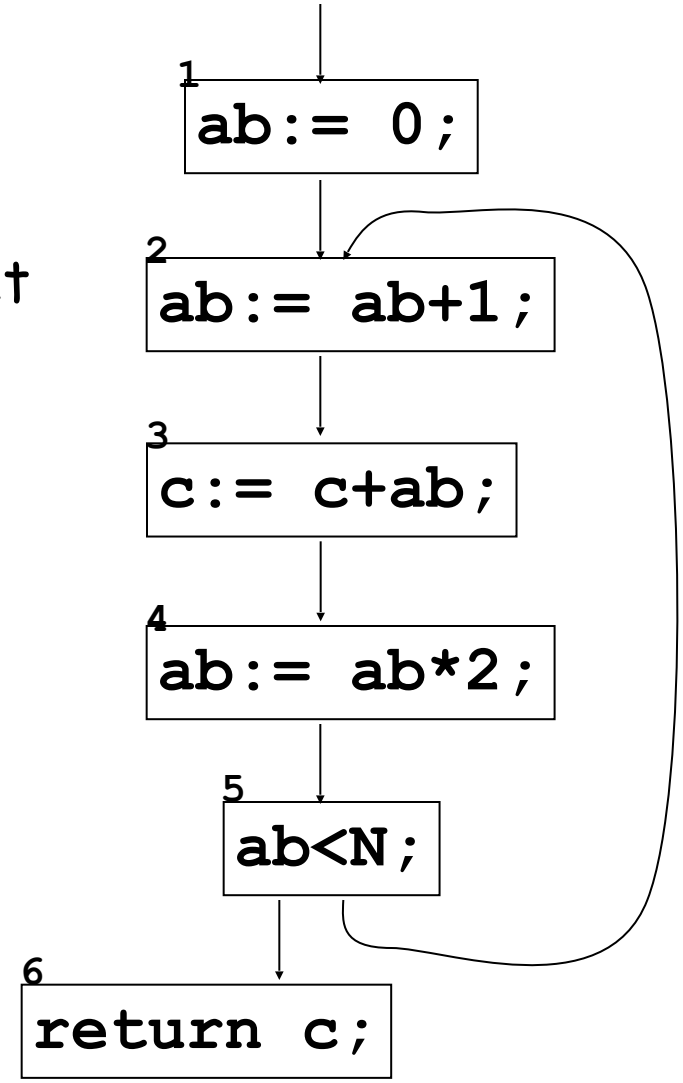






➔ Two registers are enough: variables **a** and **b** will be never simultaneously live along the same arc

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**

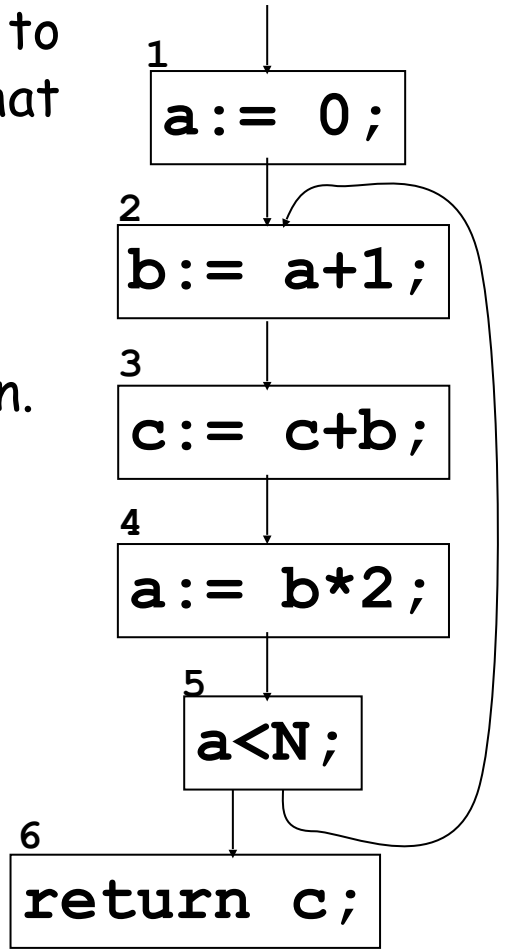


We need a way to compute live variables

- 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
  - pre[2]={1,5}; post[5]={2,6}.

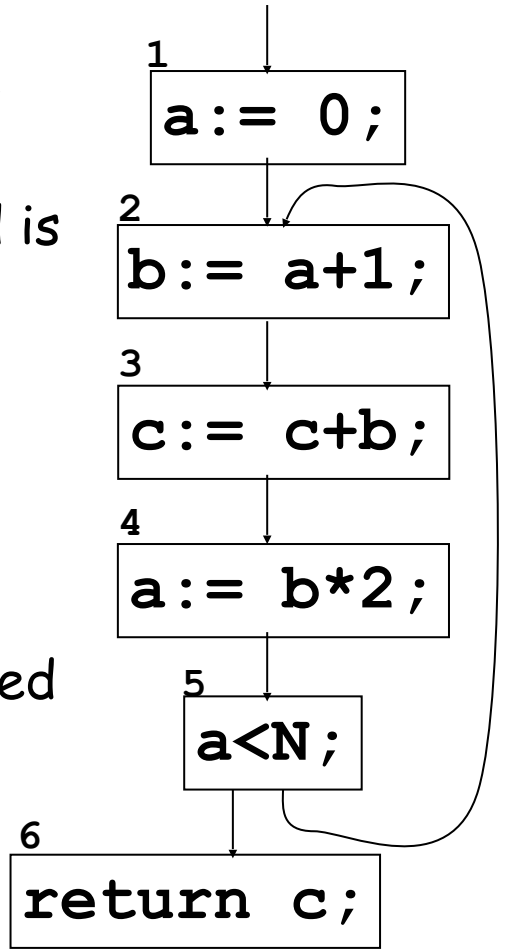


Notation

- 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:
  - def[3]={**c**}, def[5]= $\emptyset$
  - use[3]={**b**,**c**}, use[5]={**a**}

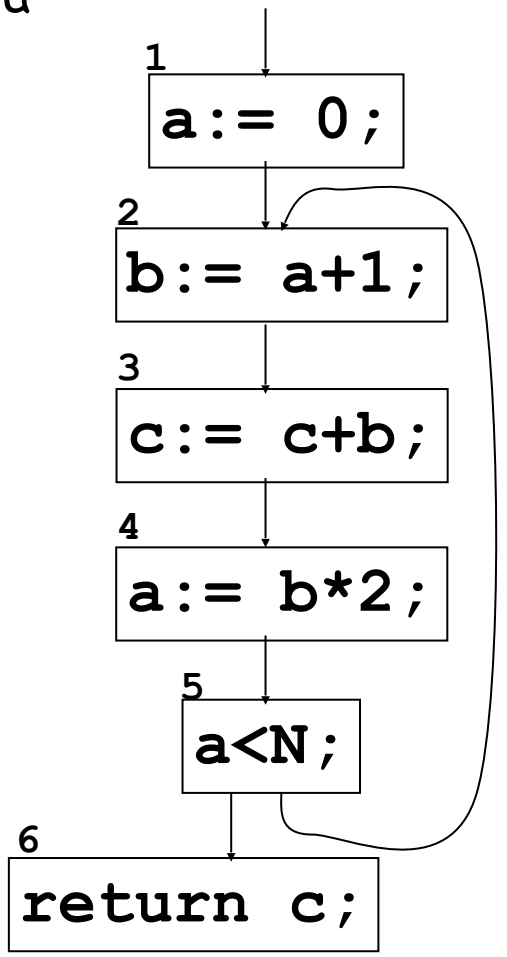


Formalization of the property:

- A variable x is live along an arc e→f if there exists a real execution path P from the node e to some node n such that:
  - e→f is the first arc of such path P
  - x ∈use[n]
  - for any node n'≠e and n'≠n in the path P, x∉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:

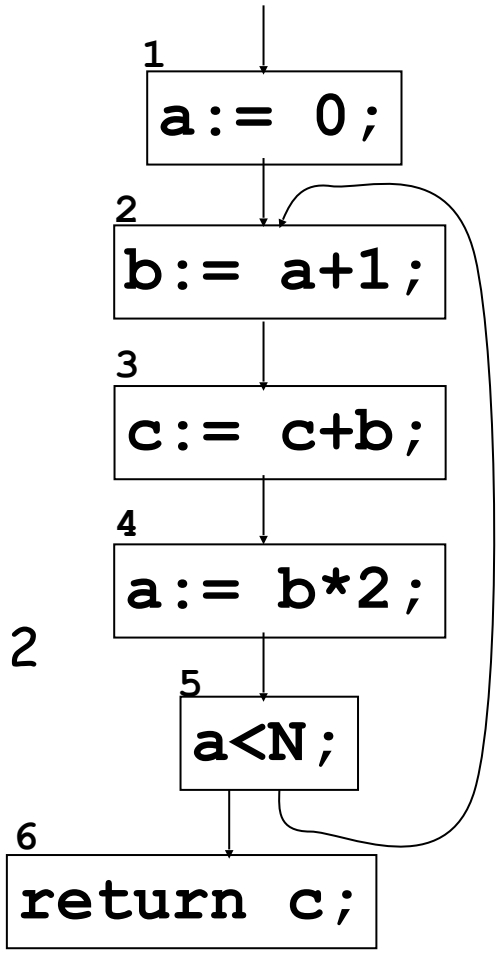
a is live along 1 → 2, 4 → 5 and 5 → 2

b is live along 2 → 3, 3 → 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 an approximation of Liveness property

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

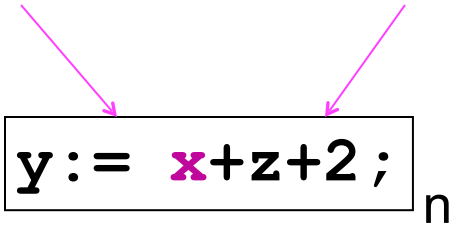
out[n] is the set of variables that the static analysis determines to be live-out at node n

Computing an approximation of Liveness property

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

n node of the CFG

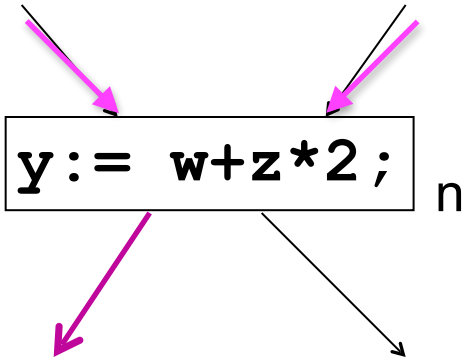
2. If a variable x∈use[n] then x is live-in in node n.
In other terms, if a node n uses a variable x as R-value then this variable x is live along each arc that enters into n.



in[n] ⊇ use[n]

Computing Liveness

2. If a variable x is live-out in a node n and x $\notin$ 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

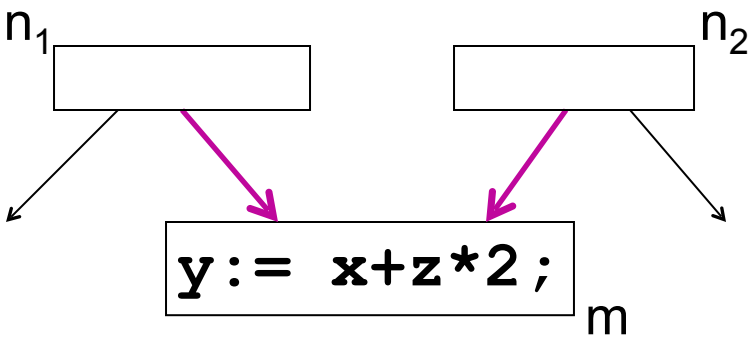


in[n] $\supseteq$ out[n] - def[n]

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∈post[n].

   This is clearly correct by definition.



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

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

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 the analysis 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]):

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

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 OUT$[B]$.

2. $x + y$ is never computed along some path. Since $x + y$ is never in OUT[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.

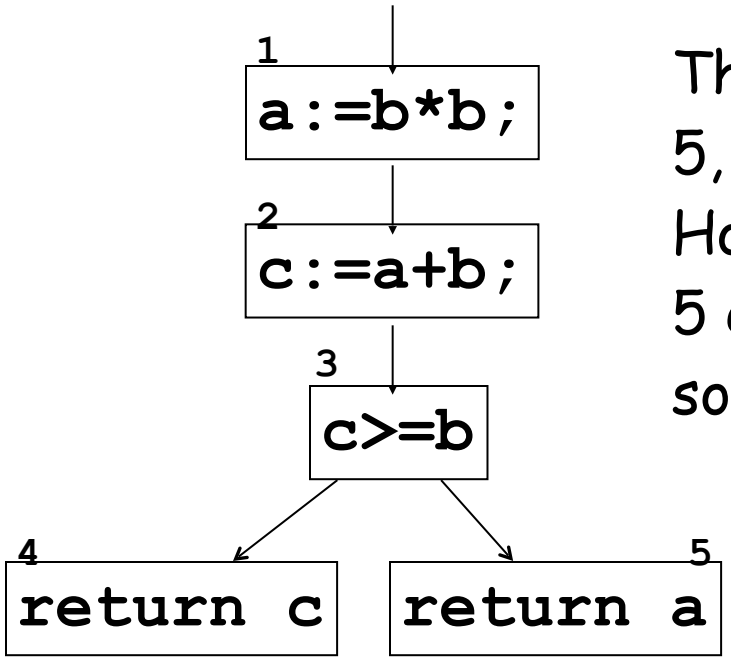
Computing Liveness

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 for each note the sets that satisfy the previous equations: fix points

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

  It depends on the domain and on the function!

Questions

Does a solution of the previous equations always exist?

If it exists, is it unique?

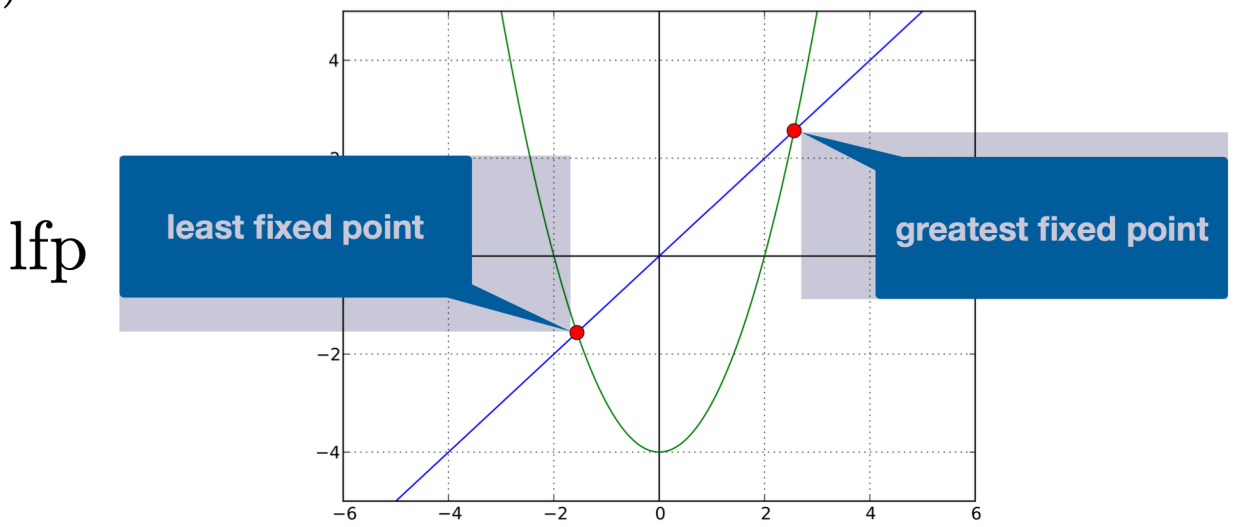
How to compute it?

Fixpoint?

fixF = X such that F(X) = X

$$x^2 - 4 = x$$

$$F(x) = x^2 - 4$$



The fix point theory

Definition (Partial Order). A binary relation $\sqsubseteq$ is a partial order on a set D

if it holds:

1. reflexivity: $a \sqsubseteq a$ for all $a \in D$

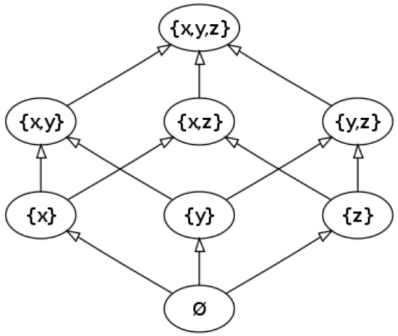
2. Antisymmetry: $a \sqsubseteq b$ and $b \sqsubseteq a$ implies $a = b$

3. Transitivity: $a \sqsubseteq b$ and $b \sqsubseteq c$ implies $a \sqsubseteq c$

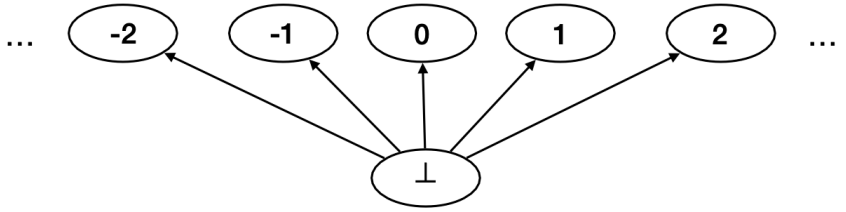
A set D with a partial order $\sqsubseteq$ is called a partially ordered set $(D, \sqsubseteq)$, or simply poset.

Examples

- Example 1: $(\wp(\{x, y, z\}), \subseteq)$

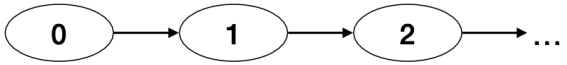


- Example 2: $(\mathbb{Z}_\perp, \sqsubseteq)$



Hasse diagrams

- Example 3: $(\mathbb{N}, \leq)$



- Example 4: $(\mathbb{N} + \{+\infty\}, \leq)$

Least Upper Bound

Definition (Least Upper Bound).

For a partial ordered set $(D, \sqsubseteq)$ and $X \subseteq D$,

$d \in D$ is an upper bound of X iff $\forall x \in X.\ x \sqsubseteq d$

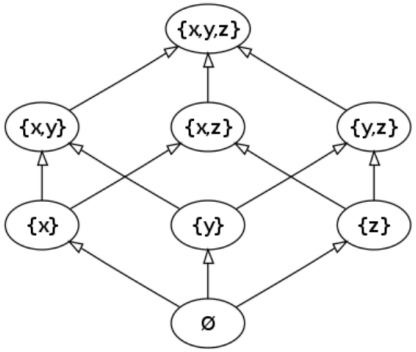
An upper bound d is the least upper bound of X iff for all upper bounds y of X

$$d \sqsubseteq y$$

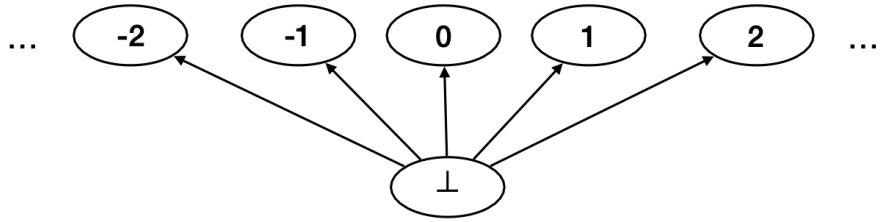
The least upper bound of X is denoted by $\bigsqcup X$

Examples

- Example 1: $(\wp(\{x, y, z\}), \subseteq)$

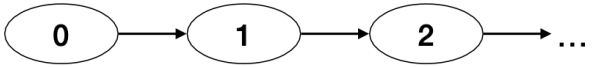


- Example 2: $(\mathbb{Z}_\perp, \sqsubseteq)$



Hasse diagrams

- Example 3: $(\mathbb{N}, \leq)$



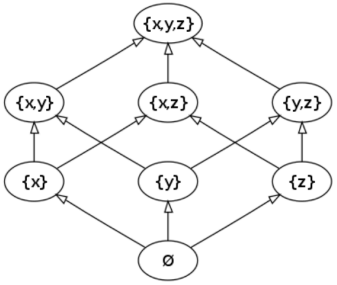
- Example 4: $(\mathbb{N} + \{+\infty\}, \leq)$

Chain

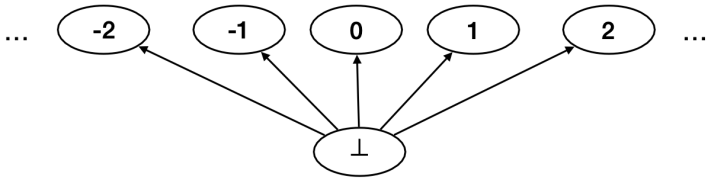
Definition (Chain). Let $(D, \sqsubseteq)$ be a partial ordered set. A subset $X \subseteq D$, is called a **chain** if X is totally ordered:

$$\forall x_1, x_2 \in X. x_1 \sqsubseteq x_2 \text{ or } x_2 \sqsubseteq x_1$$

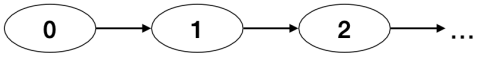
- Example 1: $(\wp(\{x, y, z\}), \subseteq)$



- Example 2: $(\mathbb{Z}_\perp, \sqsubseteq)$



- Example 3: $(\mathbb{N}, \leq)$



- Example 4: $(\mathbb{N} + \{+\infty\}, \leq)$

Complete Partial Order CPO

Definition (CPO). A poset $(D, \sqsubseteq)$ is a **CPO** (complete partial order) if every chain X of D has a

$$\bigsqcup X \in D$$

We consider CPOs $(D, \sqsubseteq)$ with a **least element** $\bot$

On finite sets

If a poset $(D, \sqsubseteq)$ is finite than it is a **CPO**

Monotone Functions

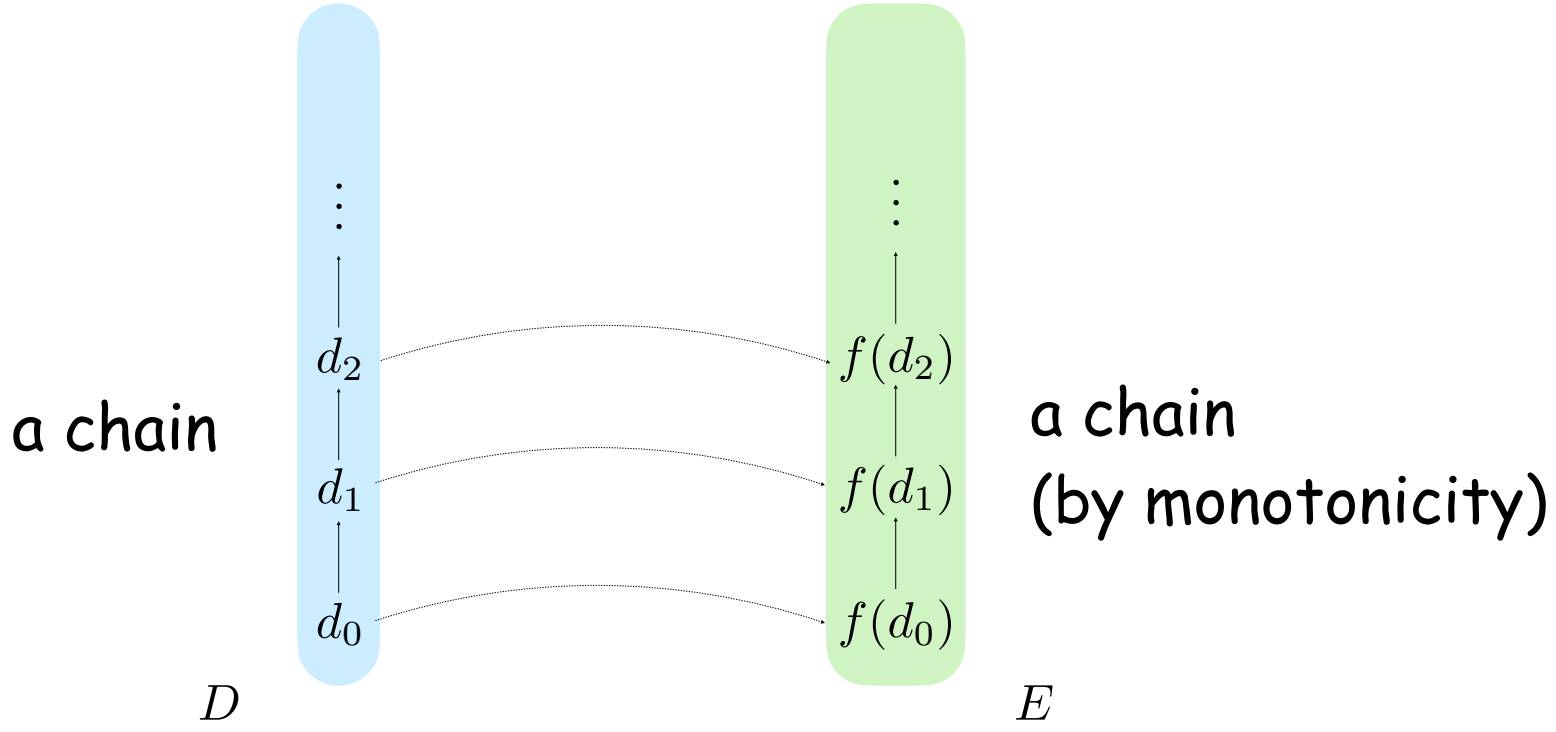
Definition (Monotone Function). Given two partially ordered sets D and E

a function f : $D \to E$ is **monotone** if it preserves orders between any two elements in D

$$\forall d_1, d_2 \in D.\ d_1 \sqsubseteq d_2 \Longrightarrow f(d_1) \sqsubseteq f(d_2)$$

Monotonicity illustrated

$f : D \to E$ monotone

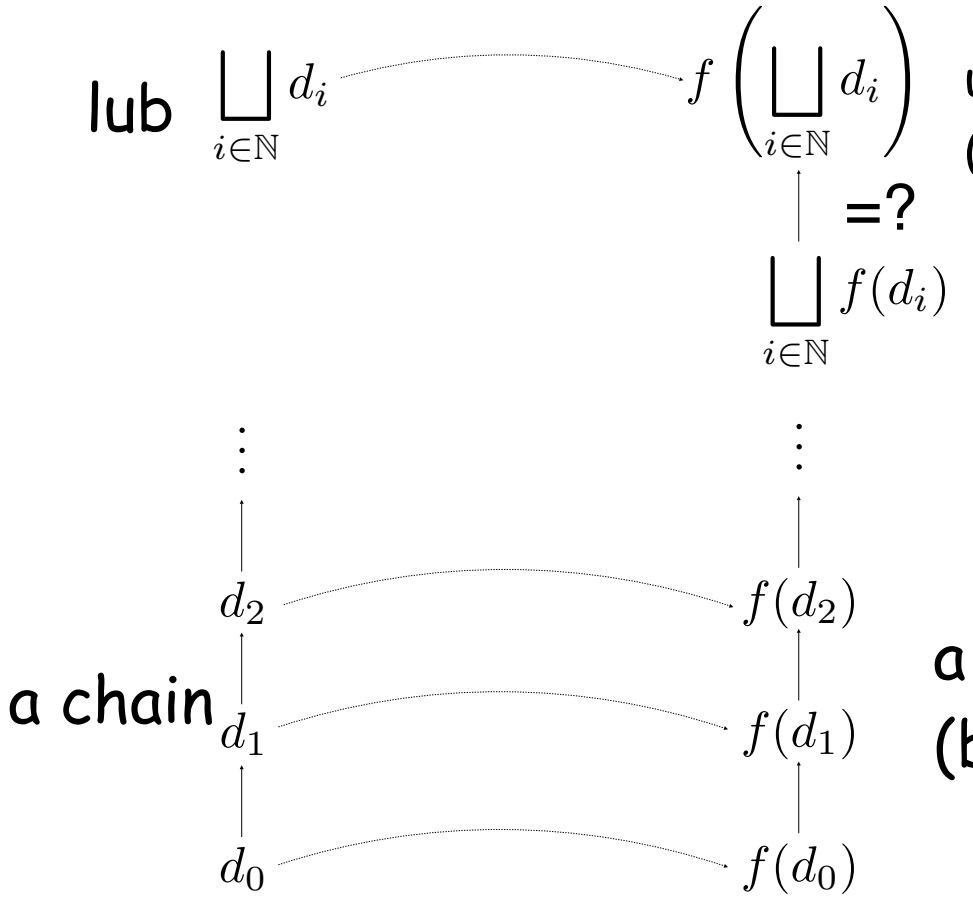


Continuous Functions

Definition (Continuous Function). Given two CPO's D and E, a monotone function f : $D \to E$ is **continuous** if it preserves least upper bounds of chains

$$\forall chain\ X \in D. \bigsqcup_{d \in X} f(d) = f(\bigsqcup X)$$

Continuity illustrated

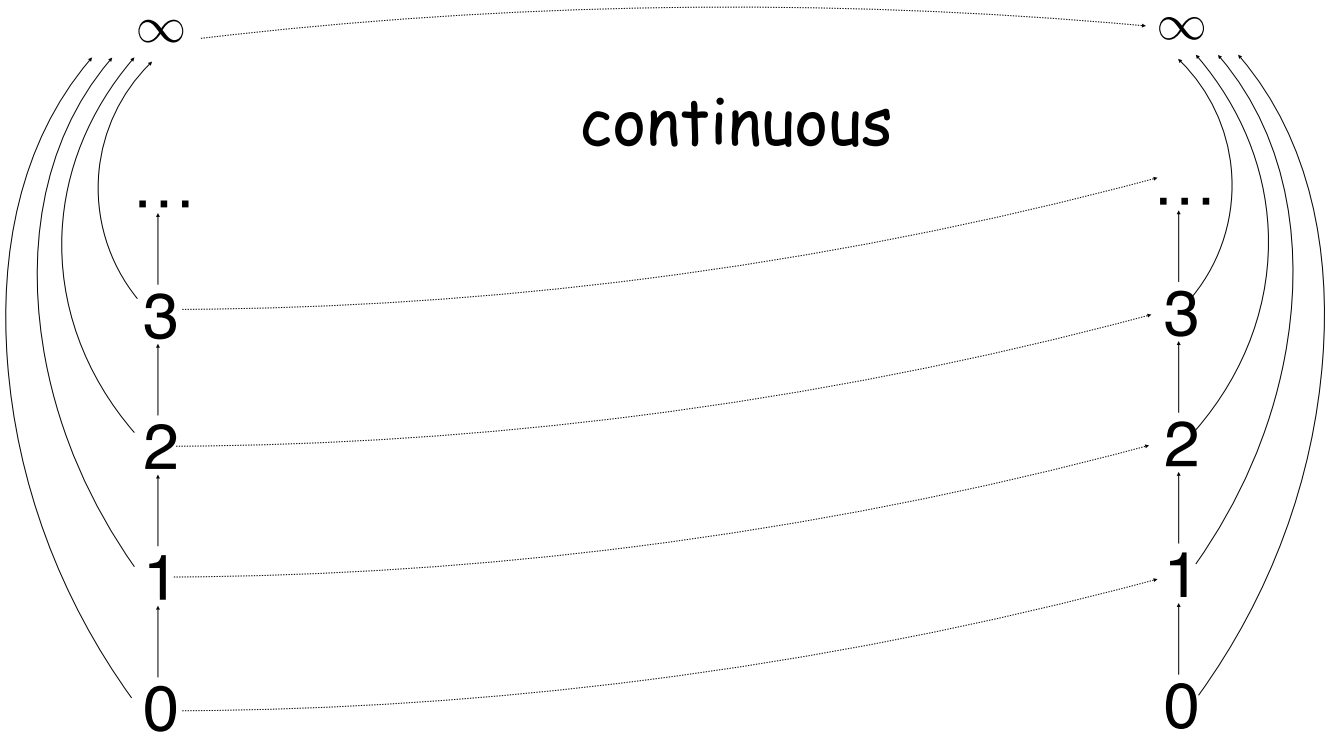


Example

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

$f(n) = n + 1$
$f(\infty) = \infty$

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



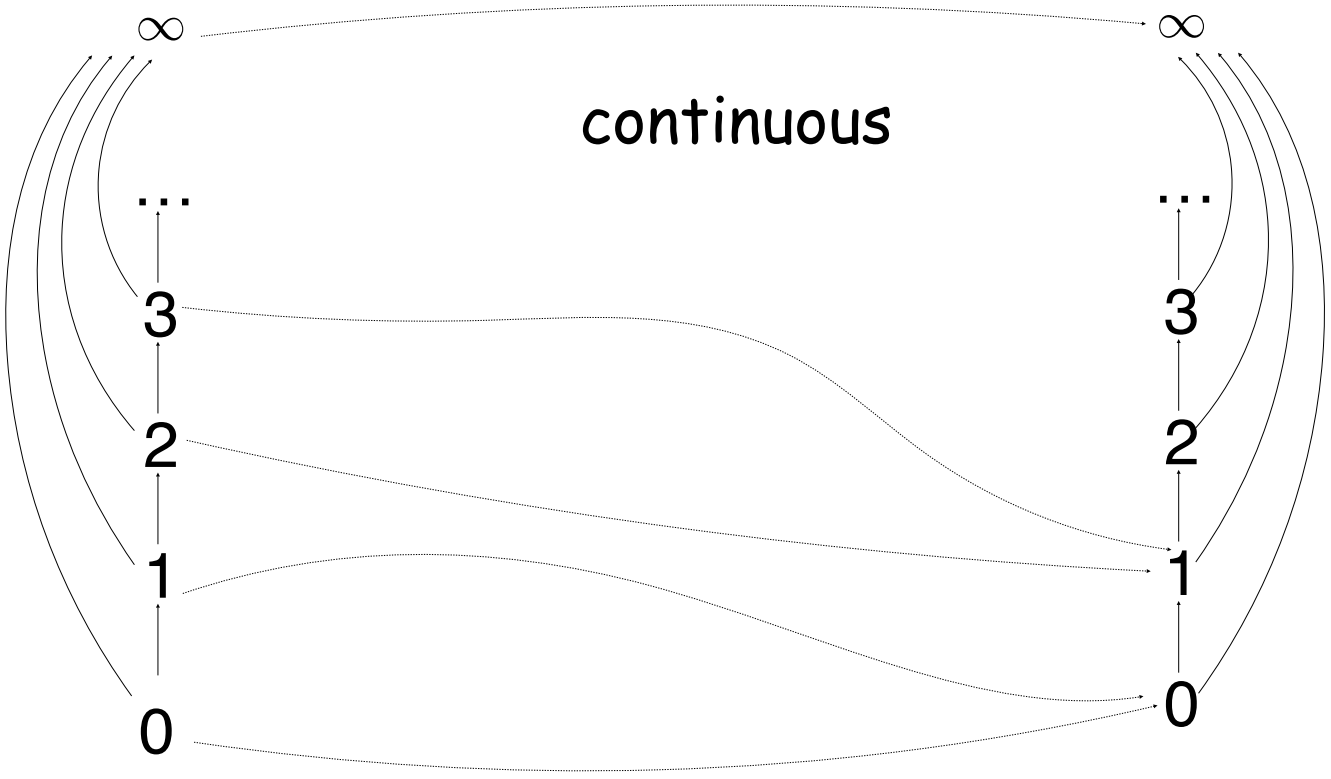
Example

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

$f(n) = n/2$

$f(\infty) = \infty$

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

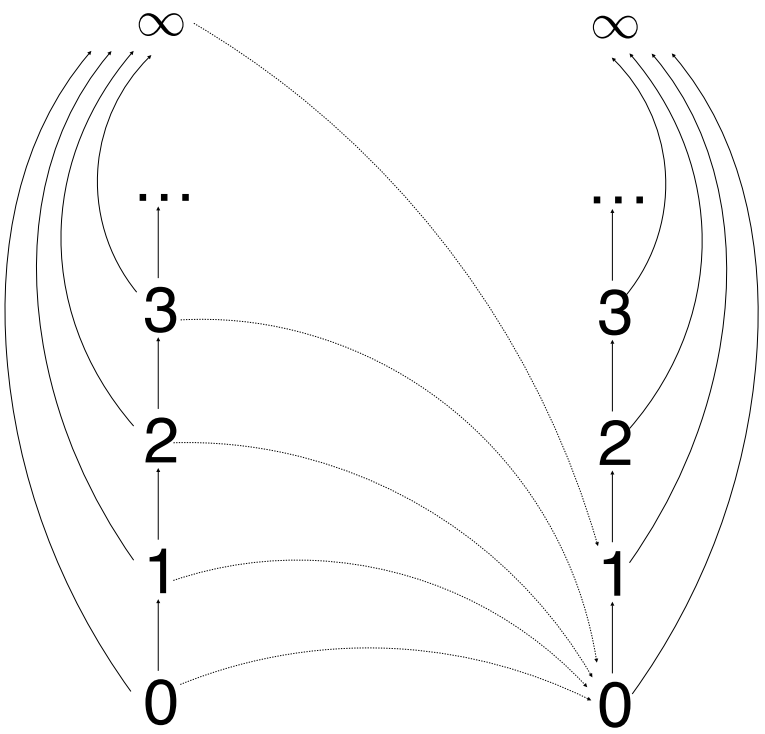


Example

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

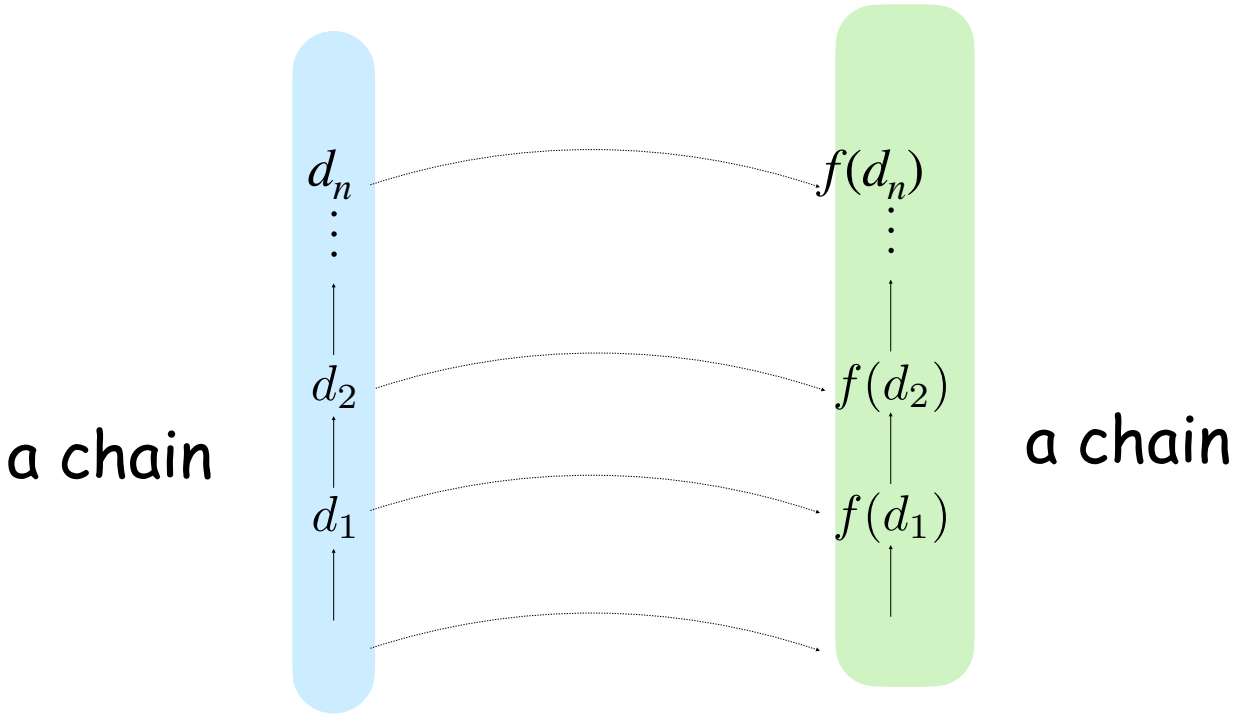
monotone function, not continuous

$$f(x) = \begin{cases} 0 & \text{if } x \in \mathbb{N} \\ 1 & \text{if } x = \infty \end{cases}$$



On finite sets

Given two CPO's D and E, if D is finite then a **monotone** function f : $D \to E$ is also **continuous.**



Repeated application

$f : D \to D$

$$f^0(d) \triangleq d$$

$$f^{n+1}(d) \triangleq f(f^n(d))$$

$$f^n(d) = \overbrace{f(\cdots(f}^{n \text{ times}}(d))\cdots)$$

$$f^n : D \to D$$

Lemma $(D, \sqsubseteq)$ $\mathrm{PO}_\perp$ $f : D \to D$ monotone $\Rightarrow$ $\{f^n(\perp)\}_{n \in \mathbb{N}}$ is a chain

Example

$(\mathbb{N} \cup \{\infty\}, \leq)$ $\quad f(n) = n + 1$ $\quad$ continuous

$f(\infty) = \infty$

$\perp = 0$

∞

...

$f^3(\perp) = f^3(0) = f(f(f(0))) = f(2) = 3$

$f^2(\perp) = f^2(0) = f(f(0)) = f(1) = 2$

$f^1(\perp) = f^1(0) = 1$

$f^0(\perp) = f^0(0) = 0$

Towards Kleene's

when $(D, \sqsubseteq)$ is a $\mathrm{CPO}_\perp$

then $\{f^n(\perp)\}_{n\in\mathbb{N}}$ is a chain

it must have a limit

$\{f^n(d)\}_{n\in\mathbb{N}}$ not necessarily a chain!

Kleene's fix point theorem states that if f is continuous, then the limit of the above chain is the least fixpoint of f

Example

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

$\bot = 0$

$f(0) = 0$
$f(n) = n - 1$
$f(\infty) = \infty$

continuous

$$\infty$$

$$\vdots$$

$$f^3(\bot) = f^3(0) = f(0) = 0$$

$$f^2(2) = f(1) = 0$$

$$f^1(2) = 1$$

$$f^0(2) = 2$$

Kleene 's Theorem (simplified)

$(D, \sqsubseteq)$ $\mathrm{CPO}_\perp$ $f : D \to D$ continuous

let $fix(f) \triangleq \bigsqcup_{n \in \mathbb{N}} f^n(\perp)$

1. $fix(f)$ is a fix point of f

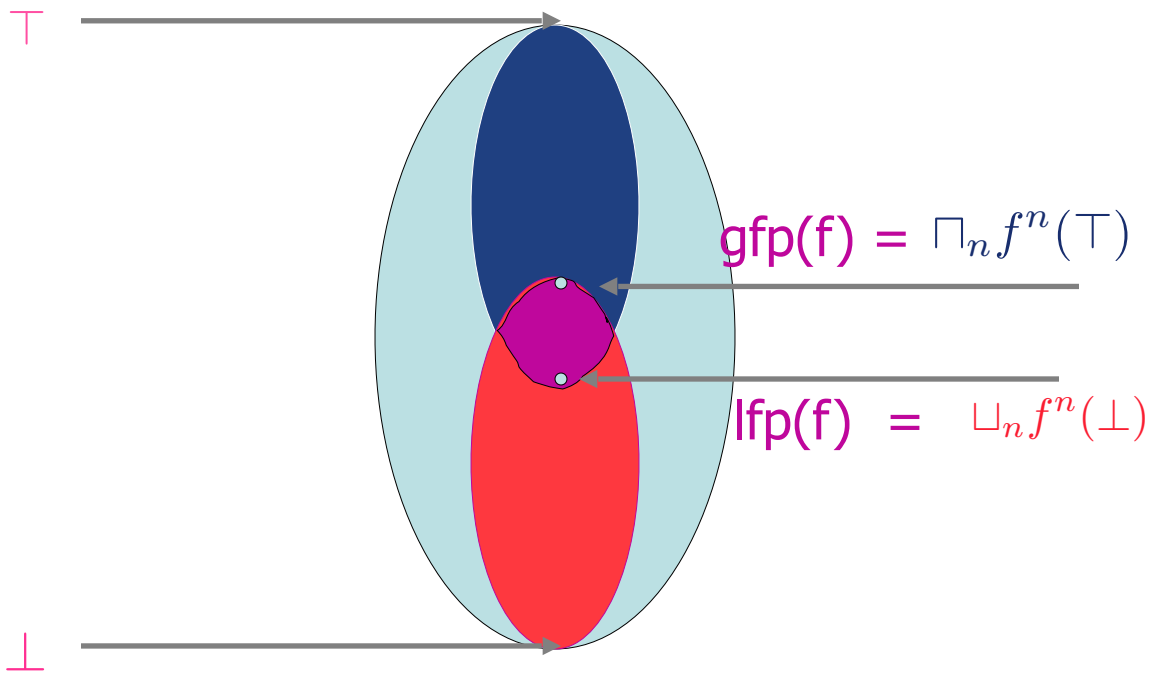
$$f(fix(f)) = fix(f)$$

2. $fix(f)$ is the least fixpoint of f

$$\forall d \in D.\ f(d) = d \Rightarrow fix(f) \sqsubseteq d$$

if d is a fixpoint then $fix(f)$ is smaller than d

Kleene's Theorem



1. in[n] = use[n] ∪ (out[n] - def[n])
2. out[n] = ∪ {in[m] | m ∈ post[n]}

Which is our domain?

Our objects:

Given a node we need to compute the set in and the set out (sets of variables)

- Let **Vars** be the finite set of variables that occur in the program P to analyze. All possible subsets: $\mathcal{P}$**(Vars)**

For each node we need a set in and a set out: $\mathcal{P}$**(Vars)x** $\mathcal{P}$**(Vars)**

Our CFG has **N** nodes, so our domain will be

($\mathcal{P}$(Vars)x $\mathcal{P}$(Vars))^N : N-tuples of pairs of subsets of **Vars**

The order : $\subseteq^{2N}$

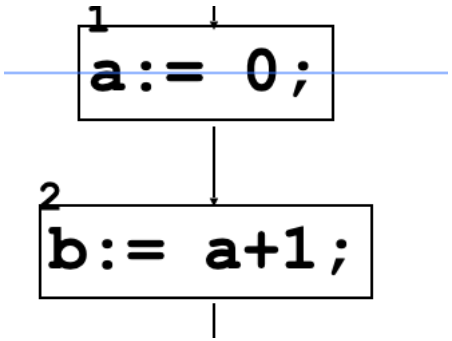
$$< in_1^1, out_1^1, \ldots, in_N^1, out_N^1 > \subseteq^{2N} < in_1^1, out_1^1, \ldots, in_N^1, out_N^1 > \text{ iff}$$

$$\forall i, \quad in_i^1 \subseteq in_i^2 \text{ and } out_i^1 \subseteq out_i^2$$

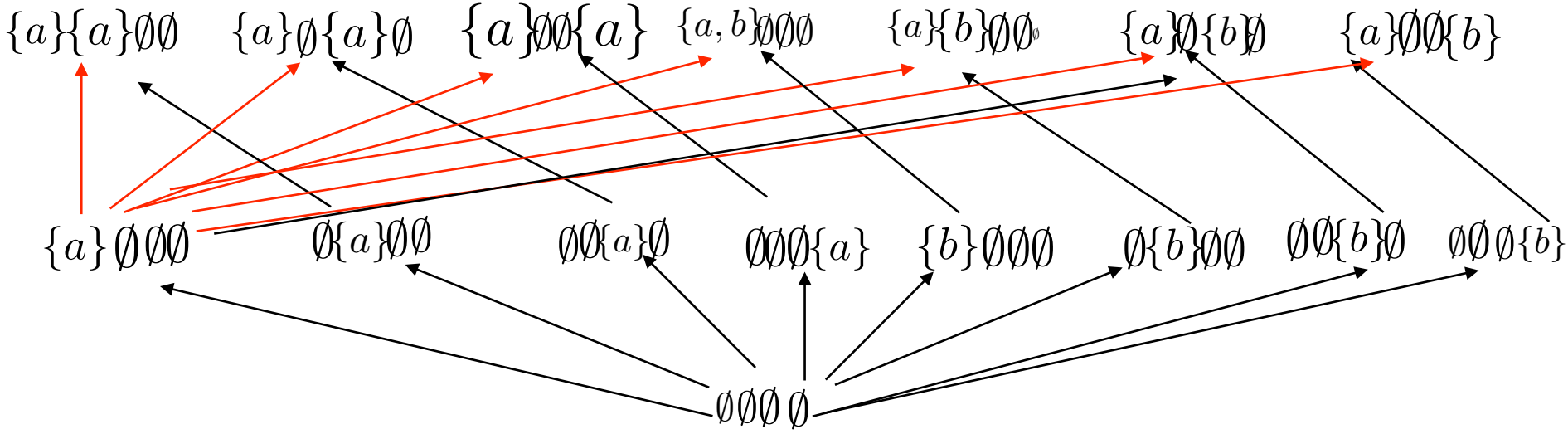
Example

Vars ={a,b} N=2.

$\langle (\mathcal{P}(\text{Vars}) \times \mathcal{P}(\text{Vars}))^2, \subseteq^4 \rangle$ is a finite domain.



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



Our domain

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

CPO with bottom?

It is a CPO because it is finite

bottom?

1. in[n] = use[n] ∪ (out[n] - def[n])
2. out[n] = ∪ {in[m] | m ∈ post[n]}

Which is our function?

The map Live:

((**Vars**)x $\mathcal{P}$(**Vars**))^N -> ($\mathcal{P}$ (**Vars**)x$\mathcal{P}$(**Vars**))^N defined by

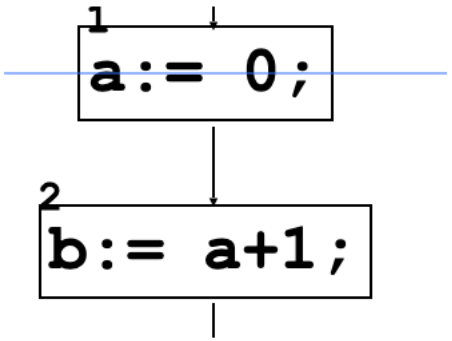
Live(<in_1,out_1,…,in_N,out_N>)=

<use[1]∪(out$_1$-def[1]),$\bigcup_{m\in post[1]}$ in_m, …., use[N]∪ (out$_N$ -def[N]), $\bigcup_{m\in post[N]}$ in_m>

- Example

Live(<{}{}{}{}>)=<{}{}{a}{}>
Live(<{}{}{a}{}>)=<{}{a}{a}{}>
Live(<{}{a}{a}{}>)=<{}{a}{a}{}>

Note that Live is monotone!



def(1)={a}, use(1)={}
def(2)={b}, use(2)={a}

Is it continuous?

The map Live:

((**Vars**)x $\mathcal{P}$(**Vars**))^N -> ($\mathcal{P}$ (**Vars**)x$\mathcal{P}$(**Vars**))^N defined by

Live(<in_1,out_1,…,in_N,out_N>)=

<use[1]∪(out_1-def[1]), $\bigcup_{m \in post[1]} in_m$, …., use[N]∪ ($out_N$ -def[N]), $\bigcup_{m \in post[N]} in_m$>

is continuous?

Yes! because it is monotone on a finite domain

In conclusion

The map Live:

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

Live($\langle in_1, out_1, \ldots, in_N, out_N \rangle$)=

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

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

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

a least fixpoint

Why a least fixpoint?

- Live is a possible analysis,

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

i.e., if a variable x will be really live in a node n during some program execution then x belongs to 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 (the more precise over approximations)

Conservative Approximation

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

in[n] ⊇ live-in[n] and **out[n] ⊇ live-out[n]**

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

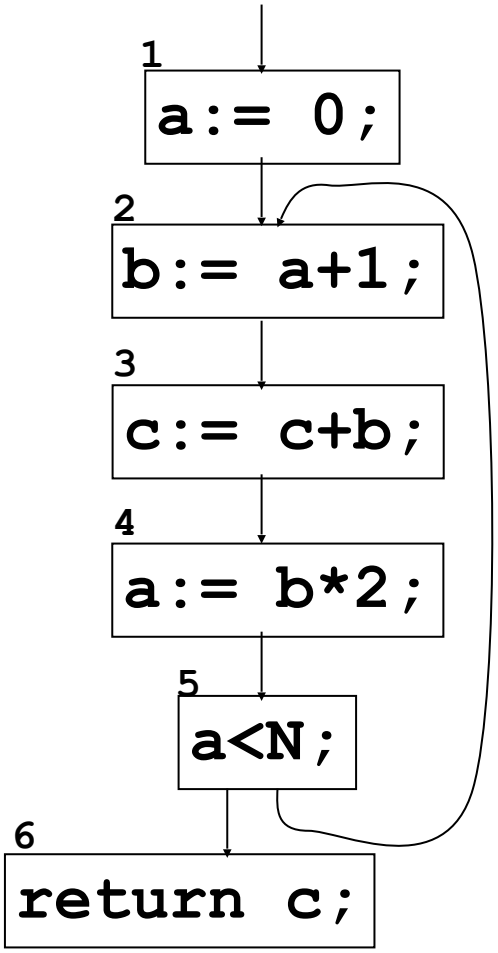
- The converse does not hold: the analysis can tell us that x is in the computed set out[n], but this does not imply that x will be necessarily live in n during some program execution
- In liveness analysis "conservative approximation" means that the analysis may erroneously derive that a variable is live, while the analysis is not allowed to erroneously derive that a variable is "dead" (i.e., not live).

★if $x \in$ in[n] then x could be live at program point n.

★if $x \notin$ in [n] then x is definitely dead at program point n.

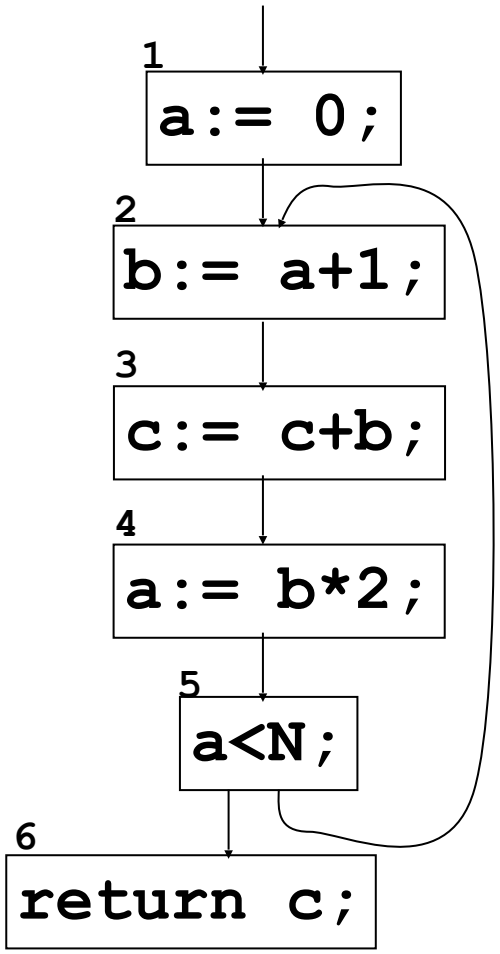
```
for all n
    in[n]:={} out[n]:={};
repeat
    for all n (1 to 6)
          in'[n]:=in[n]; out'[n]:=out[n];
          in[n]:= use[n] U (out[n] - def[n]);
          out[n]:= U { in[m] | m ∈ post[n]};
until (for all n: in'[n]=in[n] && out'[n]=out[n])
```

			Live^1		Live^2		Live^3	
	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	

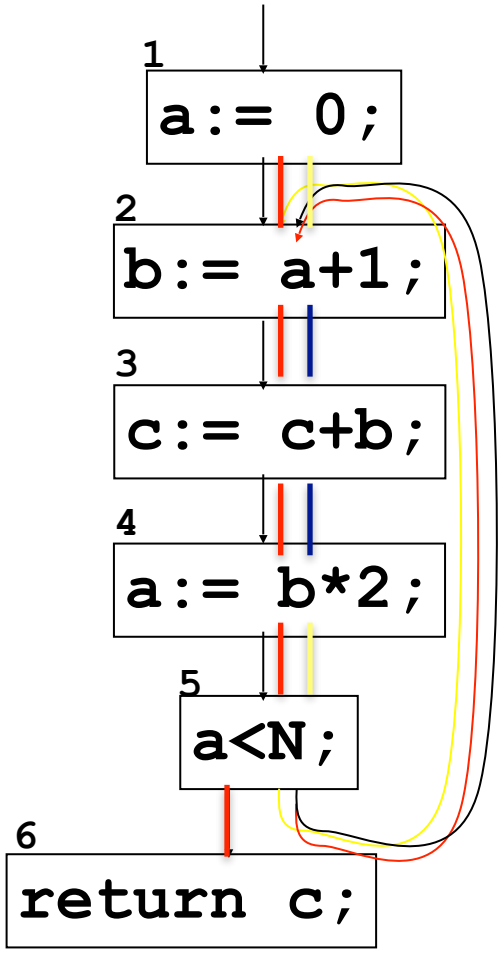


```
for all n
    in[n]:=?; out[n]:=?;
repeat
    for all n (1 to 6)
          in'[n]:=in[n]; out'[n]:=out[n];
          in[n]:= use[n] U (out[n] - def[n]);
          out[n]:= U { in[m] | m ∈ post[n]};
until (for all n: in'[n]=in[n] && out'[n]=out[n])
```

			Live^3		Live^4		Live^5	
	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[5]		Live[6]		Live[7]	
	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])
```

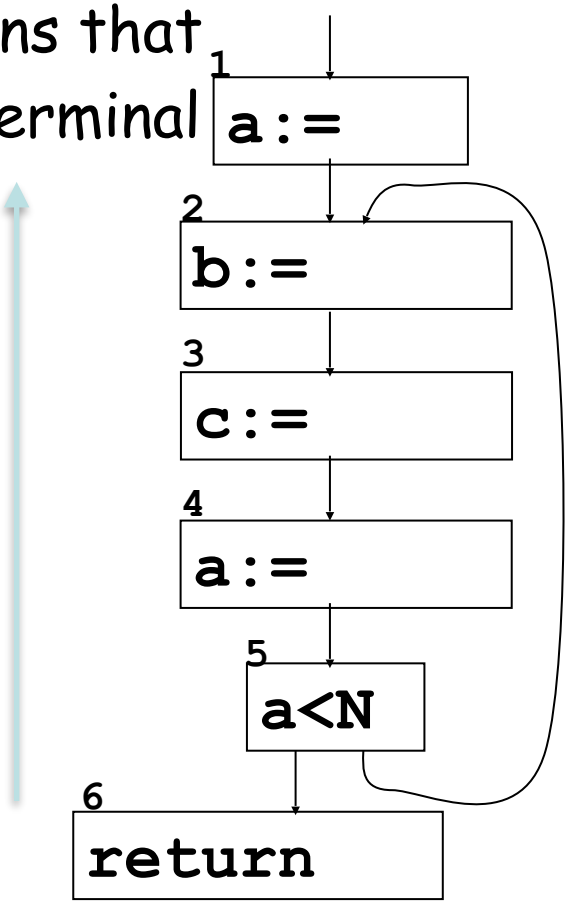
```
for all n
    in[n]:=?; out[n]:=?;
repeat
    for all n (6 to 1)
        in'[n]:=in[n]; out'[n]:=out[n];
        out[n]:= U { in[m] | m ∈ post[n]};
        in[n]:= use[n] U (out[n] - def[n]);
until (for all n: in'[n]=in[n] && out'[n]=out[n])
```

			Live[1]		Live[2]		Live[3]	
	use	def	out	in	out	in	out	in
6	c			c		c		c
5	a		c	a c	a c	a c	a c	a c
4	b	a	a c	b c	a c	b c	a c	b c
3	b c	c	b c	b c	b c	b c	b c	b c
2	a	b	b c	a c	b c	a c	b c	a c
1		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:

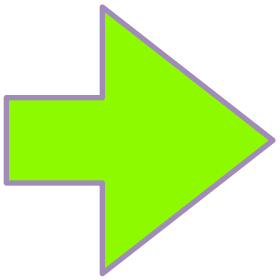
1. in[n] can be computed from out[n];
2. out[n] can be computed from in[m] for all the nodes m that are successors of n.



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



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 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?

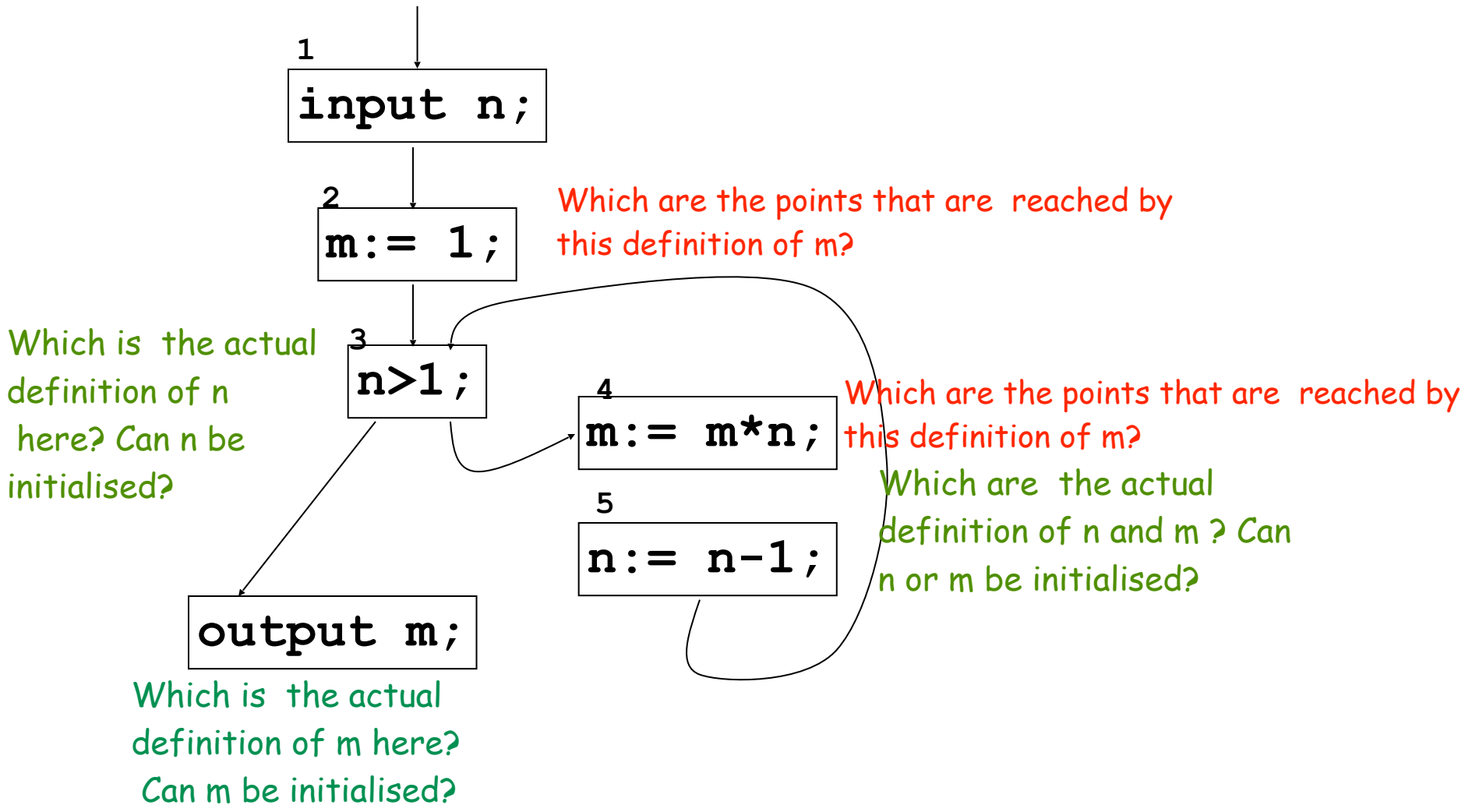
- A program point may clearly "generate" new definitions

- 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:
  $\{(x,p) \mid x \in \textbf{Vars}, p \text{ is a program point}\} \in \mathcal{P}(\textbf{Vars x Points})$
  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 **Points** and we use to represent the fact that a variable x is not initialized.
- The set $\iota = \{(x,?) \mid x \in \textbf{Vars}\}$ therefore denotes that all the program variables are not initialized.

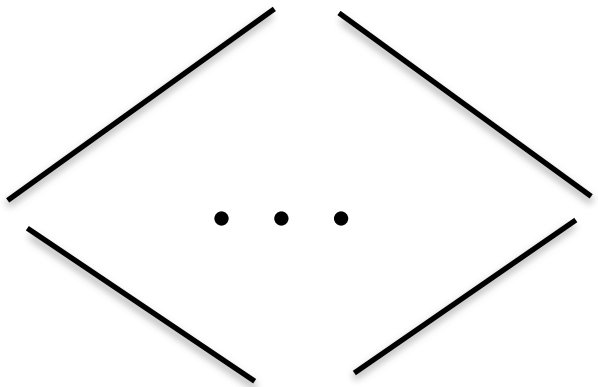
The domain for Reaching Definitions Analysis

Vars is the (finite) set of variables occuring in the program P.
Let N be the number of nodes of the CFG of P.
Let **Points**={?,1,…N}.

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

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

$$< S = \{(a,?),(a,1),(a,2),(b,?),(b,1),(b,2)\}, S, \ S, S >$$



Specification

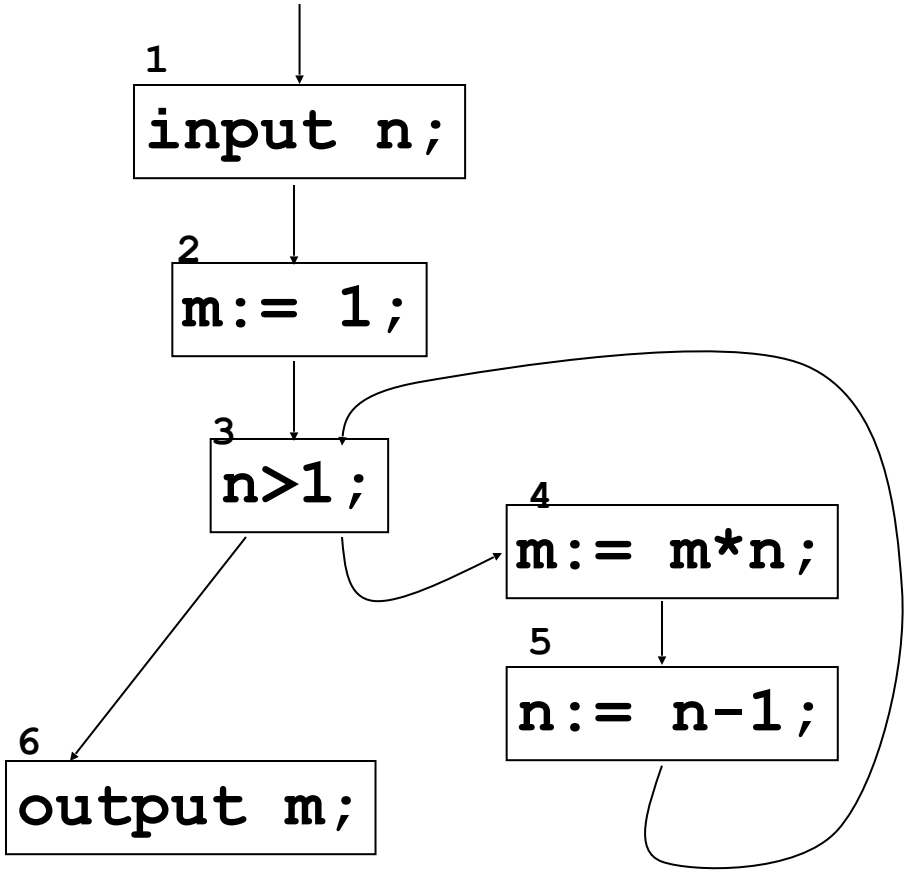
- $kill_{RD}[p] = \begin{cases} \{(x,q) \mid q \in \textbf{Points} \text{ and } \{x\}=def[q]\} & \text{if } \{x\}=def[p] \\ \emptyset & \text{if } \emptyset = def[p] \end{cases}$

- $gen_{RD}[p] = \begin{cases} \{(x,p)\} & \text{if } \{x\}=def[p] \\ \emptyset & \text{if } \emptyset = def[p] \end{cases}$

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

Kill and Gen

For simplicity we put
in kill just the points
where the var can actually be defined



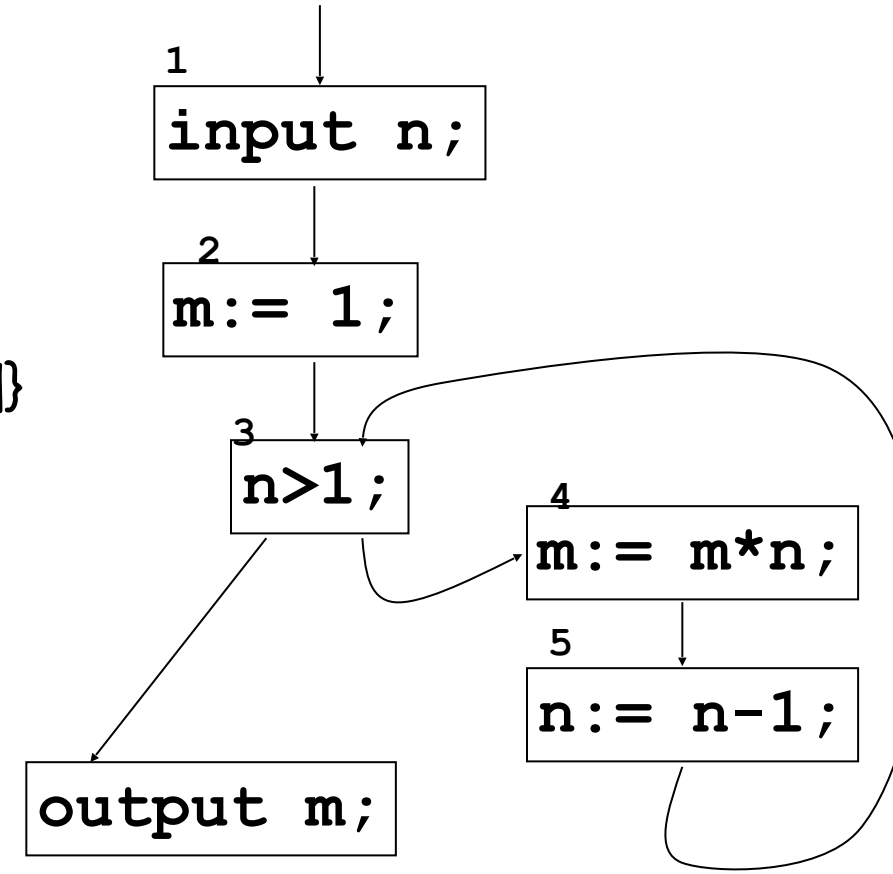
	$kill_{RD}$	gen_{RD}
1		
2	(m,?)(m,2) (m,4)	(m,2)
3		
4	(m,?)(m,2) (m,4)	(m,4)
5	(n,?) (n,5)	(n,5)
6		

Specification

- Reaching definitions analysis is specified by equations:

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

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



The solution of the previous system

Once again the solution for the equations in the previous system requires 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

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

is a CPO with bottom?

It is a CPO because it is finite

Bottom?

Point a: the function

The map Reach:

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

defined by

(assuming 1 is the only initial node)

$\text{Reach}(<RD\text{entry}_1, RD\text{exit}_1, \ldots, RD\text{entry}_N, RD\text{exit}_N>) =$

$< \{(x,?) \mid x \text{ in VARS}\}, (RD_{entry1} \setminus kill_{RD}[1]) \cup gen_{RD}[1],$

$\cup\{RD_{exit2} \mid m \text{ in pre}[2]\}, (RD_{entry2} \setminus kill_{RD}[2]) \cup gen_{RD}[2],$

$\ldots,$

$\cup\{RD_{exitm} \mid m \text{ in pre}[N]\}, (RD_{entryN} \setminus kill_{RD}[N]) \cup gen_{RD}[N] >$

Point a

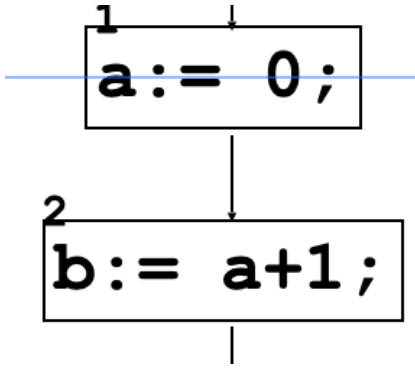
Reach(<$RDentry_1$,$RDexit_1$,…,$RDentry_N$,$RDexit_N$>)=

< {(x,?) | x in VARS}, (RD_{entry1} \ $kill_{RD}$[1]) U gen_{RD}[1],

U{RD_{exit2} |m in pre[2]} , (RD_{entry2} \ $kill_{RD}$[2]) U gen_{RD}[2]

….,

U{RD_{exitm} |m in pre[N]} , (RD_{entryN} \ $kill_{RD}$[N]) U gen_{RD}[N] >



$kill_{RD}$(1)={(a,?)}, gen_{RD}(1)={(a,1)}

$kill_{RD}$(2)={(b,?)}, gen_{RD}(2)={(b,2)}

- Example

Reach(<{(a,?)}, {}, {},{}) = <{(a,?)(b,?)},{(a,1)(b,?)},{(a,1)(b,?)},{(a,1)(b,2)}>

$\subseteq^{2N}$ $\subseteq^{2N}$

Reach(<{(a,?)(a,2)},{(a,2)},{},{(b,1)}>) = <{(a,?)(b,?)},{(a,1)(b,?)},{(a,1)(b,?)},{(a,1),(b,2)}>

Note that Reach is monotone!

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

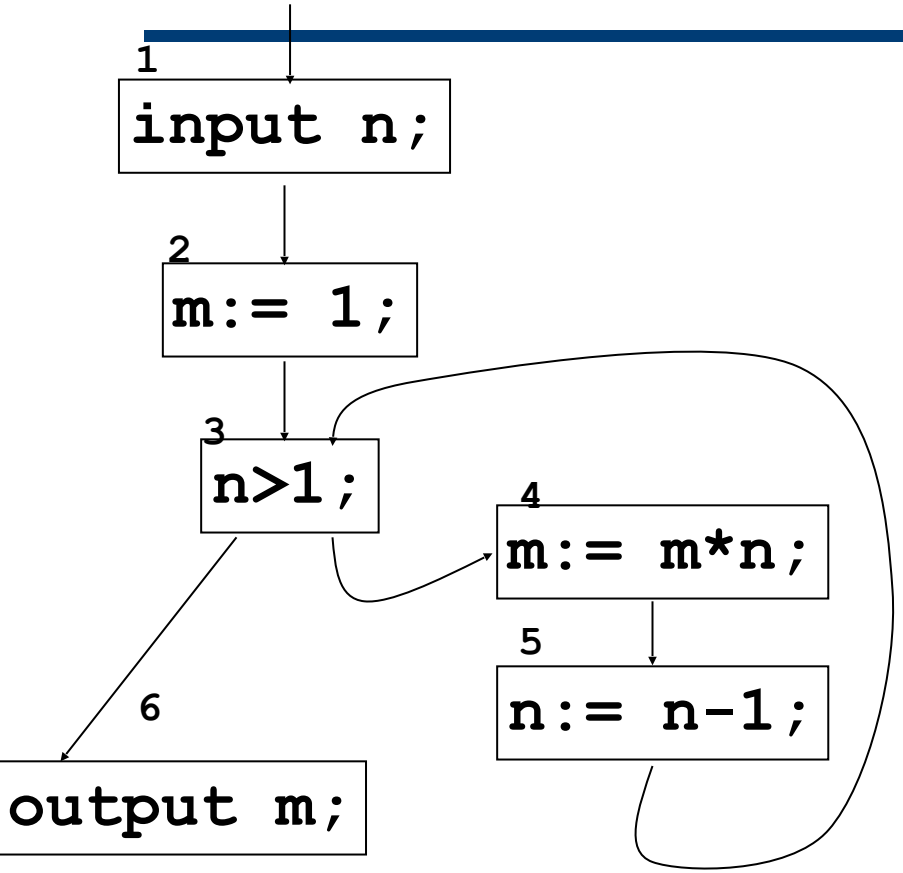
$(x,q) \in RD_{entry}(p)$

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:



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

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

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

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

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

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

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

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

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

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

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

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

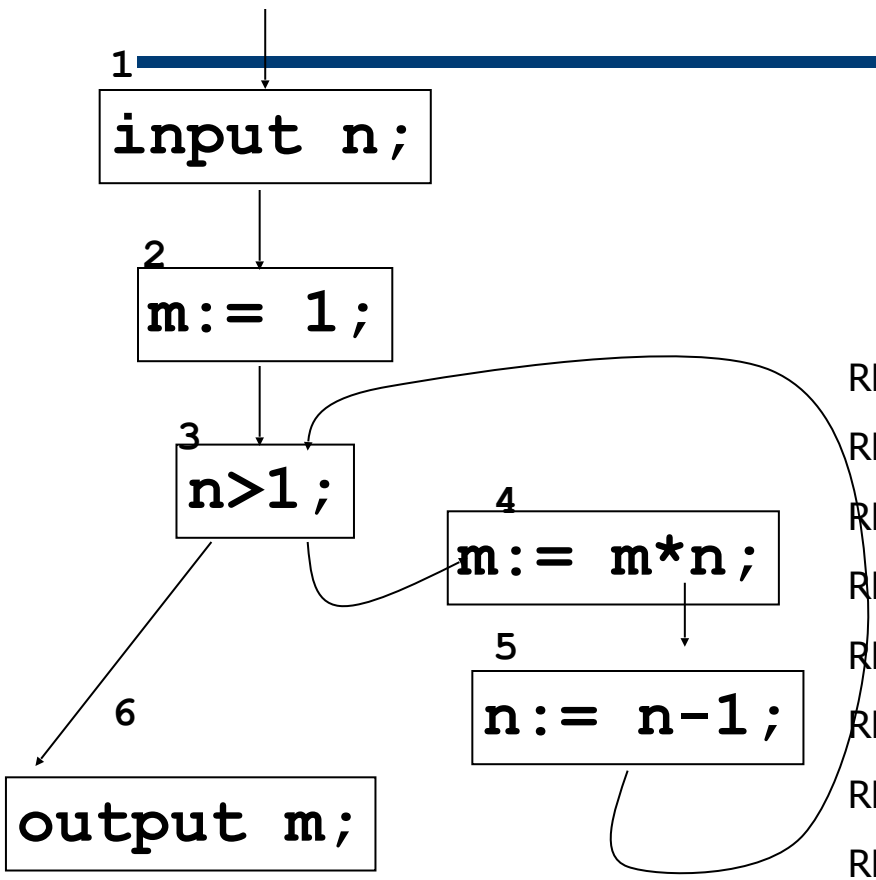
$RD_{exit}(6)= \{(n,?),(m,2)\}$

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

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

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

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
  $(x,q) \in RD_{entry}(p)$ (while the vice versa does not hold).

How can we use this?

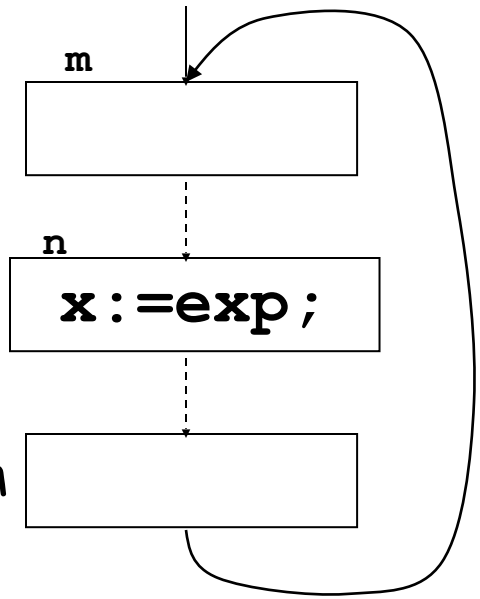
-If the analysis tells us that a variable is undefined (that is we just have the pair (x,?)) then it is

-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:=exp
3. if for any variable y occurring in exp (i.e. y$\in$**vars**(exp)) and for any program point p, we have that



$(y,p) \in RD_{entry}(m) \Leftrightarrow (y,p) \in RD_{entry}(n)$

then, the assignment x:=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

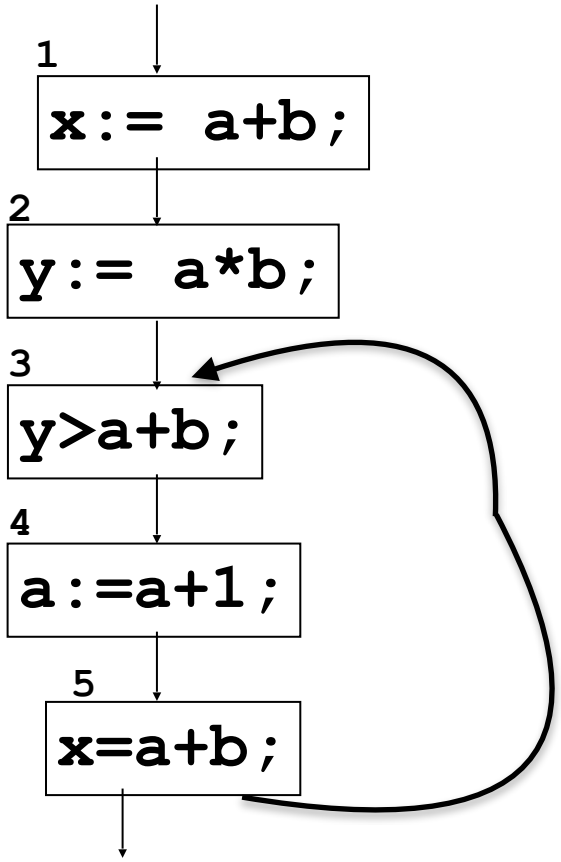
Let p be a program point. For each execution path ending in p, we want track the expressions that have already been evaluated and then not modified.

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 | e is a sub-expressions/expression appearing in P}

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

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

$Kill_{AE}$ and Gen_{AE}

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

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

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

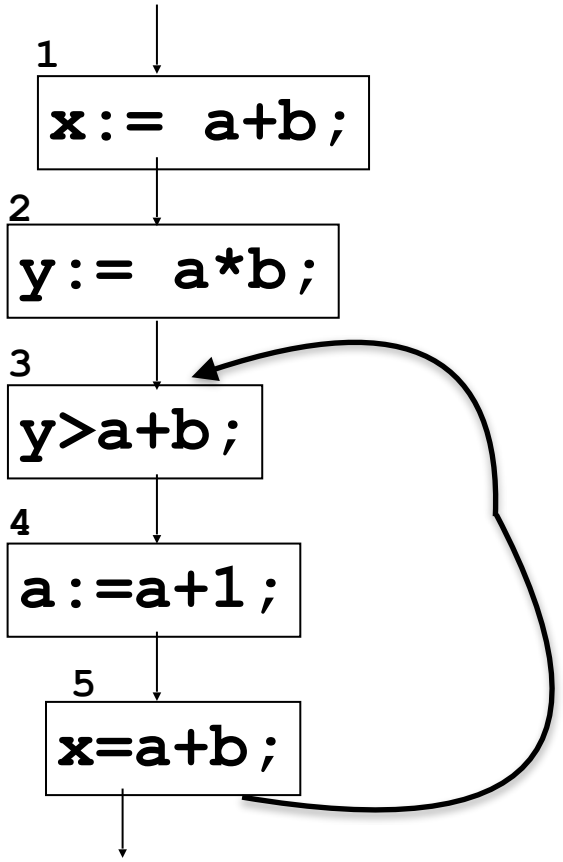
$$gen_{AE}([x:=e]^p) = \{e\} \quad \text{if } x \notin \mathbf{vars}(e),$$
$$gen_{AE}([x:=e]^p) = \emptyset \quad \text{if } x \in \mathbf{vars}(e);$$
$$gen_{AE}([e1>e2]^p) = expr(\{e1, e2\}) \quad \text{where expr(S) returns}$$
the subset of S that are expressions

Example

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

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

n	$kill_{AE}(n)$	$gen_{AE}(n)$
1	∅	{a+b}
2	∅	{a*b}
3	∅	{a+b}
4	{a+b, a*b,a+1}	∅
5	∅	{a+b}



Specification

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

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

$$AE_{exit}(p) = (AE_{entry}(p) \setminus kill_{AE}(p)) \cup gen_{AE}(p)$$

Point a and b to apply Kleene Theorem

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

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

a) The map $(\mathcal{P}(E)\times\mathcal{P}(E))^N \rightarrow (\mathcal{P}(E)\times\mathcal{P}(E))^N$ defined by

(assuming 1 is the only initial node)

$AE(<AE_{entry1}, AE_{exit1}, \ldots, AE_{entryN}, AE_{exitN}>) =$

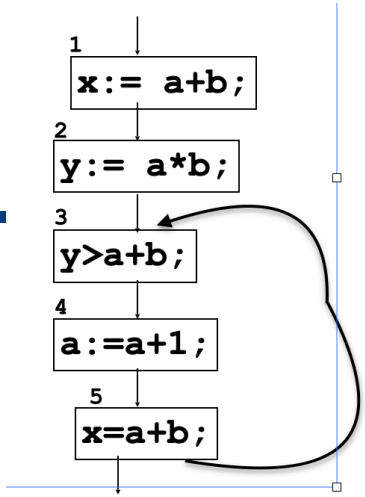
$< \varnothing, (AE_{entry1} \setminus kill_{AE}(1)) \cup gen_{AE}(1),$

$\cap\{AE_{exitq} \mid q \text{ in } pre[2]\}, (AE_{entry2} \setminus kill_{AE}(2)) \cup gen_{AE}(2),$

.....

$\cap\{AE_{exitq} \mid q \text{ in } pre[N]\}, (AE_{entryN} \setminus kill_{AE}(N)) \cup gen_{AE}(N)>$

Point a



a) The map

$AE(\langle AE_{entry1}, AE_{exit1}, \ldots, AE_{entryN}, AE_{exitN}\rangle) =$

$\langle \varnothing, (AE_{entry1} \setminus kill_{AE}(1)) \cup gen_{AE}(1),$

$\cap\{AE_{exitq} \mid q \text{ in } pre[2]\}, (AE_{entry2} \setminus kill_{AE}(2)) \cup gen_{AE}(2),$

$\ldots\ldots$

$\cap\{AE_{exitq} \mid q \text{ in } pre[N]\}, (AE_{entryN} \setminus kill_{AE}(N)) \cup gen_{AE}(N)\rangle$

is monotone on the finite domain $(\mathcal{P}(E) \times \mathcal{P}(E))^N, \subseteq^{2N} \rangle$

- Example

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

$\subseteq^{2N}$ $\subseteq^{2N}$

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

Which fix point?

AE is a definite analysis:

if $e \in AE_{entry}(p)$ 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.

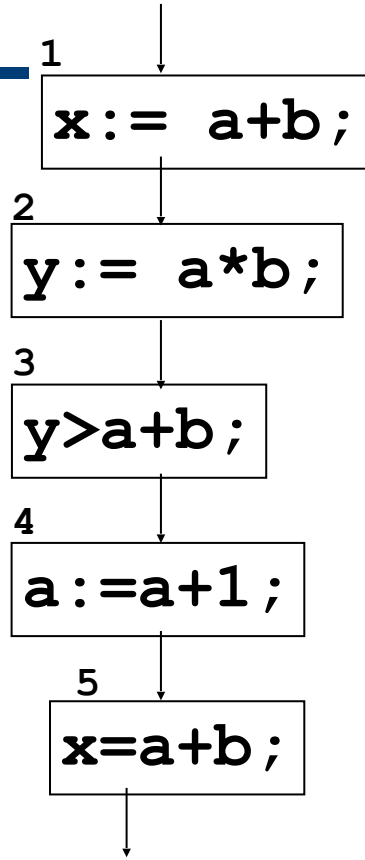
Between all fix points, we are thus interested in computing the greatest fixpoint (the more precise approximation)

Also, observe that this is a forward analysis.

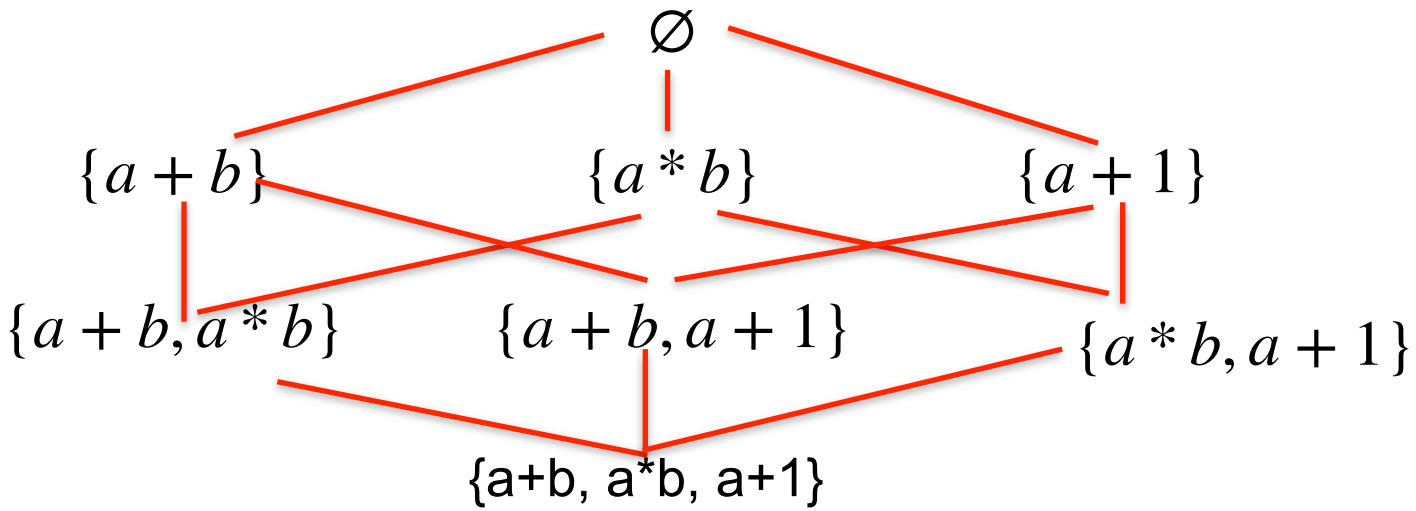
How to compute a greatest fix point?

Using Kleene's theorem with reverse order.

$$(\mathcal{P}(\mathbf{E}) \times \mathcal{P}(\mathbf{E}))^{N}, \quad \supseteq^{2N} >$$



Exemple



Computing the greatest fix point

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

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

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

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

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

$AE_{exit}(p) = (AE_{entry}(p) \setminus kill_{AE}(p)) \cup gen_{AE}(p)$

n	$kill_{AE}(n)$	$gen_{AE}(n)$
1	∅	{a+b}
2	∅	{a*b}
3	∅	{a+b}
4	{a+b, a*b,a+1}	∅
5	∅	{a+b}

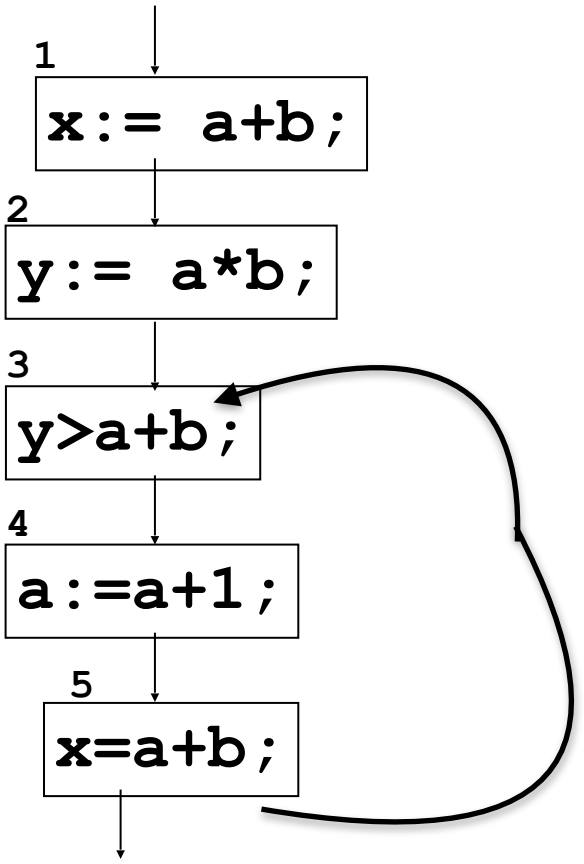
$AE_{entry}(1)= \varnothing$ $AE_{exit}(1)=\{a+b\}$

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

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

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

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



Second iteration

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

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

$AE_{exit}(p) = (AE_{entry}(p) \setminus kill_{AE}(p)) \cup gen_{AE}(p)$

Previous iteration

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

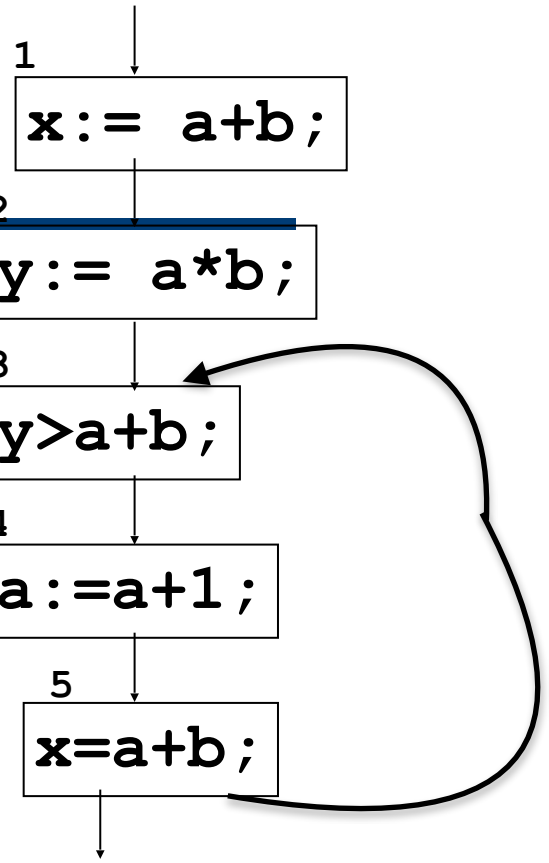
$AE_{exit}(1)= AE_{entry}(1) \cup \{a+b\}$

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

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

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

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



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

Third iteration and Greatest Fixpoint

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

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

$AE_{exit}(p) = (AE_{entry}(p) \setminus kill_{AE}(p)) \cup gen_{AE}(p)$

Previous iteration

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

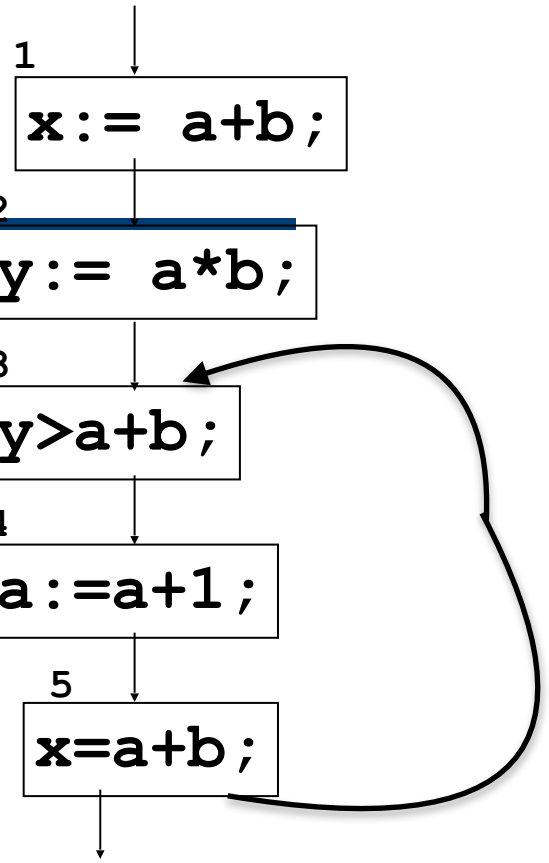
$AE_{exit}(1)= AE_{entry}(1) \cup \{a+b\}$

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

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

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

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

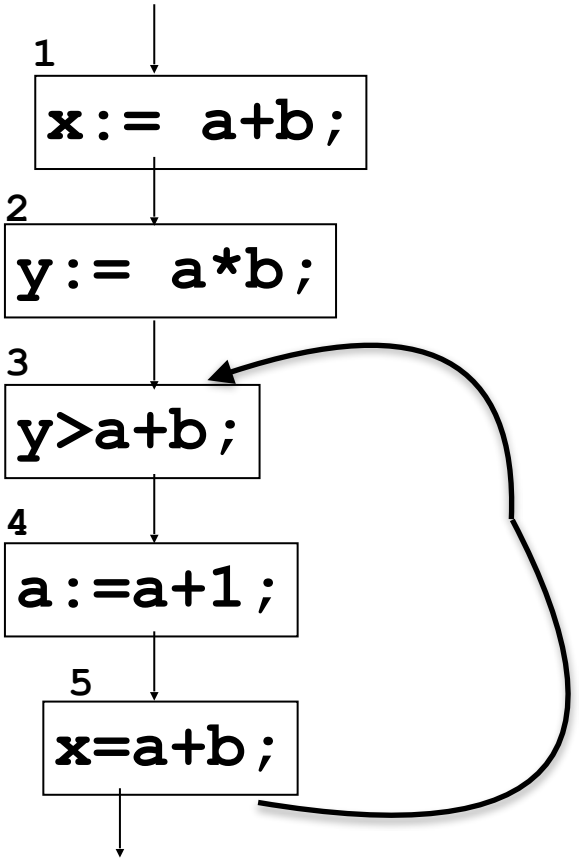


n	$AE_{entry}(n)$	$AE_{exit}(n)$
1	$\varnothing$	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	$\varnothing$
5	$\varnothing$	{a+b}

Result

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

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



Dataflow Analyses

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

	Possible Analysis **Semantics $\subseteq$ Analysis**	***Definite Analysis*** **Analysis $\subseteq$ Semantics**
Forward **in[n] $\Rightarrow$ out[n]** **pre $\Rightarrow$ post**	Reaching definitions	Available expressions
Backward **out[n] $\Rightarrow$ in[n]** **post $\Rightarrow$ pre**	Live variables	**Very busy expressions**