Dataflow analysis (ctd.)

Using Reaching Definition analysis for Global Constant Folding

Constant Folding

- By using the Reaching Definitions Analysis, we can now formally define the rules for global constant folding optimizations.
- If P is a program, we denote by RD the minimal solution of the Reaching Definition Analysis for P.
- A statement S in P can be tranformed in a more optimized statement, by applying one of the rules below, and we'll use the notation:

$$RD \vdash S \triangleright S'$$

Rule 1

1. $RD \vdash [x := a]^{\nu} \triangleright [x := a[y \to n]]^{\nu}$

   if $y \in FV(a) \quad \wedge \quad (y,?) \notin RD_{\mathtt{entry}}(\nu) \quad \wedge$
   for every $(z,\mu) \in RD_{\mathtt{entry}}(\nu)$: (z=y ⇨ [. . .]$^{\mu}$ è [y:=n]$^{\mu}$)

   The rule says that a variable can be substituted by a contant value if the Reaching Definition Analysis ensures that this is the only value that the variable can hold.

   $a[y \to n]$ means that in the expression a, variable y is substituted by value n

   FV(a) denotes the set of free variables in the expression a.

Rule 2

2. $RD \vdash [x := a]^{\nu} \triangleright [x := n]^{\nu}$

   if $FV(a)=\emptyset \wedge a \notin Num \wedge$ the value of a is n

- The rule says that an expression can be evaluated at compile time if it contains no free variables.

Composition rules

3. $RD \vdash S_1 \triangleright S'_1 \Rightarrow$

$$RD \vdash S_1 ; S_2 \triangleright S'_1 ; S_2$$

4. $RD \vdash S_2 \triangleright S'_2 \Rightarrow$

$$RD \vdash S_1 ; S_2 \triangleright S_1 ; S'_2$$

- These rules say that the transformation of a sub-statement (here a sequential statement) can be extended to the whole statement.

Composition rules

5. $RD \vdash S_1 \rhd S'_1 \Rightarrow$

   $RD \vdash \text{if } [b]^\nu \text{ then } S_1 \text{ else } S_2 \rhd$

   $\text{if } [b]^\nu \text{ then } S'_1 \text{ else } S_2$

6. $RD \vdash S_2 \rhd S'_2 \Rightarrow$

   $RD \vdash \text{if } [b]^\nu \text{ then } S_1 \text{ else } S_2 \rhd$

   $\text{if } [b]^\nu \text{ then } S_1 \text{ else } S'_2$

7. $RD \vdash S \rhd S' \Rightarrow$

   $RD \vdash \text{while } [b]^\nu \text{ do } S \rhd$

   $\text{while } [b]^\nu \text{ do } S'$

Example

- Consider the program:

  $[x:=10]^1$; $[y:=x+10]^2$; $[z:=y+10]^3$;

- The minimal solution of the Reaching Definition Analysis is:

- $RD_{\mathtt{in}}(1) = \{(x,?),(y,?),(z,?)\}$
  $RD_{\mathtt{in}}(2) = \{(x,1),(y,?),(z,?)\}$
  $RD_{\mathtt{in}}(3) = \{(x,1),(y,2),(z,?)\}$

- Using RD, we may start applying the rules above:

- RD |– $[x{:=}10]^1$; $[y{:=}x{+}10]^2$; $[z{:=}y{+}10]^3$

  $\triangleright [x{:=}10]^1$; $[y{:=}10{+}10]^2$; $[z{:=}y{+}10]^3$

- Here we apply Rule 1, with a=(x+10)

  $RD_{in}(2) = \{(x,10),(y,?),(z,?)\}$

$RD \vdash [y := a]^2 \triangleright [y := a[x \to 10]]^2$

if $x \in FV(a) \wedge (x,?) \notin RD_{in}(2) \wedge$

for every $(z,\mu) \in RD_{in}2)$: ($z=x \Rightarrow [. . .]^\mu$ is $[x{:=}10]^\mu$)

- RD $\vdash$ $[x:=10]^1; [y:=x+10]^2; [z:=y+10]^3$
  $\triangleright [x:=10]^1; [y:=10+10]^2; [z:=y+10]^3$
  $\triangleright [x:=10]^1; [y:=20]^2; [z:=y+10]^3$

- Here we apply Rule 2, whith expression a=(10+10)

  RD $\vdash$ $[y := a]^2 \triangleright [y := n]^2$

  if $FV(a)=\varnothing \wedge a \notin Num \wedge$ the value of expression a is n

- RD $\vdash$ $[x:=10]^1; [y:=x+10]^2; [z:=y+10]^3$
  $\triangleright [x:=10]^1; [y:=10+10]^2; [z:=y+10]^3$
  $\triangleright [x:=10]^1; [y:=20]^2; [z:=y+10]^3$
  $\triangleright [x:=10]^1; [y:=20]^2; [z:=20+10]^3$

- Here we apply again Rule 1, with a=(y+10)

$$RD \vdash [z := a]^3 \triangleright [z := a[y \rightarrow 20]]^3$$

if $y \in FV(a) \quad \wedge \quad (y,?) \notin RD_{in}(3) \quad \wedge$
for every $(w,\mu) \in RD_{in}(3)$: ($w=y \Rightarrow [\ldots]^\mu$ is $[y:=20]^\mu$)

- RD $\vdash$ $[x:=10]^1; [y:=x+10]^2; [z:=y+10]^3$
  $\triangleright [x:=10]^1; [y:=10+10]^2; [z:=y+10]^3$
  $\triangleright [x:=10]^1; [y:=20]^2; [z:=y+10]^3$
  $\triangleright [x:=10]^1; [y:=20]^2; [z:=20+10]^3$
  $\triangleright [x:=10]^1; [y:=20]^2; [z:=30]^3$

- Here we apply again Rule 2 with a=(20+10)

  RD $\vdash$ $[z := a]^3 \ \triangleright \ [z := n]^3$

  if $FV(a)=\emptyset \wedge a \notin Num \wedge$ the value of expression a is n

- The example above show how to get a sequence of transformations

$$RD \vdash S_1 \triangleright S_2 \triangleright S_3 \triangleright \ldots \triangleright S_k$$

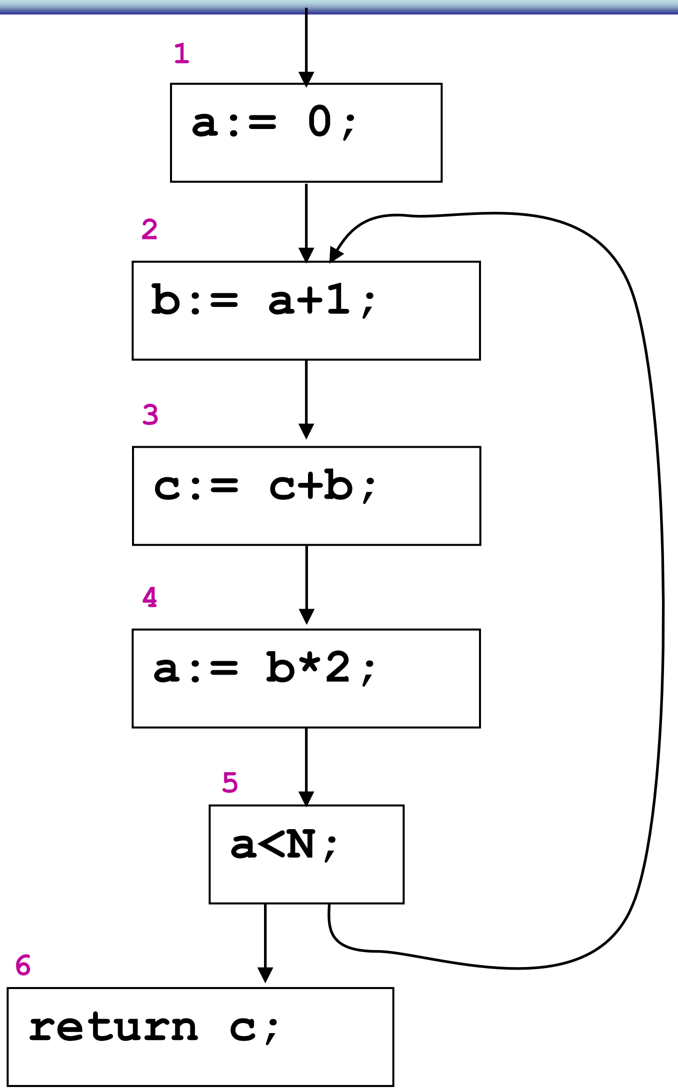
- Theoretically, once computed S_2 we shoud re-execute a reaching Definition Analysis to the new program.
- However, if RD is a solution of the Reaching Def. Analysis for S_i and $RD \vdash S_i \triangleright S_{i+1}$, then it is easy to see that RD is also a solution of the Reaching Def. Analysis for S_{i+1}.
  In fact the transformation applies to elements that do not affect at all the Reaching Def. Analysis.

Liveness: live variables

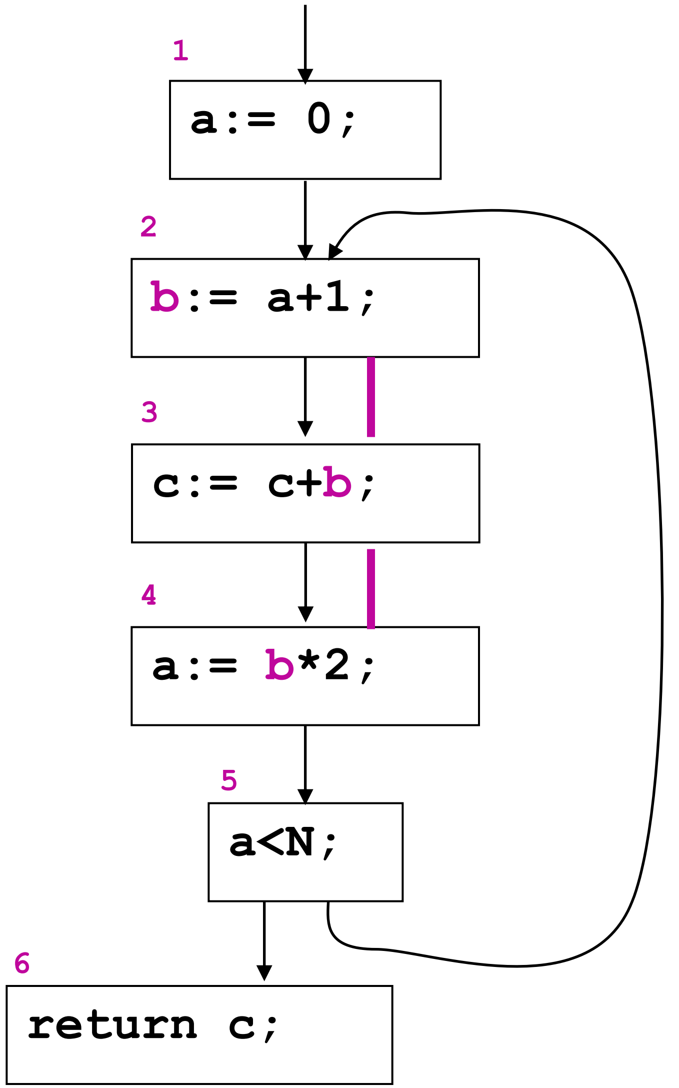
- Determine whether a given variable is used along a path from a given point to the exit.
- A variable x is *live at point p* if there is a path from p to the exit along which the value of x is used before it is redefined.
- Otherwise, the variable is dead at that point.
- Used in :
  - register allocation
  - dead code elimination

Example

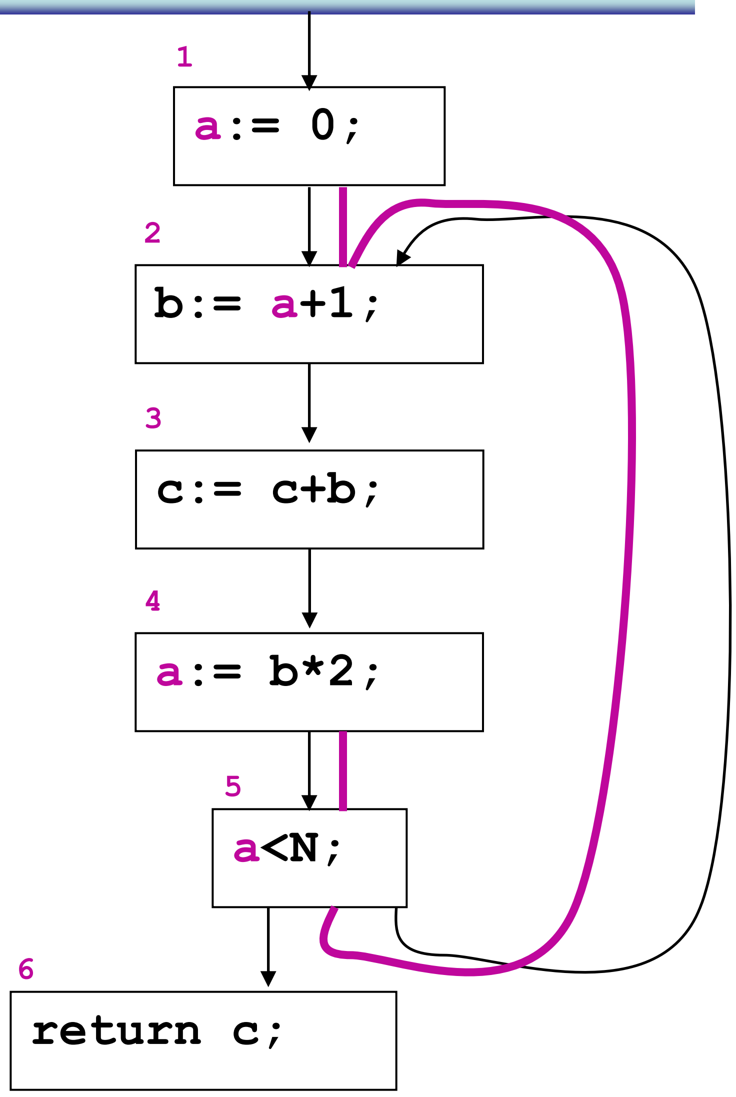
```
a = 0;
do{
   b= a+1;
   c+=b;
   a=b*2;
}
while (a<N);
return c;
```



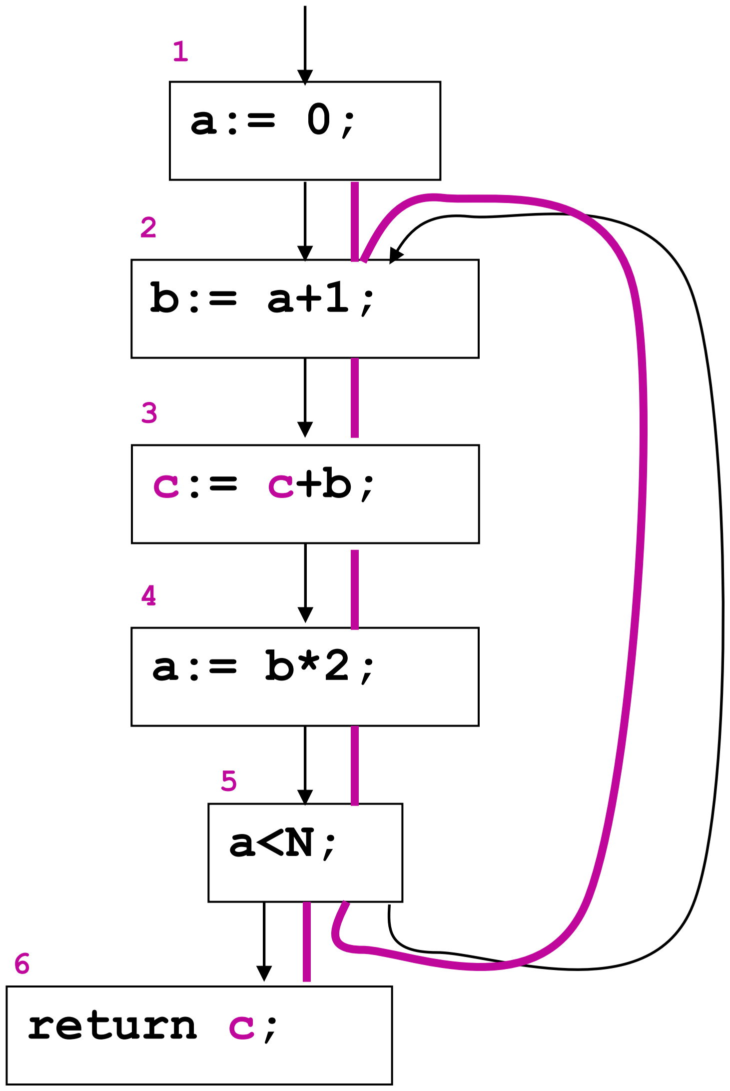
- Statement 4 makes use of variable b, then b is live in in(4) and in out(3)
- Block 3 dos not define b, then b is live also in in(3), and so in out(2)
- Block 2 defines b. Therefore the b is not live anymore in(2).
- The “live range” of variable b is: $\{2 \rightarrow 3, 3 \rightarrow 4\}$

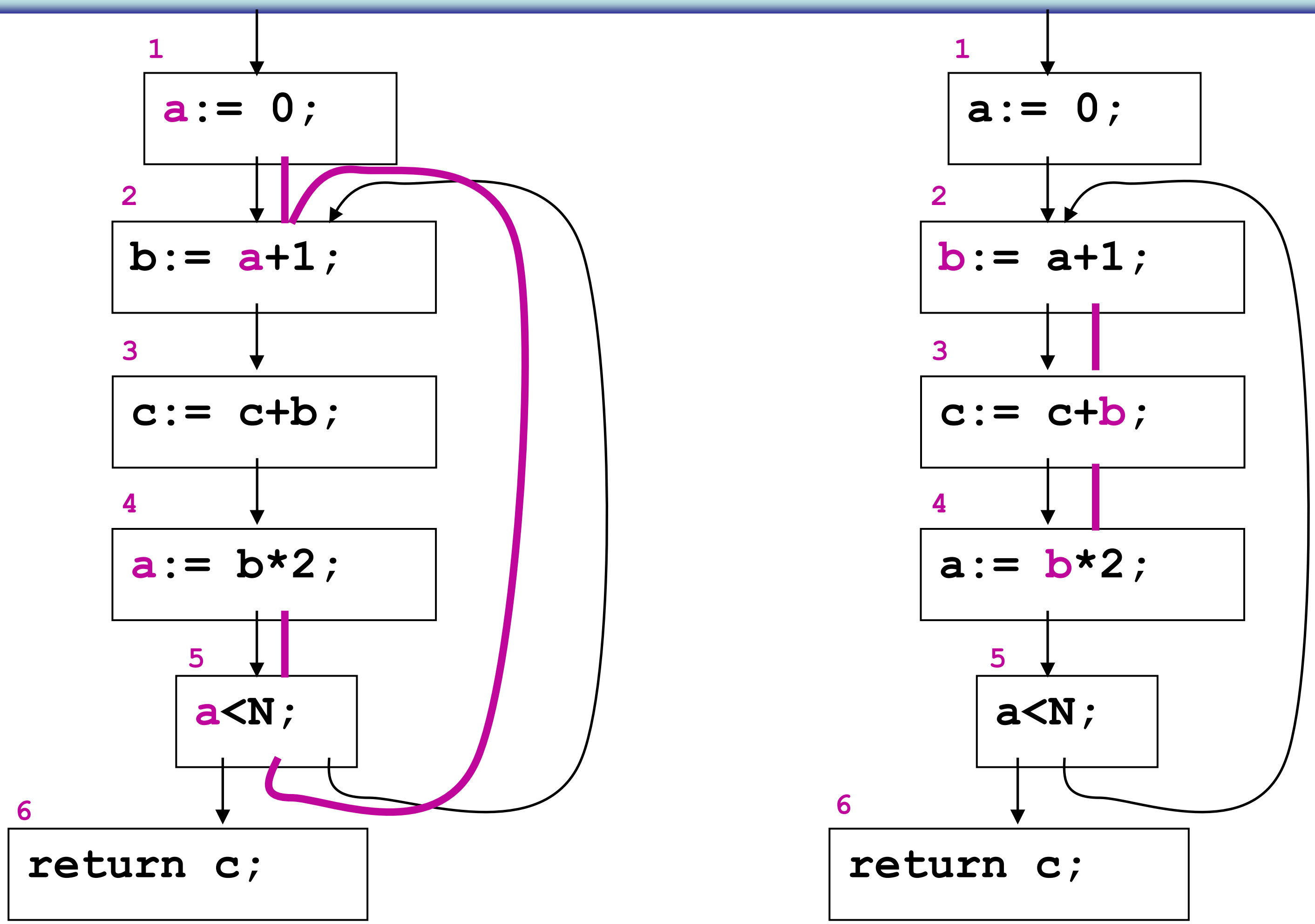


- a is live on 4 → 5 e 5 → 2
- a è live on 1 → 2
- It is dead on 2 → 3 → 4



- c is live starting from the beginning of the program:
- c is live in all points
- liveness analysis tells us that if there are no other program lines above, c is used without being initialized (and a warning message can be generated).





- Two registers are sufficient to store the three variables, as a and b are never alive at the same moment.

Live variables

- What is safe?
  - To assume that a variable is live at some point even if it may not be.
  - The computed set of live variables at point p will be a superset of the actual set of live variables at p
  - The computed set of dead variables at point p will be a subset of the actual set of dead variables at p
  - Goal : make the set of live variables as small as possible (i.e. as close to the actual set as possible)

Live variables

- How are the **def** and **use** sets defined?
  - **def**[B] = {variables defined in B before being used} /* kill */
  - **use**[B] = {variables used in B before being defined} /* gen */
- What is the direction of the analysis?
  - backward
  - **in**[B] = **use**[B] $\cup$ (**out**[B] - **def**[B])

Live variables

- What is the confluence operator?
  - union
  - **out**[B] = $\cup$ **in**[S], over the successors S of B

- How do we initialize?
  - start small
  - for each block B initialize **in**[B] = $\emptyset$

Liveness Analysis: the equations

- $gen_{LV}(p) = use[p]$
- $kill_{LV}(p) = def[n]$

$$LV_{\mathtt{exit}}(p) = \begin{cases} \emptyset & \text{if } p \text{ is a final point} \\ \cup \{ LV_{\mathtt{entry}}(q) \mid q \text{ follows } p \text{ in the CFG} \} & \end{cases}$$

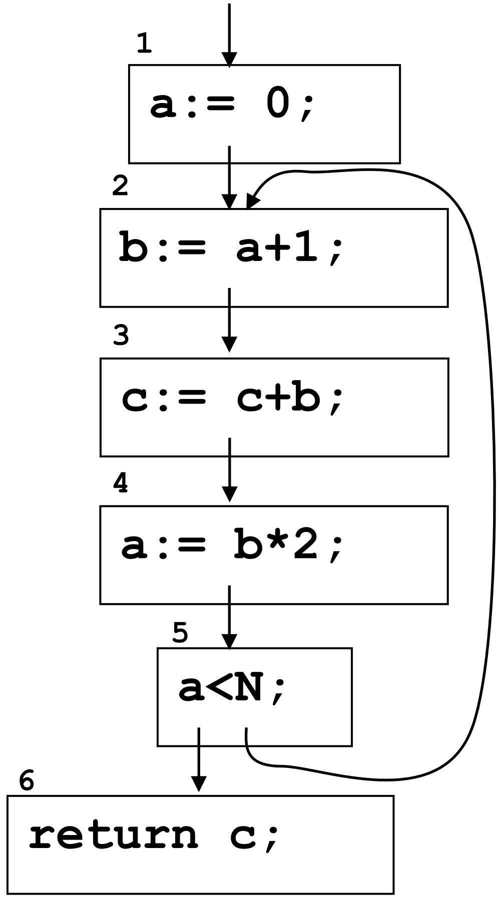
$$LV_{\mathtt{entry}}(p) = gen_{LV}(p) \cup (LV_{\mathtt{exit}}(p) \setminus kill_{LV}(p))$$

Liveness Analysis: the algorithm

```
for each n
    in[n]:={ }; out[n]:={ }
repeat
    for each n
        in'[n]:=in[n]; out'[n]:=out[n]
        in[n] := use[n] ∪ (out[n] - def[n])
        out[n]:= ∪ { in[m] | m ∈ succ[n]}
until ( for each n:   in'[n]=in[n] && out'[n]=out[n])
```

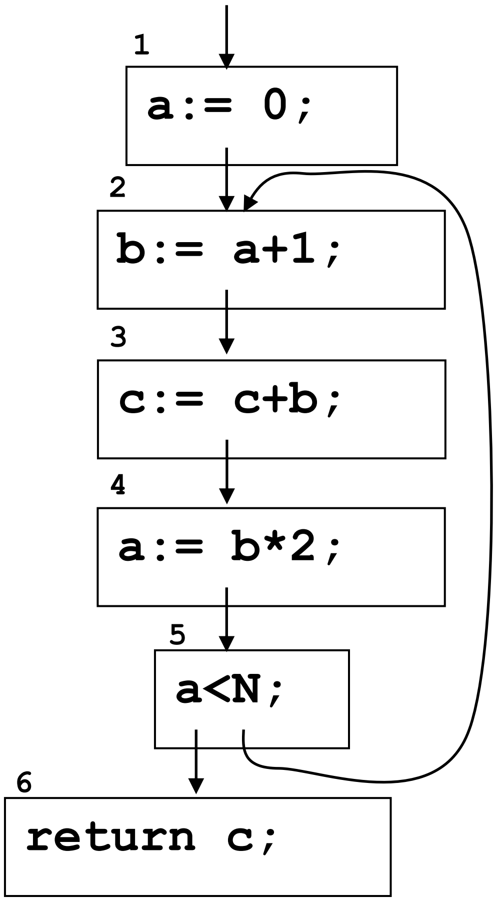
```
for each n
    in[n]:={}; out[n]:={}
repeat
    for each n
        in’[n]:=in[n]; out’[n]:=out[n]
        in[n] := use[n] U (out[n] - def[n])
        out[n]:= U { in[m] | m ∈ succ[n]}
until ( per ogni n: in’[n]=in[n] && out’[n]=out[n])
```

			1		2		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	



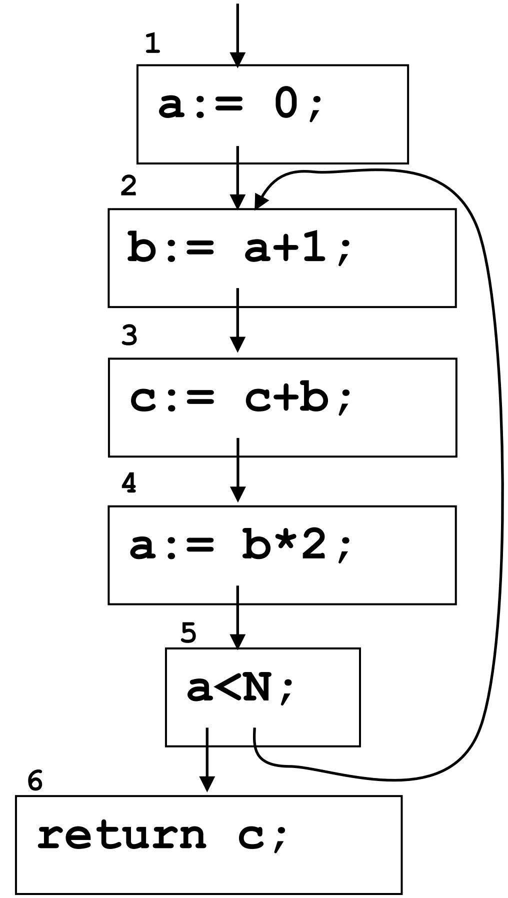
```
for each n
    in[n]:={}; out[n]:={}
repeat
    for each n
          in'[n]:=in[n]; out'[n]:=out[n]
          in[n] := use[n] U (out[n] - def[n])
          out[n]:= U { in[m] | m ∈ succ[n]}
until ( per ogni n: in'[n]=in[n] && out'[n]=out[n])
```

			3		4		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	



```
for each n
    in[n]:={}; out[n]:={}
repeat
    for each n
         in’[n]:=in[n]; out’[n]:=out[n]
         in[n] := use[n] U (out[n] - def[n])
         out[n]:= U { in[m] | m ∈ succ[n]}
until ( per ogni n: in’[n]=in[n] && out’[n]=out[n])
```

			5		6		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	



- But reordering the nodes, i.e. starting from the bottom instead of from the top, we get much faster:

			1		2		3	
	use	def	in	out	in	out	in	out
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

```
for each n
   in[n]:={}; out[n]:={}
repeat
   for each n
        in'[n]:=in[n]; out'[n]:=out[n]
        in[n] := use[n] U (out[n] - def[n])
        out[n]:= U { in[m] | m ∈ succ[n]}
until ( for each n: in'[n]=in[n] && out'[n]=out[n])
```

Time-Complexity

- A program has dimension N if the number of nodes in its CFD is N and it has at most N variables.
- Each set live-in (or live-out) has at most N elements
- Each union operation has complexity O(N)
- The for cycle computes a fixed number of union operators for each node in the graph. As the number of nodes in O(N) the for cycle has complexity $O(N^2)$

```
for each n
    in[n]:={}; out[n]:={}
repeat
    for each n
         in'[n]:=in[n]; out'[n]:=out[n]
         in[n] := use[n] U (out[n] - def[n])
         out[n]:= U { in[m] | m ∈ succ[n]}
until ( for each n: in'[n]=in[n] && out'[n]=out[n])
```

Time Complexity

- Each iteration of the repeat cycle may just add new elements to the sets live-in and live-out (it's monotonic), and the sets cannot grow indefinitely, as their size is at most N. These sets are at most 2N. Therefore there are at most $2N^2$ iterations.
- The worst overall complexity of the algorithm is $O(N^4)$.
- By reordering the nodes of the CFG, and because of the sparsity of live-in and live-out, in the practice the complexity is between $O(N)$ and $O(N^2)$.

The analysis is conservative

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

- If d is another variable unused in this code fragment, both X and Y are solutions of the two equations, while Z does not.

			X		Y		Z	
	use	def	in	out	in	out	in	out
1		a	c	a c	cd	acd	c	a c
2	a	b	a c	b c	acd	bcd	a c	b
3	b c	c	b c	b c	bcd	bcd	b	b
4	b	a	b c	a c	bcd	acd	b	a c
5	a		a c	a c	acd	acd	a c	a c
6	c		c		c		c	