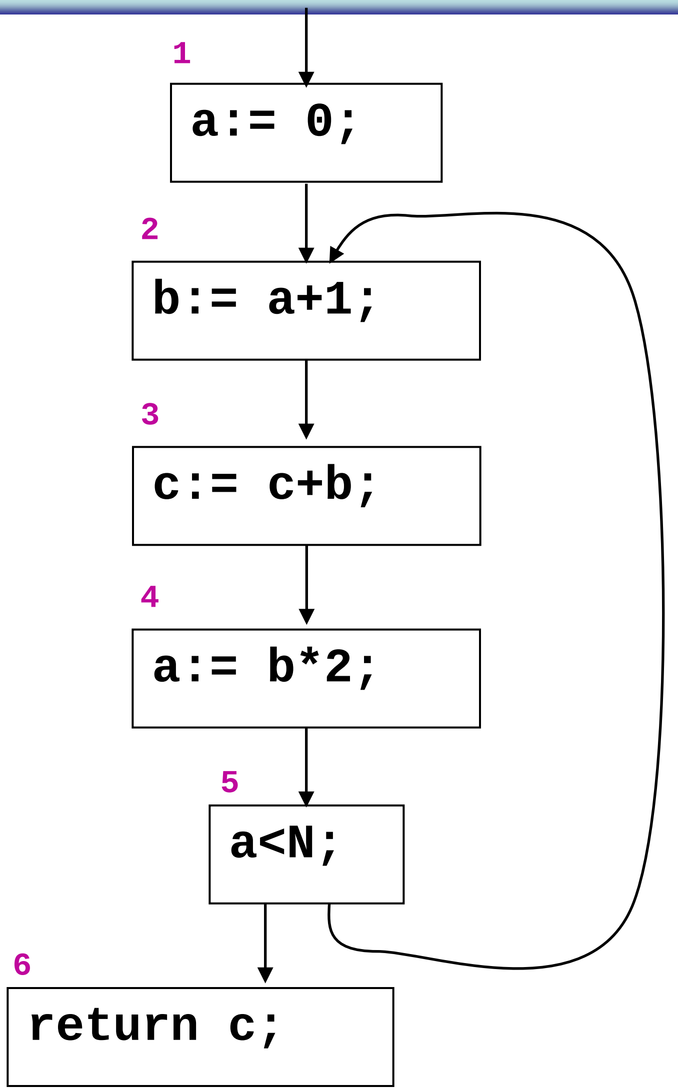
Dataflow analysis (ctd.)

Liveness: live variables

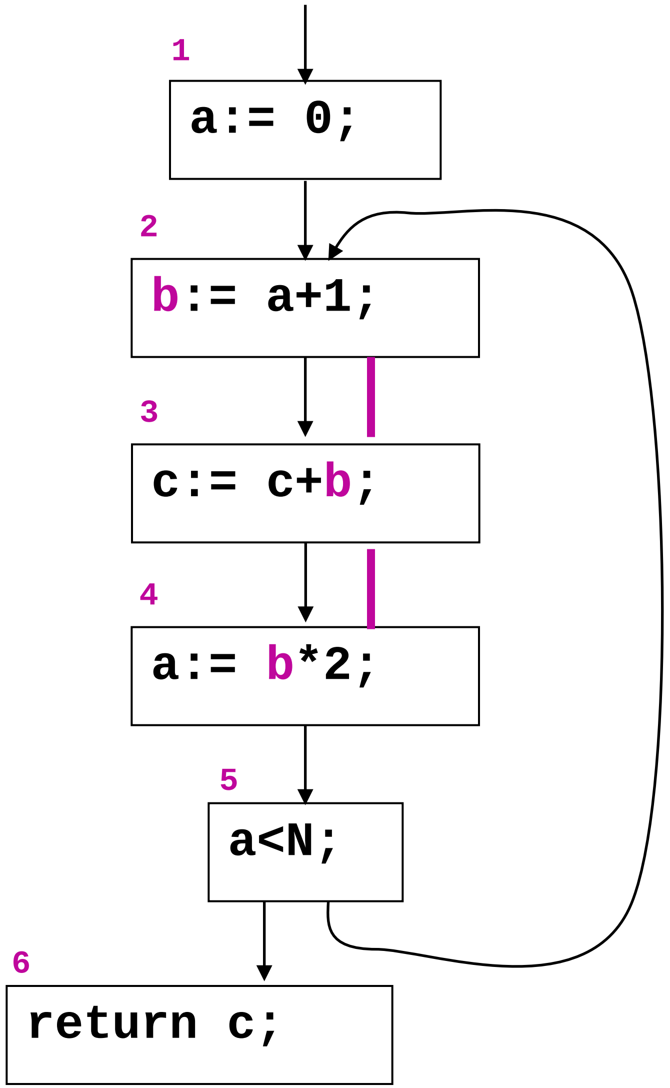
- Determine whether a given variable is used along a path from a given point to the exit.
- A variable x is *live at point p* if there is a path from p to the exit along which the value of x is used before it is redefined.
- Otherwise, the variable is dead at that point.
- Used in :
  - register allocation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

Example

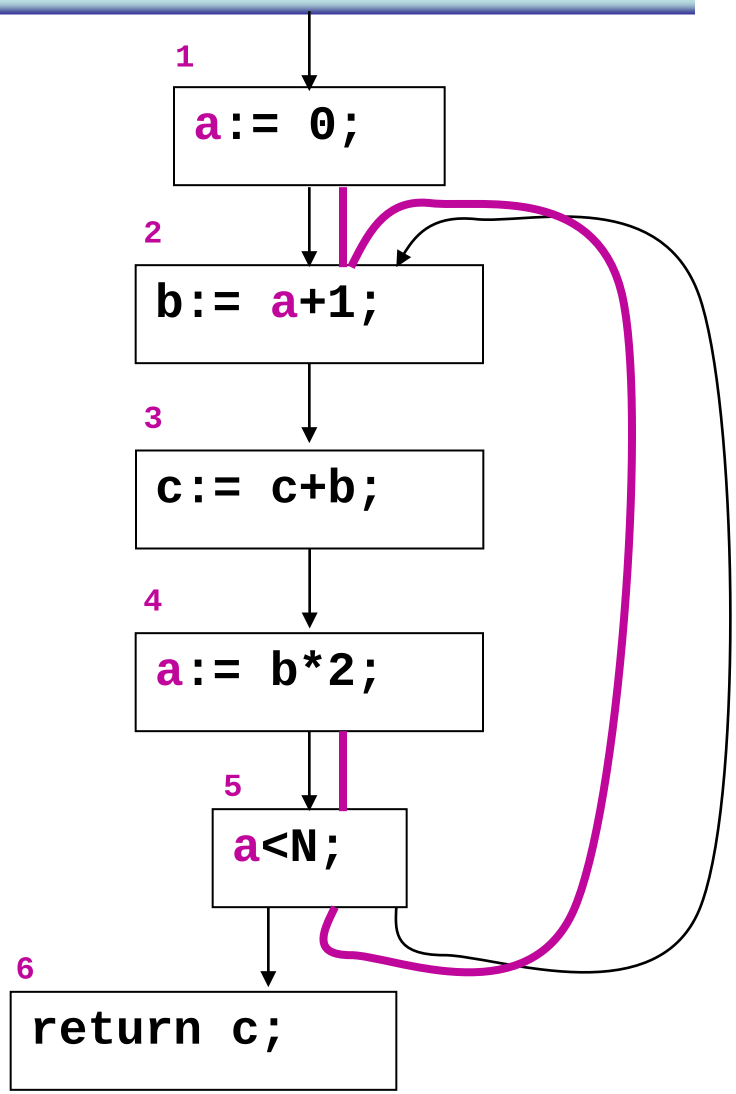
```
a = 0;
do{
  b= a+1;
  c+=b;
  a=b*2;
}
while (a<N);
return c;
```



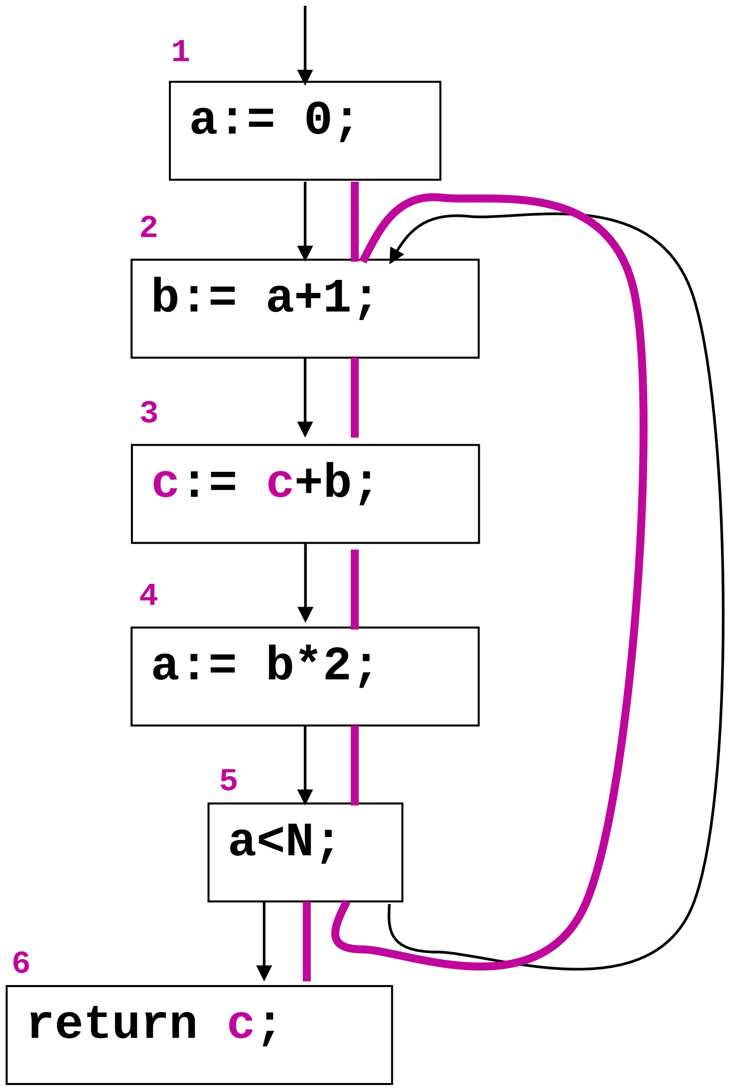
- Statement 4 makes use of variable b, then b is live in in(4) and in out(3)
- Block 3 dos not define b, then b is live also in in(3), and so in out(2)
- Block 2 defines b. Therefore the b is not live anymore in(2).
- The "live range" of variable b is: $\{2 \rightarrow 3, 3 \rightarrow 4\}$

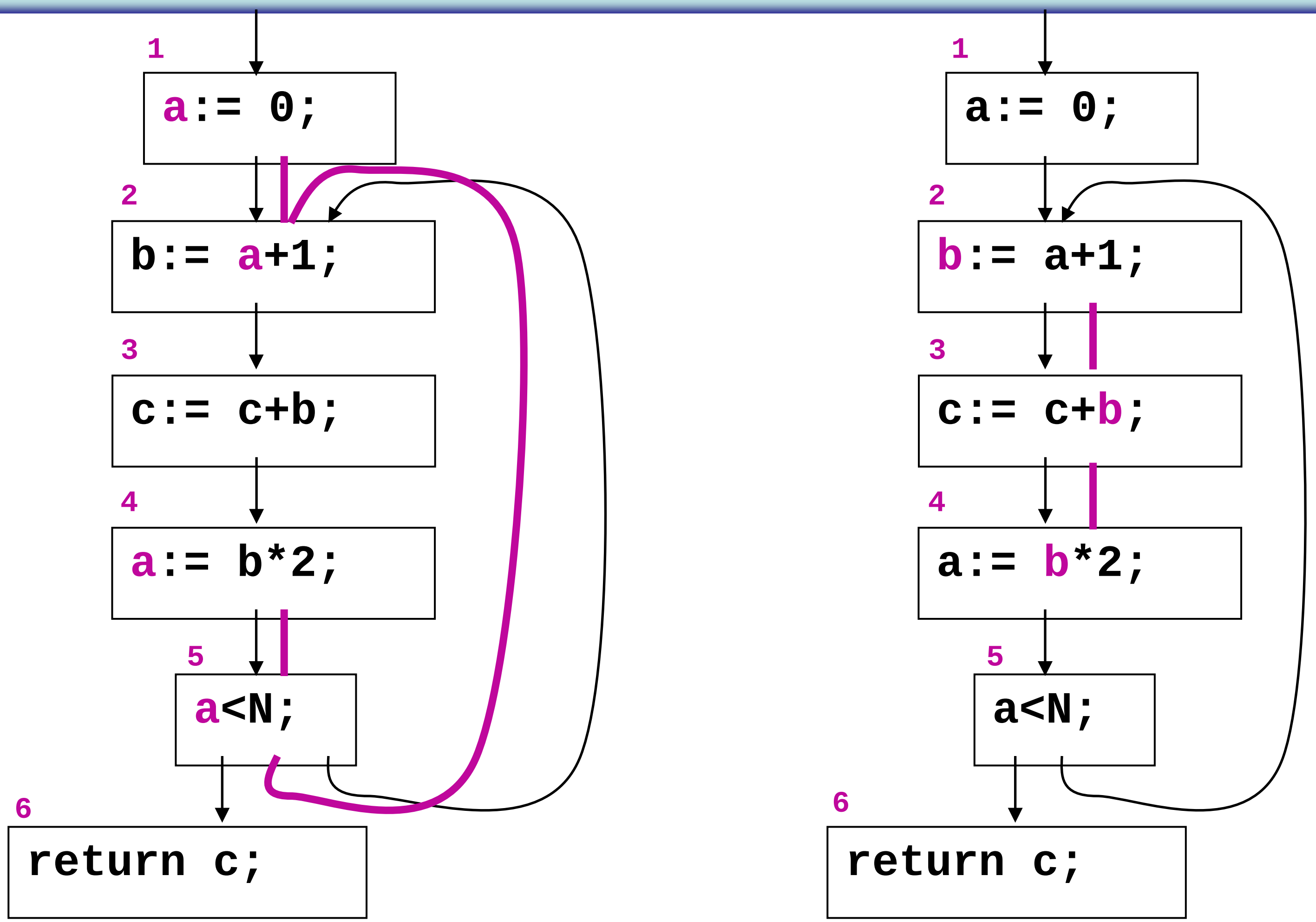


- a is live on 4 → 5 e 5 → 2
- a è live on 1 → 2
- It is dead on 2 → 3 → 4



- 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] = $\emptyset$ *or* **in**[B] = **use**[B]

Liveness Analysis: the equations

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

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

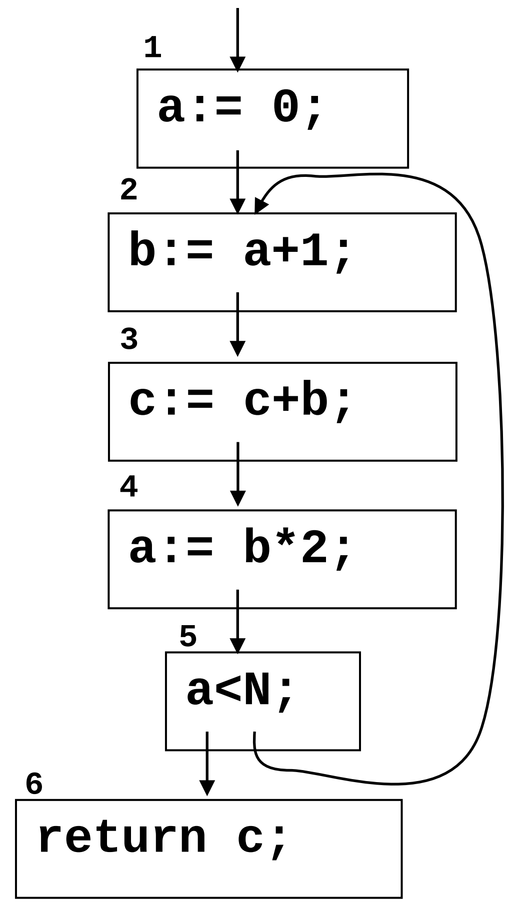
$$LV_{\texttt{entry}}(p) = gen_{LV}(p) \cup (LV_{\texttt{exit}}(p) \setminus kill_{LV}(p))$$

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])
```

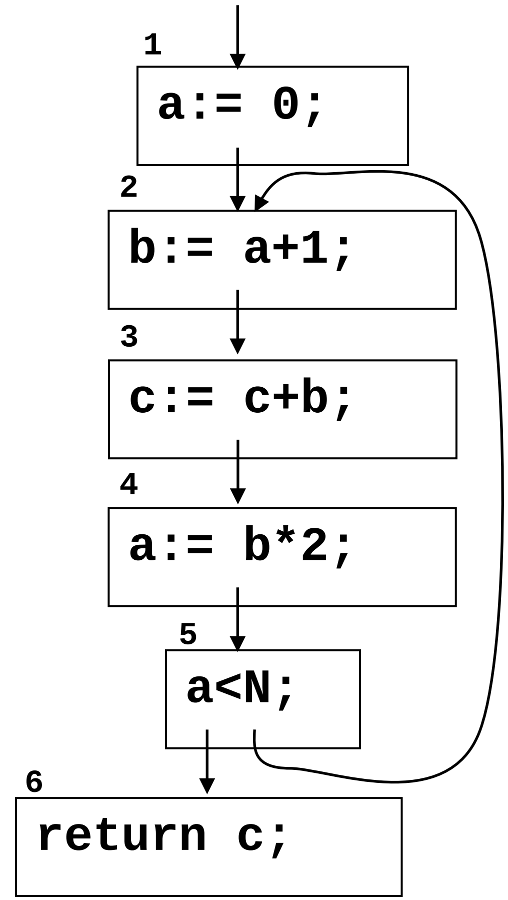
```
for each n
   in[n]:={}; out[n]:={}
repeat
   for each n
        in’[n]:=in[n]; out’[n]:=out[n]
        in[n] := use[n] ∪ (out[n] - def[n])
        out[n]:= ∪ { in[m] | m ∈ succ[n]}
until ( per ogni n: in’[n]=in[n] && out’[n]=out[n])
```

			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	



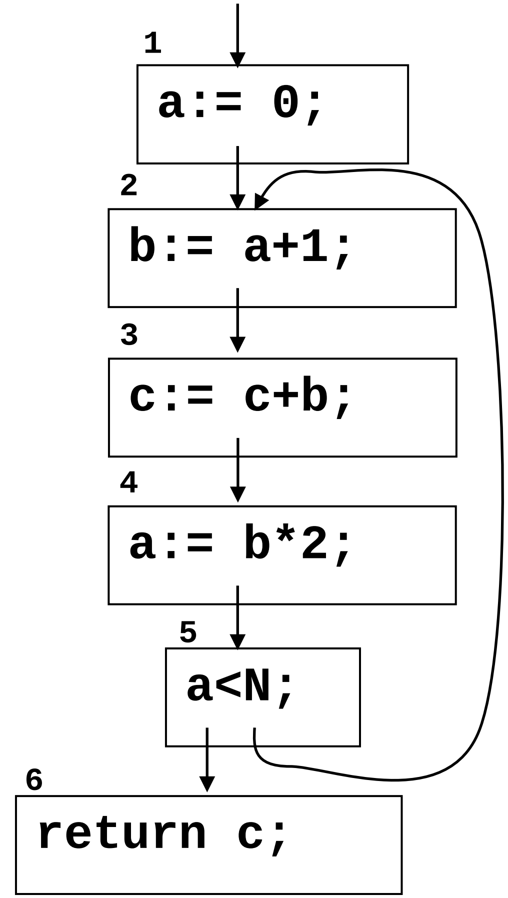
```
for each n
   in[n]:={}; out[n]:={}
repeat
   for each n
        in'[n]:=in[n]; out'[n]:=out[n]
        in[n] := use[n] ∪ (out[n] - def[n])
        out[n]:= ∪ { in[m] | m ∈ succ[n]}
until ( per ogni n: in'[n]=in[n] && out'[n]=out[n])
```

			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] ∪ (out[n] - def[n])
        out[n]:= ∪ { 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] ∪ (out[n] - def[n])
        out[n]:= ∪ { in[m] | m ∈ succ[n]}
until ( per ogni 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] ∪ (out[n] - def[n])
        out[n]:= ∪ { in[m] | m ∈ succ[n]}
until ( per ogni 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	

Available expressions

- Determine which expressions have already been evaluated at each point.
- A expression $x+y$ is *available at point p* if every path from the entry to p evaluates $x+y$ and after the last such evaluation prior to reaching p, there are no assignments to x or y
- Used in :
  - global common subexpression elimination

Example

Kills x+y,
w*x, etc.

```
x = x+y
z = a+b
```

Kills z-w,
x+z, etc.

Generates
a+b

Available expressions

- What is safe?
  - To assume that an expression is **not** available at some point even if it may be.
  - The computed set of available expressions at point p will be a subset of the actual set of available expressions at p
  - The computed set of unavailable expressions at point p will be a superset of the actual set of unavailable expressions at p
  - Goal : make the set of available expressions as large as possible (i.e. as close to the actual set as possible)

Available expressions

- How are the **gen** and **kill** sets defined?
  - **gen**[B] = {expressions evaluated in B without subsequently redefining its operands}
  - **kill**[B] = {expressions whose operands are redefined in B without reevaluating the expression afterwards}
- What is the direction of the analysis?
  - forward
  - **out**[B] = **gen**[B] $\cup$ (**in**[B] - **kill**[B])

Available expressions

- What is the confluence operator?
  - intersection
  - **in**[B] = $\cap$ **out**[P], over the predecessors P of B

- How do we initialize?
  - start large
  - for the first block B_1 initialize **out**[B_1] = **gen**[B_1]
  - for each block B initialize **out**[B] = U-**kill**[B]

Avaliable Expressions: equations

$$AE_{\mathtt{entry}}(p) = \begin{cases} \varnothing & \text{for inititial point p} \\ \\ \cap \{ AE_{\mathtt{exit}}(q) \mid (q,p) \text{ in the CFD} \} \end{cases}$$

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

Equations

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

$$AE_{entry}(p)=\begin{cases}\emptyset & \text{for iniitial point p}\\ \cap\,\{\, AE_{exit}(q) \mid (q,p) \text{ in CFD}\}\end{cases}$$

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

$$AE_{entry}(1)= \emptyset$$
$$AE_{entry}(2)=AE_{exit}(1)$$
$$AE_{entry}(3)=AE_{exit}(2) \cap AE_{exit}(5)$$
$$AE_{entry}(4)=AE_{exit}(3)$$
$$AE_{entry}(5)=AE_{exit}(4)$$

$$AE_{exit}(1)= AE_{entry}(1) \cup \{a+b\}$$
$$AE_{exit}(2)= AE_{entry}(2) \cup \{a*b\}$$
$$AE_{exit}(3)= AE_{entry}(3) \cup \{a+b\}$$
$$AE_{exit}(4)= AE_{entry}(4) - \{a+b, a*b, a+1\}$$
$$AE_{exit}(5)= AE_{entry}(5) \cup \{a+b\}$$

Solution

$AE_{entry}(1) = \varnothing$

$AE_{entry}(2) = AE_{exit}(1)$

$AE_{entry}(3) = AE_{exit}(2) \cap AE_{exit}(5)$

$AE_{entry}(4) = AE_{exit}(3)$

$AE_{entry}(5) = AE_{exit}(4)$

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

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

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

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

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

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]^1 ; [y:=a*b]^2 ; while [y>a+b]^3 do { [a:=a+1]^4 ; [x:=a+b]^5 }

n	**$AE_{\mathtt{entry}}$(n)**	**$AE_{\mathtt{exit}}$(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}

- Even though the expression a is redefined in the cycle (in 4), the expression a+b is always available ai the entry of the cycle (in 3).
- Viceversa, a*b is available at the first entry of the cycle but it is killed before the next iteration (in 4).

Very Busy Expressions

- Determine whether an expression is evaluated in all paths from a point to the exit.
- An expression e is very busy at point p if no matter what path is taken from p, e will be evaluated before any of its operands are defined.
- Used in:
  - Code hoisting
    - If e is very busy at point p, we can move its evaluation at p.

Example

if $[a>b]^1$ then $([x:=b-a]^2 ; [y:=a-b]^3)$ else $([y:=b-a]^4 ; [x:=a-b]^5)$

The two expressions a-b and b-a are both very busy in program point 1.

Very Busy Expressions

- What is safe?
  - To assume that an expression is not very busy at some point even if it may be.
  - The computed set of very busy expressions at point p will be a subset of the actual set of available expressions at p
  - Goal : make the set of very busy expressions as large as possible (i.e. as close to the actual set as possible)