Methodologies for Software Processes

Lecture 3- Dataflow Analysis

(our slides are taken from other courses that use "Principles of Program Analysis" as textbook)

Formalization:
Reaching definitions Analysis

- How can we formalize a definition D?
  By a pair (x,n) where x is the variable modified by D, and n identifies the assignment D.

- A definition D reaches a point *p* if there is a path from D to *p* along which D is not killed.

- A definition D of a variable *x* is killed when there is a redefinition of *x.*

- How can we represent the set of definitions reaching a point?

Reaching definitions

- What is safe?
  - To assume that a definition reaches a point even if it turns out not to.
  - The computed set of definitions reaching a point p will be a superset of the actual set of definitions reaching p
  - It's a "possible", not a "definite" property
  - Goal : make the set of reaching definitions as small as possible (i.e. as close to the actual set as possible)

Reaching definitions

- How are the **gen** and **kill** sets defined?
  - **gen**[B] = {definitions that appear in B and reach the end of B}
  - **kill**[B] = {all definitions that never reach the end of B}
- What is the direction of the analysis?
  - forward
  - **out**[B] = **gen**[B] $\cup$ (**in**[B] - **kill**[B])

Reaching definitions

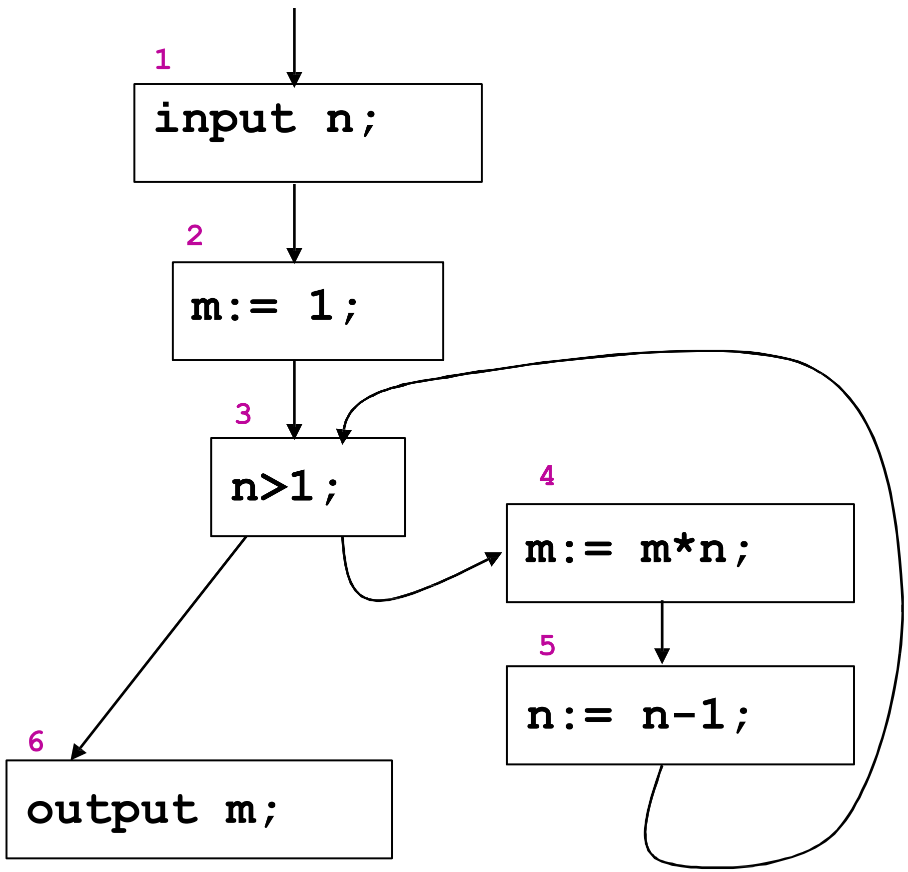
- What is the **confluence** operator?
  - union
  - **in**[P] = $\cup$ **out**[Q], over the predecessors Q of P
- How do we initialize?
  - start small
    - Why? Because we want the resulting set to be as small as possible
  - for each block B initialize **out**[B] = **gen**[B]

Formal specification

- The Reaching Definition Analysis is specified by the folllowing equations:
- For each program point,

$$RD_{\mathtt{in}}(p) = \begin{cases} \iota & \text{if p is the initial point in the control graph} \\ \bigcup \{ RD_{\mathtt{out}}(q) \mid \text{there is an arrow from q to p in the CFG} \} & \end{cases}$$

$$RD_{\mathtt{out}}(p) = gen_{\mathtt{RD}}(p) \cup (RD_{\mathtt{in}}(p) \setminus kill_{\mathtt{RD}}(p))$$



- $RD_{in}(1) = \{(n,?),(m,?)\}$
  $RD_{out}(1) = \{(n,?),(m,?)\}$
- $RD_{in}(2) = \{(n,?),(m,?)\}$
  $RD_{out}(2) = \{(n,?),(m,2)\}$
- $RD_{in}(3) = RD_{out}(2) \cup RD_{out}(5)$
  $= \{(n,?),(n,5),(m,2),(m,4)\}$
  $RD_{out}(3) = \{(n,?),(n,5),(m,2),(m,4)\}$
- $RD_{in}(4) = \{(n,?),(n,5),(m,2),(m,4)\}$
  $RD_{out}(4) = \{(n,?),(n,5),(m,4)\}$
- $RD_{in}(5) = \{(n,?),(n,5),(m,4)\}$
  $RD_{out}(5) = \{(n,5),(m,4)\}$
- $RD_{in}(6) = \{(n,?),(n,5),(m,2),(m,4)\}$
  $RD_{out}(6) = \{(n,?),(n,5),(m,2),(m,4)\}$

Algorithm

- Input: Control Flow Graph Diagram
- Output : RD
-
- Steps:
  - step 1 (initialization):
    - $RD_{in}(p)$ is the emptyset for each p
    - $RD_{in}(1) = \iota = \{(x,?) \mid x$ is a program variable$\}$

- Step 2 (iteration)
  - Flag =TRUE;

```
while Flag
        Flag = FALSE;
        for each program point p
                new = U{f(RD,q) | (q,p) is an edge of the graph}
                if RD_in(p) != new
                    Flag = TRUE;
                    RD_in(p) = new;
```

where $f(RD,q) = gen_{RD}(q) \cup (RD_{in}(q) \setminus kill_{RD}(q))$

Example

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

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

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:

$$RD \vdash S \blacktriangleright S'$$

Rule 1

1. RD |– $[x := a]^{\nu}$ ► $[x := a[y \rightarrow n]]^{\nu}$

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

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.

$a[y \rightarrow n]$ 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 $\vdash$ $[x := a]^{\nu}$ ► $[x := n]^{\nu}$

   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 |– $S_1 \blacktriangleright S'_1 \rightarrow$

   RD |– $S_1 ; S_2 \blacktriangleright S'_1 ; S_2$

4. RD |– $S_2 \blacktriangleright S'_2 \rightarrow$

   RD |– $S_1 ; S_2 \blacktriangleright S_1 ; S'_2$

- 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 |– S_1 ► S'_1 →

   RD |– if $[b]^\nu$ then S_1 else S_2 ►

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

6. RD |– S_2 ► S'_2 →

   RD |– if $[b]^\nu$ then S_1 else S_2 ►

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

7. RD |– S ► S' →

   RD |– while $[b]^\nu$ do S ►

   while $[b]^\nu$ 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 |– $[x{:=}10]^1$; $[y{:=}x+10]^2$; $[z{:=}y+10]^3$

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

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

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

RD |– $[y := a]^2$ ► $[y := a[x \longrightarrow 10]]^2$

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

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

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

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

  ► $[x:=10]^1; [y:=20]^2; [z:=y+10]^3$

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

RD |– $[y := a]^2$ ► $[y := n]^2$

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

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

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

- RD $\vdash$ $[z := a]^3$ ► $[z := a[y \longrightarrow 20]]^3$

if $y \in FV(a) \quad \wedge \quad (y,?) \notin RD_{in}(3) \quad \wedge$

for every $(w,\mu) \in RD_{in}(3)$: $(w=y \rightarrow [\,.\,.\,.\,]^\mu$ is $[y:=20]^\mu)$

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

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

  ► $[x:=10]^1; [y:=20]^2; [z:=y+10]^3$

  ► $[x:=10]^1; [y:=20]^2; [z:=20+10]^3$

  ► $[x:=10]^1; [y:=20]^2; [z:=30]^3$

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

RD |– $[z := a]^3$ ► $[z := n]^3$

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

$$RD \vdash S_1 \blacktriangleright S_2 \blacktriangleright S_3 \blacktriangleright \ldots \blacktriangleright S_\kappa$$

- 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 $RD \vdash S_i \blacktriangleright S_{i+1}$, then it is easy to see that RD is also a solution of the Reaching Def. Analysis for S_{i+1}.

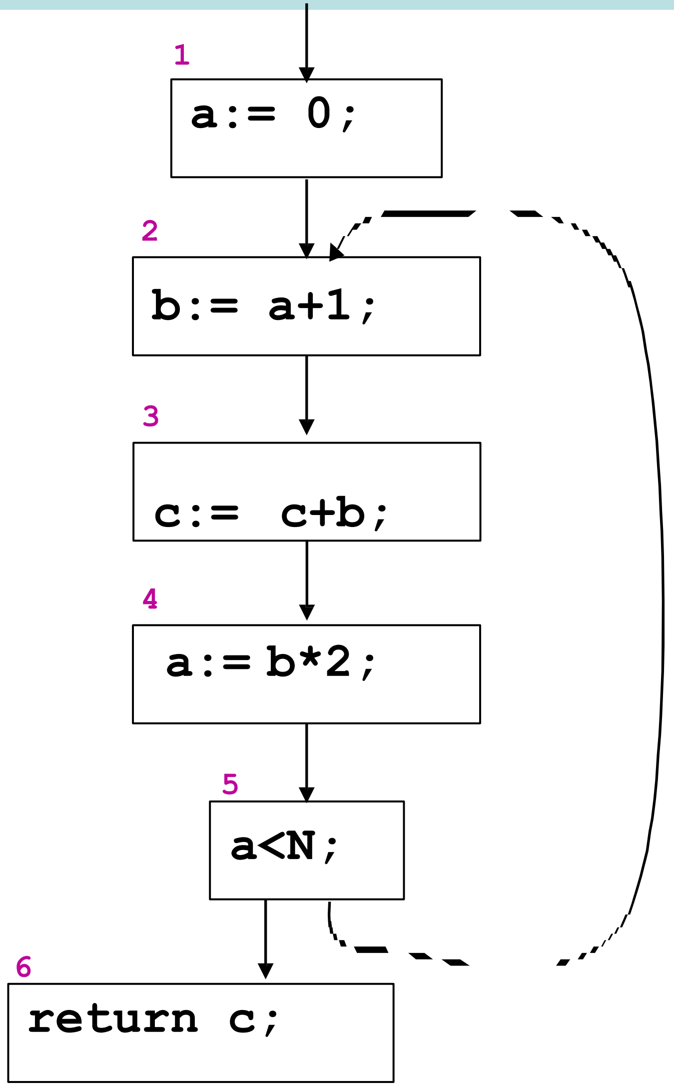
In fact the transformation applies to elements that do not affect at all the Reaching Def. Analysis.

Liveness: live variables

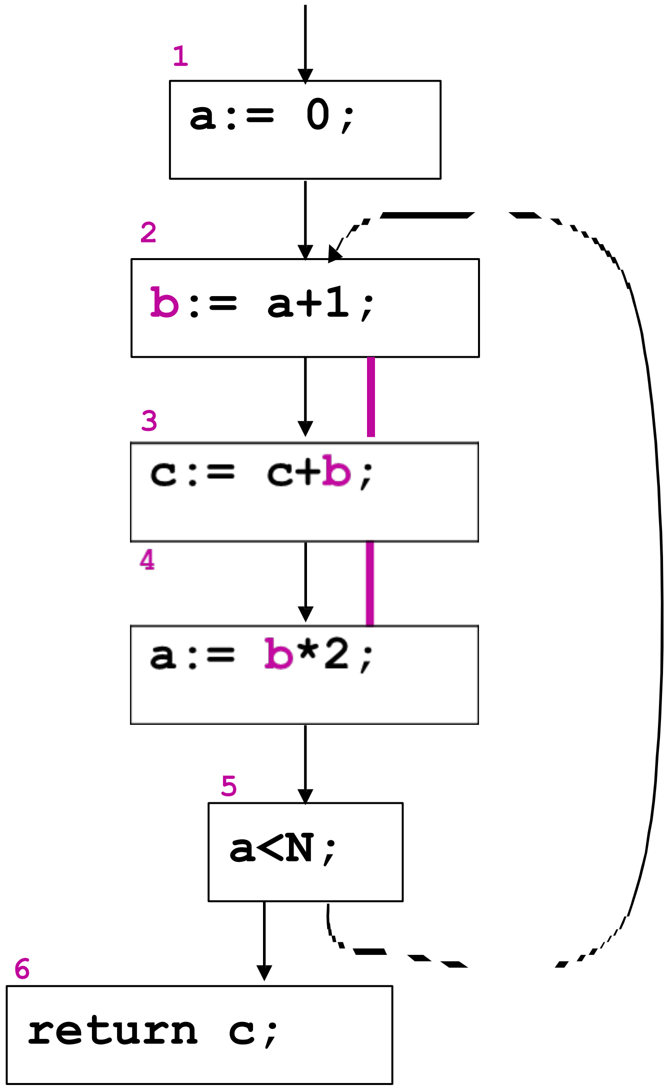
- Determine whether a given variable is used along a path from a given point to the exit.
- A variable x is *live at point p* if there is a path from p to the exit along which the value of x is used before it is redefined.
- Otherwise, the variable is dead at that point.
- Used in :
  - register allocation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

Example

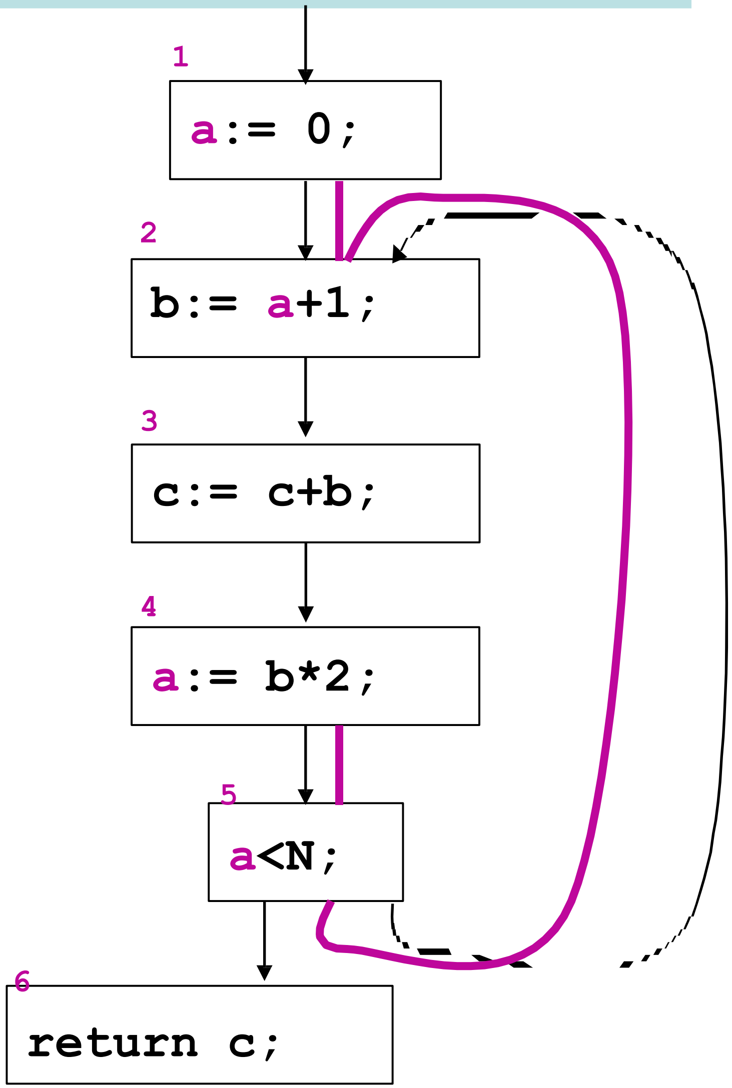
```
a = 0;
do{
  b= a+1;
  c+=b;
  a=b*2;
}
while (a<N);
return c;
```



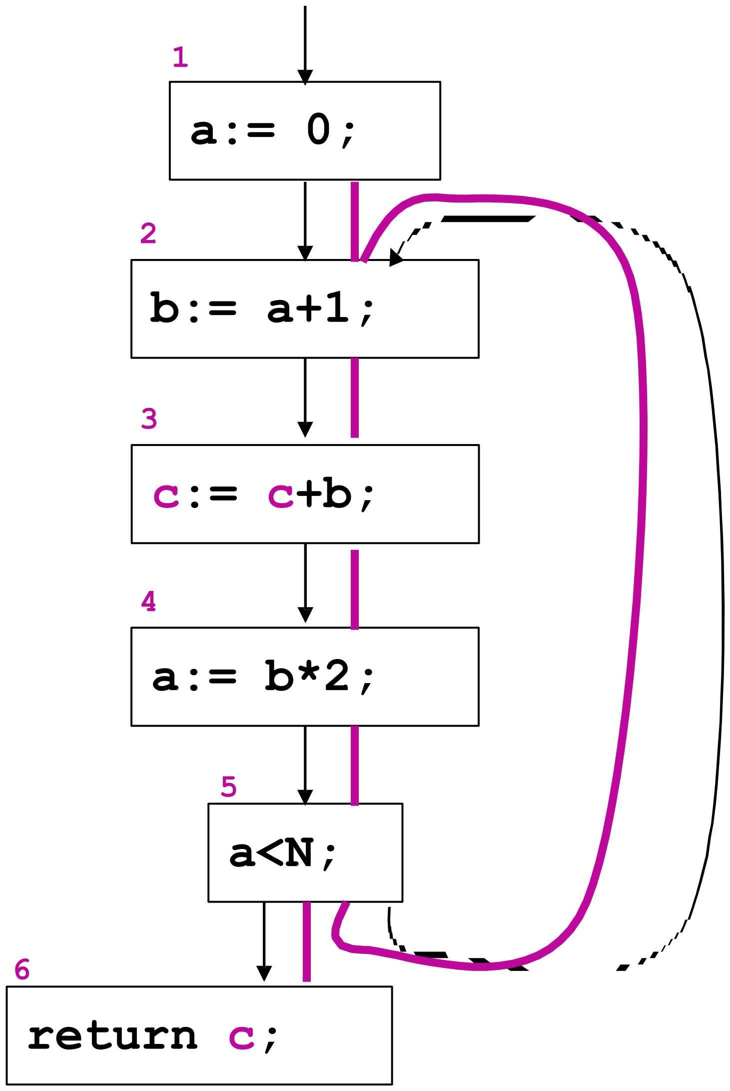
- Statement 4 makes use of variable b, then b is live in in(4) and in out(3)
- Block 3 dos not define b, then b is live also in in(3), and so in out(2)
- Block 2 defines b. Therefore the b is not live anymore in in(2).
- The "live range" of variable b is: $\{2 \rightarrow 3, 3 \rightarrow 4\}$

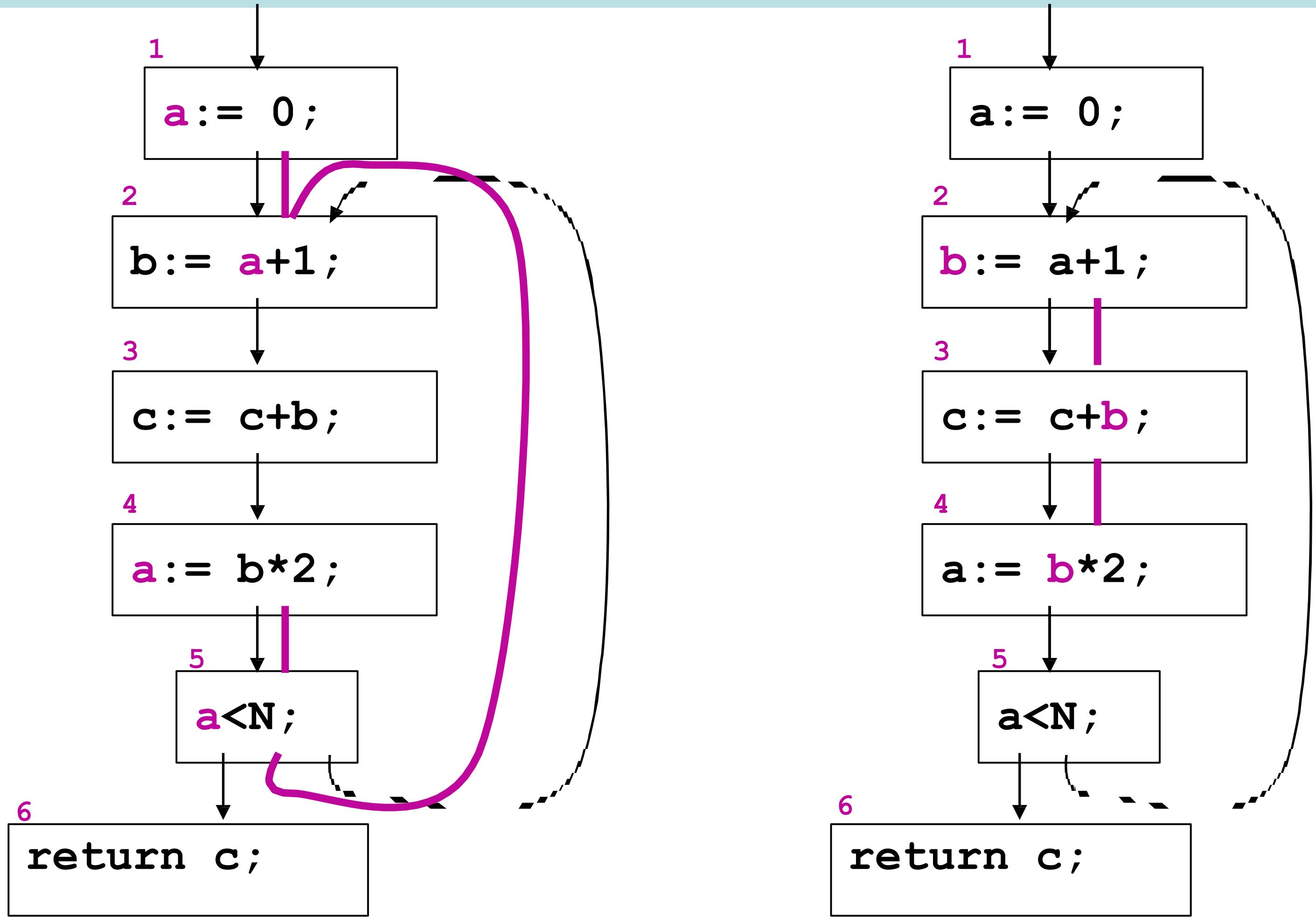


- a is live on 4 → 5 and 5 → 2
- a is live on 1 → 2
- It is dead on 2 → 3 → 4



- 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

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

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

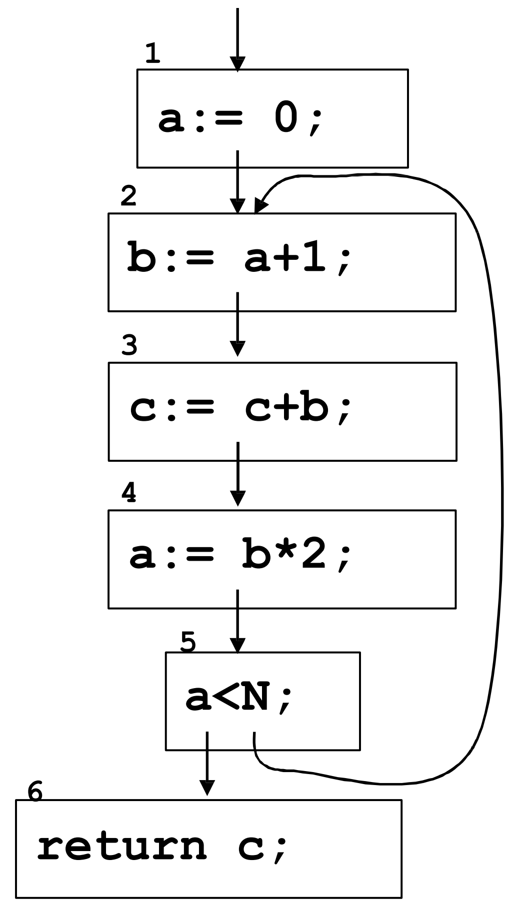
$$LV_{\texttt{entry}}(p) = gen_{LV}(p) \cup (LV_{\texttt{exit}}(p) \setminus kill_{LV}(p))$$

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])
```

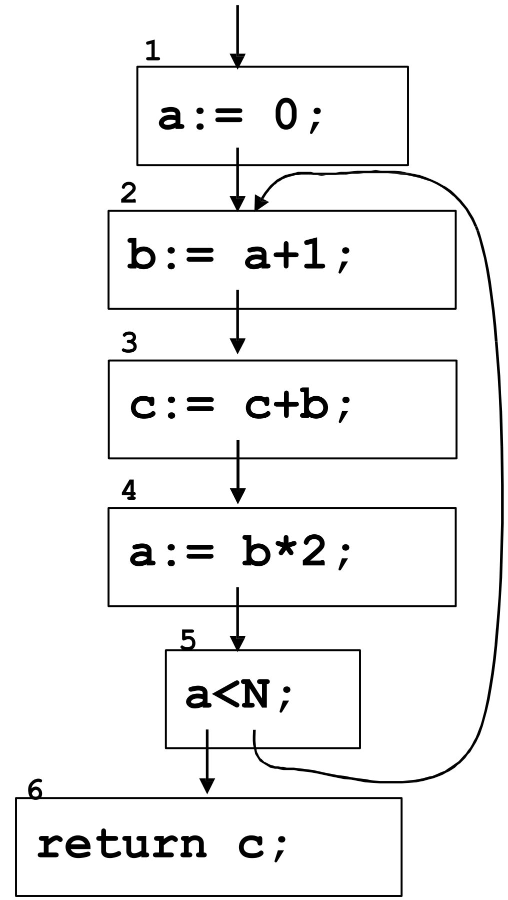
```
for each n
   in[n]:={}; out[n]:={}
repeat
   for each n
        in'[n]:=in[n]; out'[n]:=out[n]
        in[n] := use[n] U (out[n] - def[n])
        out[n]:= U { in[m] | m ∈ succ[n]}
until (for each n: in'[n]=in[n] && out'[n]=out[n])
```

			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	



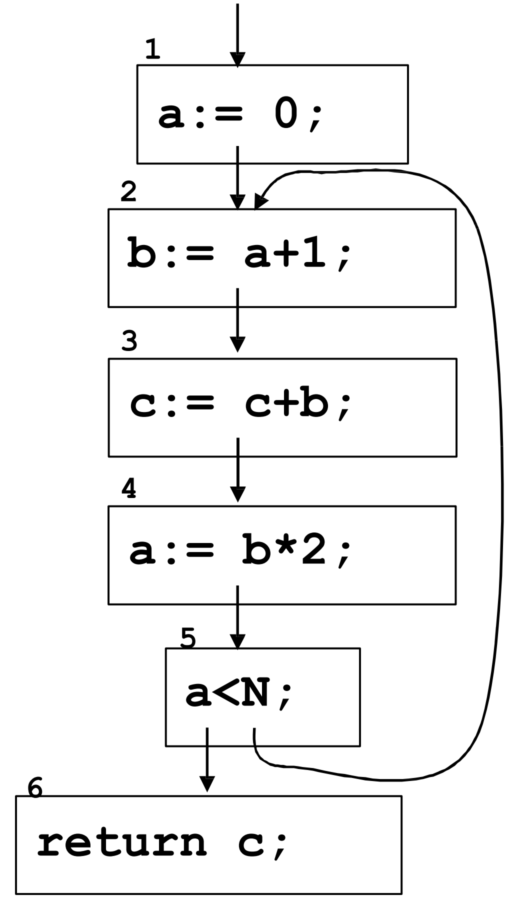
```
for each n
    in[n]:={}; out[n]:={}
repeat
    for each n
          in'[n]:=in[n]; out'[n]:=out[n]
          in[n] := use[n] U (out[n] - def[n])
          out[n]:= U { in[m] | m ∈ succ[n]}
until (for each n: in'[n]=in[n] && out'[n]=out[n])
```

			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 (for each 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 CFG 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] ∪ (out[n] - def[n])
2. out[n] = ∪ { in[m] | m ∈ 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 with emptyset!

Avaliable Expressions: equations

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

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

Equations

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

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}

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

$$AE_{exit}(p) = (AE_{entry}(p) \setminus kill_{AE}(p)) \;\mathrm{U}\; gen_{AE}(p)$$

$AE_{entry}(1)=$ ⑦

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

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

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

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

$AE_{exit}(1)= AE_{entry}(1) \;\mathrm{U}\; \{a+b\}$

$AE_{exit}(2)= AE_{entry}(2) \;\mathrm{U}\; \{a*b\}$

$AE_{exit}(3)= AE_{entry}(3) \;\mathrm{U}\; \{a+b\}$

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

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

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

- Even though the expression a is redefined in the cycle (in 4), the expression a+b is always available at 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 very busy 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)

Very Busy Expressions

- How are the **gen** and **kill** sets defined?
  - **gen**[B] = {all expressions evaluated in B before any definitions of their operands}
  - **kill**[B] = {all expressions whose operands are defined in B before any possible re-evaluation}
- What is the direction of the analysis?
  - backward
  - **in**[B] = **gen**[B] $\cup$ (**out**[B] - **kill**[B])

Very Busy Expressions

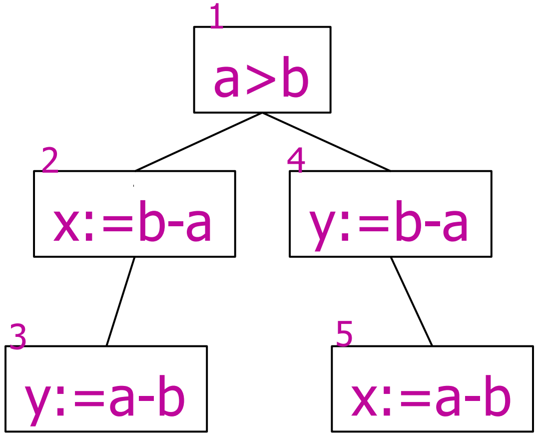
- What is the confluence operator?
  - intersection
  - **out**[B] = $\cap$ **in**[S], over the successors S of B

Very Busy Expressions: equations

$$VB_{\texttt{exit}}(p) = \begin{cases} \emptyset & \text{if } p \text{ is final} \\ \cap \{ VB_{\texttt{entry}}(q) \mid (p,q) \text{ in the CFG} \} & \end{cases}$$

$$VB_{\texttt{entry}}(p) = (VB_{\texttt{exit}}(p) \setminus kill_{VB}(p)) \cup gen_{VB}(p)$$

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



$VB_{entry}(1)=VB_{exit}(1)$

$VB_{entry}(2)=VB_{exit}(2) \cup \{b-a\}$

$VB_{entry}(3)=\{a-b\}$

$VB_{entry}(4)=VB_{exit}(4) \cup \{b-a\}$

$VB_{entry}(5)=\{a-b\}$

$VB_{exit}(1)= VB_{entry}(2) \cap VB_{entry}(4)$

$VB_{exit}(2)= VB_{entry}(3)$

$VB_{exit}(3)= \emptyset$

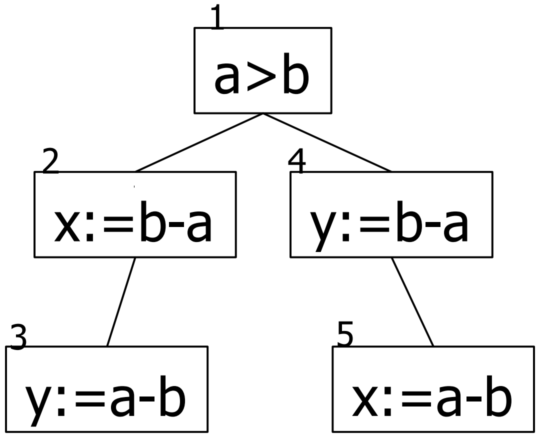
$VB_{exit}(4)= VB_{entry}(5)$

$VB_{exit}(5)= \emptyset$

n	**$kill_{VB}(n)$**	**$gen_{VB}(n)$**
1	$\emptyset$	$\emptyset$
2	$\emptyset$	{b-a}
3	$\emptyset$	{a-b}
4	$\emptyset$	{b-a}
5	$\emptyset$	{a-b}

Result

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



n	**VB$_{\mathtt{entry}}$(n)**	**VB$_{\mathtt{exit}}$(n)**
1	{a-b, b-a}	{a-b, b-a}
2	{a-b, b-a}	{a-b}
3	{a-b}	$\varnothing$
4	{a-b, b-a}	{a-b}
5	{a-b}	$\varnothing$