Iterative algorithm

Compute out[*n*] *for each* $n \in N$:

out[*n*] := $\perp$ (or TOP if MUST analysis)

Repeat

  For each *n*

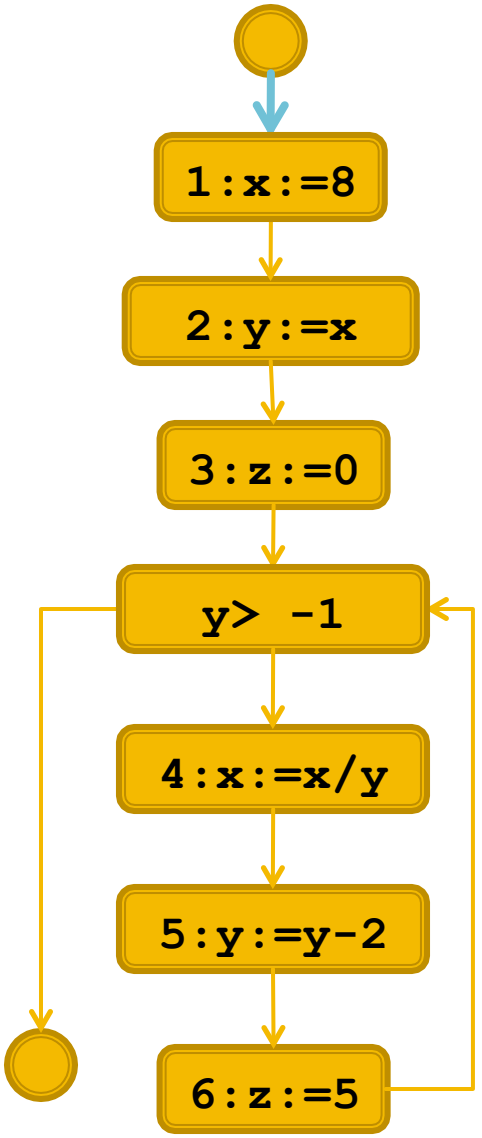
    in[*n*] := $\bigoplus$ { ***out***[*m*] | $m \in$ ***pred***(*n*) }

    out[*n*] := ***transfer***[*n*](***in***[*n*])

Until no further changes to **in**/**out**

Zero analysis

```
x := 8;

y := x;

z := 0;

while y > -1 do

    x := x / y;

    y := y-2;

    z := 5;
```



Zero analysis

lattice : (var → {bot,Z,NZ,MZ})

-bot = [] , top={x→MZ,y→MZ,z→MZ}

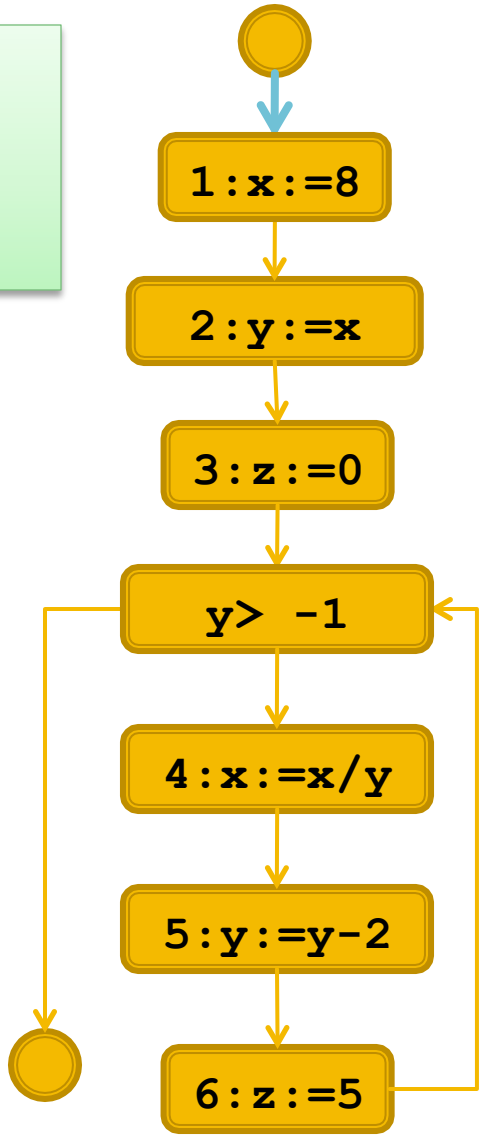
- ∪ , ⊆

-transfer:

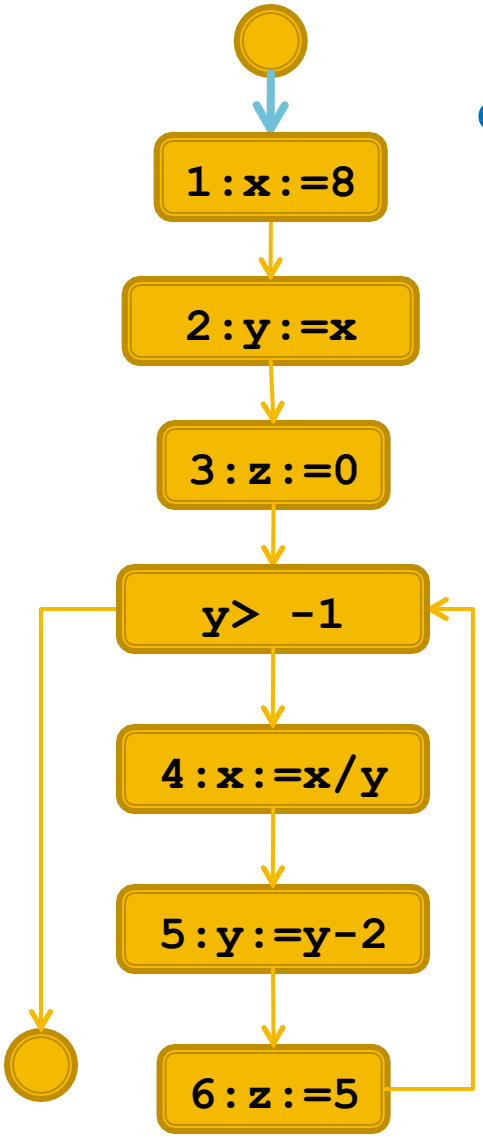
- $F_{x:=e}(\sigma) = \sigma[x \leftarrow \sigma(e)]$
- σ(Z+NZ)=NZ
- σ(Z+Z)=Z
- σ(Z-Z)=Z
- σ(NZ-NZ)=MZ
- …

$$\textbf{\textit{in}}[n] := \cup \{ \textbf{\textit{out}}[m] \mid m \in \textbf{\textit{pred}}(n) \}$$

$$\textbf{\textit{out}}[n] := \textbf{\textit{transfer}}[n](\textbf{\textit{in}}[n])$$

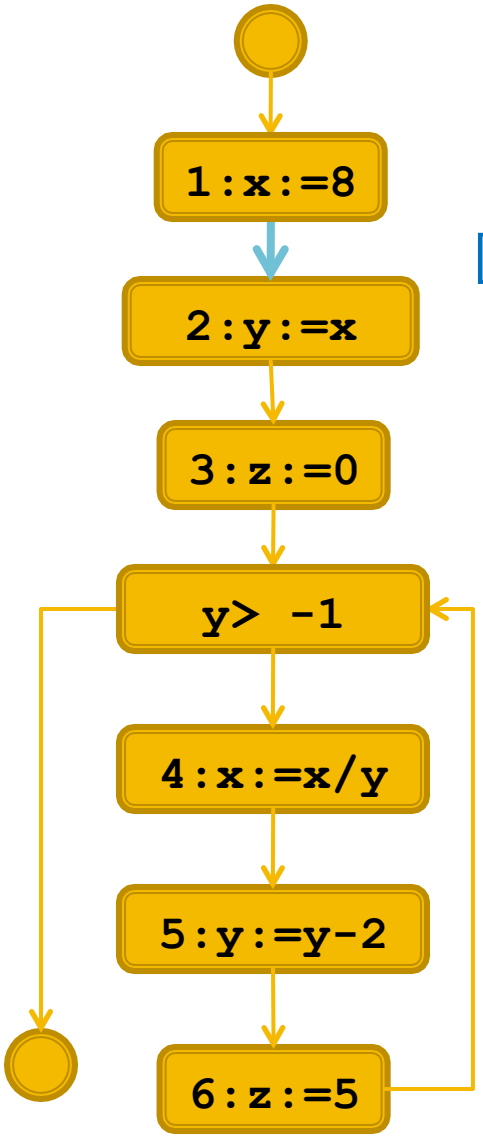


Example



$\sigma = []$

Example

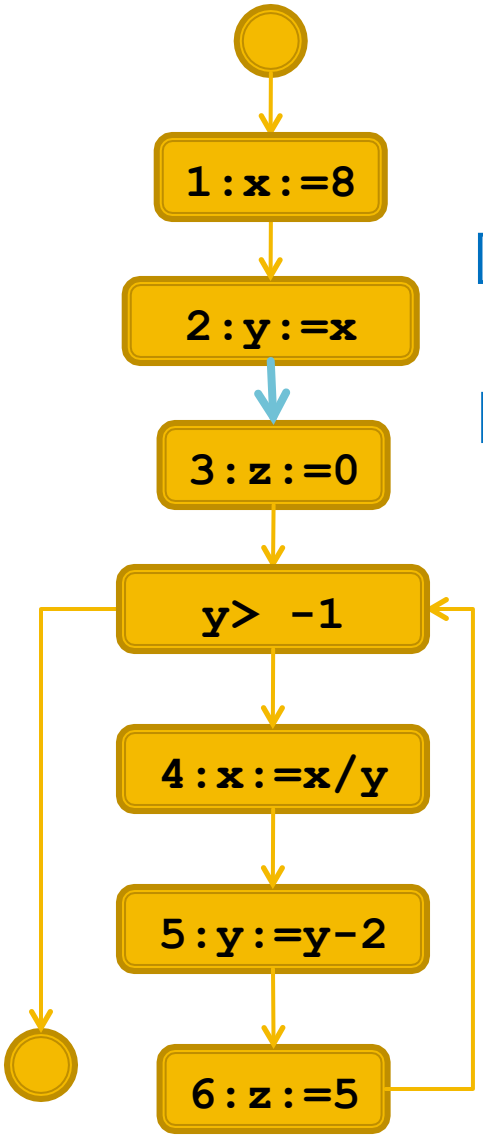


σ = []

$\alpha(8)$=NZ

[x➔ NZ]

Example

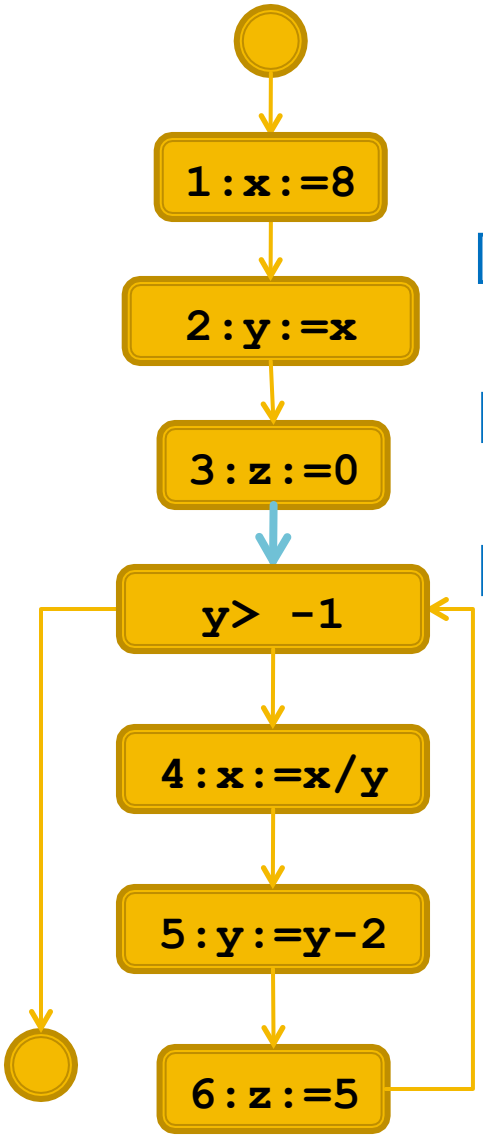


σ = []

[x➔ NZ]

[x➔ NZ, y ➔ **NZ**]

Example



$\sigma = []$

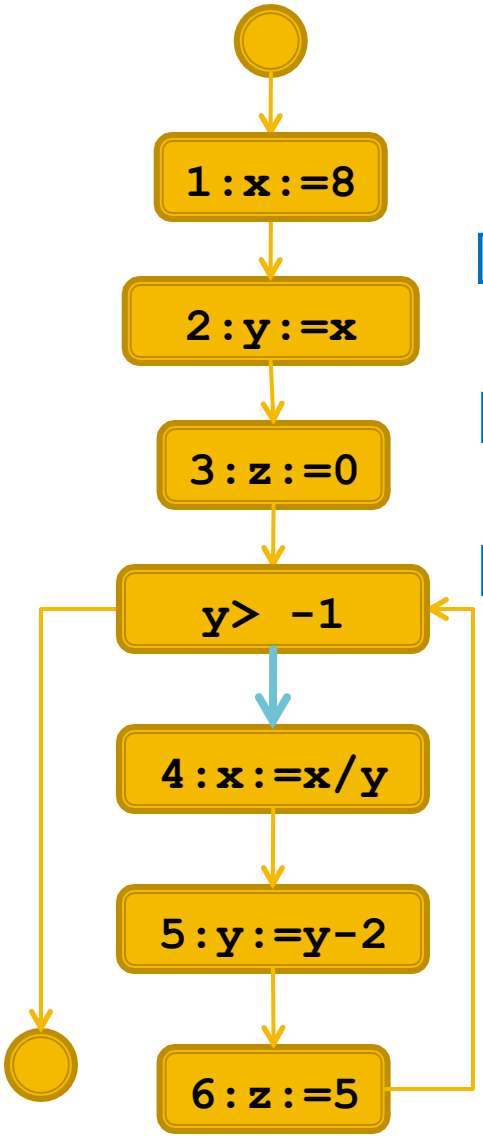
$[x \rightarrow NZ]$

$[x \rightarrow NZ, y \rightarrow NZ]$

$[x \rightarrow NZ, y \rightarrow NZ, z \rightarrow Z]$

$\alpha(o) = Z$

Example



σ = []

[x ➔ NZ]

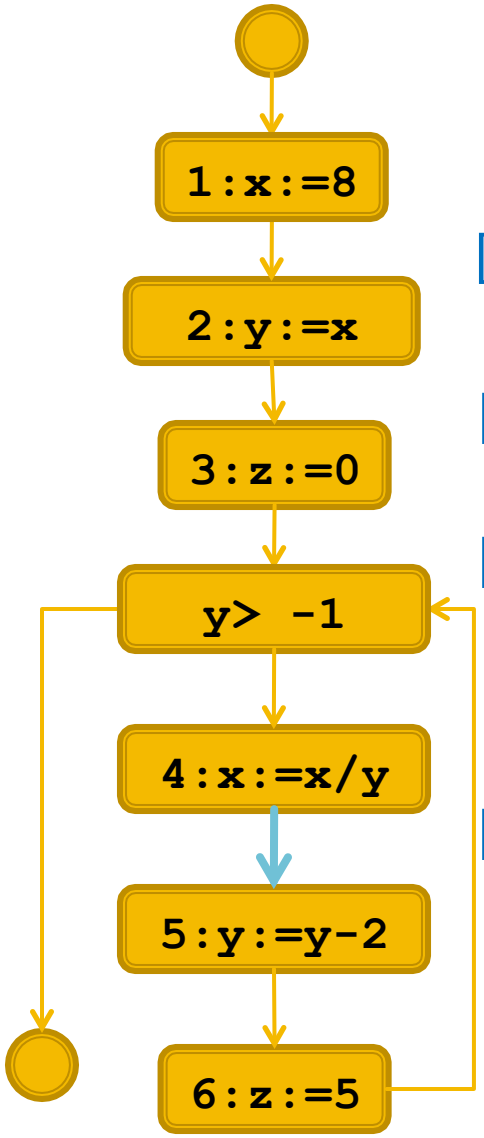
[x ➔ NZ, y ➔ NZ]

[x ➔ NZ, y ➔ NZ, z ➔ Z]

[x ➔ NZ, y ➔ NZ, z ➔ Z]

while evaluation is assignment free

Example



σ = []

[x ➔ NZ]

[x ➔ NZ, y ➔ NZ]

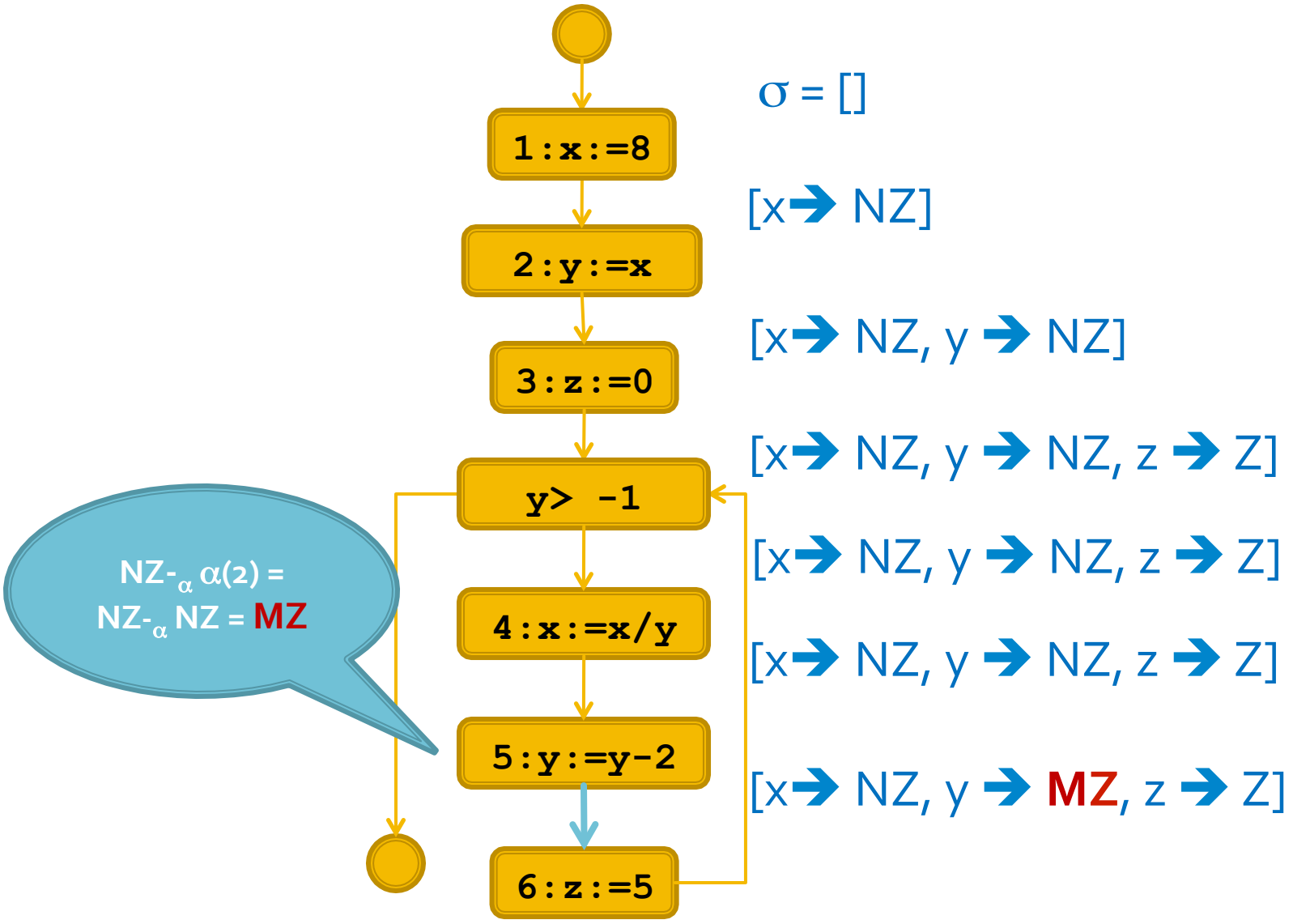
[x ➔ NZ, y ➔ NZ, z ➔ Z]

[x ➔ NZ, y ➔ NZ, z ➔ Z]

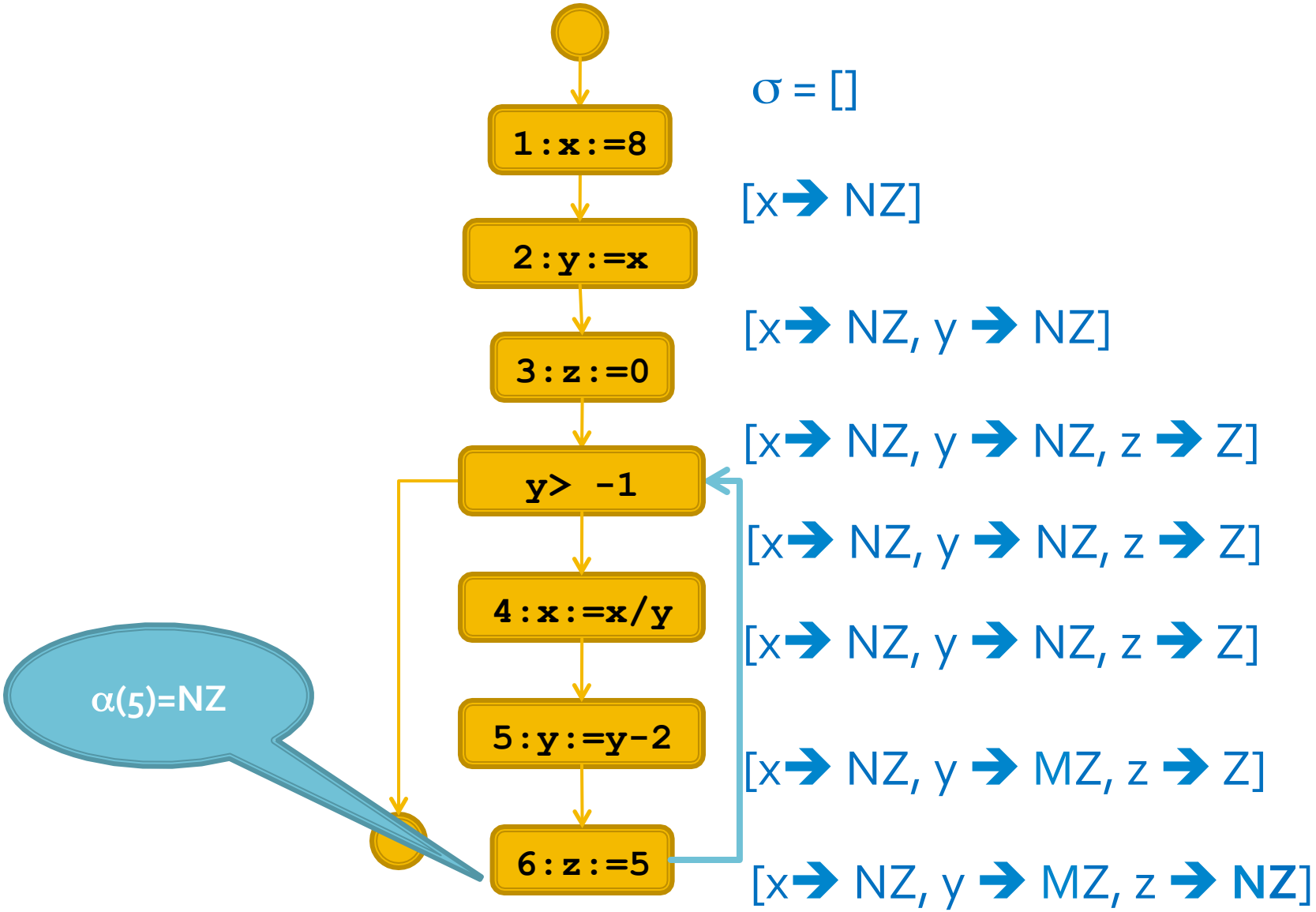
[x ➔ **NZ**, y ➔ NZ, z ➔ Z]

$NZ /_{\alpha} Y = NZ$

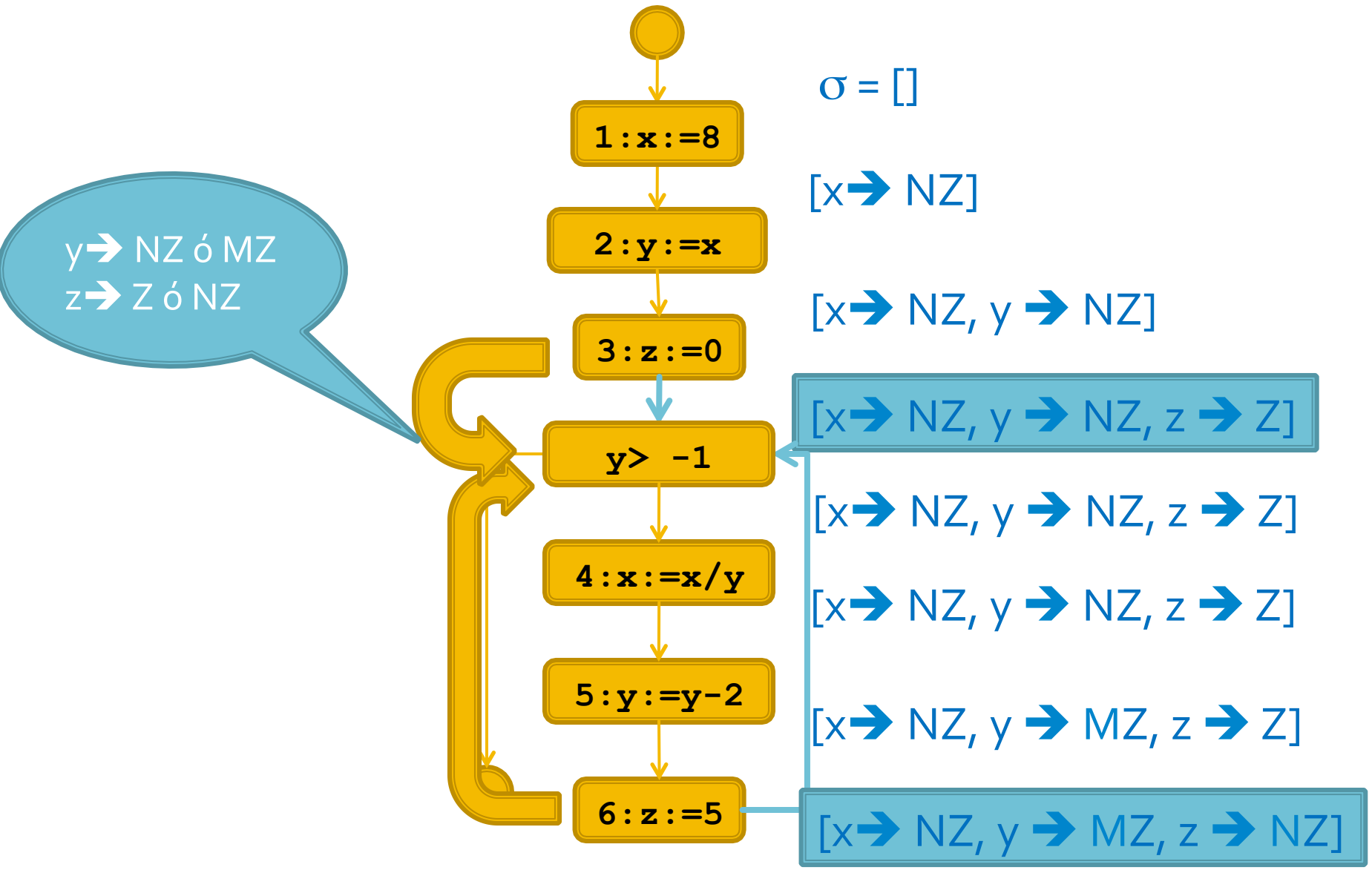
Example


1 : x := 8
2 : y := x
3 : z := 0
y > -1
4 : x := x / y
5 : y := y - 2
6 : z := 5
NZ -α α(2) =
NZ -α NZ = MZ
σ = []
[x ➔ NZ]
[x ➔ NZ, y ➔ NZ]
[x ➔ NZ, y ➔ NZ, z ➔ Z]
[x ➔ NZ, y ➔ NZ, z ➔ Z]
[x ➔ NZ, y ➔ NZ, z ➔ Z]
[x ➔ NZ, y ➔ MZ, z ➔ Z]

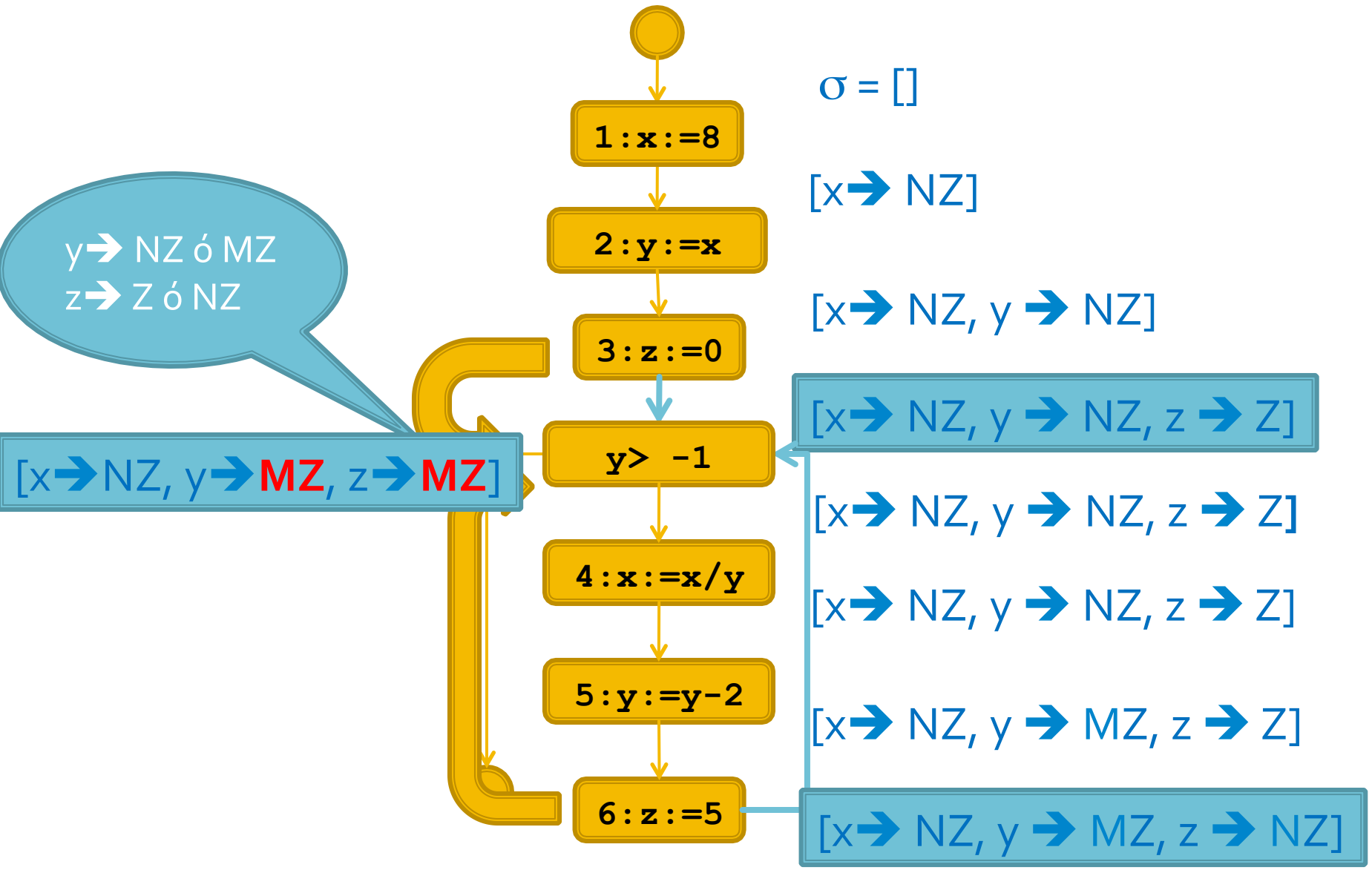
Example


1 : x := 8
2 : y := x
3 : z := 0
y > -1
4 : x := x / y
5 : y := y - 2
6 : z := 5
α(5)=NZ
σ = []
[x ➔ NZ]
[x ➔ NZ, y ➔ NZ]
[x ➔ NZ, y ➔ NZ, z ➔ Z]
[x ➔ NZ, y ➔ NZ, z ➔ Z]
[x ➔ NZ, y ➔ NZ, z ➔ Z]
[x ➔ NZ, y ➔ MZ, z ➔ Z]
[x ➔ NZ, y ➔ MZ, z ➔ NZ]

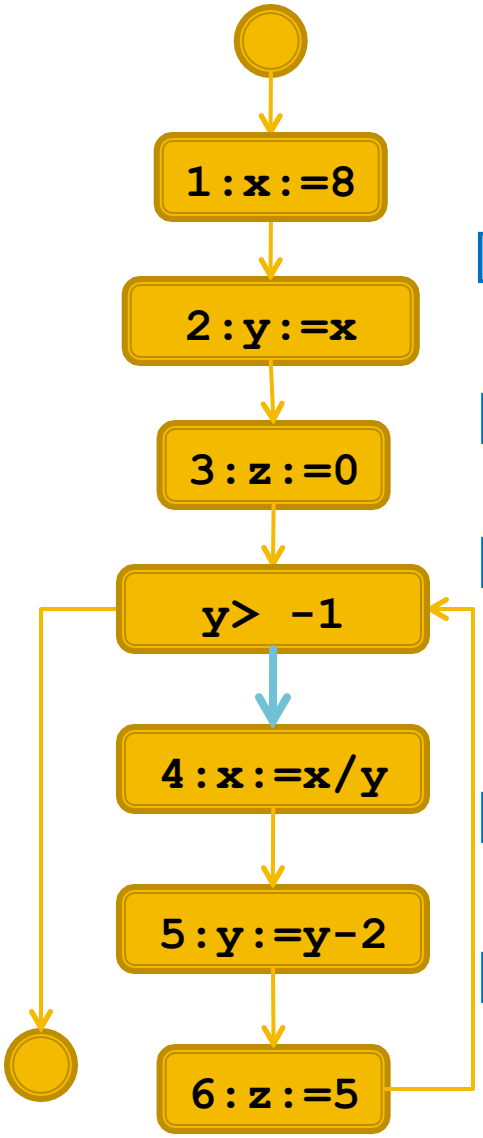
Example


y➔ NZ ó MZ
z➔ Z ó NZ
1 : x : =8
2 : y : =x
3 : z : =0
y> -1
4 : x : =x/y
5 : y : =y-2
6 : z : =5
σ = []
[x➔ NZ]
[x➔ NZ, y ➔ NZ]
[x➔ NZ, y ➔ NZ, z ➔ Z]
[x➔ NZ, y ➔ NZ, z ➔ Z]
[x➔ NZ, y ➔ NZ, z ➔ Z]
[x➔ NZ, y ➔ MZ, z ➔ Z]
[x➔ NZ, y ➔ MZ, z ➔ NZ]

Example


σ = []
1 : x := 8
[x➔ NZ]
2 : y := x
[x➔ NZ, y ➔ NZ]
3 : z := 0
[x➔ NZ, y ➔ NZ, z ➔ Z]
y➔ NZ ó MZ
z➔ Z ó NZ
[x➔NZ, y➔MZ, z➔MZ]
y> -1
[x➔ NZ, y ➔ NZ, z ➔ Z]
4 : x := x/y
[x➔ NZ, y ➔ NZ, z ➔ Z]
5 : y := y-2
[x➔ NZ, y ➔ MZ, z ➔ Z]
6 : z := 5
[x➔ NZ, y ➔ MZ, z ➔ NZ]

Example



σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

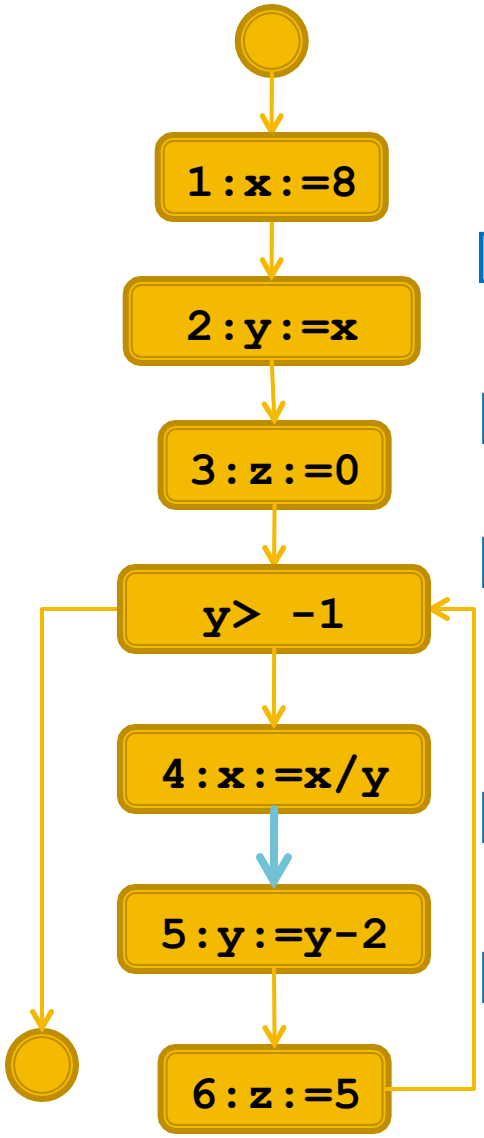
[x➔ NZ, y ➔ **MZ** ➔ **MZ**]

[x➔ NZ, y ➔ MZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ NZ]

Example



σ = []

[x ➔ NZ]

[x ➔ NZ, y ➔ NZ]

[x ➔ NZ, y ➔ NZ, z ➔ Z]

[x ➔ NZ, y ➔ MZ, z ➔ MZ]

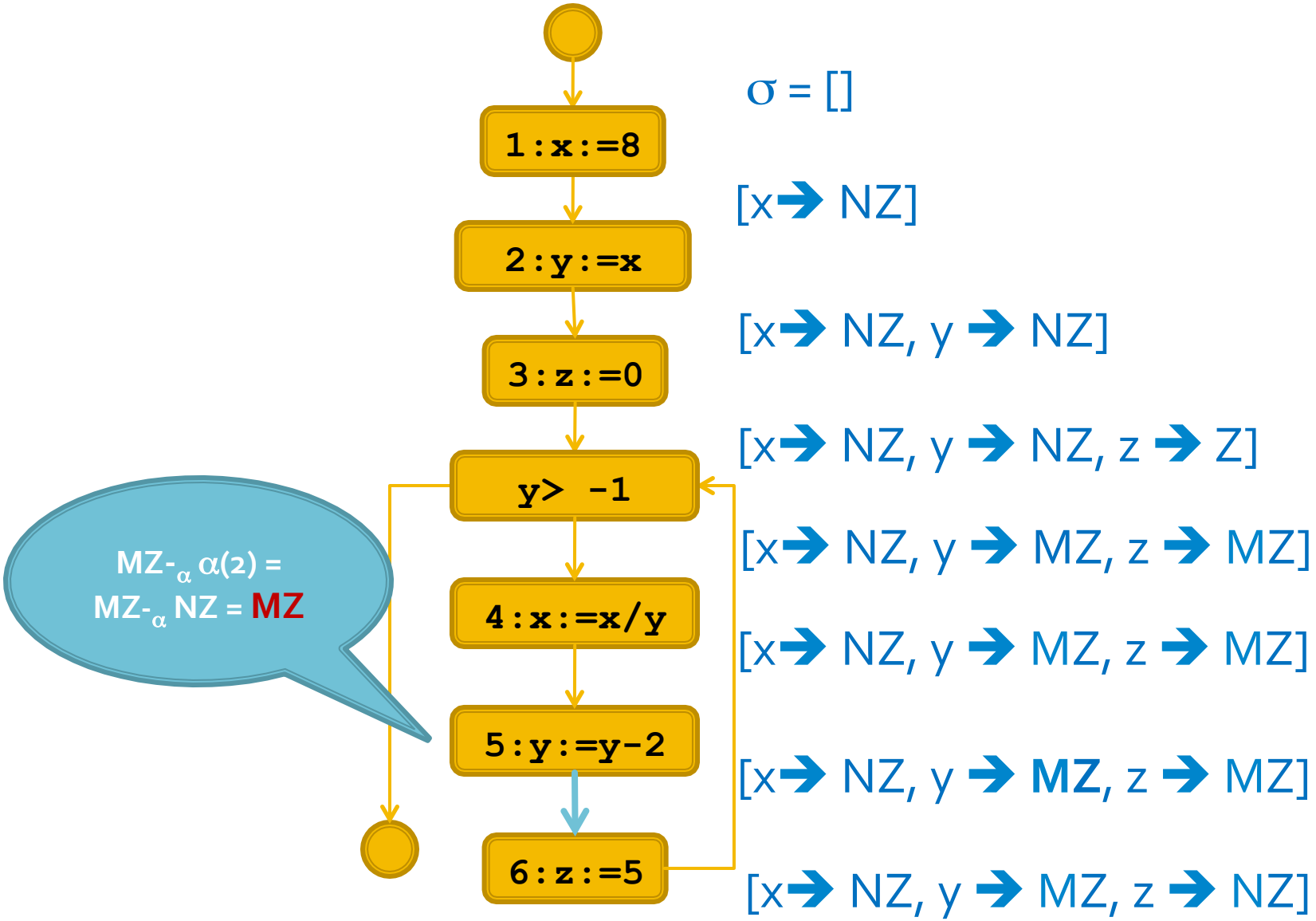
[x ➔ **NZ**, y ➔ MZ, z ➔ MZ]

[x ➔ NZ, y ➔

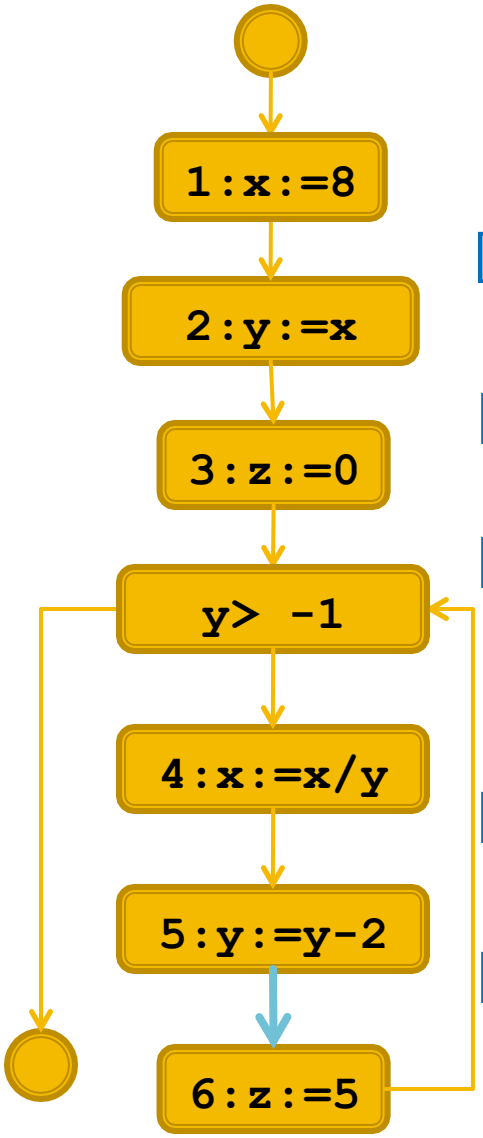
[x ➔ NZ, y ➔

$NZ /_{\alpha} Y = NZ$

Example


1 : x := 8
2 : y := x
3 : z := 0
y > -1
4 : x := x / y
5 : y := y - 2
6 : z := 5
MZ-α α(2) =
MZ-α NZ = MZ
σ = []
[x ➔ NZ]
[x ➔ NZ, y ➔ NZ]
[x ➔ NZ, y ➔ NZ, z ➔ Z]
[x ➔ NZ, y ➔ MZ, z ➔ MZ]
[x ➔ NZ, y ➔ MZ, z ➔ MZ]
[x ➔ NZ, y ➔ MZ, z ➔ MZ]
[x ➔ NZ, y ➔ MZ, z ➔ NZ]

Example



σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ **NZ**]

DF Analysis convergence

Iteration #2

σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ NZ]

No changes! Fix-point!

Iteration #3

σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ NZ]

Example: Using the result

σ = []

```
x := 8;
```

σ = [x➔ NZ]

```
y := x;
```

σ = [x➔ NZ, y➔ NZ]

```
z := 0;
```

σ = [x➔ NZ, y➔ NZ, z➔ Z]

```
while y > -1 do
```

σ = [x➔ NZ, **y➔ MZ**, z➔ MZ]

```
    x := x / y;
```

σ = [x➔ NZ, y➔ MZ, z➔ MZ]

```
    y := y-2;
```

σ = [x➔ NZ, y➔ **MZ**, z➔ MZ]

```
    z := 5;
```

σ = [x➔ NZ, y➔ MZ, z➔ NZ]

Warning: This program ***might*** produce a division by zero

Applying Zero Analysis

Interpreting the result:

- Visit each expression of the form X/Y in the program
- Look at the result from the zero analysis for that program point for the variable Y
  - IF Y = NZ, OK!

Foundamental Property: Given a program **P** y an analysis **A** searching for errors of type E.
"Sound error detection":

•If an error e exists in P => **A finds it**

Enhancing precision

- What happens with this example?
  - y = 3;
    y:= y-1;
    x = 6/y
  - Analysis is conservative… it reports a warning when it is clearly valid
- Solution?: More precise abstraction.
  - Example intervals:
  - σ = []
  - y = 3;
  - σ = [y➔ [3,3]]
  - y:= y-1;
  - σ = [y➔ [2,2]]
  - x = 6/y
  - σ = [x➔ [3,3], y➔ [2,2]]
- More precision leads to more computational cost.
  - The intervals abstraction requires 9 iterations instead of just 2.
  - Exercise!

Iterative algorithm

Compute out[*n*] *for each* $n \in N$:

out[*n*] := $\perp$ (or TOP if MUST analysis)

Repeat

  For each *n*

    in[*n*] := $\oplus$ { ***out***[*m*] | $m \in$ ***pred***(*n*) }

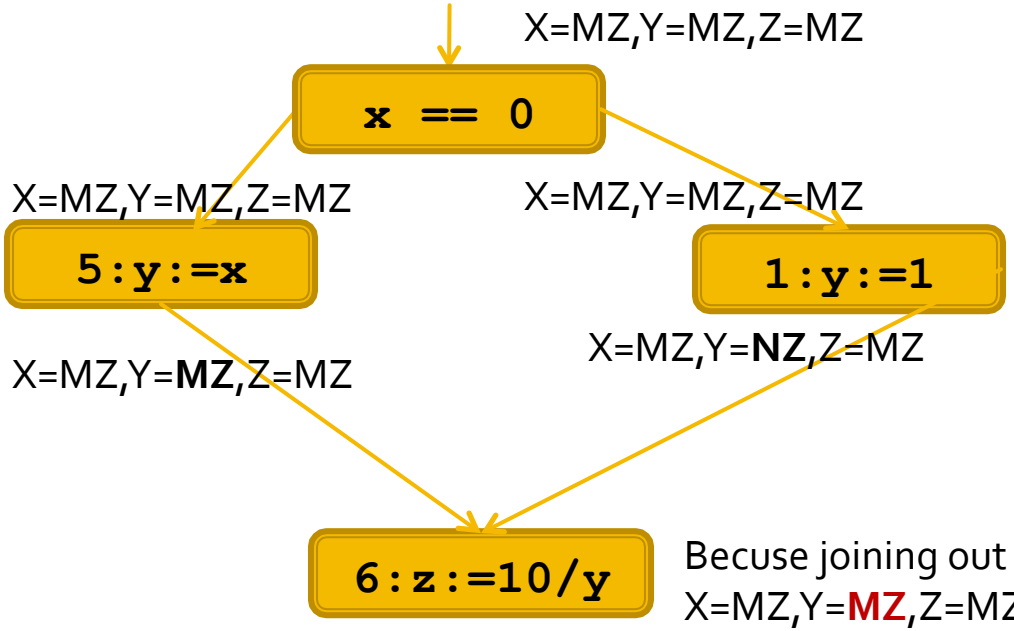
    out[*n*] := ***transfer***[*n*](***in***[*n*])

Until no further changes to **in**/**out**

- Questions:
  - Does it output a solution?
  - Does it produce the best possible solution?
  - Does the analysis terminate?
- It depends on the properties of the abstraction lattice and the transfer function

Precision problems

- What about branches?



Transfer function:

- f(σ, [x := y]) = [x -> σ(y)] σ
- f(σ, [x := n]) = if n==o
  then [x-> Z]σ
  else [x-> NZ]σ
- f(σ, [x := y op z]) = [x- > MZ] σ
- f(σ, /* any other */) = σ

- Analysis do not profit from branching flow information

Enhancing Precision

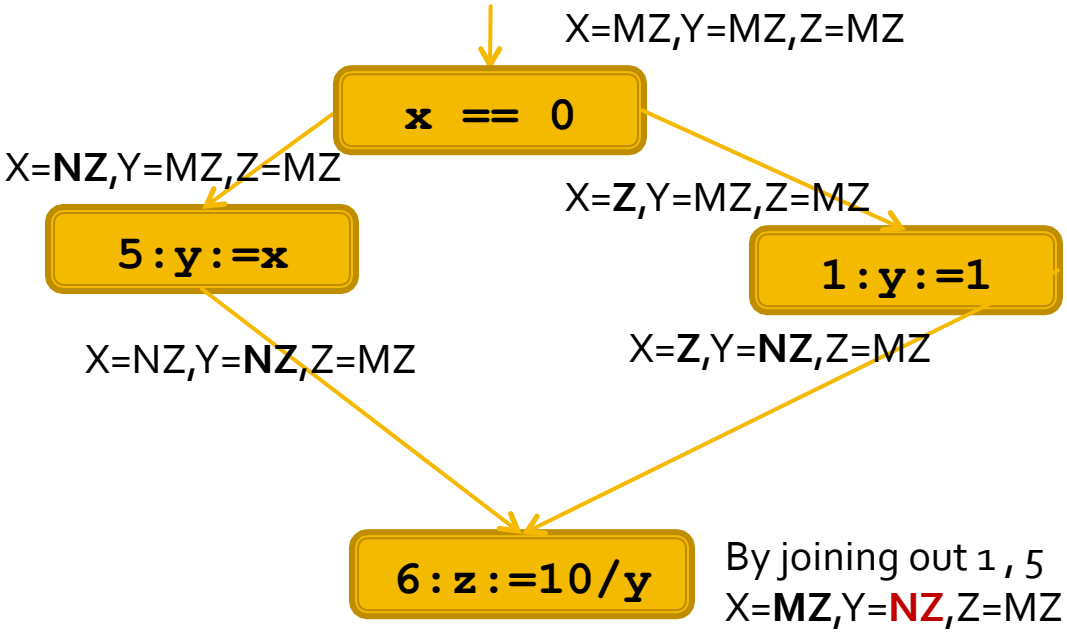
- Idea: Propagate branch information

Transfer function:

- f(σ, [x := y]) = [x -> σ(y)] σ
- f(σ, [x := n]) = if n==o
  then [x-> Z]σ
  else [x-> NZ]σ
- f(σ, [x := y op z]) = [x- > MZ] σ
- f(σ, /* any other */) = σ

- fT(σ, [x == o]) = [x -> Z] σ
- fF(σ, [x == o]) = [x -> NZ] σ

In general:
- fT(σ, [x == y]) = [x -> σ(y)] σ
- fF(σ, [x == y]) = [x -> !σ(y)] σ



(Forward) work-list algorithm

- More efficient

Compute out[*n*] *for each n* ∈ *N*:
out[*n*] := ⊥
work.add= {entry}
WHILE *work* is not empty:
  n:= *work*.pop();
  in'[*n*] := ⊕{ ***out***[*m*] | *m* ∈ ***pred***(*n*) }
  out'[*n*] := ***transfer***[*n*](***in'***[*n*])
  IF !(***out'***[*n*] ⊆ ***out***[*n*])
    for each m ∈ ***succ***(*n*) *work.add(m);*
  out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

TOP if Must Analysis

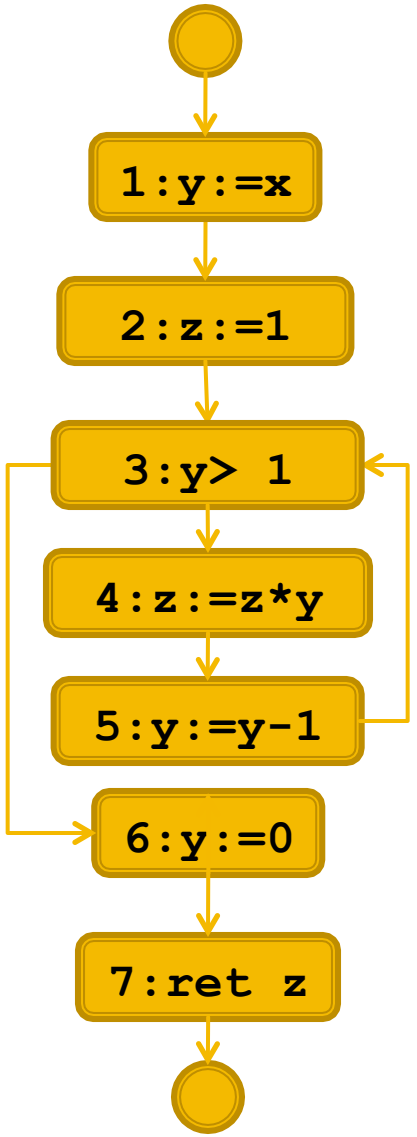
Exit if backward Analysis.

Succ if backward and replace in x out

Predecessors if backward

Example: Live variables

```
y := x;
z := 1;
while y>1
{
    z := z * y;
    y := y - 1;
}
y := 0;
return z;
```



Live variable analysis

Lattice: P({x,yz})

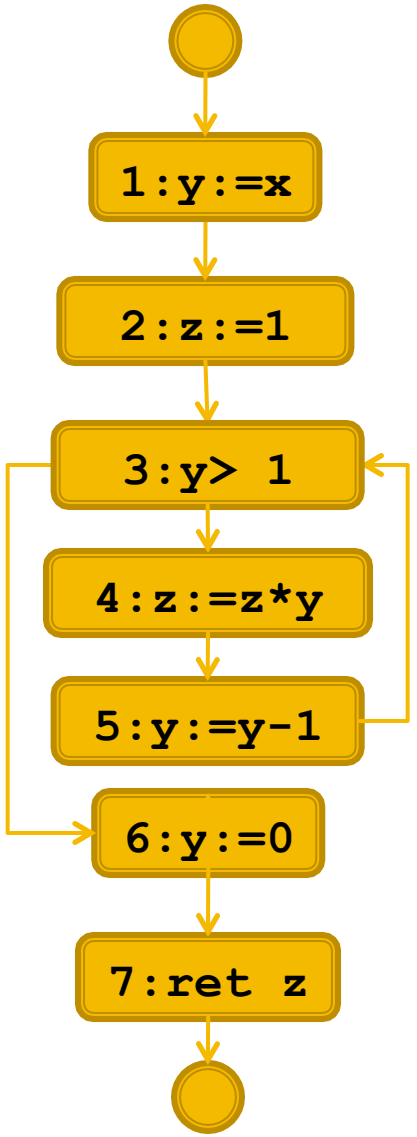
-bot = {}, top={x,y,z}, $\cup$, $\subseteq$

-Transfer:

- F(σ, *e*) = σ $\cup$ vars(*e*)
- F(σ, *x*:=*e*) = (σ - { *x* }) $\cup$ vars(*e*)
- F(σ, any other) = σ

out[*n*] := $\cup$ { ***in***[*m*] | *m* $\in$ ***succ***(*n*) }

in[n] := ***transfer***[*n*](***out***[*n*])



Live variable analysis

Lattice: P({x, y, z})

Transfer function:

- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x*:=*e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, other case) = σ

out[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in[n] := ***F(out***[*n*],*stm*[*n*])

Compute in[*n*] *for each n* ∈*N:*

out[*n*] := ⊥

work.add= {exit}

WILE *work* is not empty:

n:= *work*.pop();

out'[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

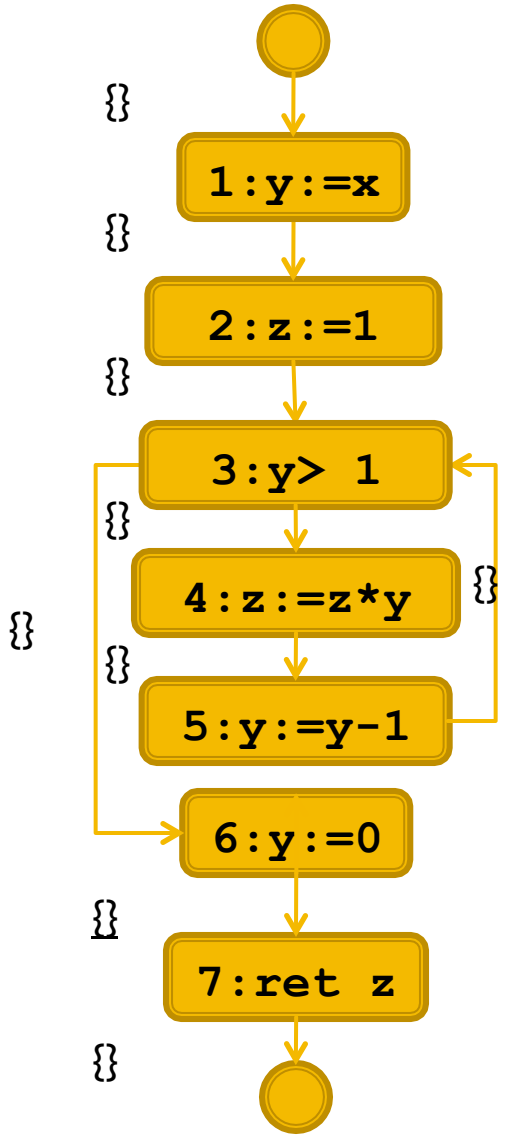
in'[*n*] := ***transfer***[*n*](***out'***[*n*])

IF !(*in'*[*n*] ⊆ ***in***[*n*])

for all m ∈ ***pred***(*n*) *work.add(m);*

out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

work = {exit}



Live variable analysis

Lattice: P({x, y, z})

Transfer function:

- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x*:=*e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, other case) = σ

out[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in[n] := ***F(out***[*n*],*stm*[*n*])

Compute in[*n*] *for each n* ∈*N:*

out[*n*] := ⊥

work.add= {exit}

WILE *work* is not empty:

n:= *work*.pop();

out'[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in'[*n*] := ***transfer***[*n*](***out'***[*n*])

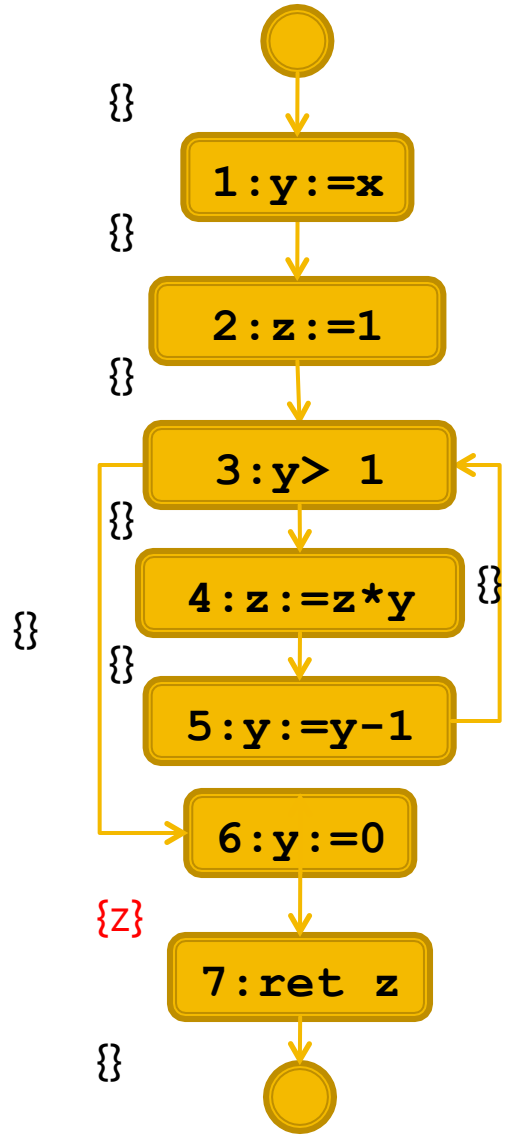
IF !(*in'*[*n*] ⊆ ***in***[*n*])

for all m ∈ ***pred***(*n*) *work.add(m);*

out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

work = {exit}
n = exit

out'[7] = {}
in' [7]= {z}
work' = {6}



Live variable analysis

Lattice: P({x, y, z})

Transfer function:

- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x*:=*e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, other case) = σ

$$\mathbf{out}[n] := \cup \{ \mathbf{in}[m] \mid m \in \mathbf{succ}(n) \}$$

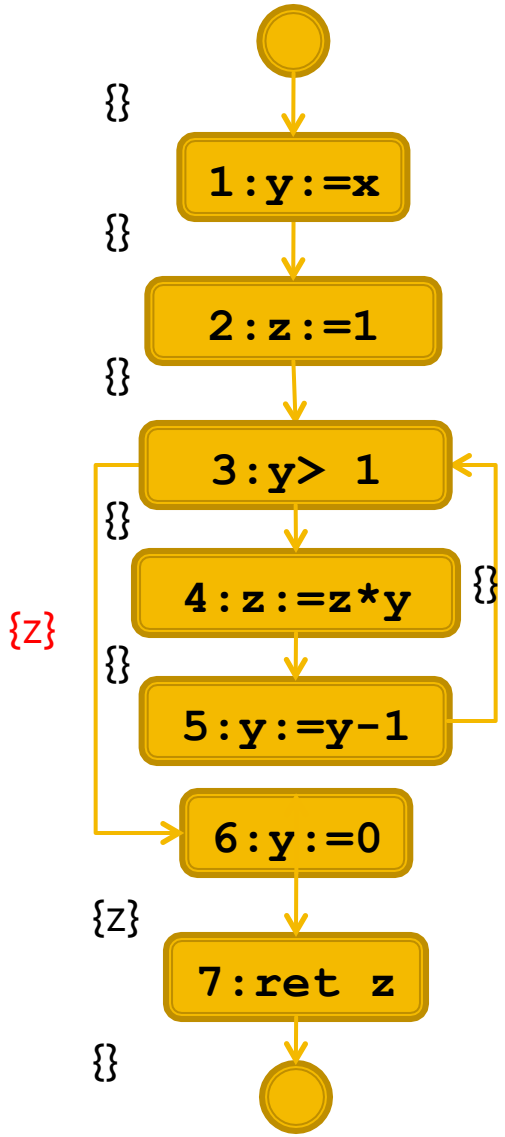
$$\mathbf{in}[n] := \mathbf{F}(\mathbf{out}[n], stm[n])$$

Compute in[*n*] *for each* $n \in N$:

```
out[n] := ⊥
work.add= {exit}
WILE work is not empty:
   n:= work.pop();
   out'[n] := ∪ { in[m] | m ∈ succ(n) }
   in'[n] := transfer[n](out'[n])
   IF !(in'[n] ⊆ in[n])
      for all m ∈ pred(n) work.add(m);
   out[n] := out'[n]; in[n] := in'[n];
```

work = {6}
n = 6

out'[6] = {z}
in' [6]= {z}
work' = {3}



Live variable analysis

Lattice: P({x, y, z})

Transfer function:

- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x*:=*e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, other case) = σ

out[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in[n] := ***F(out***[*n*],*stm*[*n*])

Compute in[*n*] *for each n* ∈*N:*

out[*n*] := ⊥

work.add= {exit}

WILE *work* is not empty:

n:= *work*.pop();

out'[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in'[*n*] := ***transfer***[*n*](***out'***[*n*])

IF !(*in'*[*n*] ⊆ ***in***[*n*])

for all m ∈ ***pred***(*n*) work.add(*m*);

out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

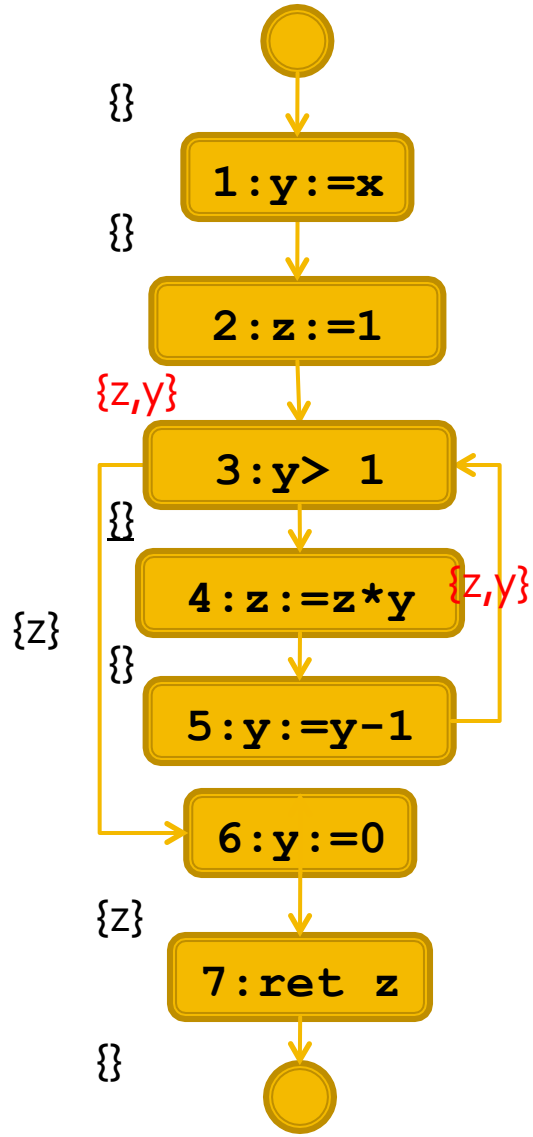
work = {3}

n = 3

out'[3] = {z} (de 6 y 4)

in'[3] = {z,y}

work' = {2,5}



Live variable analysis

Lattice: P({x, y, z})

Transfer function:

- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x*:=*e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, other case) = σ

out[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in[n] := ***F(out***[*n*],*stm*[*n*])

Compute in[*n*] *for each* *n* ∈*N:*

out[*n*] := ⊥

work.add= {exit}

WILE *work* is not empty:

n:= *work*.pop();

out'[*n*] := ∪ { ***in***[*m*] | *m* ∈***succ***(*n*) }

in'[*n*] := ***transfer***[*n*](***out'***[*n*])

IF !(*in'*[*n*] ⊆***in***[*n*])

for all m ∈***pred***(*n*) work.add(*m*);

out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

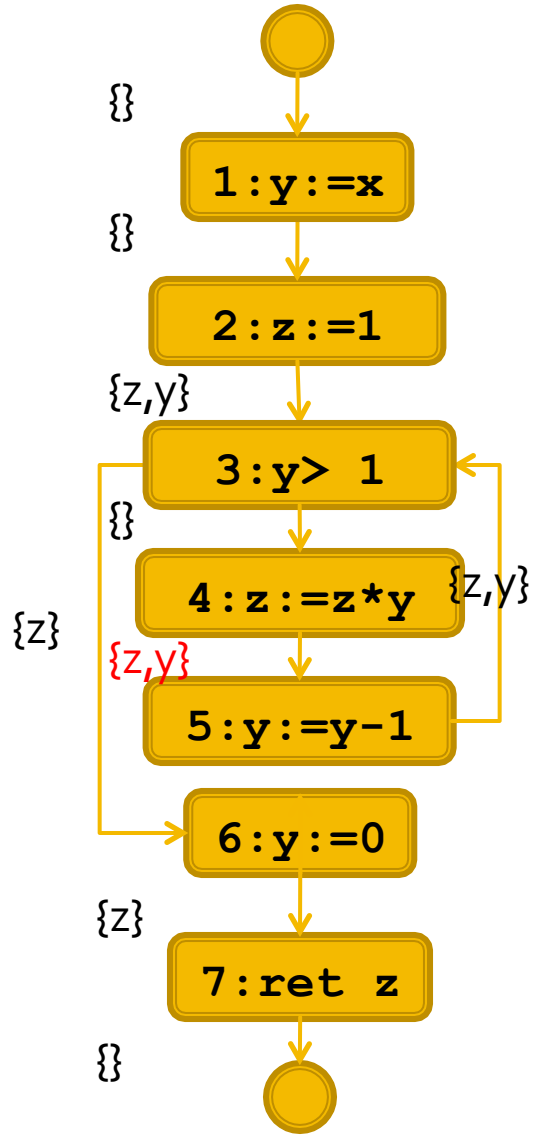
work = {2,5}

n = 5

out'[5] = {z,y}

in' [5]= {z,y}

work' = {2,4}



Live variable analysis

Lattice: P({x, y, z})

Transfer function:

- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x*:=*e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, other case) = σ

out[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in[n] := ***F(out***[*n*],*stm*[*n*])

Compute in[*n*] *for each n* ∈*N:*

out[*n*] := ⊥

work.add= {exit}

WILE *work* is not empty:

n:= *work*.pop();

out'[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in'[*n*] := ***transfer***[*n*](***out'***[*n*])

IF !(*in'*[*n*] ⊆ ***in***[*n*])

for all m ∈ ***pred***(*n*) *work.add(m);*

out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

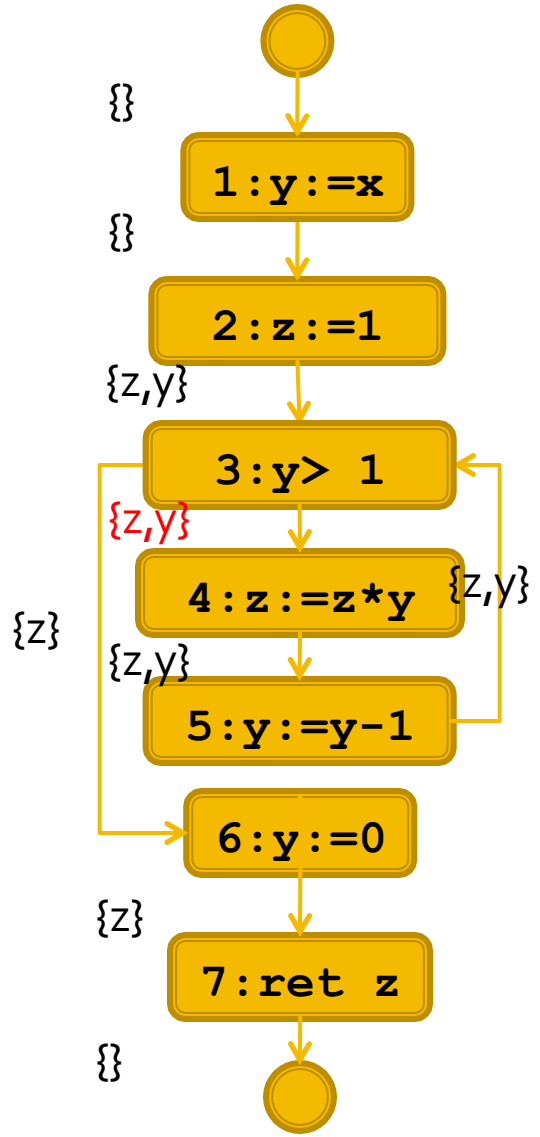
work = {2,4}

n = 4

out'[4] = {z,y}

in' [4]= {z,y}

work' = {2,3}



Live variable analysis

Lattice: P({x, y, z})

Transfer function:

- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x*:=*e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, other case) = σ

out[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in[n] := ***F(out***[*n*],*stm*[*n*])

Compute in[*n*] *for each n* ∈*N:*

out[*n*] := ⊥

work.add= {exit}

WILE *work* is not empty:

n:= *work*.pop();

out'[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in'[*n*] := ***transfer***[*n*](***out'***[*n*])

IF !(*in'*[*n*] ⊆ ***in***[*n*])

for all m ∈ ***pred***(*n*) work.add(*m*);

out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

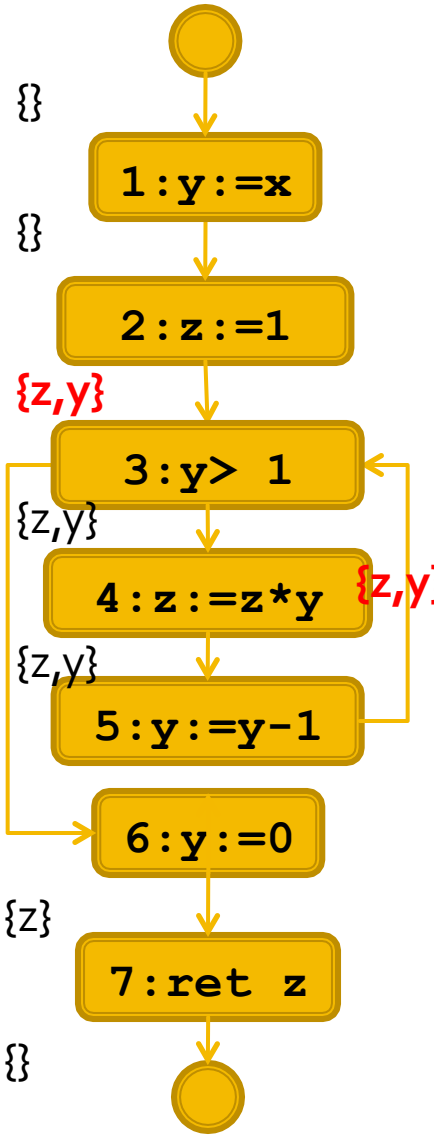
work = {2,3}

n = 3

out'[3] = {z,y} (of 6 and 4)

in' [3]= {z,y}

work' = {2}



Live variable analysis

Lattice: P({x, y, z})

Transfer function:

- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x*:=*e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, other case) = σ

out[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in[n] := ***F(out***[*n*],*stm*[*n*])

Compute in[*n*] *for each n* ∈*N:*

out[*n*] := ⊥

work.add= {exit}

WILE *work* is not empty:

n:= *work*.pop();

out'[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in'[*n*] := ***transfer***[*n*](***out'***[*n*])

IF !(*in'*[*n*] ⊆ ***in***[*n*])

for all m ∈ ***pred***(*n*) work.add(*m*);

out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

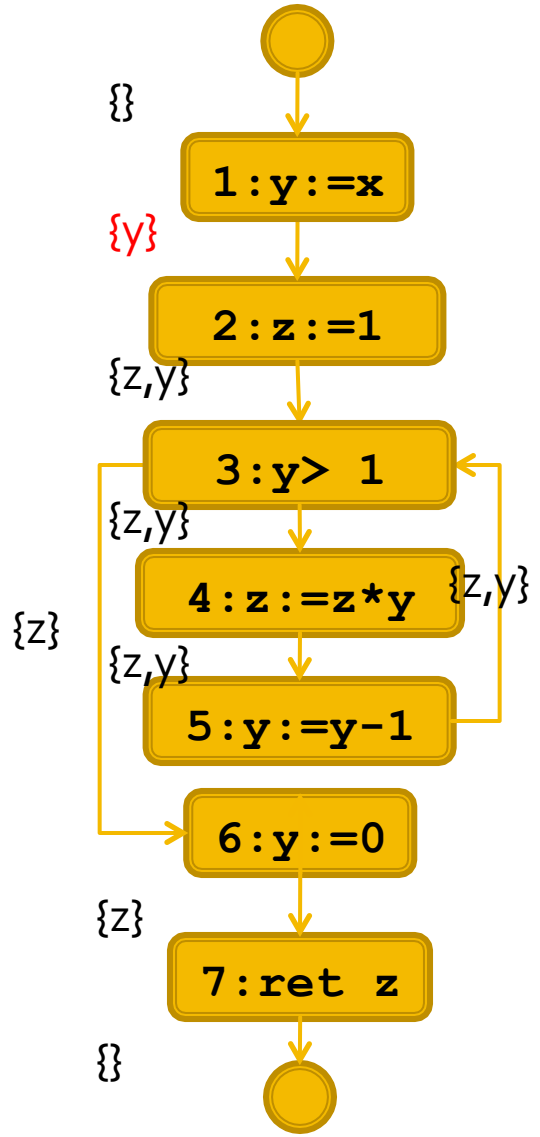
work = {2}

n = 2

out'[2] = {z,y}

in' [2]= {y}

work' = {1}



Live variable analysis

Lattice: P({x, y, z})

Transfer function:

- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x*:=*e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, other case) = σ

out[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in[n] := ***F(out***[*n*],*stm*[*n*])

Compute in[*n*] *for each n* ∈*N:*

out[*n*] := ⊥

work.add= {exit}

WILE *work* is not empty:

n:= *work*.pop();

out'[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }

in'[*n*] := ***transfer***[*n*](***out'***[*n*])

IF !(*in'*[*n*] ⊆ ***in***[*n*])

for all m ∈ ***pred***(*n*) *work.add(m);*

out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

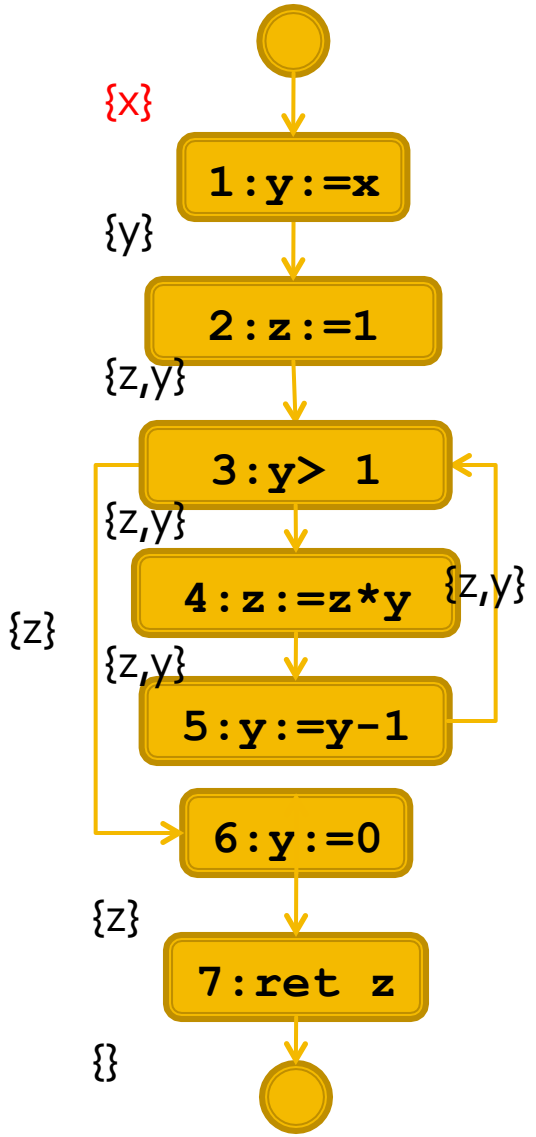
work = {1}

n = 1

out'[1] = {y}

in' [1]= {x}

work' = {entry}



Termination

- How do we know the algorithm terminates?

- Because…
  - All operations are monotonous
  - Domain is finite
    - Live variables: set of variables in the program
    - Available Expressions: set of expressions
    - Zero analysis: bot, Z,NZ, MZ
    - Etc.
  - If the domain is not finite termination can still be achieved
    - Acceleration (widening)

Flow sensitive

- In a flow sensitive analysis
  - Execution order of instructions does matter
    - Example: Zero analysis
- In flow insensitive analysis
  - Execution order **does not** matter
  - Result is independent from the order in which instructions are executed
    - Example: type inference in typed languages

Flow-insensitive analysis

- Variables updated by a procedure
  - $M(x := e) = \{ x \}$
  - $M(S1;S2) = M(S1) \cup M(S2)$
- Observe that $M(S1;S2) = M(S2;S1)$

Sensitive vs. Insensitive

- Sensitive analysis require a program model for each program point
  - More precise
  - Scale less than a insensitive analysis
- Insensitive analyses require **only** a global state
  - Example: Set of modified variables
  - Less precise
  - Scale better than sensitive counterparts

Taste & Sensibility

- Sensibility: which aspects of the code will we consider?
  - Instruction-oder? Flow-sensitive
  - Call Stack, parameters? Context-sensitive
  - Branch conditions? Path-sentivive

Analysis	Sensibility
Typing	Insensitive
Dataflow	Flow-sensitive
Model Checking	Flow & Path sensitive
Points-to	C :Flow Java: Flow insensitive although context sensitive