

Program analysis

Mooly Sagiv

<http://www.cs.tau.ac.il/~msagiv/courses/wcc11-12.html>

Abstract Interpretation

Static analysis

- Automatically identify program properties
 - No user provided loop invariants
- Sound but incomplete methods
 - But can be rather precise
- Non-standard interpretation of the program operational semantics
- Applications
 - Compiler optimization
 - Code quality tools
 - Identify potential bugs
 - Prove the absence of runtime errors
 - Partial correctness

Control Flow Graph(CFG)

$z = 3$

while ($x > 0$) {

 if ($x = 1$)

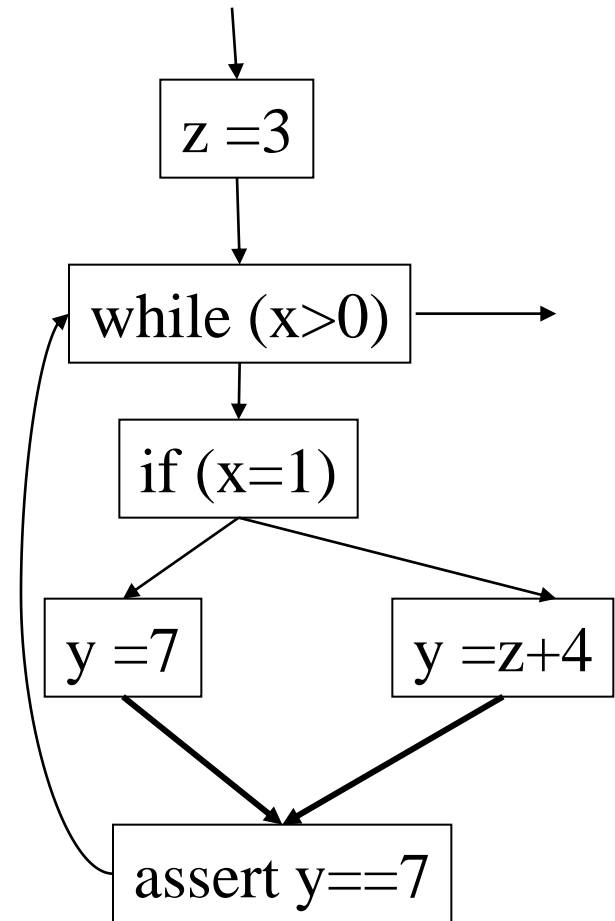
$y = 7$;

 else

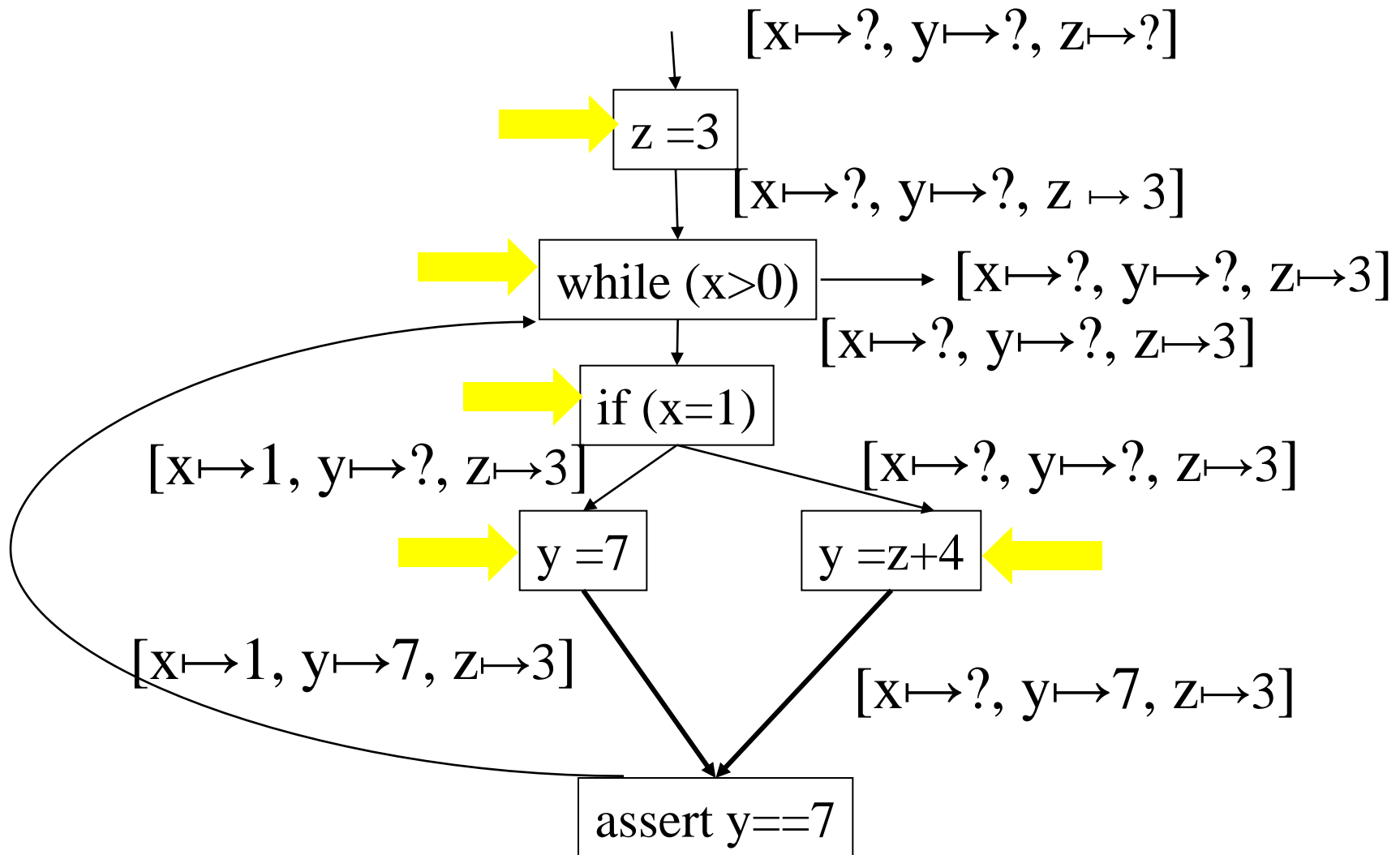
$y = z + 4$;

 assert $y == 7$

}



Constant Propagation



Memory Leakage

```
List reverse(Element *head)
{
    List rev, n;
    rev = NULL;
    while (head != NULL) {
        n = head →next;
        head → next = rev;
        head = n;
        rev = head;
    }
    return rev;
}
```

***potential leakage of address
pointed to by head***

Memory Leakage

```
Element* reverse(Element *head)
```

```
{
```

```
    Element *rev, *n;
```

```
    rev = NULL;
```

```
    while (head != NULL) {
```

```
        n = head → next;
```

```
        head → next = rev;
```

```
        rev = head;
```

```
        head = n;
```

```
    }
```

```
    return rev; }
```

👍 No memory leaks

A Simple Example

```
void foo(char *s )
```

```
{
```

```
    while ( *s != ' ' )
```

```
        s++;
```

```
    *s = 0;
```

```
}
```

Potential buffer overrun:
 $\text{offset}(s) \geq \text{alloc}(\text{base}(s))$

A Simple Example

```
void foo(char *s) @require string(s)
{
    while ( *s != ' ' && *s != 0)
        s++;
    *s = 0;
}
```

👍 No buffer overruns

Example Static Analysis Problem

- Find variables which are **live** at a given program location
- Used before set on some execution paths from the current program point

A Simple Example

/ c */*

L0: a := 0

/ ac */*

L1: b := a + 1

/ bc */*

c := c + b

/ bc */*

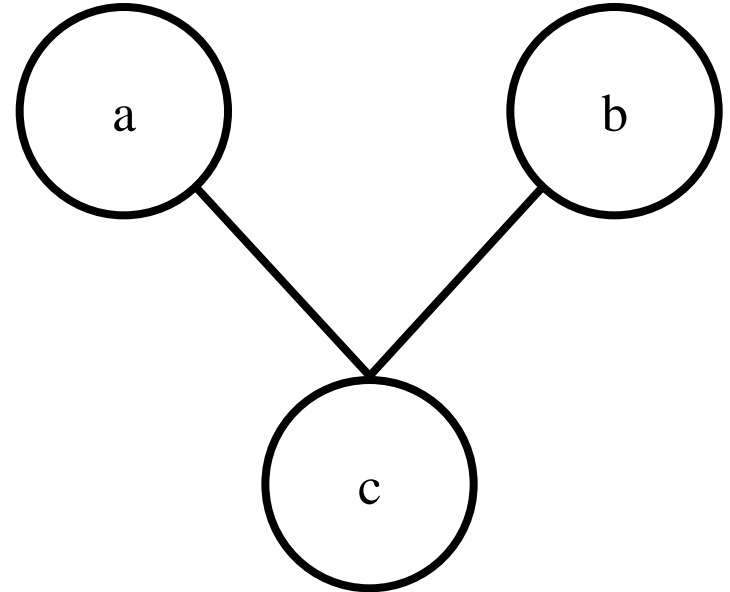
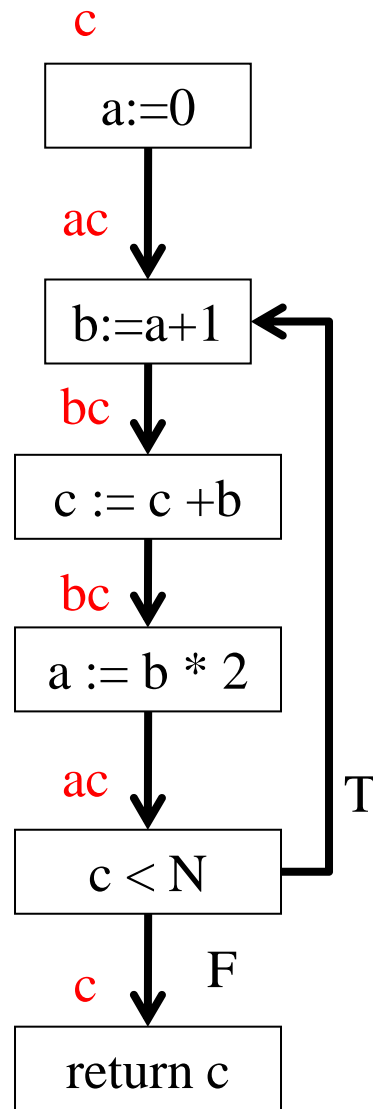
a := b * 2

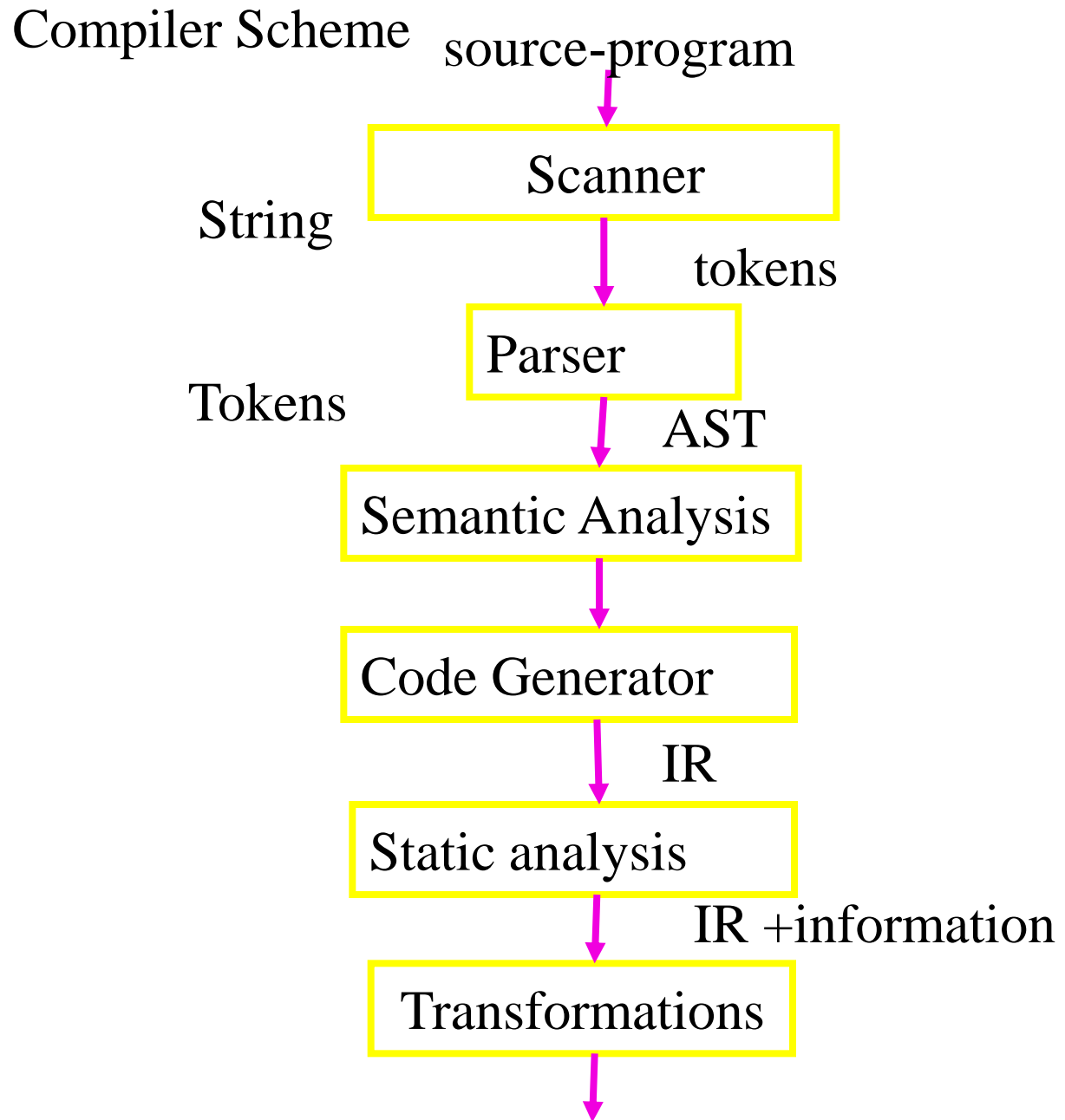
/ ac */*

if c < N goto L1

/ c */*

return c





Undecidability issues

- It is impossible to compute exact static information
- Finding if a program point is reachable
- Difficulty of interesting data properties

Undecidability

- A variable is **live** at a given point in the program
 - if its current value **is used** after this point prior to a definition in **some execution path**
- It is undecidable if a variable is live at a given program location

Proof Sketch

Pr

L: $x := y$

Is y live at L?

Conservative (Sound)

- The compiler need not generate the optimal code
- Can use more registers (“spill code”) than necessary
- Find an upper approximation of the live variables
- Err on the safe side
- A superset of edges in the interference graph
- Not too many superfluous live variables

Conservative(Sound) Software Quality Tools

- Can never miss an error
- But may produce false alarms
 - Warning on non existing errors

Data Flow Values

- Order data flow values
 - $a \sqsubseteq b \Leftrightarrow a$ “is more precise than” b
 - In live variables
 - $a \sqsubseteq b \Leftrightarrow a \subseteq b$
 - In constant propagation
 - $a \sqsubseteq b \Leftrightarrow a$ includes more constants than b
- Compute the least solution
- Merge control flow paths optimistically
 - $a \sqcup b$
 - In live variables
 - $a \sqcup b = a \cup b$

Transfer Functions

- Program statements operate on data flow values conservatively

Transfer Functions (Constant Propagation)

- Program statements operate on data flow values conservatively
- If $a=3$ and $b=7$ before
“ $z = a + b;$ ”
 - then $a=3$, $b =7$, and $z =10$ after
- If $a=?$ and $b=7$ before
“ $z = a + b;$ ”
 - then $a=?$, $b =7$, and $z =?$ After
- For $x = \text{exp}$
 - $\text{CpOut} = \text{CpIn} [x \mapsto [[\text{exp}]](\text{CpIn})]$

Transfer Functions

Live Variables

- If **a** and **c** are potentially live after “**a** = **b** * 2”
 - then **b** and **c** are potentially live before
- For “**x** = exp;”
 - $\text{LiveIn} = \text{LiveOut} - \{x\} \cup \text{arg}(\text{exp})$

Iterative computation of conservative static information

- Construct a control flow graph(CFG)
- Optimistically start with the best value at every node
- “Interpret” every statement in a conservative way
- Forward/Backward traversal of CFG
- Stop when no changes occur

Pseudo Code (forward)

```
forward(G(V, E): CFG, start: CFG node, initial: value){
```

```
  // initialization
```

```
  value[start]:= initial
```

```
  for each  $v \in V - \{start\}$  do value[v] :=  $\perp$ 
```

```
  // iteration
```

```
  WL = V
```

```
  while WL != {} do
```

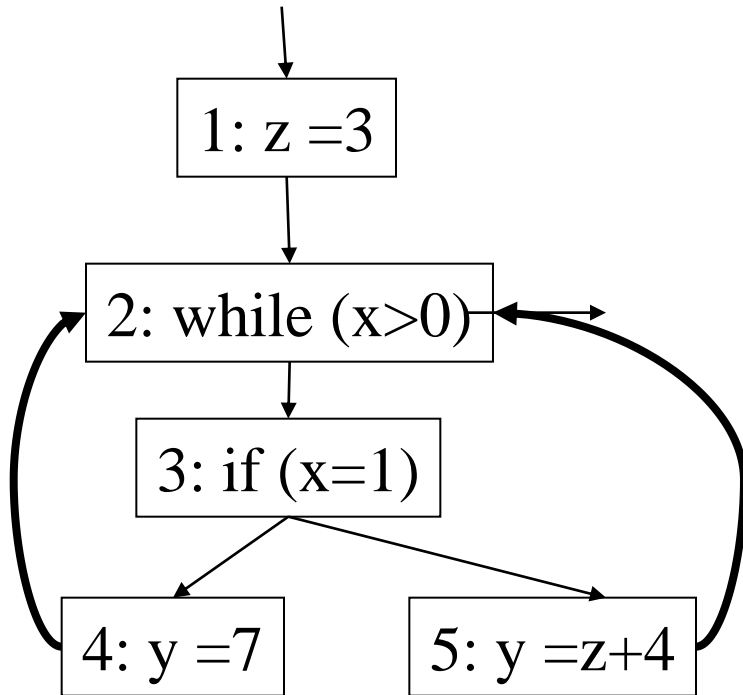
```
    select and remove a node  $v \in WL$ 
```

```
    for each  $u \in V$  such that  $(v, u) \in E$  do
```

```
      value[u] := value[u]  $\sqcup$  f(v, u)(value[v])
```

```
      if value[u] was changed WL := WL  $\cup$  {u}
```

Constant Propagation



N	Val	WL
	$v[1]=[x \mapsto ?, y \mapsto ?, z \mapsto ?]$	{1, 2, 3, 4, 5}
1	$v[2]=[x \mapsto ?, y \mapsto ?, z \mapsto 3]$	{2, 3, 4, 5}
2	$v[3]=[x \mapsto ?, y \mapsto ?, z \mapsto 3]$	{3, 4, 5}
3	$v[4]=[x \mapsto ?, y \mapsto ?, z \mapsto 3]$ $v[5]=[x \mapsto ?, y \mapsto ?, z \mapsto 3]$	{4, 5}
4		{5}
5		{}

Only values before CFG are shown

Pseudo Code (backward)

```
backward(G(V, E): CFG, exit: CFG node, initial: value){
```

```
  // initialization
```

```
  value[exit]:= initial
```

```
  for each  $v \in V - \{\text{exit}\}$  do value[v] :=  $\perp$ 
```

```
  // iteration
```

```
  WL = V
```

```
  while WL != {} do
```

```
    select and remove a node  $v \in \text{WL}$ 
```

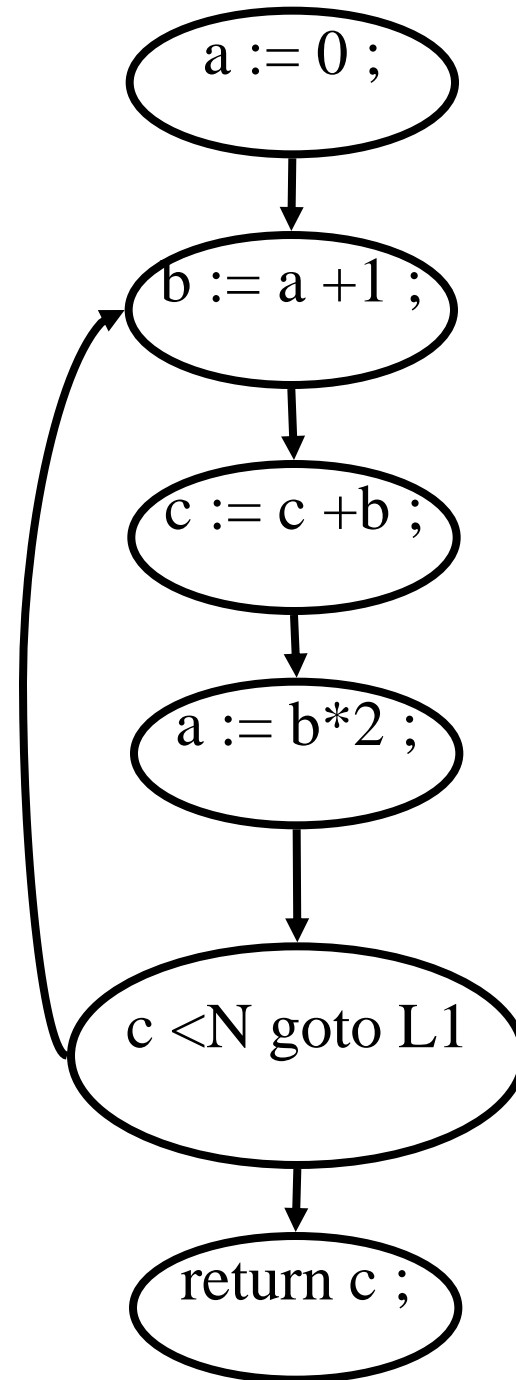
```
    for each  $u \in V$  such that  $(u, v) \in E$  do
```

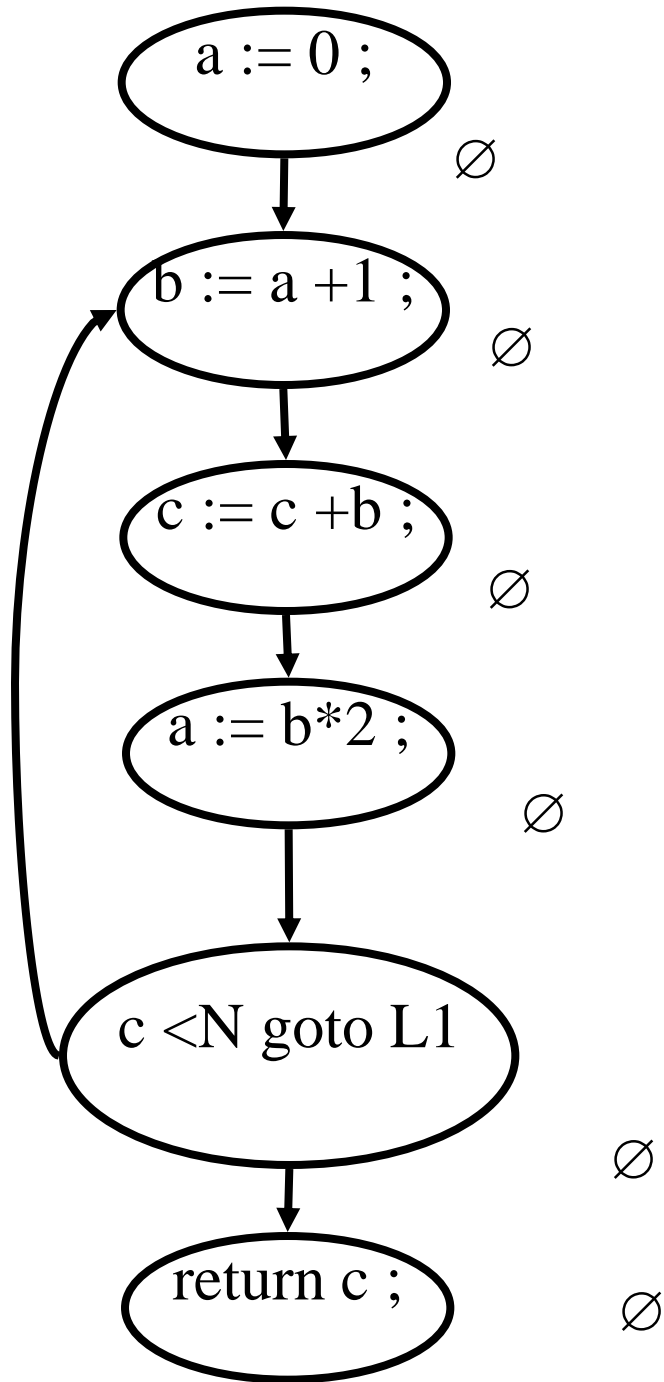
```
      value[u] := value[u]  $\sqcup$  f(v, u)(value[v])
```

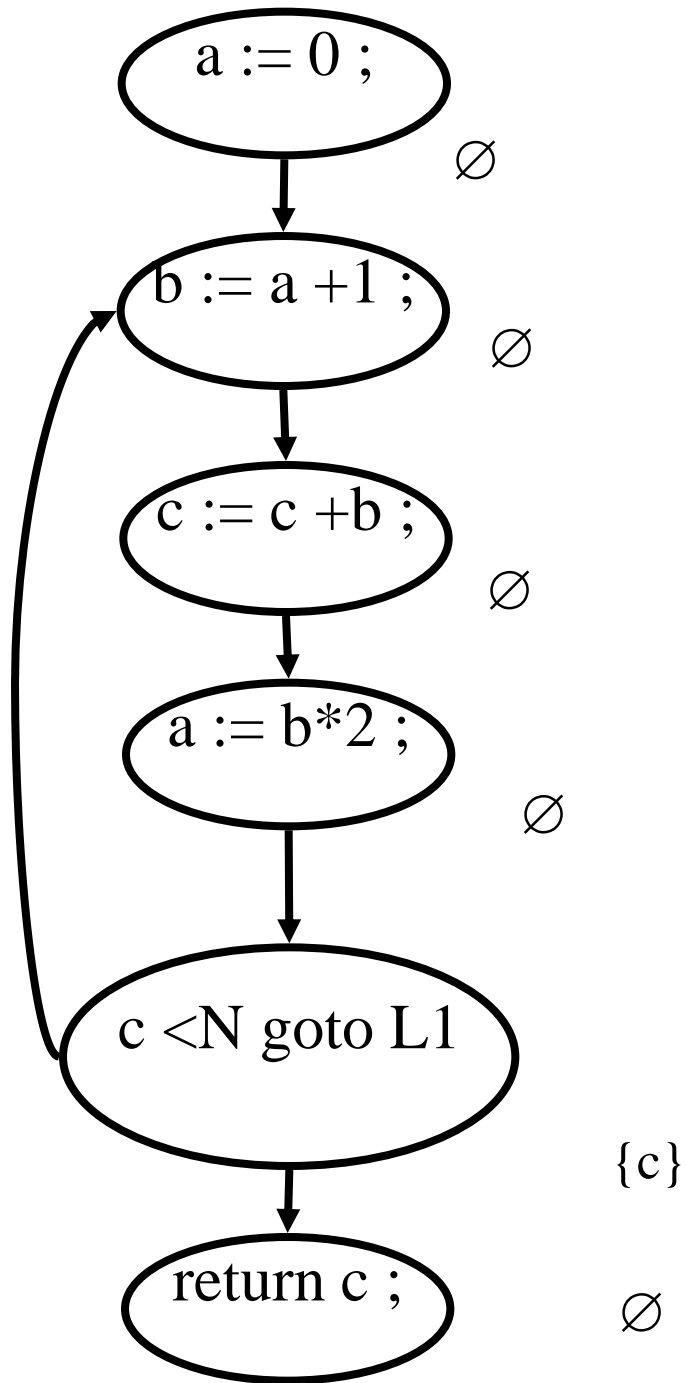
```
      if value[u] was changed WL := WL  $\cup$  {u}
```

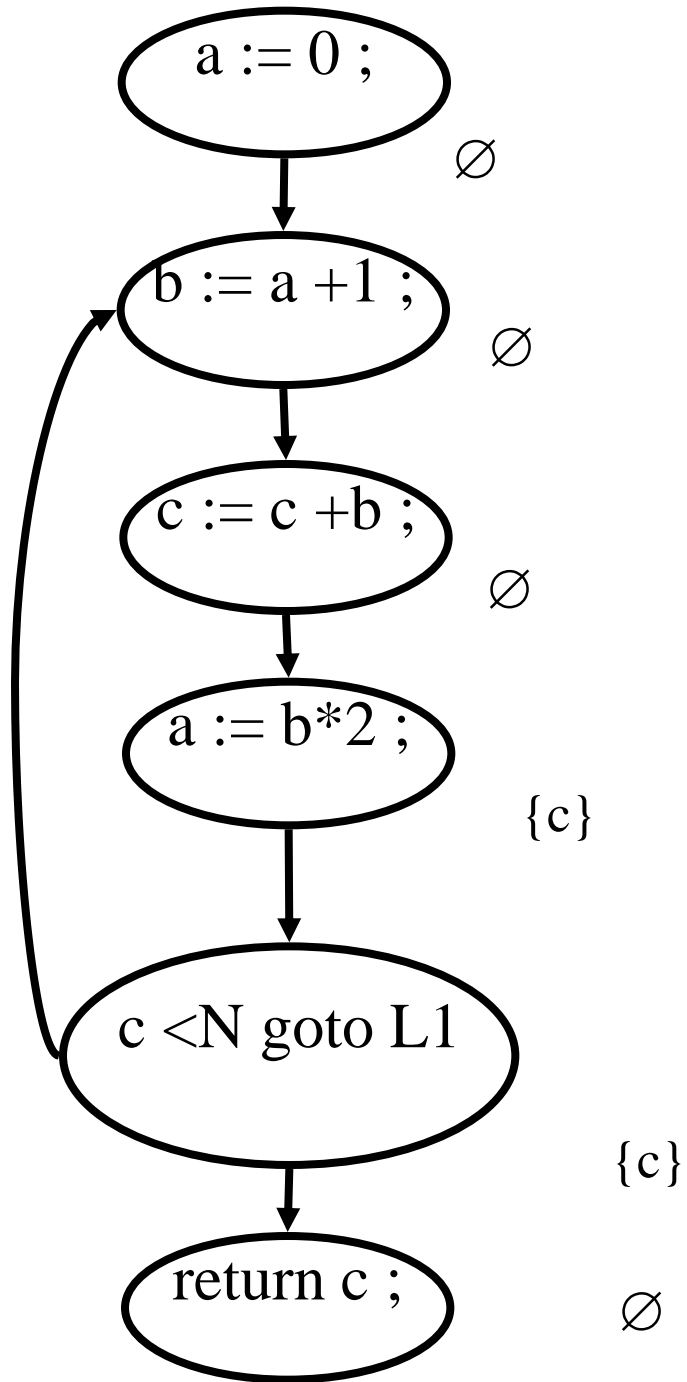


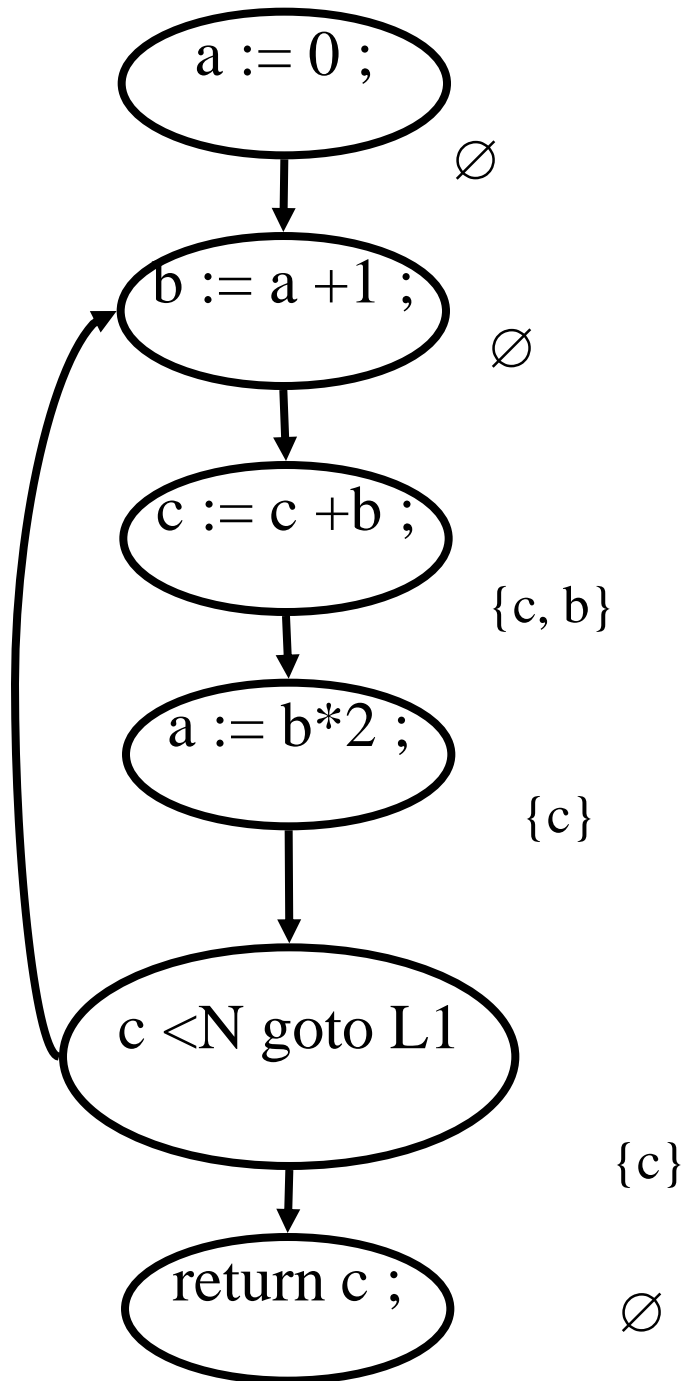
```
/* c */
L0:   a := 0
/* ac */
L1:   b := a + 1
/* bc */
      c := c + b
/* bc */
      a := b * 2
/* ac */
      if c < N goto L1
/* c */
      return c
```

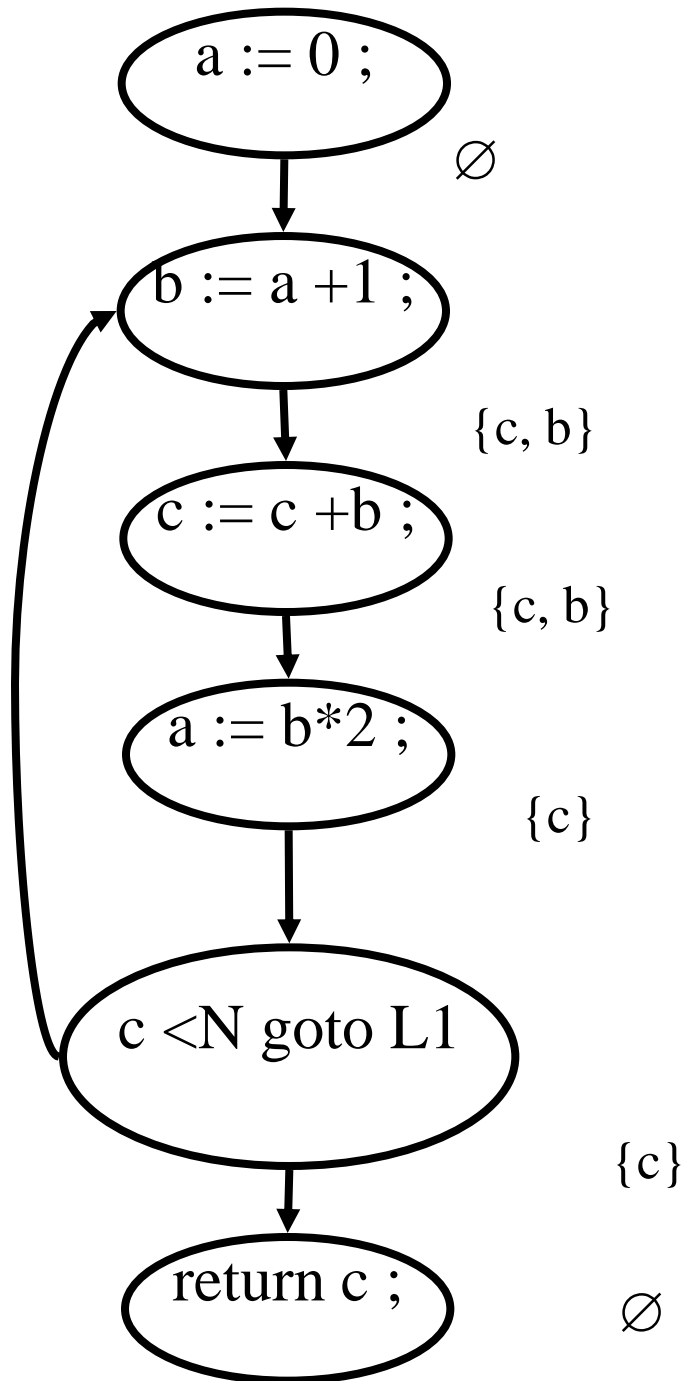


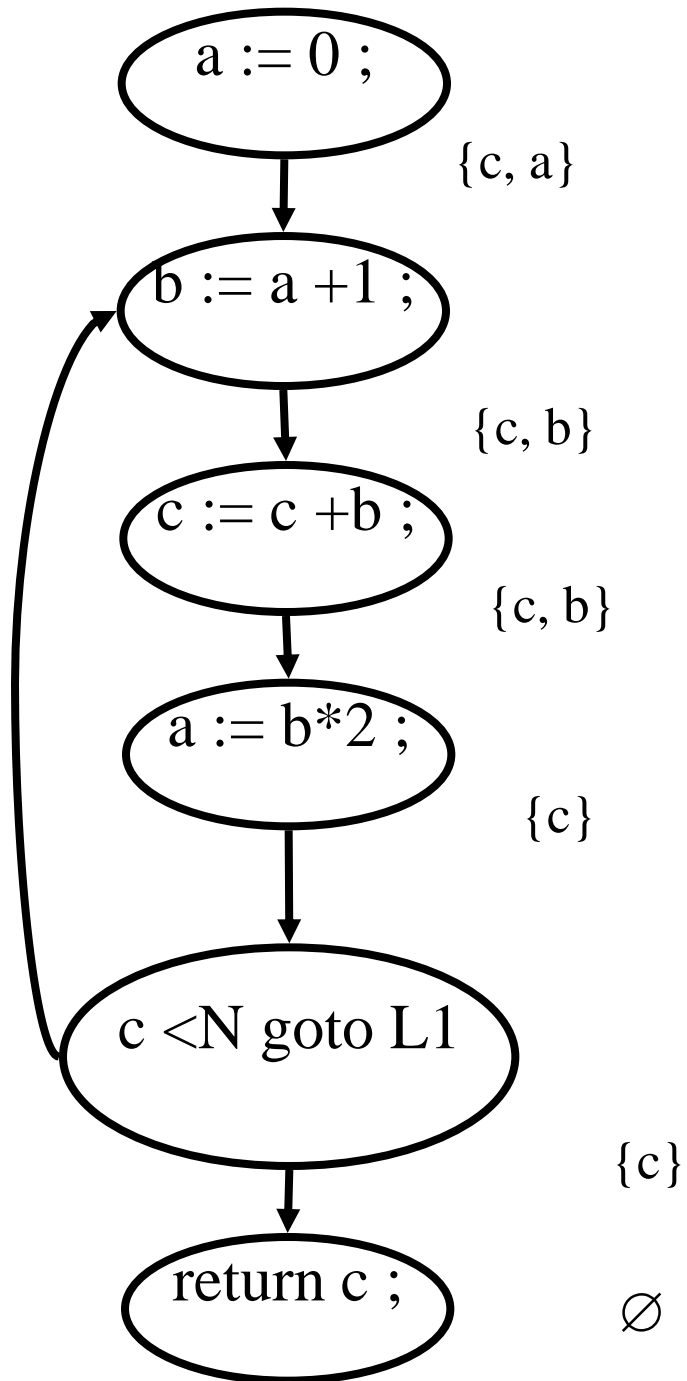


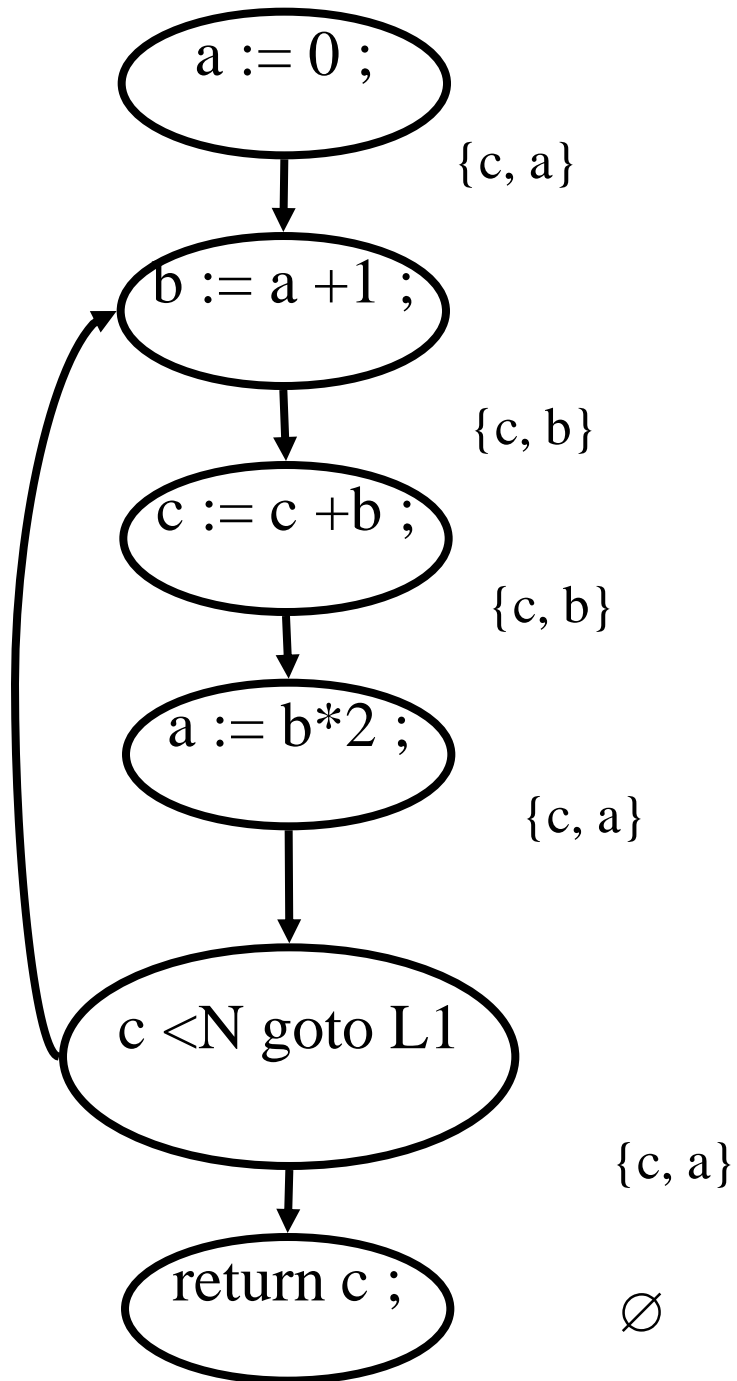












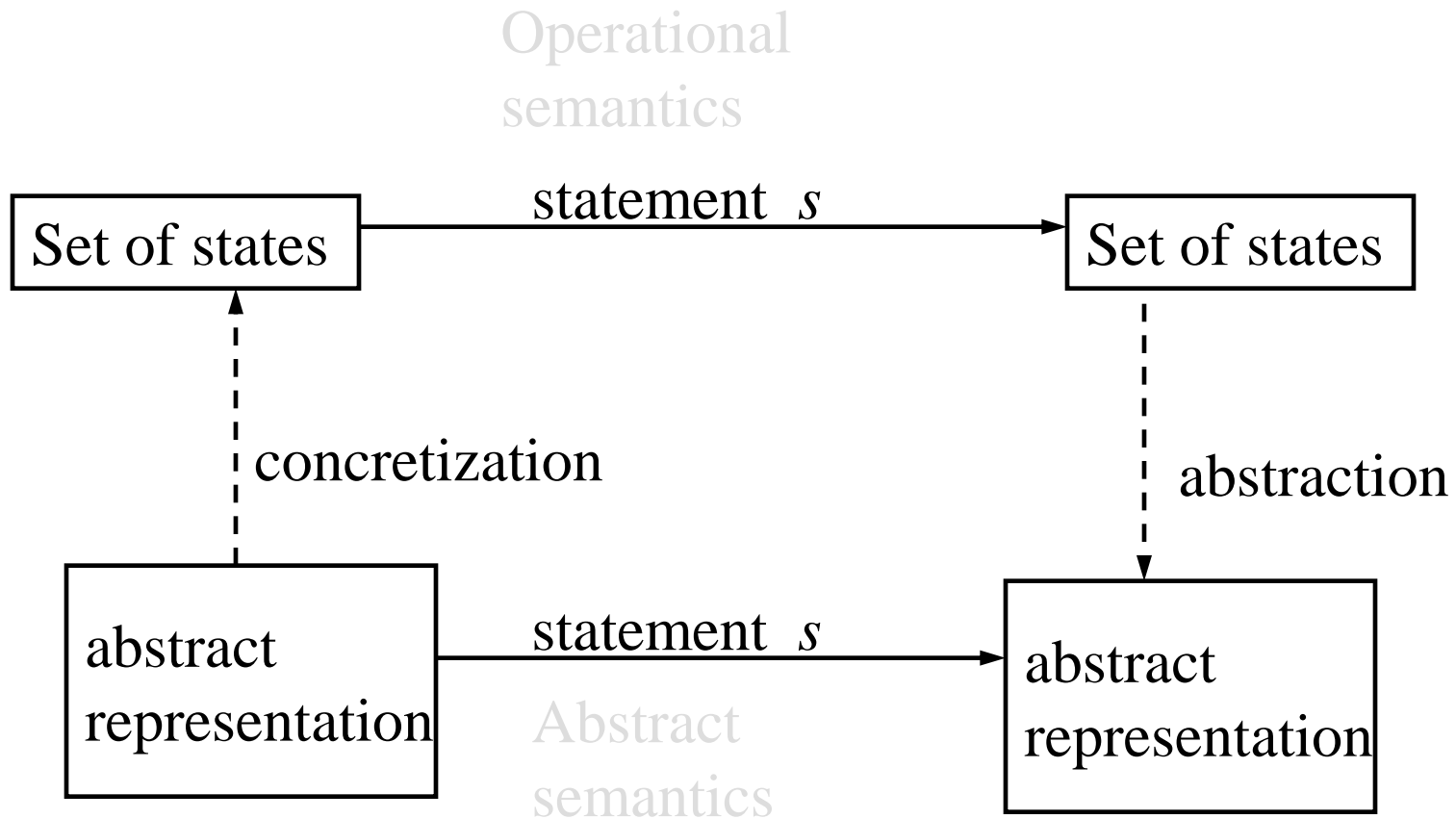
Summary Iterative Procedure

- Analyze one procedure at a time
 - More precise solutions exist
- Construct a control flow graph for the procedure
- Initializes the values at every node to the most optimistic value
- Iterate until convergence

Abstract Interpretation

- The mathematical foundations of program analysis
- Established by Cousot and Cousot 1979
- Relates static and runtime values

Abstract (Conservative) interpretation



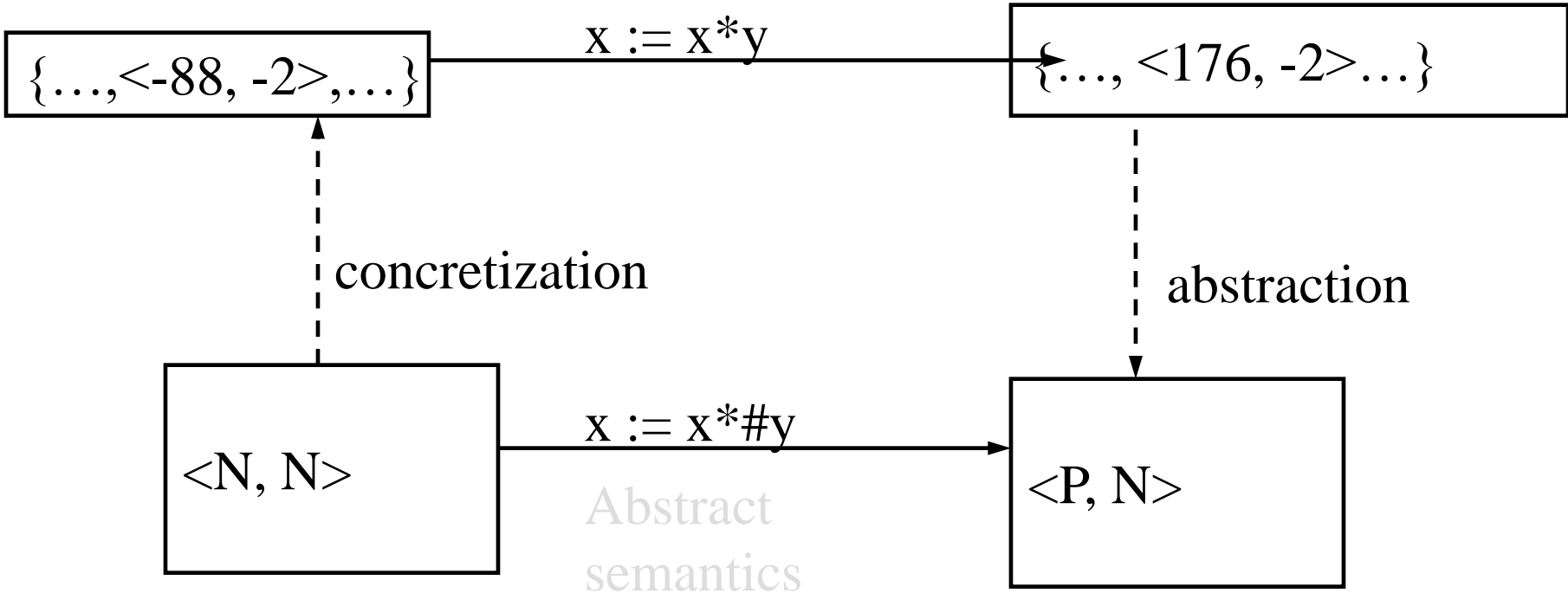
Example rule of signs

- Safely identify the sign of variables at every program location
- Abstract representation {P, N, ?}
- Abstract (conservative) semantics of *

*#	P	N	?
P	P	N	?
N	N	P	?
?	?	?	?

Abstract (conservative) interpretation

Operational semantics



Example rule of signs

- Safely identify the sign of variables at every program location
- Abstract representation $\{P, N, ?\}$
- $\alpha(C) =$ if all elements in C are positive
then return P
else if all elements in C are negative
then return N
else return $?$
- $\gamma(a) =$ if $(a==P)$ then
return $\{0, 1, 2, \dots\}$
else if $(a==N)$
return $\{-1, -2, -3, \dots, \}$
else return Z

Example Constant Propagation

- Abstract representation
 - set of integer values and an extra value “?” denoting variables not known to be constants
- Conservative interpretation of +

+#	?	0	1	2
?	?	?	?	?
0	?	0	1	2
1	?	1	2	3
2	?	2	3	4

Example Program

```
x = 5;
```

```
y = 7;
```

```
if (getc())
```

```
    y = x + 2;
```

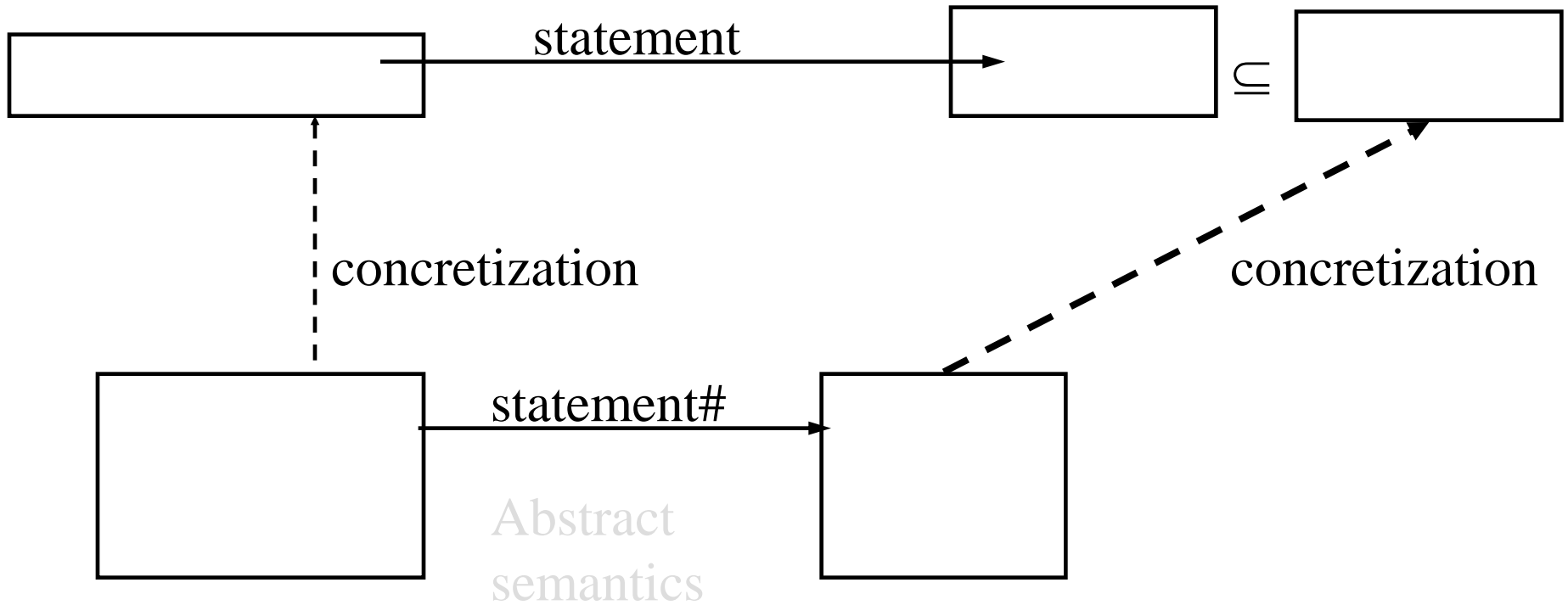
```
z = x + y;
```


Example Program (2)

```
if (getc())  
    x = 3 ; y = 2;  
  
    else  
  
        x = 2; y = 3;  
  
z = x + y;
```

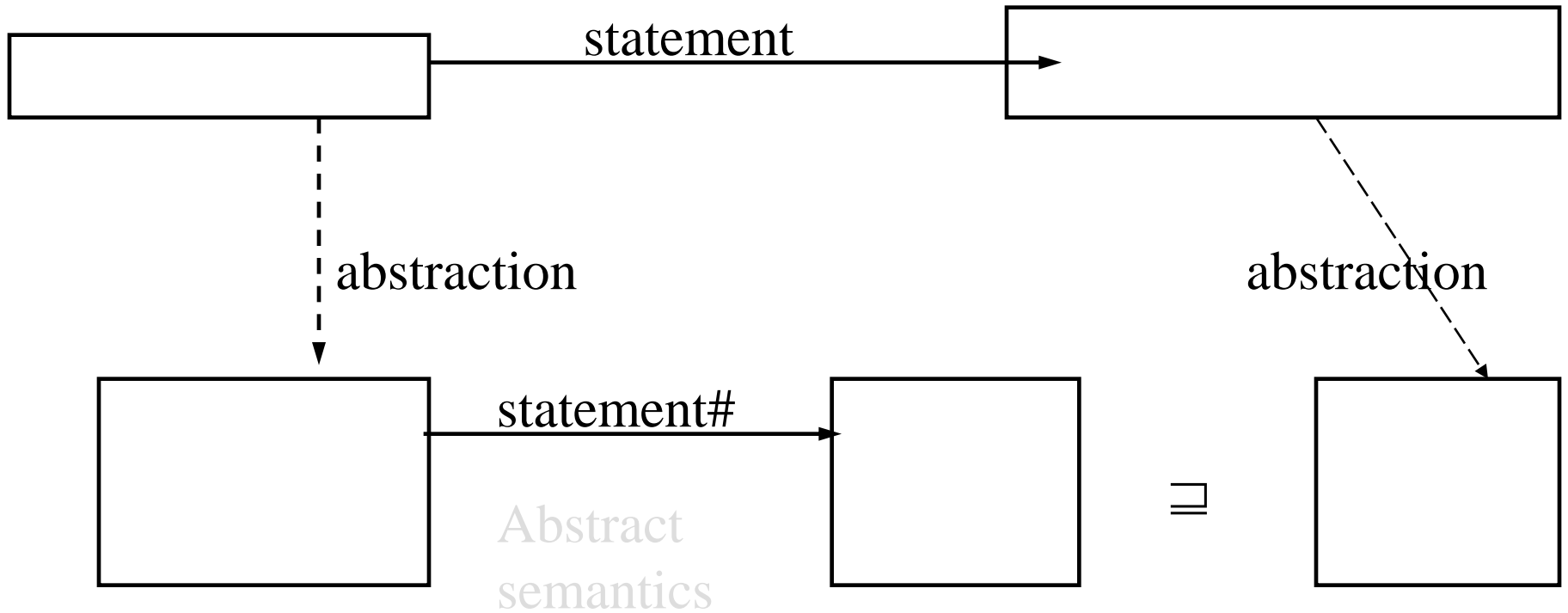
Local Soundness of Abstract Interpretation

Operational
semantics



Local Soundness of Abstract Interpretation

Operational
semantics



Some Success Stories

Software Quality Tools

- The prefix Tool identified interesting bugs in Windows
- The Microsoft SLAM tool checks correctness of device driver
 - Driver correctness rules
- Astree checks floating point operation

Summary

- Program analysis provides non-trivial insights on the runtime executions of the program
 - Degenerate case – types (flow insensitive)
- Mathematically justified
 - Operational semantics
 - Abstract interpretation (lattice theory)
- Employed in compilers
- Will be employed in software quality tools