

Formal Semantics of Programming Languages

Mooly Sagiv

Reference: Semantics with Applications

Chapter 2

H. Nielson and F. Nielson

http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html

Benefits of formal definitions

- Intellectual
- Better understanding
- Formal proofs
- Mechanical checks by computer
- Tool generations

What is a good formal definition?

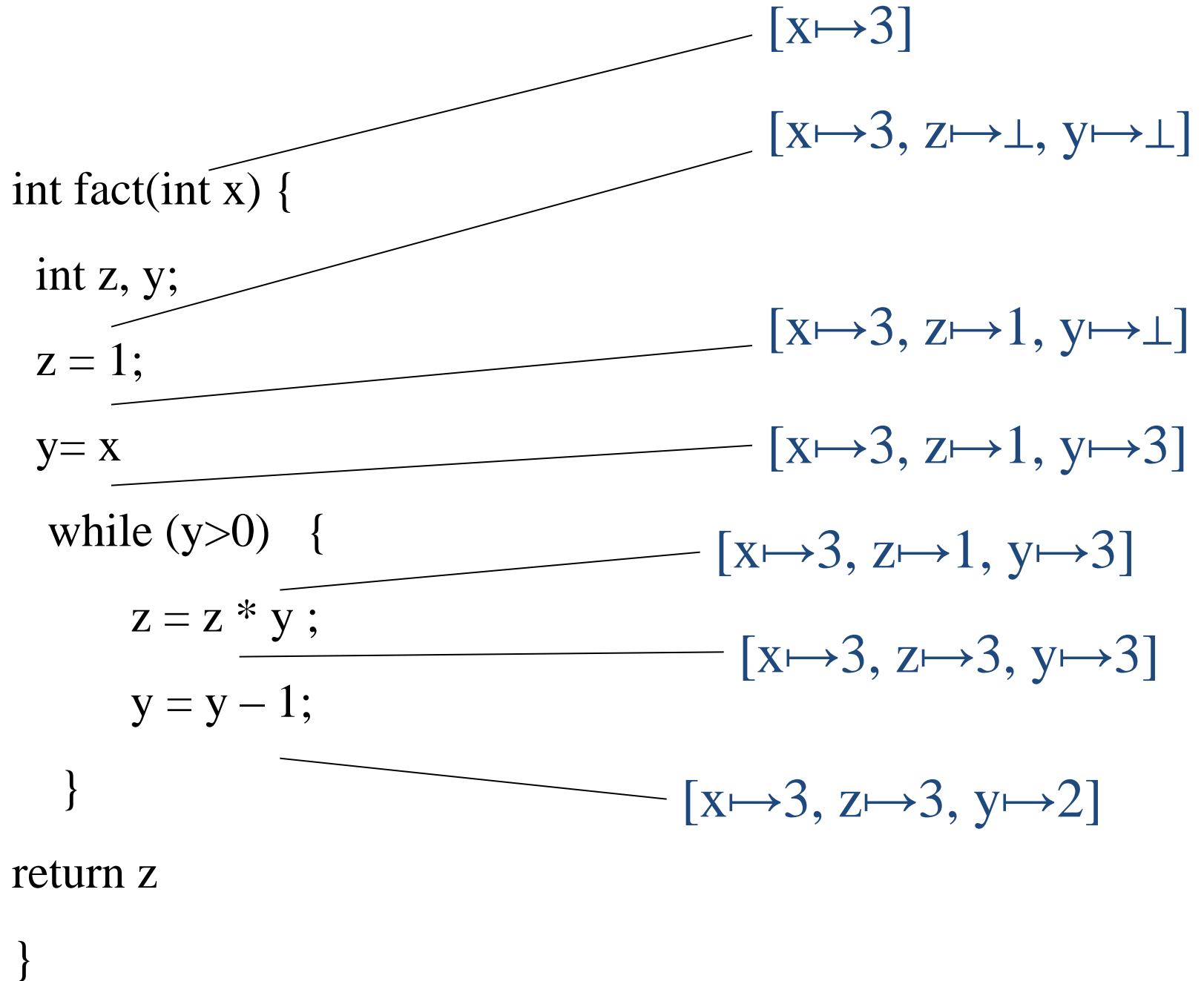
- Natural
- Concise
- Easy to understand
- Permits effective mechanical reasoning

Syntax vs. Semantics

- The pattern of formation of sentences or phrases in a language
- Examples
 - Regular expressions
 - Context free grammars
- The study or science of meaning in language
- Examples
 - Interpreter
 - Compiler
 - Better mechanisms will be given in the course

Alternative Formal Semantics

- Operational Semantics
 - The meaning of the program is described “operationally”
 - Natural Operational Semantics
 - Structural Operational Semantics
- Denotational Semantics
 - The meaning of the program is an input/output relation
 - Mathematically challenging but complicated
- Axiomatic Semantics
 - The meaning of the program are observed properties



```

int fact(int x) {
    int z, y;
    z = 1;
    y = x
    while (y > 0) {
        z = z * y;
        y = y - 1;
    }
    return z
}

```

$[x \mapsto 3, z \mapsto 3, y \mapsto 2]$
 $[x \mapsto 3, z \mapsto 3, y \mapsto 2]$
 $[x \mapsto 3, z \mapsto 6, y \mapsto 2]$
 $[x \mapsto 3, z \mapsto 6, y \mapsto 1]$

```

int fact(int x) {
    int z, y;
    z = 1;
    y = x
    while (y > 0) {
        z = z * y;
        y = y - 1;
    }
    return z
}

```

$[x \mapsto 3, z \mapsto 6, y \mapsto 1]$
 $[x \mapsto 3, z \mapsto 6, y \mapsto 1]$
 $[x \mapsto 3, z \mapsto 6, y \mapsto 1]$
 $[x \mapsto 3, z \mapsto 6, y \mapsto 0]$


```

int fact(int x) {
    int z, y;
    z = 1;
    y = x;
    while (y > 0) {
        z = z * y;
        y = y - 1;
    }
    return z
}

```

$[x \mapsto 3, z \mapsto 6, y \mapsto 0]$

$[x \mapsto 3, z \mapsto 6, y \mapsto 0]$

```
int fact(int x) {  
    int z, y;  
    z = 1;  
    y = x;  
    while (y > 0) {  
        z = z * y;  
        y = y - 1;  
    }  
    return 6 ——— [x ↦ 3, z ↦ 6, y ↦ 0]  
}
```

Denotational Semantics

```
int fact(int x) {
```

```
    int z, y;
```

```
    z = 1;
```

```
    y = x ;
```

$f = \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } x * f(x - 1)$

```
    while (y > 0) {
```

```
        z = z * y ;
```

```
        y = y - 1;
```

```
    }
```

```
    return z;
```

```
}
```

Axiomatic Semantics

$\{x=n\}$

int fact(int x) { int z, y;

z = 1;

$\{x=n \wedge z=1\}$

y = x

$\{x=n \wedge z=1 \wedge y=n\}$

while

$\{x=n \wedge y \geq 0 \wedge z=n! / y!\}$

(y>0) {

$\{x=n \wedge y > 0 \wedge z=n! / y!\}$

z = z * y ;

$\{x=n \wedge y > 0 \wedge z=n! / (y-1)!\}$

y = y - 1;

$\{x=n \wedge y \geq 0 \wedge z=n! / y!\}$

} return z } $\{x=n \wedge z=n!\}$

The **While** Programming Language

- Abstract syntax

$S ::= x := a \mid \mathbf{skip} \mid S_1 ; S_2 \mid \mathbf{if} \ b \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 \mid$
 $\mathbf{while} \ b \ \mathbf{do} \ S$

- Use parentheses for precedence
- Informal Semantics
 - **skip** behaves like no-operation
 - Import meaning of arithmetic and Boolean operations

Example While Program

$y := 1;$

while $\neg(x=1)$ do (

$y := y * x;$

$x := x - 1;$

)

General Notations

- Syntactic categories
 - Var the set of program variables
 - Aexp the set of arithmetic expressions
 - Bexp the set of Boolean expressions
 - Stm set of program statements
- Semantic categories
 - Natural values $N = \{0, 1, 2, \dots\}$
 - Truth values $T = \{ff, tt\}$
 - States $\text{State} = \text{Var} \rightarrow N$
 - Lookup in a state $s: s \ x$
 - Update of a state $s: s \ [\ x \mapsto 5 \]$

Example State Manipulations

- $[x \mapsto 1, y \mapsto 7, z \mapsto 16] y =$
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16] t =$
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16][x \mapsto 5] =$
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16][x \mapsto 5] x =$
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16][x \mapsto 5] y =$

Semantics of arithmetic expressions

- Assume that arithmetic expressions are side-effect free
- $A \llbracket \text{Aexp} \rrbracket : \text{State} \rightarrow \mathbb{N}$
- Defined by **structural** induction on the syntax tree
 - $A \llbracket n \rrbracket s = n$
 - $A \llbracket x \rrbracket s = s \ x$
 - $A \llbracket e_1 + e_2 \rrbracket s = A \llbracket e_1 \rrbracket s + A \llbracket e_2 \rrbracket s$
 - $A \llbracket e_1 * e_2 \rrbracket s = A \llbracket e_1 \rrbracket s * A \llbracket e_2 \rrbracket s$
 - $A \llbracket (e_1) \rrbracket s = A \llbracket e_1 \rrbracket s$ --- not needed
 - $A \llbracket - e_1 \rrbracket s = -A \llbracket e_1 \rrbracket s$

Properties of arithmetic expressions

- The semantics is **compositional**
 - $A[[e_1 \text{ op } e_2]] = [[\text{op}]] (A[[e_1]], A[[e_2]])$
 - Properties can be proved by structural induction
- We say that e_1 is **semantically equivalent** to e_2 ($e_1 \approx e_2$) when $A[[e_1]] = A[[e_2]]$

Commutativity of expressions

- Theorem: for every expressions $e_1, e_2: e_1 + e_2 \approx e_2 + e_1$

- Proof:

$$A[e_1 + e_2]s = A[e_1]s + A[e_2]s = A[e_2]s + A[e_1]s = A[e_2 + e_1]s$$

Semantics of Boolean expressions

- Assume that Boolean expressions are side-effect free
- $B \llbracket \text{Bexp} \rrbracket : \text{State} \rightarrow T$
- Defined by induction on the syntax tree
 - $B \llbracket \text{true} \rrbracket s = \text{tt}$
 - $B \llbracket \text{false} \rrbracket s = \text{ff}$
 - $B \llbracket e_1 = e_2 \rrbracket s =$
 - $B \llbracket e_1 \wedge e_2 \rrbracket s = \begin{cases} \text{tt} & \text{if } A \llbracket e_1 \rrbracket s = A \llbracket e_2 \rrbracket s \\ \text{ff} & \text{if } A \llbracket e_1 \rrbracket s \neq A \llbracket e_2 \rrbracket s \end{cases}$
 - $B \llbracket e_1 \geq e_2 \rrbracket s = \begin{cases} \text{tt} & \text{if } B \llbracket e_1 \rrbracket s = \text{tt} \text{ and } B \llbracket e_2 \rrbracket s = \text{tt} \\ \text{ff} & \text{if } B \llbracket e_1 \rrbracket s = \text{ff} \text{ or } B \llbracket e_2 \rrbracket s = \text{ff} \end{cases}$
 - $B \llbracket e_1 \geq e_2 \rrbracket s = \begin{cases} \text{tt} & \text{if } A \llbracket e_1 \rrbracket s \geq A \llbracket e_2 \rrbracket s \\ \text{ff} & \text{if } A \llbracket e_1 \rrbracket s < A \llbracket e_2 \rrbracket s \end{cases}$

Natural Operational Semantics

- Describe the “overall” effect of program constructs
- Ignores non terminating computations

Natural Semantics

- Notations
 - $\langle S, s \rangle$ - the program statement S is executed on input state s
 - s representing a terminal (final) state
- For every statement S , write meaning rules
 $\langle S, i \rangle \rightarrow o$
“If the statement S is executed on an input state i , it terminates and yields an output state o ”
- The meaning of a program P on an input state s is the set of outputs states o such that $\langle P, i \rangle \rightarrow o$
- The meaning of compound statements is defined using the meaning immediate constituent statements

Natural Semantics for While

$$[\text{ass}_{\text{ns}}] \langle x := a, s \rangle \rightarrow s[x \mapsto \mathbf{A}[[a]]s]$$

axioms

$$[\text{skip}_{\text{ns}}] \langle \mathbf{skip}, s \rangle \rightarrow s$$

$$[\text{comp}_{\text{ns}}] \frac{\langle S_1, s \rangle \rightarrow s', \langle S_2, s' \rangle \rightarrow s''}{\langle S_1; S_2, s \rangle \rightarrow s''}$$

rules

$$\langle S_1; S_2, s \rangle \rightarrow s''$$

$$[\text{if}^{\text{tt}}_{\text{ns}}] \frac{\langle S_1, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'}$$

$$\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'$$

if $\mathbf{B}[[b]]s = \text{tt}$

$$[\text{if}^{\text{ff}}_{\text{ns}}] \frac{\langle S_2, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'}$$

$$\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'$$

if $\mathbf{B}[[b]]s = \text{ff}$

Natural Semantics for While (More rules)

$$[\text{while}_{\text{ns}}^{\text{ff}}] \frac{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s}{\text{if } \mathbf{B}[[b]]s = \text{ff}}$$

$$[\text{while}_{\text{ns}}^{\text{tt}}] \frac{\langle S, s \rangle \rightarrow s', \langle \text{while } b \text{ do } S, s' \rangle \rightarrow s''}{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s''} \text{if } \mathbf{B}[[b]]s = \text{tt}$$

Simple Examples

- Let s_0 be the state which assigns zero to all program variables

- Assignments

$$[\text{ass}_{ns}] \langle x := x+1, s_0 \rangle \rightarrow s_0[x \mapsto 1]$$

- Skip statement

$$[\text{skip}_{ns}] \langle \text{skip}, s_0 \rangle \rightarrow s_0$$

- Composition

$$\frac{[\text{comp}_{ns}] \langle \text{skip}, s_0 \rangle \rightarrow s_0, \langle x := x+1, s_0 \rangle \rightarrow s_0[x \mapsto 1]}{\langle \text{skip}; x := x + 1, s_0 \rangle \rightarrow s_0[x \mapsto 1]}$$

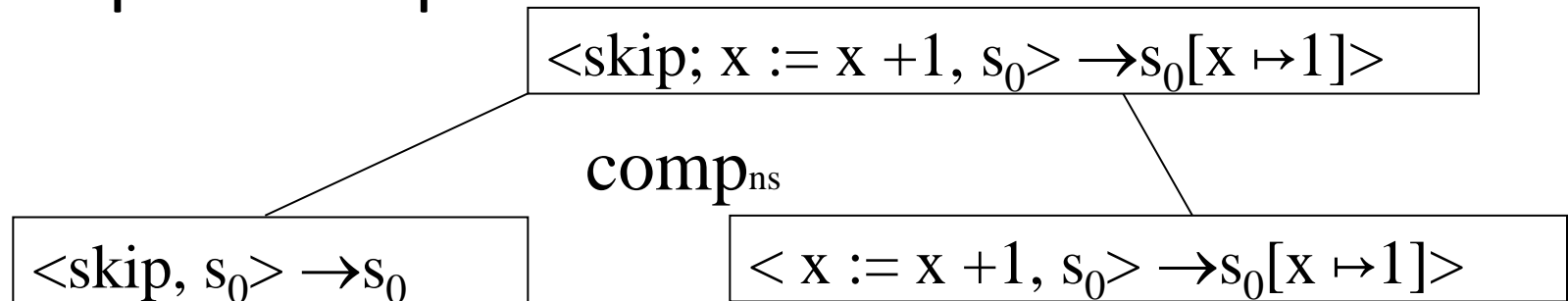
Simple Examples (Cont)

- Let s_0 be the state which assigns zero to all program variables
- if-construct

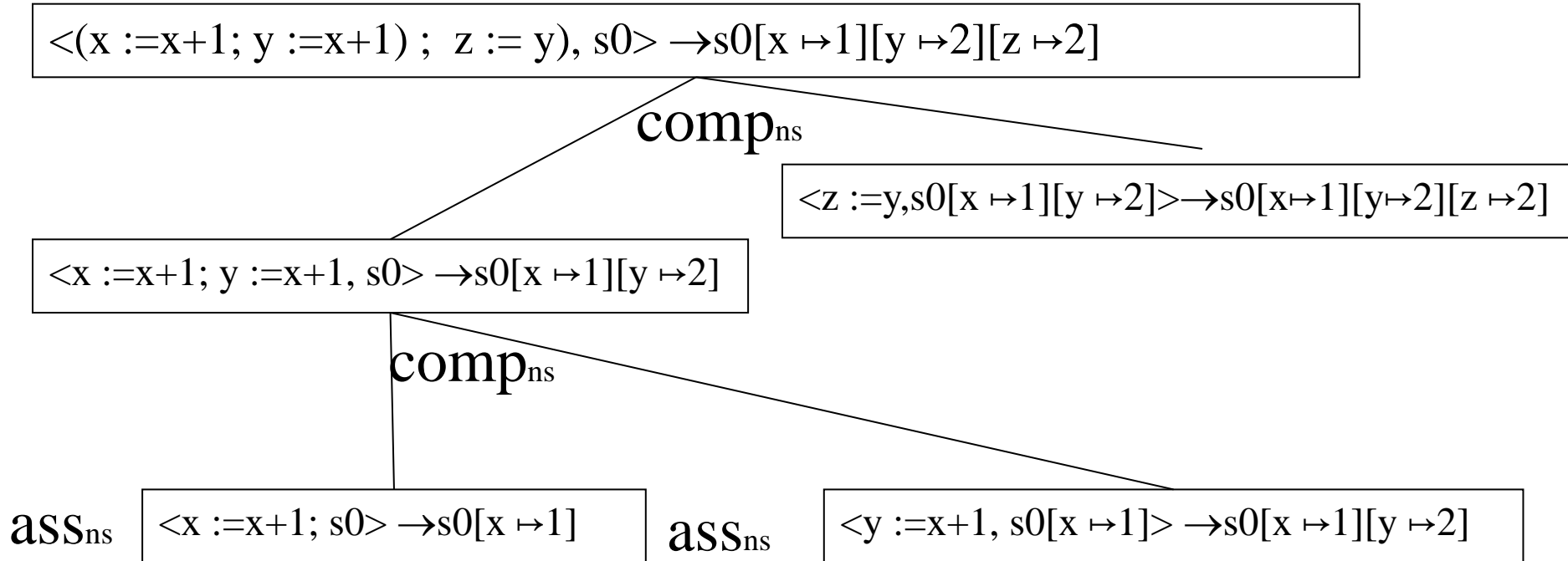
$$\boxed{\begin{array}{l} [\text{if}_{\text{ns}}^{\text{tt}}] \quad \langle \text{skip}, s_0 \rangle \rightarrow s_0 \\ \hline \langle \text{if } x=0 \text{ then skip else } x := x + 1, s_0 \rangle \rightarrow s_0 \end{array}}$$

A Derivation Tree

- A “proof” that $\langle S, s \rangle \rightarrow s'$
- The root of tree is $\langle S, s \rangle \rightarrow s'$
- Leaves are instances of axioms
- Internal nodes rules
 - Immediate children match rule premises
- Simple Example



An Example Derivation Tree



Top Down Evaluation of Derivation Trees

- Given a program S and an input state s
- Find an output state s' such that
 $\langle S, s \rangle \rightarrow s'$
- Start with the root and repeatedly apply rules until the axioms are reached
- Inspect different alternatives in order
- In While s' and the derivation tree is unique

Example of Top Down Tree Construction

- Input state s such that $s\ x = 2$
- Factorial program

$\langle y := 1; \text{while } \neg(x=1) \text{ do } (y := y * x; x := x - 1), s \rangle \rightarrow s[y \mapsto 2][x \mapsto 1] \quad \triangleright$

comp_{ns}

$\langle W, s[y \mapsto 1] \rangle \rightarrow s[y \mapsto 2][x \mapsto 1] \quad \triangleright$

$\langle y := 1, s \rangle \rightarrow s[y \mapsto 1]$

ass_{ns}

$\text{while}_{\text{ns}}^{\text{tt}}$

$\langle W, s[y \mapsto 2][x \mapsto 1] \rangle \rightarrow s[y \mapsto 2][x \mapsto 1] \quad \triangleright$

$\text{while}_{\text{ns}}^{\text{ff}}$

$\langle (y := y * x; x := x - 1, s[y \mapsto 1]) \rangle \rightarrow s[y \mapsto 2][x \mapsto 1] \quad \triangleright$

comp_{ns}

$\langle y := y * x; s[y \mapsto 1] \rangle \rightarrow s[y \mapsto 2]$

ass_{ns}

$\langle x := x - 1, s[y \mapsto 2] \rangle \rightarrow s[y \mapsto 2][x \mapsto 1] \quad \triangleright$

ass_{ns}

Program Termination

- Given a statement S and input s
 - S terminates on s if there exists a state s' such that $\langle S, s \rangle \rightarrow s'$
 - S loops on s if there is no state s' such that $\langle S, s \rangle \rightarrow s'$
- Given a statement S
 - S always terminates if for every input state s , S terminates on s
 - S always loops if for every input state s , S loops on s

Semantic Equivalence

- S_1 and S_2 are **semantically equivalent** if for all s and s' ($s \approx s'$)
 $\langle S_1, s \rangle \rightarrow s'$ if and only if $\langle S_2, s \rangle \rightarrow s'$

Example of Semantic Equivalence

- $\text{skip} ; \text{skip} \approx \text{skip}$
- $(S_1 ; S_2) ; S_3 \approx (S_1 ; (S_2 ; S_3))$
- $x := 5 ; y := x * 8 \approx x := 5 ; y := 40$
- $\text{while } b \text{ do } S \approx$
 $\text{if } b \text{ then } (S ; \text{while } b \text{ do } S) \text{ else skip}$

Deterministic Semantics for While

- If $\langle S, s \rangle \rightarrow s_1$ and $\langle S, s \rangle \rightarrow s_2$ then $s_1 = s_2$
- The proof uses induction on the shape of derivation trees
 - Prove that the property holds for all simple derivation trees by showing it holds for axioms
 - Prove that the property holds for all composite trees:
 - For each rule assume that the property holds for its premises (induction hypothesis) and prove it holds for the conclusion of the rule

The Semantic Function S_{ns}

- The meaning of a statement S is defined as a partial function from **State** to **State**
- $S_{ns}: \mathbf{Stm} \rightarrow (\mathbf{State} \hookrightarrow \mathbf{State})$
- $S_{ns} \llbracket S \rrbracket s = s'$ if $\langle S, s \rangle \rightarrow s'$ and otherwise $S_{ns} \llbracket S \rrbracket s$ is undefined
- Examples
 - $S_{ns} \llbracket \text{skip} \rrbracket s = s$
 - $S_{ns} \llbracket x := 1 \rrbracket s = s [x \mapsto 1]$
 - $S_{ns} \llbracket \text{while true do skip} \rrbracket s = \text{undefined}$

Summary Natural Semantics

- Simple
- Useful
- Enables simple proofs of language properties
- Automatic generation of interpreters
- But limited
 - Concurrency