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

Programming Languages

- Syntax
  - Which string is a legal program?
  - Usually defined using context free grammar+ contextual constraints
- Semantics
  - What does a program mean?
  - What is the output of the program on a given run?
  - When does a runtime error occur?
  - A formal definition

Who need formal semantics for PL?

- Language designers
- Compiler designers
- [Programmers]

Example C++

- Designed with a source to source compiler to C
- Many issues
  - Especially later

Type Safety

- A programming language is type safe if every well typed program has no undefined semantics
- No runtime surprise
- Is C type safe?
- How about Java?

Breaking Safety in C

```
void foo(s) {
    char c[100];
    strcpy(c, s);
}
```

Pointers

- a = malloc(…) ;
- b = a;
- free (a);
- c = malloc (…);
- if (b == c) printf(“unexpected equality”);

A Pathological C Program

```
a =  malloc(…) ;
b = a;
free (a);
c = malloc (…);
if  (b == c)  printf(“unexpected equality”);
```

A Pathological C Program

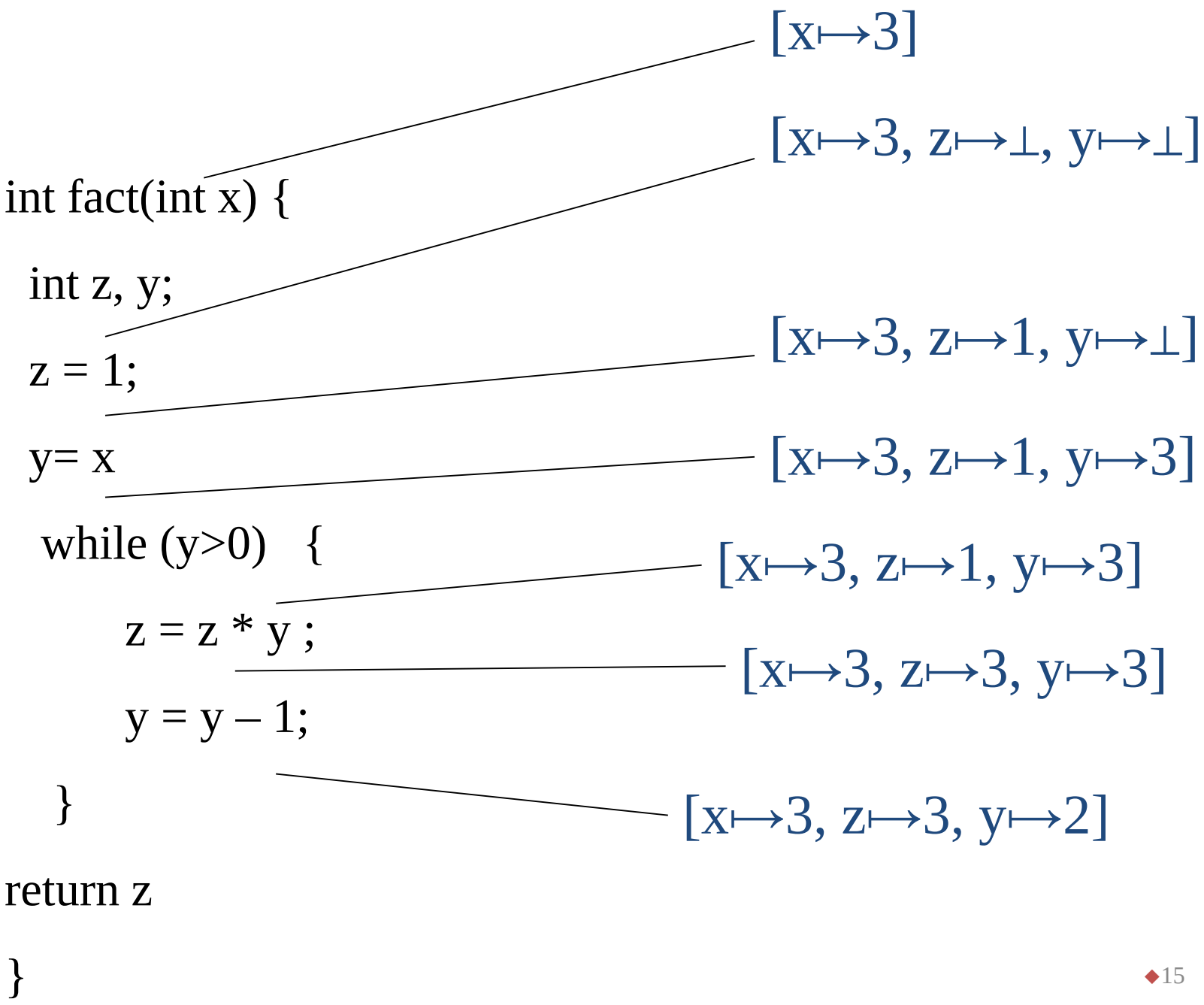
- a = malloc(…) ;
- b = a;
- free (a);
- c = malloc (…);
- if (b == c) printf(“unexpected equality”);

A Pathological C Program

```
a =  malloc(…) ;
b = a;
free (a);
c = malloc (…);
if  (b == c)  printf(“unexpected equality”);
```

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↦3]
[x↦3, z↦⊥, y↦⊥]
[x↦3, z↦1, y↦⊥]
[x↦3, z↦1, y↦3]
[x↦3, z↦1, y↦3]
[x↦3, z↦3, y↦3]
[x↦3, z↦3, y↦2]

```
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 ;
   while (y>0)   {
         z = z * y ;
         y = y – 1;
    }
return z;
}
```

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

Axiomatic Semantics

{x=n}

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

z = 1;

{x=n ∧ z=1}

y= x

{x=n ∧ z=1 ∧ y=n}

while

{x=n ∧ y ≥0 ∧ z=n! / y!}

(y>0) {

{x=n ∧ y >0 ∧ z=n! / y!}

z = z * y ;

{x=n ∧ y>0 ∧ z=n!/(y-1)!}

y = y – 1;

{x=n ∧ y ≥0 ∧ z=n!/y!}

} return z} {x=n ∧ z=n!}

The **While** Programming Language

- Abstract syntax
  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 |
  while b do S
- Use parenthesizes for precedence
- Informal Semantics
  - **skip** behaves like no-operation
  - Import meaning of arithmetic and Boolean operations

Example While Program

```
y := 1;
while ¬(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, …}
  - Truth values T={ff, tt}
  - States State = 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
- $A[\![Aexp]\!] : State \rightarrow N$
- Defined by structural induction on the syntax tree
  - $A[\![n]\!] s = n$
  - $A[\![x]\!] s = s\ x$
  - $A[\![e_1 + e_2]\!] s = A[\![e_1]\!] s + A[\![e_2]\!] s$
  - $A[\![e_1 * e_2]\!] s = A[\![e_1]\!] s * A[\![e_2]\!] s$
  - $A[\![(e_1)]\!] s = A[\![e_1]\!] s$ --- not needed
  - $A[\![- e_1]\!] s = -A[\![e_1]\!] s$

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
- $B[\![Bexp]\!] : State \rightarrow T$
- Defined by induction on the syntax tree
  - $B[\![true]\!] s = tt$
  - $B[\![false]\!] s = ff$
  - $B[\![e_1 = e_2]\!] s =$

$$\begin{cases} tt \text{ if } A[\![e_1]\!] s = A[\![e_2]\!] s \\ ff \text{ if } A[\![e_1]\!] s \neq A[\![e_2]\!] s \end{cases}$$

  - $B[\![e_1 \wedge e_2]\!] s =$

$$\begin{cases} tt \text{ if } B[\![e_1]\!] s = tt \text{ and } B[\![e_2]\!] = tt \\ ff \text{ if } B[\![e_1]\!] s = ff \text{ or } B[\![e_2]\!] s = ff \end{cases}$$

  - $B[\![e_1 \geq e_2]\!] s =$

$$\begin{cases} tt \text{ if } A[\![e_1]\!] s \geq A[\![e_2]\!] s \\ ff \text{ if } A[\![e_1]\!] s < A[\![e_2]\!] s \end{cases}$$

Natural Operational Semantics

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

Natural Semantics

- Notations
  - <S, s> - the program statement S is executed on input state s
  - s representing a terminal (final) state
- For every statement S, write meaning rules
  <S, *i*> $\rightarrow$ *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 i is the set of outputs states *o* such that <P, *i*> $\rightarrow$ *o*
- The meaning of compound statements is defined using the meaning immediate constituent statements

Natural Semantics for While

axioms

$[ass_{ns}] \langle x := a, s\rangle \rightarrow s[x \mapsto \mathbf{A}[\![a]\!]s]$

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

rules

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

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

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

Natural Semantics for While (More rules)

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

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

Simple Examples

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

  $[ass_{ns}] \langle x := x+1, s_0\rangle \rightarrow s_0[x \mapsto 1]$
- Skip statement

  $[skip_{ns}] \langle skip, s_0\rangle \rightarrow s_0$
- Composition

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

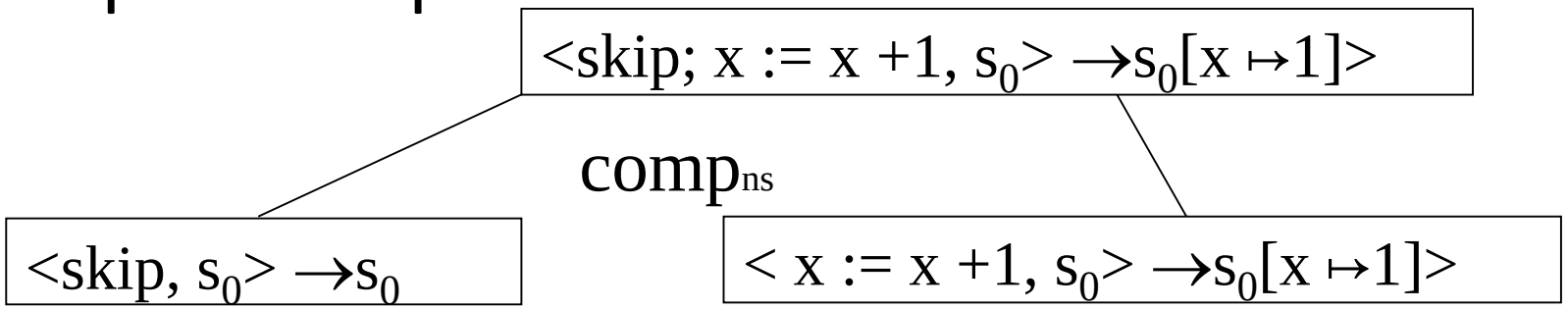
Simple Examples (Cont)

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

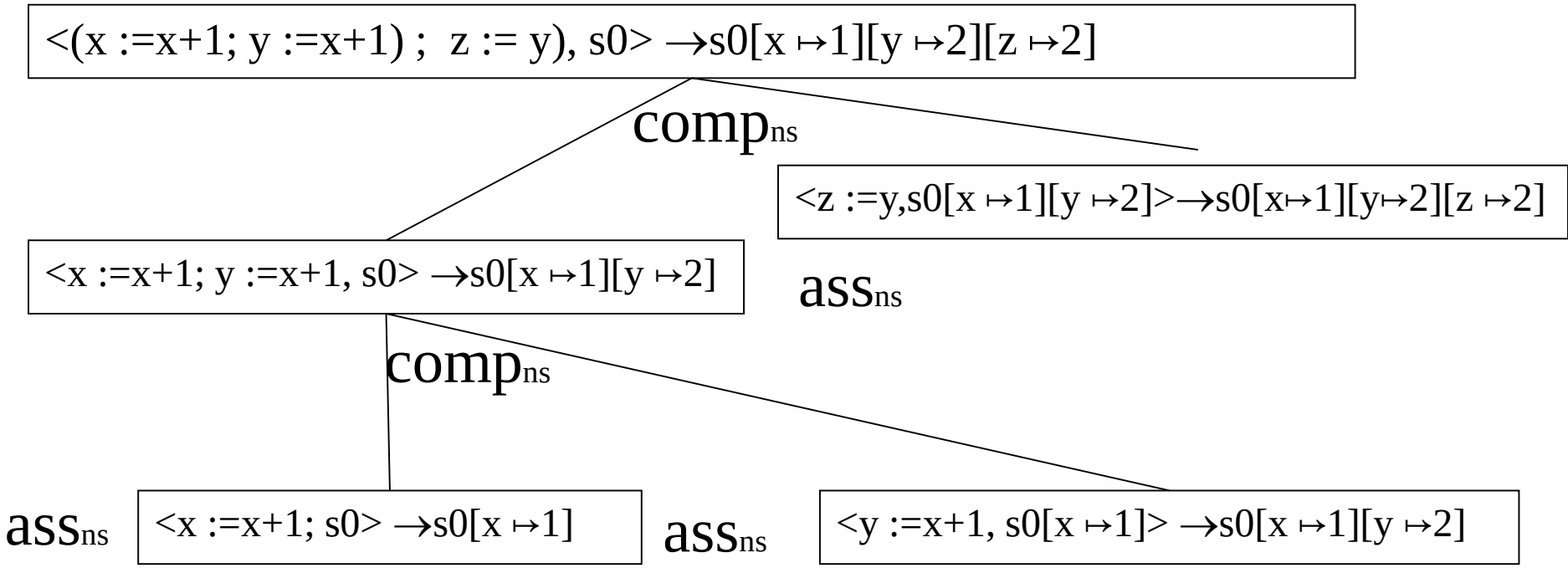
$$[\text{if}^{tt}_{ns}] \quad \frac{\langle \text{skip}, s_0 \rangle \rightarrow s_0}{\langle \text{if } x=0 \text{ then skip else } x := x+1, s_0 \rangle \rightarrow s_0}$$

A Derivation Tree

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



An Example Derivation Tree

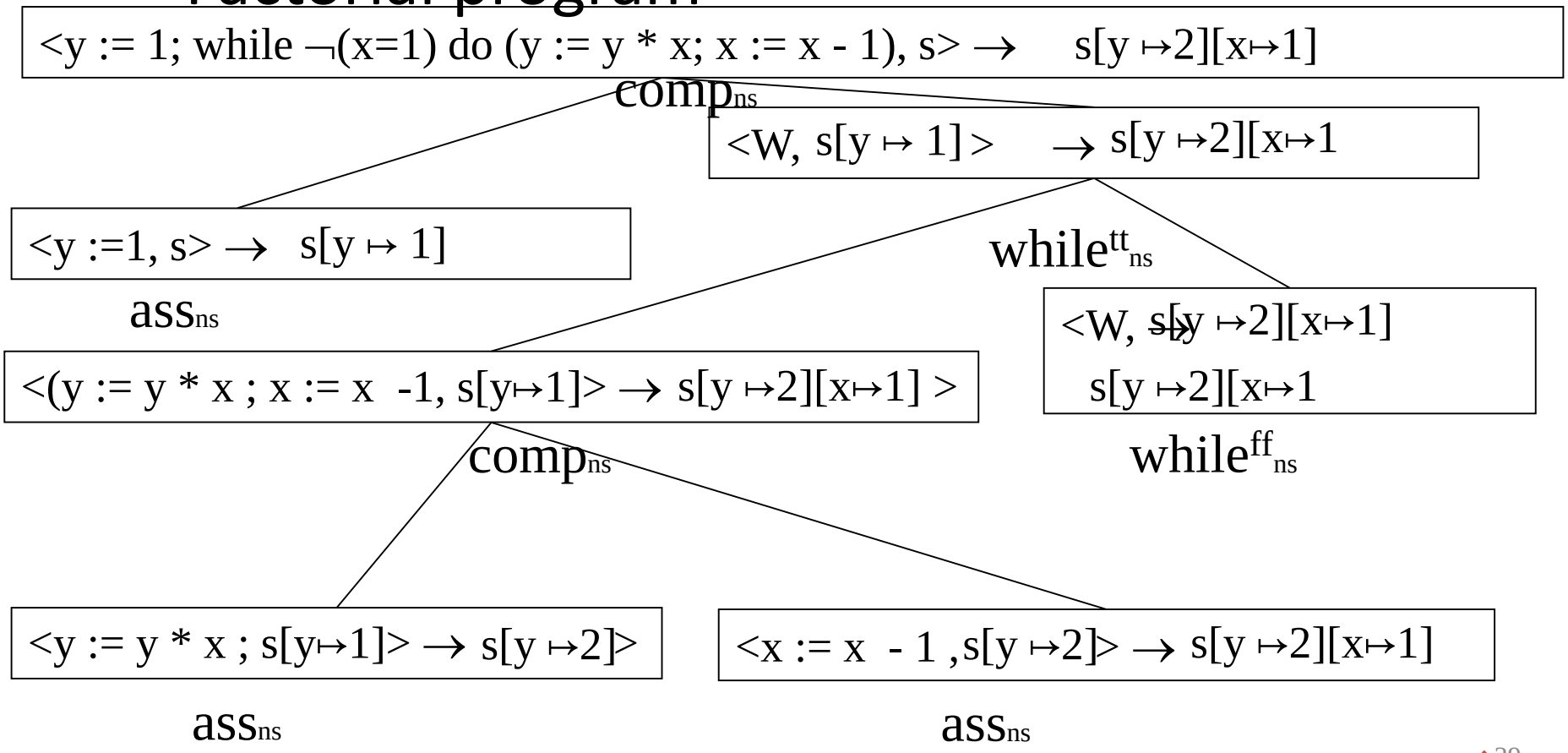


Top Down Evaluation of Derivation Trees

- 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



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 ≈ s’)
  $\langle S_1, s\rangle \rightarrow s'$ if and only if $\langle S_2, s\rangle \rightarrow s'$

Example of Semantic Equivalence

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

Deterministic Semantics for While

- If <S, s> $\rightarrow s_1$ and <S, s> $\rightarrow s_2$ 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}: **Stm** $\rightarrow$ **(State** $\hookrightarrow$ **State)**
- $S_{ns}\,[\![S]\!]s = s'$ if $\langle S, s\rangle \rightarrow s'$ and otherwise $S_{ns}\,[\![S]\!]s$ is undefined
- Examples
  - $S_{ns}\,[\![skip]\!]s = s$
  - $S_{ns}\,[\![x := 1]\!]s = s\,[x \mapsto 1]$
  - $S_{ns}\,[\![while\ true\ do\ skip]\!]s$ = undefined

Summary Natural Semantics

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