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

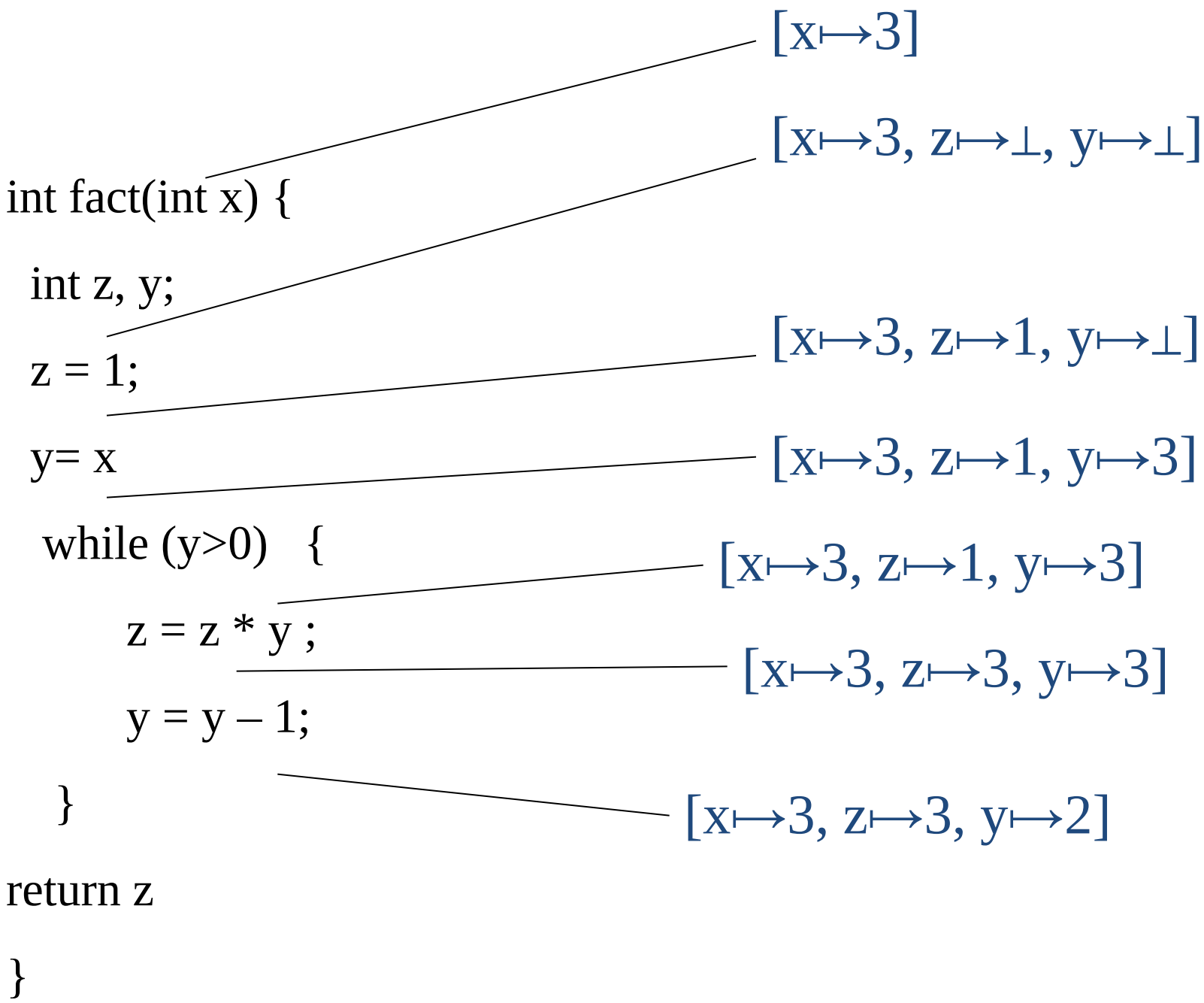
Syntax vs. Semantics

- 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


[x↦3]
[x↦3, z↦⊥, y↦⊥]
int fact(int x) {
int z, y;
z = 1;
[x↦3, z↦1, y↦⊥]
y= x
[x↦3, z↦1, y↦3]
while (y>0) {
[x↦3, z↦1, y↦3]
z = z * y ;
[x↦3, z↦3, y↦3]
y = y – 1;
}
[x↦3, z↦3, y↦2]
return z
}

```
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]

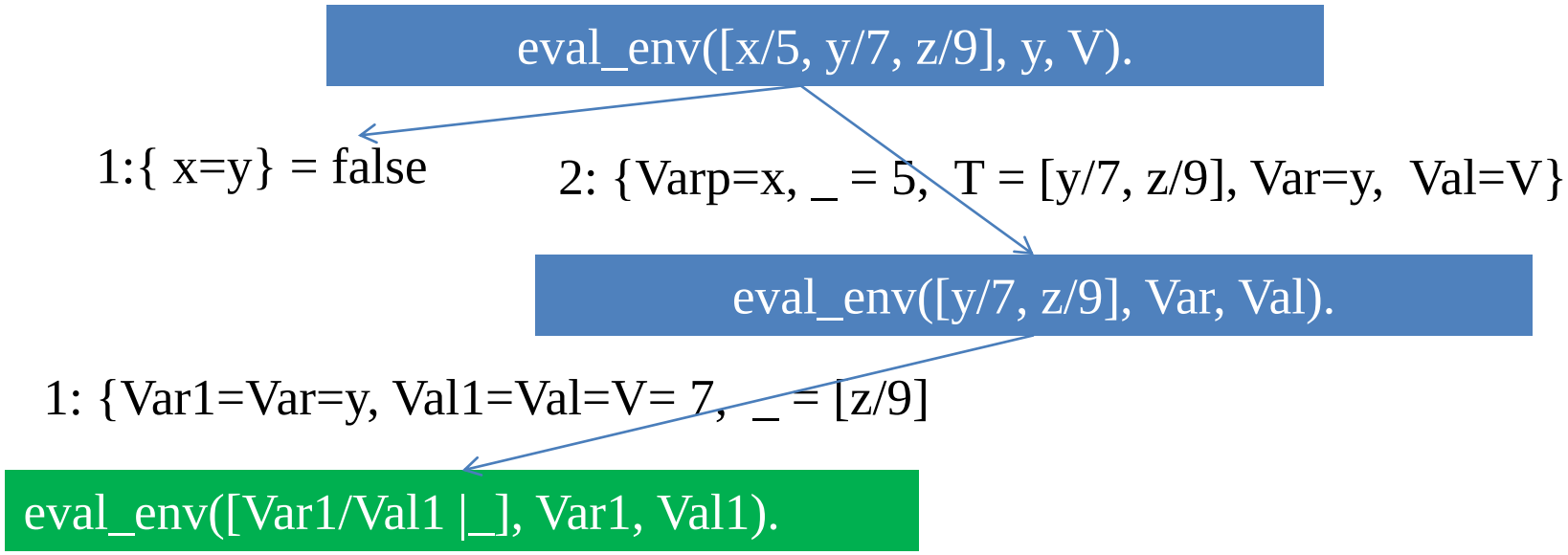
Example State Manipulations

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

Prolog Implementation Environment

1: eval_env([Var/Val |_], Var, Val).

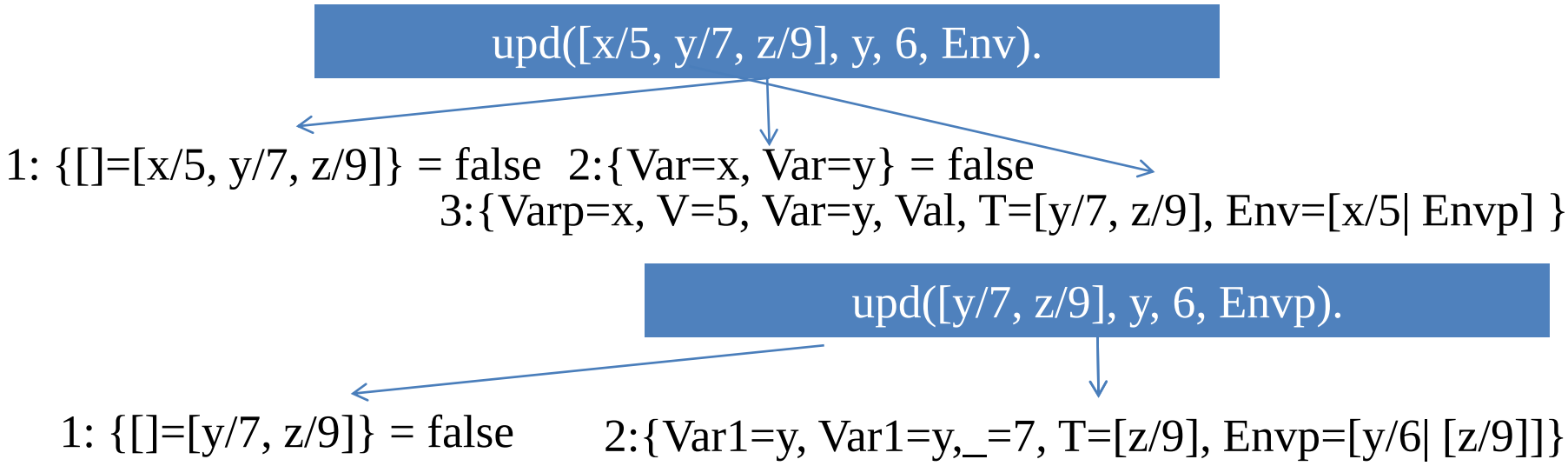
2: eval_env([Varp/_ | T], Var, Val) :- Varp \= Var,
eval_env(T, Var, Val).



V=7

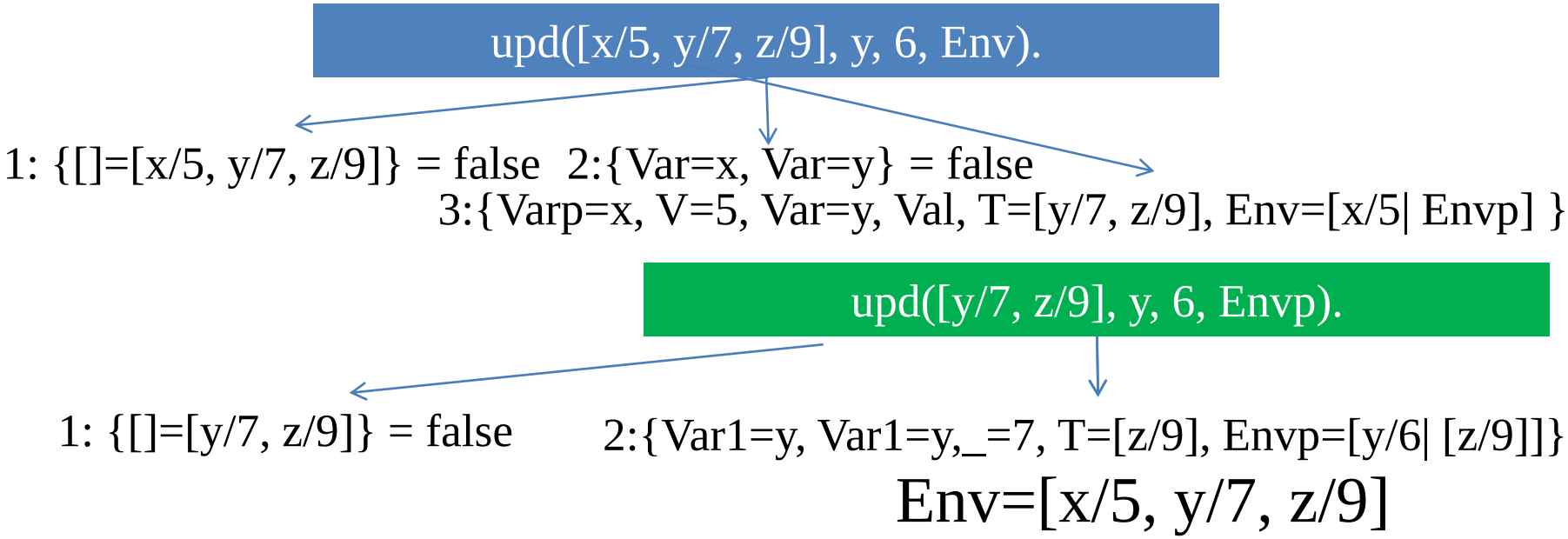
Prolog Implementation Environment(2)

```
1: upd([], Var, Val, [Var/Val]).
2: upd([Var/_ | T], Var, Val, [Var/Val | T]).
3: upd([Varp/V | T], Var, Val, [Varp/V | Envp])  :-
        Varp \= Var, upd(T, Var, Val, Envp).
```



Prolog Implementation Environment(2)

```
1: upd([], Var, Val, [Var/Val]).
2: upd([Var/_ | T], Var, Val, [Var/Val | T]).
3: upd([Varp/V | T], Var, Val, [Varp/V | Envp])  :-
        Varp \= Var, upd(T, Var, Val, Envp).
```



Semantics of arithmetic expressions

- Assume that arithmetic expressions are side-effect free
- $A[\![Aexp]\!] : State \rightarrow N$
- Defined by 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
  - $A[\![- e_1]\!] s = -A[\![e_1]\!] s$

Semantics of arithmetic expressions

- Assume that arithmetic expressions are side-effect free
- $A[\![Aexp]\!] : State \rightarrow N$
- Defined by induction on the syntax tree
  - $A[\![n]\!] s = n$
  - $A[\![x]\!] s = s\ x$
  - $A[\![e_1 + e_2]\!] s = A[\![e_1]\!] s + A[\![e_2]\!] s$
  - $A[\![e_1 * e_2]\!] s = A[\![e_1]\!] s * A[\![e_2]\!] s$
  - $A[\![- e_1]\!] s = -A[\![e_1]\!] s$
- Compositional
  - Properties can be proved by structural induction
- We say that e_1 is semantically equivalent to e_2 ($e_1 \approx e_2$) when $A[\![e_1]\!] = A[\![e_2]\!]$
- Theorem: for every expressions e_1, e_2: $e_1 + e_2 \approx e_2 + e_1$

Prolog Implementation (Expressions)

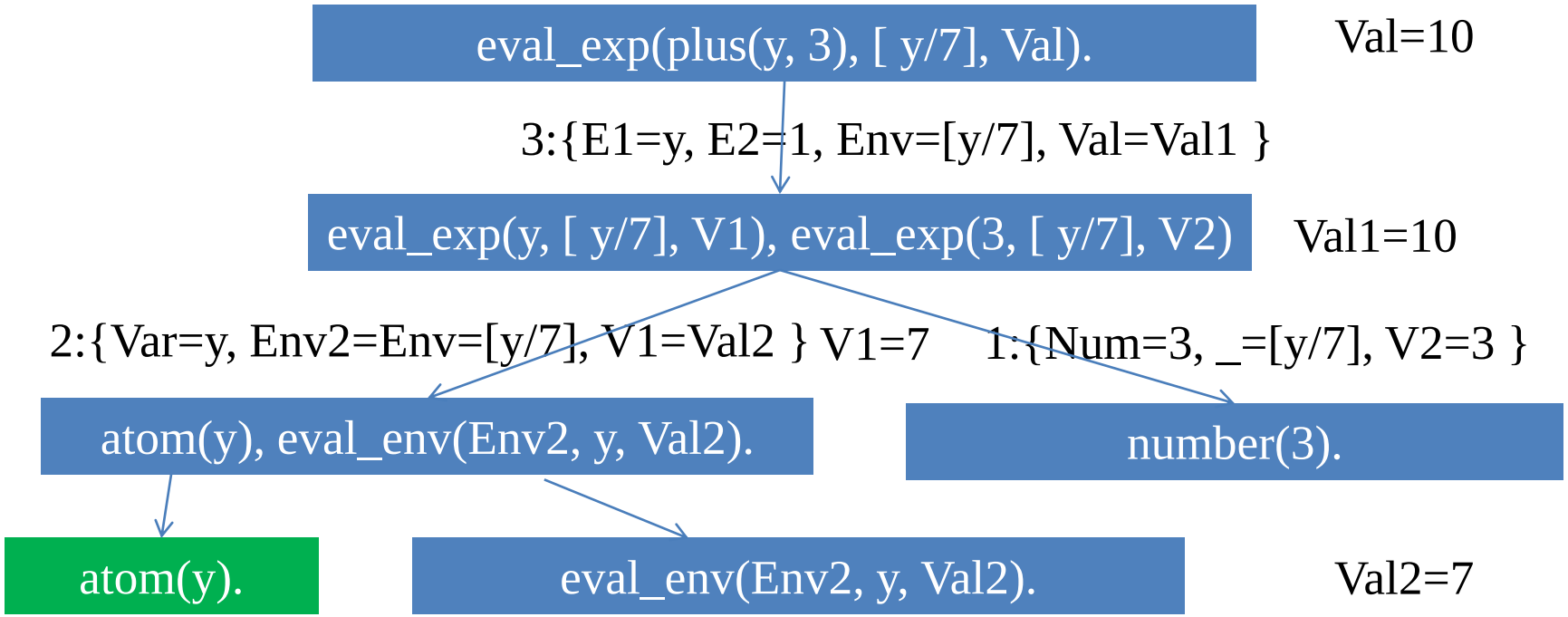
```
eval_exp(Num, _, Num) :- number(Num).
eval_exp(Var, Env, Val) :- atom(Var),
                           eval_env(Env, Var, Val).
eval_exp(plus(E1, E2), Env, Val) :- eval_exp(E1, Env, V1),
    eval_exp(E2, Env, V2),
    Val is V1 + V2.
eval_exp(minus(E1, E2), Env, Val) :- eval_exp(E1, Env, V1),
    eval_exp(E2, Env, V2),
    Val is V1 - V2.
eval_exp(mul(E1, E2), Env, Val) :- eval_exp(E1, Env, V1),
     eval_exp(E2, Env, V2),
     Val is V1 * V2.
```

Prolog Implementation (Expressions)

1: eval_exp(Num, _, Num) :- number(Num).

2: eval_exp(Var, Env, Val) :- atom(Var),
eval_env(Env, Var, Val).

3: eval_exp(plus(E1, E2), Env, Val) :- eval_exp(E1, Env, V1),
eval_exp(E2, Env, V2),
Val is V1 + V2.



Semantics of Boolean expressions

- Assume that Boolean expressions are side-effect free
- $B[\![Bexp]\!] : State \rightarrow T$
- Defined by induction on the syntax tree
  - $B[\![true]\!] s = tt$
  - $B[\![false]\!] s = ff$
  - $B[\![e_1 = e_2]\!] s =$
  - $B[\![e_1 \wedge e_2]\!] s =$

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

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

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

Prolog Implementation Boolean expression

```
eval_boolean_exp(true,_).
eval_boolean_exp(eq(E1, E2), Env) :-
      eval_exp(E1, Env, V1), eval_exp(E2, Env, V2),
       V1 = V2.
eval_boolean_exp(le(E1, E2), Env) :-
     eval_exp(E1, Env, V1), eval_exp(E2, Env, V2),
     V1 =< V2.
eval_boolean_exp(land(E1, E2), Env) :- eval_boolean_exp(E1, Env),
                                       eval_boolean_exp(E2, Env).
eval_boolean_exp(lor(E1, _), Env) :- eval_boolean_exp(E1, Env).
eval_boolean_exp(lor(_, E2), Env) :- eval_boolean_exp(E2, Env).
eval_boolean_exp(lneg(E), Env) :- \+ eval_boolean_exp(E, Env).
```

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 s 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

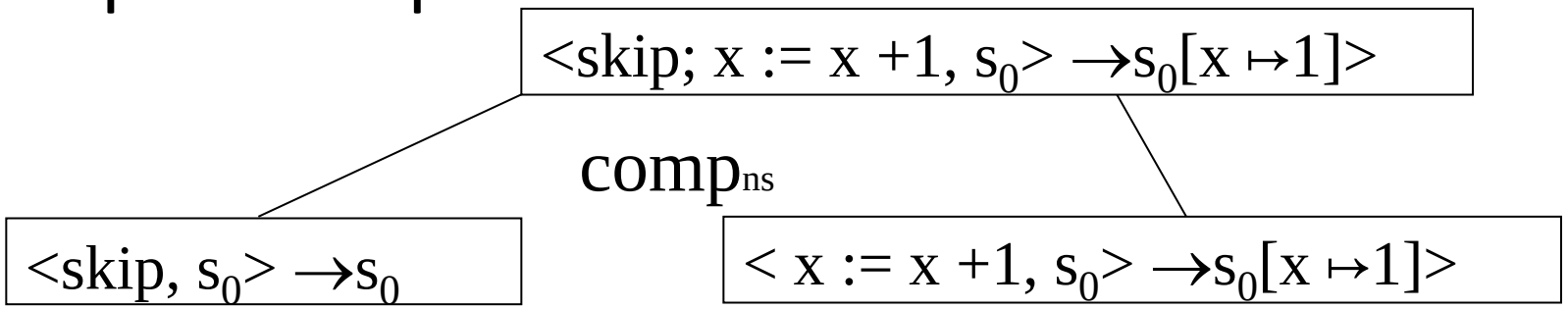
  $[ass_{ns}] \langle x := x+1, s_0 \rangle \rightarrow s_0[x \mapsto 1]$
- Skip statement

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

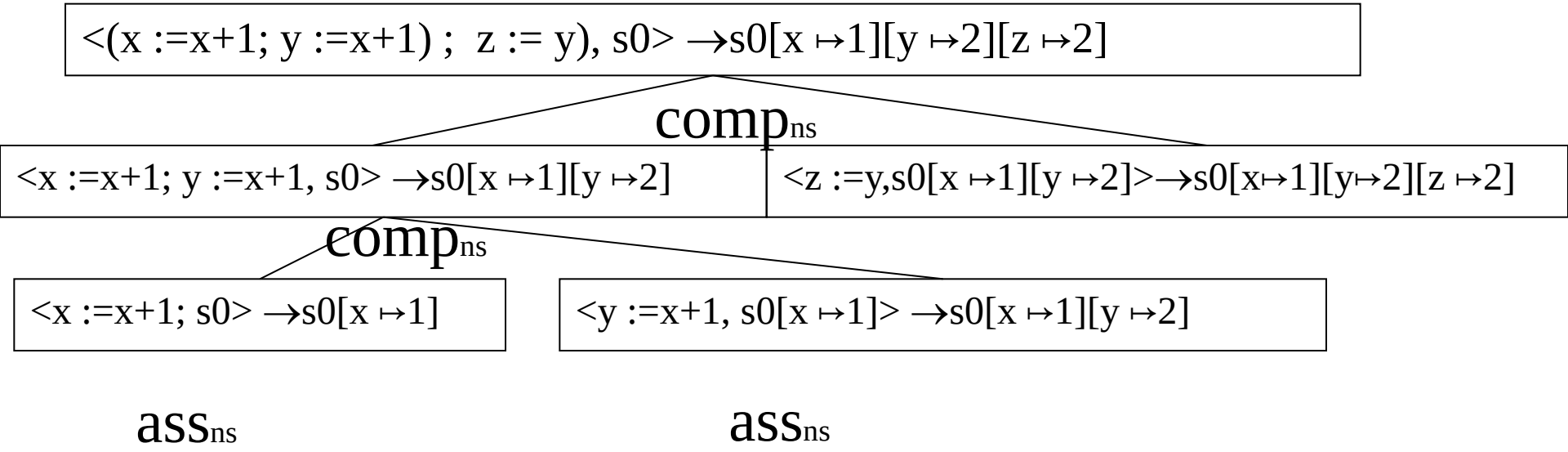
$$\frac{[comp_{ns}] \quad \langle skip, s_0 \rangle \rightarrow s_0, \ \langle x := x+1, s_0 \rangle \rightarrow s_0[x \mapsto 1]}{\langle skip; x := x+1, s_0 \rangle \rightarrow s_0[x \mapsto 1]}$$

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

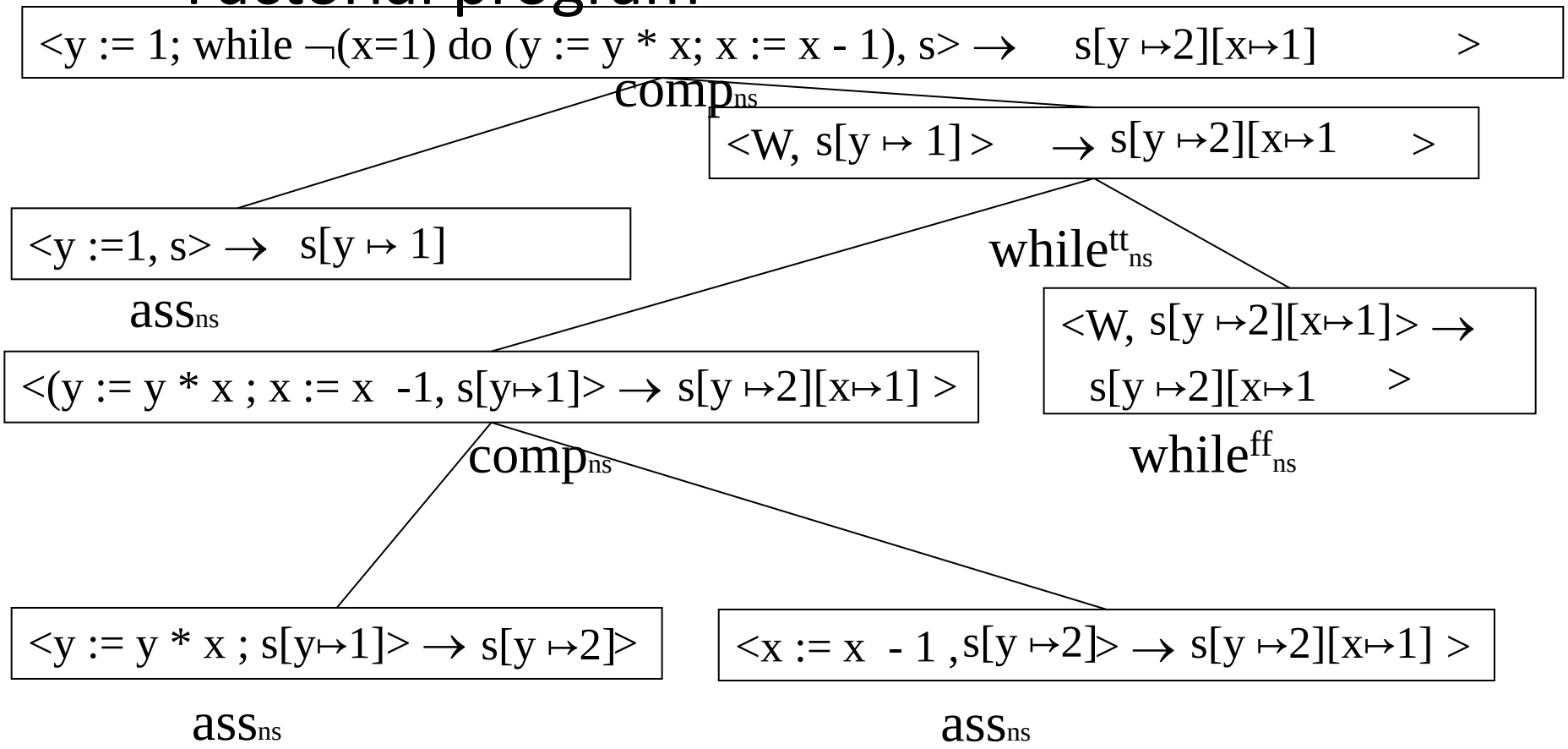


Top Down Evaluation of Derivation Trees

- 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



Natural Semantics Prolog

```
1: nat(skip, Env, Env).

2: nat(ass(LHS, RHS), Env, Envp) :-
       eval_exp(RHS, Env, Val), upd(Env, LHS, Val, Envp).

3: nat(seq(S1, S2), Env, Envp) :- nat(S1, Env, Envpp),
                                  nat(S2, Envpp, Envp).

4: nat(if(B, S1, _), Env, Envp) :- eval_boolean_exp(B, Env),
                                   nat(S1, Env, Envp).

5: nat(if(B, _, S2), Env, Envp) :- \+ eval_boolean_exp(B, Env),
                                   nat(S2, Env, Envp).

6: nat(while(B,S), Env, Envp) :- eval_boolean_exp(B, Env),
                                 nat(S, Env, Envpp),
                                 nat(while(B,S), Envpp, Envp).

7: nat(while(B,_), Env, Env) :- \+ eval_boolean_exp(B, Env).
```

```
nat(seq(ass(y, 1),
    while(lneg(eq(x, 1)),
          seq(ass(y, mul(y, x)),
          ass(x, minus(x, 1)))))),
    [x/3], E)
```

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'
  $\langle S_1, s\rangle \rightarrow s'$ if and only if $\langle S_2, s\rangle \rightarrow s'$
- Simple example
  "while b do S"
  is semantically equivalent to:
  "if b then (S ; while b do S) else skip"

Properties of Natural Semantics

- Equivalence of program constructs
  - "skip ; skip" is semantically equivalent to "skip"
  - "((S$_1$; S$_2$) ; S$_3$)" is semantically equivalent to "(S$_1$;(S$_2$; S$_3$))"
  - "(x := 5 ; y := x * 8)" is semantically equivalent to "(x :=5; y := 40)"
- Deterministic
  - If <S, s> $\rightarrow$ s$_1$ and <S, s> $\rightarrow$ s$_2$ then s$_1$=s$_2$

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