Semantics with Applications Operational Semantics of Extensions of WHILE

Pablo López

University of Málaga

November 5, 2024

Outline

Introduction

Non-sequential Language Constructs

Blocks and Procedures

Outline

Introduction

Non-sequential Language Constructs

Blocks and Procedures

A Matter of (Semantic) Style

The natural and structural operational semantics of `WHILE` are equivalent; however:

- for some constructs it is **easy** to specify the semantics in one style but **difficult** or even **impossible** in the other
- some **tools** or **applications** are easier to implement in one style

In this lecture we introduce some new constructs in `WHILE` and illustrate the power and weakness of both operational semantics styles.

Outline

Introduction

Non-sequential Language Constructs

Blocks and Procedures

The `abort` Statement

`abort` **stops** the execution of the complete program

Note that `abort` is different from:

- `while true do skip` which *loops*
- `skip` since the statement following `abort` is never executed

We first extend the syntax of `WHILE`:

$$\begin{array}{lll} S & ::= & x := a \mid \texttt{skip} \mid S_1 \; ; S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\ & \mid & \texttt{while } b \texttt{ do } S \mid \texttt{abort} \end{array}$$

The Semantics of `abort`

We assume that configurations $\langle S,\ s\rangle$ have been trivially extended to deal with `abort`.

The intended meaning is that configurations of the form:

$$\langle \texttt{abort},\ s\rangle$$

are **stuck**; i.e. a program in such a configuration does not progress any more.

We need to extend the definitions of $\rightarrow$ and $\Rightarrow$ with the appropriate axioms or rules.

No Semantic Definition, No Progress

Indeed, neither rules nor axioms are required to define the semantics of `abort`!
Since there is no definition for $\langle \texttt{abort},\ s\rangle$:

- in Natural Semantics we cannot complete a derivation tree for

$$\frac{?}{\langle \texttt{abort},\ s\rangle \to ?}$$

- in Structural Operational Semantics, the derivation sequence

$$\cdots \Rightarrow \langle \texttt{abort},\ s\rangle \Rightarrow ?$$

  is stuck

Structural Operational Semantics of `abort`

The Structural Operational Semantics captures the informal behavior of `abort`:

- `abort` is not semantically equivalent to `skip`:

$$\langle \texttt{abort},\ s\rangle \ \neq\ \langle \texttt{skip},\ s\rangle \Rightarrow s$$

- `abort` is not semantically equivalent to `while true do skip`:

$$\begin{aligned}
&\langle \texttt{while true do skip},\ s\rangle \\
&\Rightarrow \langle \texttt{if true then (skip; while true do skip) else skip},\ s\rangle \\
&\Rightarrow \langle \texttt{skip; while true do skip},\ s\rangle \\
&\Rightarrow \langle \texttt{while true do skip},\ s\rangle \\
&\Rightarrow \cdots
\end{aligned}$$

Natural Semantics of `abort`

The Natural Semantics does **not** capture the informal behavior of `abort`:

- `abort` is not semantically equivalent to `skip`:

$$\frac{?}{\langle \texttt{abort},\ s\rangle \to ?} \neq \frac{}{\langle \texttt{skip},\ s\rangle \to s}$$

- `abort` **is semantically equivalent** to `while true do skip`:

$$\frac{?}{\langle \texttt{abort},\ s\rangle \to ?} = \frac{?}{\langle \texttt{while true do skip},\ s\rangle \to ?}$$

  In both cases the derivation tree cannot be finished.

Natural Semantics versus Structural Operational Semantics

- if we have no derivation tree for $\langle S,\ s\rangle \to s'$, then we cannot tell whether it is because we entered a stuck configuration or a looping execution
- thus in Natural Semantics we cannot distinguish **looping** from **abnormal termination**
- on the other hand, in Structural Operational Semantics **looping** is reflected by **infinite** derivation sequences and **abnormal termination** by finite derivation sequences ending in a **stuck** configuration

Quiz

Is there anything we can do to make **looping** and **abnormal termination** semantically different in natural semantics?

Exercises

Exercise 2.31 We know that the natural semantics and structural operational semantics of `WHILE` are equivalent. Discuss whether a similar result holds for `WHILE` extended with `abort`.

Exercise 2.32 Extend `WHILE` with the statement

```
assert b before S
```

The idea is that if `b` evaluates to true, then we execute `S`; otherwise the execution of the complete program aborts. Extend the definition of the structural operational semantics to support this new construct. You are not allowed to rely on the definition of `abort`. Show that `assert true before S` is semantically equivalent to `S`, but `assert false before S` is equivalent to neither `while true do skip` nor `skip`.

The `or` Statement – Non-determinism

`S1 or S2` non-deterministically chooses to execute either `S1` or `S2`. Thus the statement:

```
x:= 1 or (x:= 2; x:= x+2);
```

yields one and only one of the two final states:

- $s\ \mathtt{x} = 1$, or
- $s\ \mathtt{x} = 4$

We first extend the syntax of `WHILE`:

$$\begin{array}{rcl} S & ::= & x := a \mid \mathtt{skip} \mid S_1\ ;\ S_2 \mid \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 \\ & \mid & \mathtt{while}\ b\ \mathtt{do}\ S \mid S_1\ \mathtt{or}\ S_2 \end{array}$$

and then proceed to extend the rules for → and ⇒.

Natural Semantics for `or`

We add the following two rules:

$$[\texttt{or}^1_{\text{ns}}] \quad \frac{\langle S_1,\ s\rangle \to s'}{\langle S_1 \ \texttt{or}\ S_2,\ s\rangle \to s'} \qquad\qquad [\texttt{or}^2_{\text{ns}}] \quad \frac{\langle S_2,\ s\rangle \to s'}{\langle S_1 \ \texttt{or}\ S_2,\ s\rangle \to s'}$$

Thus for the configuration

$$\langle \texttt{x} := 1 \ \texttt{or}\ (\texttt{x} := 2; \texttt{x} := \texttt{x} + 2),\ s\rangle$$

we have two derivation trees:

$$[\texttt{or}^1_{\text{ns}}] \quad \frac{\langle \texttt{x} := 1,\ s\rangle \to s[\texttt{x} \mapsto 1]}{\langle \texttt{x} := 1 \ \texttt{or}\ (\texttt{x} := 2; \texttt{x} := \texttt{x} + 2),\ s\rangle \to s[\texttt{x} \mapsto 1]}$$

$$[\texttt{or}^2_{\text{ns}}] \quad \frac{\begin{array}{c}\vdots\\ \langle (\texttt{x} := 2; \texttt{x} := \texttt{x} + 2),\ s\rangle \to s[\texttt{x} \mapsto 4]\end{array}}{\langle \texttt{x} := 1 \ \texttt{or}\ (\texttt{x} := 2; \texttt{x} := \texttt{x} + 2),\ s\rangle \to s[\texttt{x} \mapsto 4]}$$

Quiz

How many derivation trees are there for the configuration

$$\langle (\texttt{while true do skip}) \texttt{ or } (\texttt{x} := 2; \texttt{x} := \texttt{x} + 2),\ s \rangle$$

Structural Operational Semantics for `or`

We add the following two axioms:

$[\texttt{or}^1_{\text{sos}}] \quad \langle S_1 \texttt{ or } S_2, s\rangle \Rightarrow \langle S_1, s\rangle \qquad [\texttt{or}^2_{\text{sos}}] \quad \langle S_1 \texttt{ or } S_2, s\rangle \Rightarrow \langle S_2, s\rangle$

Thus for the configuration

$$\langle \texttt{x} := 1 \texttt{ or } (\texttt{x} := 2; \texttt{x} := \texttt{x} + 2),\ s\rangle$$

we have two derivation sequences:

$$\langle \texttt{x} := 1 \texttt{ or } (\texttt{x} := 2; \texttt{x} := \texttt{x} + 2),\ s\rangle \Rightarrow^\star s[\texttt{x} \mapsto 1]$$

and

$$\langle \texttt{x} := 1 \texttt{ or } (\texttt{x} := 2; \texttt{x} := \texttt{x} + 2),\ s\rangle \Rightarrow^\star s[\texttt{x} \mapsto 4]$$

Quiz

How many derivation sequences are there for the configuration

$$\langle (\texttt{while true do skip}) \ \texttt{or} \ (\texttt{x} := 2; \texttt{x} := \texttt{x} + 2), \ s \rangle$$

Natural Semantics versus Structural Operational Semantics

- if we do have a derivation tree for $\langle S,\ s\rangle \to s'$, then we always chose the *right* branch of execution
- thus in Natural Semantics non-determinism suppresses **looping**, if possible (we can only deal with derivation trees)
- on the other hand, in a Structural Operational Semantics non-determinism does not suppress **looping**; we deal with both finite and infinite derivations

Exercise

Exercise 2.34 Extend the WHILE language with the statement

```
random(x)
```

that changes the value of x to be any positive natural number. Extend the natural and structural operational semantics to support this statement. Discuss whether random would be redundant if WHILE were extended with the or statement as well.

The `par` Statement – Parallelism

`S1 par S2` executes both statements, but the execution can be **interleaved**.

Thus the statement:

```
x:= 1 par (x:= 2; x:= x+2);
```

yields one and only one of the three final states:

- $s\ \mathtt{x} = 1$, or
- $s\ \mathtt{x} = 3$, or
- $s\ \mathtt{x} = 4$

We first extend the syntax of `WHILE`:

$$\begin{array}{lcl} S & ::= & x := a \mid \mathtt{skip} \mid S_1 \ ; S_2 \mid \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 \\ & \mid & \mathtt{while}\ b\ \mathtt{do}\ S \mid S_1\ \mathtt{par}\ S_2 \end{array}$$

and then proceed to extend the rules for $\rightarrow$ and $\Rightarrow$.

Structural Operational Semantics for `par`

We add the following rules:

$$[\texttt{par}^1_{\text{sos}}] \qquad \frac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1 \texttt{ par } S_2,\ s\rangle \Rightarrow \langle S_1' \texttt{ par } S_2,\ s'\rangle}$$

$$[\texttt{par}^2_{\text{sos}}] \qquad \frac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1 \texttt{ par } S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$$

$$[\texttt{par}^3_{\text{sos}}] \qquad \frac{\langle S_2,\ s\rangle \Rightarrow \langle S_2',\ s'\rangle}{\langle S_1 \texttt{ par } S_2,\ s\rangle \Rightarrow \langle S_1 \texttt{ par } S_2',\ s'\rangle}$$

$$[\texttt{par}^4_{\text{sos}}] \qquad \frac{\langle S_2,\ s\rangle \Rightarrow s'}{\langle S_1 \texttt{ par } S_2,\ s\rangle \Rightarrow \langle S_1,\ s'\rangle}$$

- execute the first step of either S_1 or S_2 and proceed with remaining `par` statement
- complete the execution of either S_1 or S_2 and proceed with the remaining S_1 or S_2 statement

Derivation Sequences for par – Interleaving

For the statement

```
x:= 1 par (x:= 2; x:= x+2);
```

we get up to three derivation sequences, corresponding to three different execution orders of the atomic actions (shown in the next slide).

Example Derivation Sequences – Interleaving

$$\begin{array}{lcl}
\langle \texttt{x := 1 par (x := 2; x := x+2)},\ s\rangle & \Rightarrow & \langle \texttt{x := 2; x := x+2},\ s[\texttt{x}\mapsto\mathbf{1}]\rangle \\
& \Rightarrow & \langle \texttt{x := x+2},\ s[\texttt{x}\mapsto\mathbf{2}]\rangle \\
& \Rightarrow & s[\texttt{x}\mapsto\mathbf{4}]
\end{array}$$

$$\begin{array}{lcl}
\langle \texttt{x := 1 par (x := 2; x := x+2)},\ s\rangle & \Rightarrow & \langle \texttt{x := 1 par x := x+2},\ s[\texttt{x}\mapsto\mathbf{2}]\rangle \\
& \Rightarrow & \langle \texttt{x := 1},\ s[\texttt{x}\mapsto\mathbf{4}]\rangle \\
& \Rightarrow & s[\texttt{x}\mapsto\mathbf{1}]
\end{array}$$

and

$$\begin{array}{lcl}
\langle \texttt{x := 1 par (x := 2; x := x+2)},\ s\rangle & \Rightarrow & \langle \texttt{x := 1 par x := x+2},\ s[\texttt{x}\mapsto\mathbf{2}]\rangle \\
& \Rightarrow & \langle \texttt{x := x+2},\ s[\texttt{x}\mapsto\mathbf{1}]\rangle \\
& \Rightarrow & s[\texttt{x}\mapsto\mathbf{3}]
\end{array}$$

Natural Semantics for `par`

We add the following two rules:

$$[\texttt{par}^1_{\text{ns}}] \qquad \frac{\langle S_1,\ s\rangle \to s',\quad \langle S_2,\ s'\rangle \to s''}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \to s''}$$

$$[\texttt{par}^2_{\text{ns}}] \qquad \frac{\langle S_2,\ s\rangle \to s',\quad \langle S_1,\ s'\rangle \to s''}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \to s''}$$

- either execute `S1; S2` or execute `S2; S1`

Natural Semantics for `par`

We add the following two rules:

$$[\texttt{par}^{1}_{\text{ns}}] \qquad \frac{\langle S_1,\ s\rangle \to s',\quad \langle S_2,\ s'\rangle \to s''}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \to s''}$$

$$[\texttt{par}^{2}_{\text{ns}}] \qquad \frac{\langle S_2,\ s\rangle \to s',\quad \langle S_1,\ s'\rangle \to s''}{\langle S_1\ \texttt{par}\ S_2,\ s\rangle \to s''}$$

- either execute `S1; S2` or execute `S2; S1`

Does this work?

Natural Semantics for `par`

We add the following two rules:

$$[\texttt{par}^1_{\text{ns}}] \qquad \frac{\langle S_1,\ s\rangle \to s',\quad \langle S_2,\ s'\rangle \to s''}{\langle S_1 \texttt{ par } S_2,\ s\rangle \to s''}$$

$$[\texttt{par}^2_{\text{ns}}] \qquad \frac{\langle S_2,\ s\rangle \to s',\quad \langle S_1,\ s'\rangle \to s''}{\langle S_1 \texttt{ par } S_2,\ s\rangle \to s''}$$

- either execute `S1; S2` or execute `S2; S1`

Does this work?

Not really, we cannot **interleave** the executions of `S1` and `S2`

Natural Semantics versus Structural Operational Semantics

- in Natural Semantics, the execution of immediate constituents is an **atomic** entity thus we cannot express interleaving of computations
- in Structural Operational Semantics, we concentrate on the **small steps** of the computation, thus we can easily express interleaving of computations

Exercises

Exercise 2.35 Consider an extension of `WHILE` that in addition to `par` also includes the construct:

```
protect S end
```

so that `S` must be executed as an atomic entity. For example:

```
x:= 1 par protect (x:= 2; x:= x+2) end
```

has only two possible outcomes, namely $s\ \mathrm{x} = \mathbf{1}$ or $s\ \mathrm{x} = \mathbf{4}$. Extend the Structural Operational Semantics to support `protect` and get all the possible derivations for the sentence above.
Can you specify a Natural Semantics for `protect`?
Exercise 2.36 Specify the Structural Operational Semantics for arithmetic expressions $\mathbf{Aexp}$ where the individual parts of an expression may be computed in parallel. Prove that the new semantics is equivalent to $\mathcal{A}$.

Outline

Introduction

Non-sequential Language Constructs

Blocks and Procedures

Extending `WHILE` with Declarations

We now turn `WHILE` into a block-structured language:

- Blocks contain variable and procedure declarations
- We introduce new concepts:
  - **environments** for both variables and procedures
  - **locations** and **stores**
  - **scopes** for both variables and procedures
- We cover:
  - dynamic and static scope
  - recursive and non-recursive procedures

We shall concentrate on the Natural Semantics.

The BLOCK Language

We first extend WHILE with blocks containing declarations of local variables. The new language is called BLOCK:

$$\begin{array}{rcl} S & ::= & x := a \mid \texttt{skip} \mid S_1 \,;\, S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{begin}\ D_V\ S\ \texttt{end} \end{array}$$

where D_V is a meta-variable ranging over a new syntactic category $\mathbf{Dec}_V$ of **variable declarations**:

$$D_V \quad ::= \quad \texttt{var}\ x := a;\ D_V \mid \varepsilon$$

where ε is the empty declaration.

Global and Local Variables in BLOCK (I)

```
begin
   var y:= 1;            (* 1, local *)
   (
     x:= 1;              (* 2, global *)
     begin
        var x:= 2;       (* 3, local *)
        y:= x + 1        (* x refers to 3 *)
     end;
     x:= y + x           (* x refers to 2 *)
   )
end
```

What is the final value of `x`?

Global and Local Variables in BLOCK (I)

```
begin
   var y:= 1;              (* 1, local *)
   (
     x:= 1;                (* 2, global *)
     begin
        var x:= 2;         (* 3, local *)
        y:= x + 1          (* x refers to 3 *)
     end;
     x:= y + x             (* x refers to 2 *)
   )
end
```

What is the final value of `x`?

4

Global and Local Variables in BLOCK (II)

```
x:= 1;
y:= 2;
begin
   var x:= 10;
   var y:= 2 * x;

   s := x + y
end
```

What are the final values of `x`, `y`, `s`, and `z`?

Global and Local Variables in BLOCK (II)

```
x:= 1;
y:= 2;
begin
   var x:= 10;
   var y:= 2 * x;

   s  := x + y
end
```

What are the final values of x, y, s, and z?

`x = 1, y = 2, z = 0, s = 30`

Semantics of block statement = initialization + restoration

```
x:= 1;
y:= 2;
begin
   var x:= 10;
   var y:= 2 * x;
   (* initialize local variables *)

   s:= x + y
end
(* restore external variables *)
```

Natural Semantics for Variable Declarations

We consider configurations of the form:

- $\langle D_V,\ s\rangle$
- s

and the transition relation $\rightarrow_D$:

$$\langle D_V,\ s\rangle \rightarrow_D s'$$

the intended meaning is that we update the current state s with new bindings from D_V, obtaining the final state s'.

$$[\mathtt{var}_{\mathrm{ns}}] \quad \frac{\langle D_V,\ s[x \mapsto \mathcal{A}[\![a]\!]s]\rangle \rightarrow_D s'}{\langle \mathtt{var}\ x{:=}\ a;\ D_V,\ s\rangle \rightarrow_D s'} \qquad [\mathtt{none}_{\mathrm{ns}}] \quad \langle \varepsilon,\ s\rangle \rightarrow_D s$$

Exercise

Build the derivation tree for the declaration:

```
var x:= 1;
var y:= 2;
var x:= 3;
```

Set of Variables Declared in a Block

To define the semantics of variable declarations, we first introduce the function DV computing the set of variables declared in a block:

$$\begin{aligned}\mathrm{DV}(\texttt{var } x := a;\, D_V) &= \{x\} \cup \mathrm{DV}(D_V)\\ \mathrm{DV}(\varepsilon) &= \emptyset\end{aligned}$$

Generalization of Substitutions on States

Given states s and s' and a set of variables X, $s'[X \longmapsto s]$ denotes a state that is like s' except for variables in X, where it is as specified by s; that is:

$$(s'[X \longmapsto s])x = \begin{cases} s\ x & \text{if } x \in X \\ s'\ x & \text{if } x \notin X \end{cases}$$

Exercise. Given

$$s'\ \mathtt{x} = \mathbf{3} \quad s'\ \mathtt{y} = \mathbf{2} \quad s'\ \mathtt{z} = \mathbf{1} \quad s'\ \_ = \mathbf{0}$$

and:

$$s\ \mathtt{x} = \mathbf{1} \quad s\ \mathtt{y} = \mathbf{2} \quad s\ \mathtt{z} = \mathbf{3} \quad s\ \_ = \mathbf{0}$$

compute the values of `x`, `y`, `z`, and `w` in:

$$s'[\{\mathtt{x},\mathtt{y},\mathtt{z}\} \longmapsto s] \quad s[\{\mathtt{x},\mathtt{y},\mathtt{z}\} \longmapsto s'] \quad s'[\{\mathtt{w},\mathtt{x}\} \longmapsto s]$$

Natural Semantics for BLOCK

The Natural Semantics for BLOCK is identical to that of WHILE, except that we add a new rule for a block with variable declarations:

$$[\text{block}_{\text{ns}}] \quad \frac{\langle D_V,\ s\rangle \to_D s',\quad \langle S,\ s'\rangle \to s''}{\langle \texttt{begin}\ D_V\ S\ \texttt{end},\ s\rangle \to s''[DV(D_V) \longmapsto s]}$$

To execute a block from state s:

1. add local variables in D_V to s, yielding s'
2. execute the body S from s', yielding s''
3. restore the variables shadowed in s to their original values

Exercise

Exercise 2.37 Use the Natural Semantics of BLOCK to compute the final state of the statement:

```
begin
   var y:= 1;
   (
     x:= 1;
     begin
        var x:= 2;
        y:= x + 1
     end;
     x:= y + x
   )
end
```

The PROC Language

We now extend BLOCK to include (parameterless) procedure declarations and invocations. The resulting language is called PROC:

$$\begin{array}{lcl}
S & ::= & x := a \mid \texttt{skip} \mid S_1 \;; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\
 & \mid & \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{begin}\ D_V\ D_P\ S\ \texttt{end} \mid \texttt{call}\ p \\
D_V & ::= & \texttt{var}\ x := a;\ D_V \mid \varepsilon \\
D_P & ::= & \texttt{proc}\ p\ \texttt{is}\ S;\ D_P \mid \varepsilon
\end{array}$$

where:

- p ranges over procedures names $\mathbf{Pname}$
- D_P ranges over procedure declarations $\mathbf{Dec}_{\mathrm{P}}$

PROC and Scope Rules

A **binding** maps an identifier to a program entity (variable, procedure, etc.)

The **scope** of a binding is the region of the program where it is valid.

The scope can be either **static** or **dynamic**:

- **static scope** (or lexical) is determined at compile-time, according to the lexical structure of the program
- **dynamic scope** is determined at run-time, according to the run-time environment

We shall define three different semantics for PROC, differing in the scope rules:

- static scope for variables and procedures
- dynamic scope for variables and procedures
- dynamic scope for variables, static scope for procedures

Scope Rules in Action

The result of the statement below depends on the scope rules:

```
begin
   var x:= 0;
   proc p is x:= x * 2;
   proc q is call p;
   begin
      var x:= 5
      proc p is x:= x + 1;
      call q;
      y:= x
   end
end
```

Static Scope for Variables and Procedures

Assuming static scope for both variables and procedures:

```
begin
   var x:= 0;
   proc p is x:= x * 2;
   proc q is call p;
   begin
      var x:= 5
      proc p is x:= x + 1;
      call q;
      y:= x
   end
end
```

What is the final value of `y`?

Static Scope for Variables and Procedures

Assuming static scope for both variables and procedures:

```
begin
   var x:= 0;
   proc p is x:= x * 2;
   proc q is call p;
   begin
      var x:= 5
      proc p is x:= x + 1;
      call q;
      y:= x
   end
end
```

What is the final value of y?

5

Dynamic Scope for Variables and Procedures

Assuming dynamic scope for both variables and procedures:

```
begin
   var x:= 0;
   proc p is x:= x * 2;
   proc q is call p;
   begin
      var x:= 5
      proc p is x:= x + 1;
      call q;
      y:= x
   end
end
```

What is the final value of `y`?

Dynamic Scope for Variables and Procedures

Assuming dynamic scope for both variables and procedures:

```
begin
   var x:= 0;
   proc p is x:= x * 2;
   proc q is call p;
   begin
      var x:= 5
      proc p is x:= x + 1;
      call q;
      y:= x
   end
end
```

What is the final value of `y`?

6

Dynamic Scope for Variables, Static for Procedures

Assuming dynamic scope for variables and static for procedures:

```
begin
   var x:= 0;
   proc p is x:= x * 2;
   proc q is call p;
   begin
      var x:= 5
      proc p is x:= x + 1;
      call q;
      y:= x
   end
end
```

What is the final value of `y`?

Dynamic Scope for Variables, Static for Procedures

Assuming dynamic scope for variables and static for procedures:

```
begin
   var x:= 0;
   proc p is x:= x * 2;
   proc q is call p;
   begin
      var x:= 5
      proc p is x:= x + 1;
      call q;
      y:= x
   end
end
```

What is the final value of `y`?
10

Procedure Environments

The statement

```
call p
```

executes the body of a procedure named `p`.

Procedure Environments

The statement

```
call p
```

executes the body of a procedure named `p`.
Thus we need somehow to keep track of the association of procedure names $\mathbf{Pname}$ with procedure bodies $\mathbf{Stm}$.

Procedure Environments

The statement

```
call p
```

executes the body of a procedure named `p`.
Thus we need somehow to keep track of the association of procedure names $\mathbf{Pname}$ with procedure bodies $\mathbf{Stm}$.
A **procedure environment** env_P is a partial function that given a procedure name $p \in \mathbf{Pname}$ returns the statement $S \in \mathbf{Stm}$ corresponding to its body.

Procedure Environments

The statement

```
call p
```

executes the body of a procedure named `p`.
Thus we need somehow to keep track of the association of procedure names $\mathbf{Pname}$ with procedure bodies $\mathbf{Stm}$.
A **procedure environment** env_P is a partial function that given a procedure name $p \in \mathbf{Pname}$ returns the statement $S \in \mathbf{Stm}$ corresponding to its body.
We introduce the type $\mathbf{Env}_P$:

$$\mathbf{Env}_P = \mathbf{Pname} \hookrightarrow \mathbf{Stm}$$

thus $env_P \in \mathbf{Env}_P$.

Quiz

```
begin
   var x:= 0;
   proc p is x:= x + 2;
   proc q is call p;
   begin
      var x:= 5
      proc p is x:= x+1;
      call q;
      y:= x
   end
end
```

What is the value of env_P q ?
What is the value of env_P p ?

Natural Semantics with Procedure Environments

We extend the transition system to take into account procedure environments:

$$env_P \vdash \langle S,\ s\rangle \to s'$$

the intended meaning is that we execute S from s and env_P.
Axioms and rules for `WHILE` are simply recast with the new configurations, for example:

$$[\mathtt{skip}_{\mathrm{ns}}] \quad env_P \vdash \langle \mathtt{skip},\ s\rangle \to s$$

$$[\mathrm{comp}_{\mathrm{ns}}] \quad \frac{env_P \vdash \langle S_1,\ s\rangle \to s', \quad env_P \vdash \langle S_2,\ s'\rangle \to s''}{env_P \vdash \langle S_1;\ S_2,\ s\rangle \to s''}$$

Thus env_P is just propagated to where it is needed.

Managing the Procedure Environment

The axioms and rules of `WHILE` simply propagate env_P to make sure it is available where needed.
We need rules to:

- **update** the environment (the procedure declarations in `begin` ... `end` statements)
- **retrieve** the environment (the `call` statement)

The definitions of procedure environment and these rules determine the scope rules for procedures.

Dynamic Scope for Variables and Procedures

To define the semantics for a language with **dynamic** scope we need:

- a proper definition of the procedure environment (done)
- a strategy to update the procedure environment (`begin` . . . `end`)
- a strategy to retrieve the procedure environment (`call`)

Updating the Procedure Environment

Given a procedure environment env_P and a procedure declaration D_P the updated procedure environment $\mathrm{upd_P(D_P, env_P)}$ is defined by:

$$\begin{array}{rcl}\mathrm{upd_P}(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P, env_P) & = & \mathrm{upd_P}(D_P, env_P[p \mapsto S]) \\ \mathrm{upd_P}(\varepsilon, env_P) & = & env_P\end{array}$$

The `begin` ... `end` Statement

To execute the statement

```
begin Dv Dp S end
```

1. update the state with variables from `Dv`
2. update the procedure environment with procedures from `Dp`
3. execute `S` in the updated environment/state
4. restore the shadowed variables

$$[\text{block}_{ns}] \quad \frac{\langle D_V,\ s\rangle \to_D s',\quad \text{upd}_{\text{P}}(D_P, env_P) \vdash \langle S,\ s'\rangle \to s''}{env_P \vdash \langle \texttt{begin}\ D_V\ D_P\ S\ \texttt{end},\ s\rangle \to s''[\text{DV}(D_V) \longmapsto s]}$$

Quiz

To process variable declarations we are using transitions of the form $\langle D_V,\ s\rangle \rightarrow_D s'$ instead of $env_P \vdash \langle D_V,\ s\rangle \rightarrow_D s'$.
Do we need to take env_P into account to process D_V? Why?
Can you think of an extension to `PROC` so that transitions of the form $env_P \vdash \langle D_v,\ s\rangle \rightarrow_D s'$ are needed?

The `call` Statement

To execute the statement

```
call p
```

just retrieve the definition of `p` from the current environment and execute its body.

$$[\texttt{call}_{\text{ns}}] \quad \frac{env_P \vdash \langle S,\ s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\ s\rangle \to s'} \quad \text{where } env_P\ p = S$$

Note that invoking a non-existing procedure (env_P is a partial function) will abort the execution.

Quiz

Does this semantics

$$[\text{block}_{ns}] \quad \frac{\langle D_V,\ s\rangle \to_D s',\quad \text{upd}_\text{P}(D_V, env_P) \vdash \langle S,\ s'\rangle \to s''}{env_P \vdash \langle \texttt{begin}\ D_V\ D_P\ S\ \texttt{end},\ s\rangle \to s''[\text{DV}(D_V) \longmapsto s]}$$

$$[\texttt{call}_\text{ns}] \quad \frac{env_P \vdash \langle S,\ s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\ s\rangle \to s'} \quad \text{where } env_P\ p = S$$

support recursive procedure calls? Why?

Exercises (I)

Exercise 2.38 Consider the following statement of PROC:

```
begin
   proc fac is
      begin
         var z:= x;
         if x = 1 then skip
         else (
            x:= x - 1;
            call fac;
            y:= z * y
         )
      end
   (
     y:= 1;
     call fac
   )
end
```

construct a derivation tree for the execution from state s where $s\ \mathrm{x} = \mathbf{3}$.

Exercises (II)

Exercise 2.39 Use the semantics to verify that the statement:

```
begin
   var x:= 0;
   proc p is x:= x * 2;
   proc q is call p;
   begin
      var x:= 5
      proc p is x:= x + 1;
      call q;
      y:= x
   end
end
```

does indeed assign $\mathbf{6}$ to y.

Dynamic Scope for Variables, Static for Procedures (I)

To define the semantics for a language with **mixed** scope we need:

- a proper definition of the procedure environment
- a strategy to update the procedure environment (`begin ... end`)
- a strategy to retrieve the procedure environment (`call`)

Dynamic Scope for Variables, Static for Procedures (II)

Static scope for procedures means that a procedure only knows the procedures that were **already declared** when it itself is declared; thus future re-declarations will not interfere with its semantics.

Dynamic Scope for Variables, Static for Procedures (II)

Static scope for procedures means that a procedure only knows the procedures that were **already declared** when it itself is declared; thus future re-declarations will not interfere with its semantics.
To achieve this, we must take a *snapshot* of the procedure environment whenever we declare a procedure.

Dynamic Scope for Variables, Static for Procedures (II)

Static scope for procedures means that a procedure only knows the procedures that were **already declared** when it itself is declared; thus future re-declarations will not interfere with its semantics.
To achieve this, we must take a *snapshot* of the procedure environment whenever we declare a procedure.
A **procedure environment** env_P is a partial function that given a procedure name $p \in \mathbf{Pname}$ returns a pair with:

- the statement $S \in \mathbf{Stm}$ corresponding to its body, and
- the procedure environment env_P at the point of declaration (the *snapshot*)

Dynamic Scope for Variables, Static for Procedures (II)

Static scope for procedures means that a procedure only knows the procedures that were **already declared** when it itself is declared; thus future re-declarations will not interfere with its semantics.
To achieve this, we must take a *snapshot* of the procedure environment whenever we declare a procedure.
A **procedure environment** env_P is a partial function that given a procedure name $p \in \mathbf{Pname}$ returns a pair with:

- the statement $S \in \mathbf{Stm}$ corresponding to its body, and
- the procedure environment env_P at the point of declaration (the *snapshot*)

We introduce the type $\mathbf{Env}_P$:

$$\mathbf{Env}_P = \mathbf{Pname} \hookrightarrow (\mathbf{Stm}, \mathbf{Env}_P)$$

thus $env_P \in \mathbf{Env}_P$.

Updating the Procedure Environment

Given a procedure environment env_P and a procedure declaration D_P the updated procedure environment $\mathrm{upd_P(D_P, env_P)}$ is defined by:

$$\begin{aligned}\mathrm{upd_P}(\texttt{proc}\ p\ \texttt{is}\ S;\, D_P, env_P) &= \mathrm{upd_P}(D_P, env_P[p \mapsto (S, env_P)]) \\ \mathrm{upd_P}(\varepsilon, env_P) &= env_P\end{aligned}$$

Note that the definition is similar to the previous one, but we bind the current environment env_P to `p`.

Quiz

Are these definitions circular?

$$\mathbf{Env}_P = \mathbf{Pname} \hookrightarrow (\mathbf{Stm}, \mathbf{Env}_P)$$

$$\begin{array}{rcl}
\mathrm{upd_P}(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P, env_P) & = & \mathrm{upd_P}(D_P, env_P[p \mapsto (S, env_P)]) \\
\mathrm{upd_P}(\varepsilon, env_P) & = & env_P
\end{array}$$

The `call` Statement

To execute the statement

```
call p
```

just retrieve the definition of `p` from the current environment and execute its body S in the recorded environment env'_P.

$$[\texttt{call}_{\text{ns}}] \quad \frac{env'_P \vdash \langle S,\ s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\ s\rangle \to s'} \quad \text{where } env_P\ p = (S, env'_P)$$

Note that invoking a non-existing procedure (env_P is a partial function) will abort the execution.

Quiz

Does this semantics

$$[\mathrm{block}_{ns}] \quad \frac{\langle D_V,\ s\rangle \rightarrow_D s', \quad \mathrm{upd_P}(D_P, env_P) \vdash \langle S,\ s'\rangle \rightarrow s''}{env_P \vdash \langle \texttt{begin}\ D_V\ D_P\ S\ \texttt{end},\ s\rangle \rightarrow s''[\mathrm{DV}(D_V) \longmapsto s]}$$

$$[\texttt{call}_{\mathrm{ns}}] \quad \frac{env'_P \vdash \langle S,\ s\rangle \rightarrow s'}{env_P \vdash \langle \texttt{call}\ p,\ s\rangle \rightarrow s'} \quad \text{where } env_P\ p = (S, env'_P)$$

support recursive procedure calls? Why?

The `call` Statement – with Recursive Procedures

We can reconcile static scope for procedures and recursive procedures with the following alternative definition of `call`:

$$[\texttt{call}^{\text{rec}}_{\text{ns}}] \quad \frac{env'_P[p \mapsto (S, env'_P)] \vdash \langle S,\ s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\ s\rangle \to s'}$$

where $env_P\ p = (S, env'_P)$.

Exercises

Exercise 2.40 Construct a statement that illustrates the difference between the rules $[\texttt{call}_{\mathrm{ns}}]$ and $[\texttt{call}^{\mathrm{rec}}_{\mathrm{ns}}]$. Validate your claim constructing derivation trees for the execution of the statements from a given initial state.

Exercise 2.41 Use the semantics to verify that the statement:

```
begin
   var x:= 0;
   proc p is x:= x + 2;
   proc q is call p;
   begin
      var x:= 5
      proc p is x:= x + 1;
      call q;
      y:= x
   end
end
```

does indeed assign $\mathbf{10}$ to $\texttt{y}$.

Static Scope for Variables and Procedures

To define the semantics for a language with **static** scope we need:

- proper definitions of the procedure and variable environments
- a strategy to update the variable and procedure environments (`begin ... end`)
- a strategy to retrieve the procedure environment (`call`)

Variable Environments

A **variable environment** env_V associates a **location** with each variable:

$$\mathbf{Env}_V = \mathbf{Var} \to \mathbf{Loc}$$

where $\mathbf{Loc}$ is a set of **locations** or abstract addresses.
For the sake of simplicity, we take:

$$\mathbf{Loc} = \mathbb{Z}$$

furthermore, we assume that locations are non-negative.

Stores

A **store** sto associates a value with each location:

$$\mathbf{Store} = \mathbf{Loc} \cup \{\mathrm{next}\} \to \mathbb{Z}$$

where next is a special token that holds the *next free location* in the store.
We assume a function new:

$$\mathrm{new} : \mathbf{Loc} \to \mathbf{Loc}$$

that given a location l returns the next free location $\mathrm{new}\ l$.
For the sake of simplicity, we take:

$$\mathrm{new}\ l = l + 1$$

Splitting the State into Variable Environments and Stores

The idea is to replace the state s:

$$s : \mathbf{Var} \to \mathbb{Z}$$

by the function composition:

$$(sto \circ env_V) : \mathbf{Var} \to \mathbb{Z}$$

thus:

$$s = (sto \circ env_V)$$

Reading and Writing Variables

To determine the value of a variable x ($s\ x$):

1. determine the location $l = env_V\ x$ associated with x
2. determine the value $sto\ l$ associated with l

To assign a value v to a variable x ($s[x \mapsto v]$):

1. determine the location $l = env_V\ x$ associated with x
2. update the store ($sto[l \mapsto v]$) to have $sto\ l = v$

Initialization of Variable Environment and Store

When we worked with states in `WHILE`, we had $s\ \_ = 0$.
To get a similar initialization:

- the initial variable environment maps all variables to the location $\mathbf{0}$
- the initial store maps next to $\mathbf{1}$

Quiz

Is the proposed initialization for PROC:

- the initial variable environment maps all variables to the location $\mathbf{0}$
- the initial store maps next to $\mathbf{1}$

equivalent to having $s\ \_ = 0$ in WHILE?

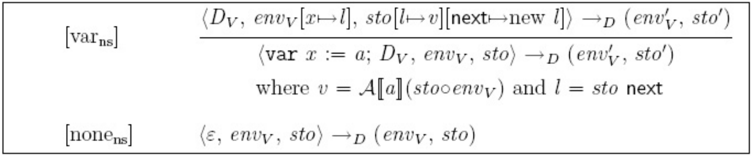
Variable Declarations

Both the variable environment and the store are updated by variable declarations.

Variable Declarations

Both the variable environment and the store are updated by variable declarations.
The transition system $\to_D$ is therefore modified to:

$$\langle D_V,\ env_V,\ sto\rangle \to_D \langle env'_V,\ sto'\rangle$$



$[\mathrm{var}_{\mathrm{ns}}]$	$\dfrac{\langle D_V,\ env_V[x\mapsto l],\ sto[l\mapsto v][\mathsf{next}\mapsto \mathrm{new}\ l]\rangle \to_D (env'_V,\ sto')}{\langle \mathtt{var}\ x := a;\ D_V,\ env_V,\ sto\rangle \to_D (env'_V,\ sto')}$ where $v = \mathcal{A}[\![a]\!](sto\circ env_V)$ and $l = sto\ \mathsf{next}$
$[\mathrm{none}_{\mathrm{ns}}]$	$\langle \varepsilon,\ env_V,\ sto\rangle \to_D (env_V,\ sto)$

Table 3.5: Natural semantics for variable declarations using locations

Extended Procedure Environments

We extend the procedure environment env_P to hold the variable environment env_V at the point of declaration (a *snapshot* of the current variables):

$$\mathbf{Env}_P = \mathbf{Pname} \hookrightarrow (\mathbf{Stm},\ \mathbf{Env}_V,\ \mathbf{Env}_P)$$

and modify the $\mathrm{upd_P}$ function accordingly:

$$\begin{array}{rcl} \mathrm{upd_P}(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P,\ env_V,\ env_P) & = & \mathrm{upd_P}(D_P, env_V, env_P[p \mapsto (S, env_V, env_P)]) \\ \mathrm{upd_P}(\varepsilon, env_V, env_P) & = & env_P \end{array}$$

Extended Transition System

The transition system is extended to have the form:

$$env_V, env_P \vdash \langle S,\ sto \rangle \rightarrow sto'$$

thus we get a relation between the initial store and the final store. Most rules and axioms are straightforward: just propagate the variable and procedure environments thus they are available where needed.

The Assignment (`:=`) and `skip` Statements

$$[\text{ass}_{\text{ns}}] \qquad env_V, env_P \vdash \langle x{:=}\ a, sto\rangle \rightarrow sto[l \mapsto v]$$

where:

$$l = env_V\ x \quad \text{and} \quad v = \mathcal{A}[\![a]\!](sto \circ env_V)$$

$$[\text{skip}_{\text{ns}}] \qquad env_V, env_P \vdash \langle \texttt{skip}, sto\rangle \rightarrow sto$$

The Sequential Composition (;) Statement

[comp$_{\text{ns}}$]

$$\frac{env_V, env_P \vdash \langle S_1, sto\rangle \rightarrow sto' \quad env_V, env_P \vdash \langle S_2, sto'\rangle \rightarrow sto''}{env_V, env_P \vdash \langle S_1;\, S_2, sto\rangle \rightarrow sto''}$$

The `if then else` Statement

$$[\mathrm{if}_{\mathrm{ns}}^{\mathrm{tt}}] \qquad \frac{env_V, env_P \vdash \langle S_1, sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, sto\rangle \to sto'}$$

where:

$$\mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{tt}$$

$$[\mathrm{if}_{\mathrm{ns}}^{\mathrm{ff}}] \qquad \frac{env_V, env_P \vdash \langle S_2, sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, sto\rangle \to sto'}$$

where:

$$\mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{ff}$$

The `while` Statement

$[\text{while}_{\text{ns}}^{\texttt{tt}}]$

$$\frac{env_V, env_P \vdash \langle S, sto\rangle \to sto' \quad env_V, env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, sto'\rangle \to sto''}{env_V, env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, sto\rangle \to sto''}$$

where:

$$\mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{tt}$$

$[\text{while}_{\text{ns}}^{\texttt{ff}}]$

$$env_V, env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, sto\rangle \to sto$$

where:

$$\mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{ff}$$

The Block (`begin` ... `end`) Statement

[block$_{\mathrm{ns}}$]

$$\frac{\langle D_V, env_V, sto\rangle \to_D (env'_V, sto') \quad env'_V, env'_P \vdash \langle S,\ sto'\rangle \to sto''}{env_V, env_P \vdash \langle \texttt{begin}\ D_V\ D_P\ S\ \texttt{end}, sto\rangle \to sto''}$$

where:

$$env'_P = \mathrm{upd_P}(D_P, env'_V, env_P)$$

Quiz

Compare:

$$\frac{\langle D_V, env_V, sto\rangle \rightarrow_D (env'_V, sto') \quad env'_V, env'_P \vdash \langle S,\ sto'\rangle \rightarrow sto''}{env_V, env_P \vdash \langle \texttt{begin}\ D_V\ \ D_P\ \ S\ \texttt{end}, sto\rangle \rightarrow sto''}$$

where:

$$env'_P = \mathrm{upd_P}(D_P, env'_V, env_P)$$

to:

$$\frac{\langle D_V,\ s\rangle \rightarrow_D s', \quad \mathrm{upd_P}(D_P, env_P) \vdash \langle S,\ s'\rangle \rightarrow s''}{env_P \vdash \langle \texttt{begin}\ D_V\ \ D_P\ \ S\ \texttt{end},\ s\rangle \rightarrow s''[\mathrm{DV}(D_V) \longmapsto s]}$$

What happened to $[\mathrm{DV}(D_V) \longmapsto s]$?

The `call` Statement – Non-recursive Procedures

[call$_{\text{ns}}$]

$$\frac{env'_V, env'_P \vdash \langle S,\ \ sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{call}\ p, sto\rangle \to sto'}$$

where:

$$env_P\ p = (S, env'_V, env'_P)$$

The `call` Statement – Recursive Procedures

$[\text{call}_{\text{ns}}^{\text{rec}}]$

$$\frac{env'_V, env'_P[p \mapsto (S, env'_V, env'_P)] \vdash \langle S,\ sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{call}\ p, sto\rangle \to sto'}$$

where:

$$env_P\ p = (S, env'_V, env'_P)$$

Exercises

Exercises 2.42 to 2.47 (old, on-line edition)
Exercises 3.12 to 3.17 (new edition)