Programming Languages and Compiler Design

Natural Operational Semantics of Languages **Block** and **Proc**

Yliès Falcone, Jean-Claude Fernandez

Master of Sciences in Informatics at Grenoble (MoSIG)
Univ. Grenoble Alpes
(Université Joseph Fourier, Grenoble INP)

Academic Year 2015 - 2016

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Blocks and variable declarations: syntax

Extending language **While**.

Definition (Language **Block**)

$$
\begin{array}{lcl}
S & \in & \mathbf{Stm} \\
S & ::= & x := a \mid \mathsf{skip} \mid S; S \\
 & & \mid \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S\ \mathsf{fi} \\
 & & \mid \mathsf{while}\ b\ \ \mathsf{do}\ S\ \mathsf{od} \\
 & & \mid \mathsf{begin}\ D_V\ \ S\ \mathsf{end}
\end{array}
$$

Definition (Syntactic category $\mathbf{Dec}_V$)

$$D_V ::= \mathsf{var}\ \ x;\ \ D_V \mid \mathsf{var}\ \ x := a;\ \ D_V \mid \epsilon$$

Example of program in **Block**

Example (Example of program in **Block**)

$$
\begin{array}{ll}
\text{begin} & \text{var } y := 1; \\
 & \text{var } x := 1; \\
 & \quad \text{begin} \quad \text{var } x := 2 \\
 & \qquad\qquad\quad y := x + 1 \\
 & \quad \text{end}; \\
 & \quad x := y + x \\
\text{end} &
\end{array}
$$

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Extending the Syntax of **While** with Blocks and Procedures
- Language **Block**
- Language **Proc**

Motivating Examples

Preliminaries

Natural Operational Semantics of Language **Block**

Natural Operational Semantics of **Proc**

Summary

Introducing Procedures in the syntax

Extending **Block** with procedure declarations.

Definition (Language **Proc**)

- Statements

$$
\begin{array}{rcl}
S & \in & \textbf{Stm} \\
S & ::= & x := a \mid \text{skip} \mid S_1; S_2 \\
 & & \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \\
 & & \mid \text{while } b \text{ do } S \text{ od} \\
 & & \mid \text{begin } D_V\ D_P\ \ S \text{ end} \mid \text{call } p
\end{array}
$$

- Variable declarations:

$$D_V \quad ::= \quad \text{var } x;\ D_V \mid \text{var } x := a;\ D_V \mid \epsilon$$

Definition (Syntactic category $\textbf{Dec}_P$)

$$D_P ::= \text{proc } p \text{ is } S;\ D_P \mid \epsilon$$

Example: a program with procedures

Example (Program in **Proc**)

$$
\begin{array}{ll}
\text{begin} & \text{var } x := 0; \\
& \text{var } y := 1; \\
& \text{proc } p \text{ is } x := x * 2; \\
& \text{proc } q \text{ is call } p; \\
& \text{begin var } x := 5; \\
& \quad \text{proc } p \text{ is } x := x + 1; \\
& \quad \text{call } q; y := x; \\
& \text{end}; \\
\text{end} &
\end{array}
$$

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Example of program in **Block**

Example (Program in **Block**)

$$
\begin{array}{ll}
\text{begin} & \text{var } y := 1; \\
 & \text{var } x := 1; \\
 & \quad \begin{array}{ll} \text{begin} & \text{var } x := 2 \\ & \quad y := x + 1 \\ \text{end}; & \end{array} \\
 & \quad x := y + x \\
\text{end} &
\end{array}
$$

Questions:

1. Are the declarations active during declaration execution?
2. Which order to choose when executing the declarations?
3. Do we need to restore the initial state?
4. If so, how to restore the initial state?

Example of program in **Proc**

Example (Program in **Proc**)

$$
\begin{array}{ll}
\text{begin} & \text{var } x := 0; \\
 & \text{var } y := 1; \\
 & \text{proc } p \text{ is } x := x * 2; \\
 & \text{proc } q \text{ is call } p; \\
 & \text{begin var } x := 5; \\
 & \quad\quad \text{proc } p \text{ is } x := x + 1; \\
 & \quad\quad \text{call } q;\, y := x; \\
 & \text{end}; \\
\text{end} &
\end{array}
$$

What is the final value of y?

Example: a program with procedures

Example (Dynamic binding for variables and procedures)

$$
\begin{array}{ll}
\text{begin} & \text{var } x := 0; \\
& \text{var } y := 1; \\
& \text{proc } p \text{ is } x := x * 2; \\
& \text{proc } q \text{ is call } p; \\
& \text{begin var } x := 5; \\
& \quad \text{proc } p \text{ is } x := x + 1; \\
& \quad \text{call } q; y := x; \\
& \text{end;} \\
\text{end} &
\end{array}
$$

We need to have some “memorization” of the current “procedure mapping”

$\hookrightarrow$ when we call q we call p and modify x

Example: a program with procedures

Example (Static binding for procedures)

$$
\begin{array}{ll}
\text{begin} & \text{var } x := 0; \\
& \text{var } y := 1; \\
& \text{proc } p \text{ is } x := x * 2; \\
& \text{proc } q \text{ is call } p; \\
& \text{begin var } x := 5; \\
& \quad \text{proc } p \text{ is } x := x + 1; \\
& \quad \text{call } q;\, y := x; \\
& \text{end}; \\
\text{end} &
\end{array}
$$

We need to:

- have some "memorization" of the current "procedure mapping" that "remembers the current procedure definitions when it has been defined"

$\hookrightarrow$ when we call q we call p and modify x

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Some preliminary notations: stacks

We use a stack structure to *manage local declarations*.

Let $\mathcal{F}$ be a set of (partial) functions.

Definition (Stack notations)

- The set of stacks over $\mathcal{F}$ is noted $\mathcal{F}^*$.
- Elements of $\mathcal{F}^*$ are noted $\hat{f}, \hat{f}_1, \hat{f}_2 \ldots$
- The empty stack is denoted by $\emptyset$.

Remark A stack can be seen as a sequence where:

- the push operation consists in appending to the right,
- the pop operation consists in suppressing from the right.

□

Remark When a stack $\hat{f}$ is reduced to one element we use sometimes notation f instead of $\hat{f}$. □

Some preliminary notations: stacks (ctd)

Definition (Evaluation on stacks)

Evaluation is defined inductively on stacks:

- $$(\hat{f} \oplus f')(x) = \begin{cases} f'(x) & \text{if } x \in \mathtt{Dom}(f'), \\ \hat{f}(x) & \text{otherwise.} \end{cases}$$

  ($\hat{f} \oplus f'$ is the stack resulting in pushing local function f' to stack $\hat{f}$.)
- $\emptyset(x) = \mathtt{undef}$.

Remark Consider the stack $\hat{f} = \hat{f}_1 \oplus \hat{f}_2$, $\hat{f}_1$ is a prefix of $\hat{f}$. □

Definition (Substitution (reminder))

$$f[y \mapsto v](x) = \begin{cases} v & \text{if } x = y, \\ f(x) & \text{otherwise.} \end{cases}$$

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Semantic domains

States are replaced by a symbol table plus a memory:

- a symbol table associates a memory address to a variable (an identifier);
- a memory associates a value to an address.

Definition (Symbol table: variable environment)

$$\mathbf{Env}_V = \mathbf{Var} \overset{part.}{\to} \mathbf{Loc} \ni \rho$$

Thus, $\hat{\rho}$ denotes a stack of tables.

Definition (Memory)

$$\mathbf{Store} = \mathbf{Loc} \overset{part.}{\to} \mathbb{Z} \ni \sigma$$

Intuition: function state corresponds to $\sigma \circ \hat{\rho}$.

Notation: new() is a function that returns a fresh memory location.

Semantic functions for arithmetical and boolean expressions

Definition (Semantic function for arithmetical expressions)

$$\mathcal{A} : \textbf{Aexp} \rightarrow ((\textbf{Env}_V{}^* \times \textbf{Store}) \rightarrow \mathbb{Z})$$

$$\begin{array}{ll}
\mathcal{A}[n](\hat{\rho},\sigma) & = \mathcal{N}[n] \\
\mathcal{A}[x](\hat{\rho},\sigma) & = \sigma(\hat{\rho}(x)) \\
\mathcal{A}[a_1 + a_2](\hat{\rho},\sigma) & = \mathcal{A}[a_1](\hat{\rho},\sigma) +_I \mathcal{A}[a_2](\hat{\rho},\sigma) \\
\mathcal{A}[a_1 * a_2](\hat{\rho},\sigma) & = \mathcal{A}[a_1](\hat{\rho},\sigma) *_I \mathcal{A}[a_2](\hat{\rho},\sigma) \\
\mathcal{A}[a_1 - a_2](\hat{\rho},\sigma) & = \mathcal{A}[a_1](\hat{\rho},\sigma) -_I \mathcal{A}[a_2](\hat{\rho},\sigma)
\end{array}$$

Exercise

Give the semantic function for boolean expressions.

Transition rules for assignment, skip, and sequential composition

Definition (Transition system for **While**)

Configurations:

$$\mathbf{Stm} \times {\mathbf{Env}_V}^* \times \mathbf{Store} \cup \mathbf{Store}$$

Transitions:

- Assignment:

$$(x := a, \hat{\rho}, \sigma) \to \sigma[\hat{\rho}(x) \mapsto \mathcal{A}[a](\hat{\rho}, \sigma)]$$

- Skip:

$$(\mathsf{skip}, \hat{\rho}, \sigma) \to \sigma$$

- Sequential composition:

$$\frac{(S_1, \hat{\rho}, \sigma) \to \sigma' \quad (S_2, \hat{\rho}, \sigma') \to \sigma''}{(S_1; S_2, \hat{\rho}, \sigma) \to \sigma''}$$

Transition rules for while and if

Definition (Transition system for **While**)

- While:
  - if $\mathcal{B}[b](\hat{\rho}, \sigma) = \mathbf{tt}$

$$\frac{(S, \hat{\rho}, \sigma) \to \sigma', \quad (\text{while } b \ \text{ do } S \text{ od }, \hat{\rho}, \sigma') \to \sigma''}{(\text{while } b \ \text{ do } S \text{ od }, \hat{\rho}, \sigma) \to \sigma''}$$

  - if $\mathcal{B}[b](\hat{\rho}, \sigma) = \mathbf{ff}$

$$(\text{while } b \ \text{ do } S \text{ od }, \hat{\rho}, \sigma) \to \sigma$$

Exercise

Give the rules for the if ... then ... else ... fi statement.

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Transition rules for blocks

To define the semantics, we define:

- a transition system for declarations, and
- an extended transition system for statements.

Definition (Transition system for Variable Declarations)

- Configurations:

$$(\mathbf{Dec}_V \times {\mathbf{Env}_V}^* \times \mathbf{Env}_V \times \mathbf{Store}) \cup (\mathbf{Env}_V \times \mathbf{Store})$$

  (i.e., of the form $(D_v, \hat{\rho}, \rho', \sigma)$ or (ρ', σ))

Transition rules for blocks

To define the semantics, we define:

- a transition system for declarations, and
- an extended transition system for statements.

Definition (Transition system for Variable Declarations)

- Transitions given by the transition relation $\rightarrow_D$ (where $l = \text{new}()$):

$$(\epsilon, \hat{\rho}, \rho', \sigma) \rightarrow_D (\rho', \sigma)$$

$$\frac{(D_V, \hat{\rho}, \rho[x \mapsto l], \sigma) \rightarrow_D (\rho', \sigma')}{(\text{var } x;\ D_V, \hat{\rho}, \rho, \sigma) \rightarrow_D (\rho', \sigma')}$$

$$\frac{(D_V, \hat{\rho}, \rho[x \mapsto l], \sigma[l \mapsto \mathcal{A}[a](\hat{\rho} \oplus \rho, \sigma)]) \rightarrow_D (\rho', \sigma')}{(\text{var } x := a;\ D_V, \hat{\rho}, \rho, \sigma) \rightarrow_D (\rho', \sigma')}$$

($\hat{\rho}$ means that the global env. is used to evaluate expressions.)
(ρ means that the local env. is used to evaluate expressions.)

Transition rules for blocks

If we allow only declarations of the form var x:

$$D_V \quad ::= \quad \text{var } x;\ D_V \mid \epsilon$$

Then, the transition system for declarations can be simplified.

Definition (Transition system for Variable Declarations)

- Configurations:

$$(\mathbf{Dec}_V \times \mathbf{Env}_V) \cup \mathbf{Env}_V$$

(i.e., of the form (D_v, ρ) or ρ)

- Transitions given by the transition relation $\rightarrow_D$ (where $l = \text{new}()$):

$$(\epsilon, \rho') \rightarrow_D \rho'$$

$$\frac{(D_V, \rho[x \mapsto l]) \rightarrow_D \rho'}{(\text{var } x;\ D_V, \rho) \rightarrow_D \rho'}$$

Transition rules for blocks

To define the semantics we define:

- a transition system for declarations, and
- a transition system for statements.

Definition (Natural Semantics for statements of **Block**)

- Configurations:

$$\mathbf{Stm} \times {\mathbf{Env}_V}^* \times \mathbf{Store} \cup \mathbf{Store}$$

- Transitions:

$$\frac{(D_V, \hat{\rho}, \emptyset, \sigma) \to_D (\rho_l, \sigma') \quad (S, \hat{\rho} \oplus \rho_l, \sigma') \to \sigma''}{(\text{begin } D_V \;\; S \text{ end}, \hat{\rho}, \sigma) \to \sigma''}$$

Execution of one statement of **Block**

Example

begin var $y := 1$;
var $x := 1$;
begin var $x := 2\ \ y := x + 1$ end
$x := y + x$
end

Let us note:

- D_{V_0} : var $y := 1;$ var $x := 1;$
- S_0 : (begin var $x := 2\ \ y := x + 1$ end); $x := y + x$
- S_{00} : (begin var $x := 2\ \ y := x + 1$ end)
- S_{01} : $x := y + x$
- D_{V_1} : var $x := 2$
- $S_1 = y := x + 1$

Let us compute a derivation tree of root
(begin $D_{V_0}\ \ S_0$ end$, \hat{\rho}_0, \sigma_0) \rightarrow \sigma_0''$ starting from $\sigma_0 = \emptyset, \hat{\rho}_0 = \emptyset$.

Execution of one statement of **Block** (ctd)

Rule of block

$$\frac{(D_{V_0}, \hat{\rho}_0, \emptyset, \sigma_0) \to_D (\rho_1, \sigma_1) \quad (S_0, \hat{\rho}_0 \oplus \rho_1, \sigma_1) \to \sigma_0''}{(\text{begin } D_{V_0} \;\; S_0 \text{ end}, \hat{\rho}_0, \sigma_0) \to \sigma_0''}$$

Rules of sequential composition and block

$$\frac{\dfrac{(D_{V_1}, \hat{\rho}_1, \emptyset, \sigma_1) \to_D (\rho_2, \sigma_2) \quad (S_1, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \to \sigma_3}{(S_{00}, \hat{\rho}_0 \oplus \rho_1, \sigma_1) \to \sigma_3} \qquad (S_{01}, \hat{\rho}_0 \oplus \rho_1, \sigma_3) \to \sigma_0''}{(S_0, \hat{\rho}_0 \oplus \rho_1, \sigma_1) \to \sigma_0''}$$

where

$$\begin{array}{rcl}
\rho_0 & = & \emptyset \\
\rho_1 & = & [y \mapsto l_1, x \mapsto l_2] \\
\sigma_1 & = & [l_1 \mapsto 1, l_2 \mapsto 1] \\
\hat{\rho}_1 & = & \hat{\rho}_0 \oplus \rho_1 = \emptyset \oplus \rho_1 = \rho_1 \\
\rho_2 & = & [x \mapsto l_3] \\
\sigma_2 & = & [l_1 \mapsto 1, l_2 \mapsto 1, l_3 \mapsto 2] \\
\sigma_3 & = & [l_1 \mapsto 3, l_2 \mapsto 1, l_3 \mapsto 2] \\
\sigma_0'' & = & [l_1 \mapsto 3, l_2 \mapsto 4, l_3 \mapsto 2]
\end{array}$$

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Dynamic bindings: remember the intuition

Example (Dynamic binding for variables and procedures)

```
begin   var  x := 0;
        var  y := 1
        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
```

We need to have some "memorization" of the current "procedure mapping"
$\hookrightarrow$ when we call q we call p and modify x

Semantics with dynamic bindings

Procedure names belong to a syntactic category called **Pname**.

$$
\begin{array}{lcll}
\mathbf{Env}_V & = & \mathbf{Var} \overset{part.}{\rightarrow} \mathbf{Loc} \ni \rho & \text{Variable environment} \\
\mathbf{Store} & = & \mathbf{Loc} \overset{part.}{\rightarrow} \mathbb{Z} \ni \sigma & \text{Store} \\
\mathbf{Env}_P & = & \mathbf{Pname} \overset{part.}{\longrightarrow} \mathbf{Stm} \ni \lambda & \text{Procedure environment}
\end{array}
$$

Example (Environment)

- $[p \mapsto x := x + 1]$: procedure name p is associated to statement $x := x + 1$.
- $[q \mapsto \text{call } p]$: procedure name q is associated to procedure call to p.

Semantics with dynamic bindings: transition system

Configurations: $(\mathbf{Stm} \times \mathbf{Env}_P{}^* \times \mathbf{Env}_V{}^* \times \mathbf{Store}) \cup \mathbf{Store}$

Transition rules:

$$\frac{(D_V, \hat{\rho}, \emptyset, \sigma) \to_D (\rho_I, \sigma') \quad (S, \hat{\lambda} \oplus \mathsf{upd}(\emptyset, D_P), \hat{\rho} \oplus \rho_I, \sigma') \to \sigma''}{(\mathsf{begin}\ D_V\ D_P\ \ S\ \mathsf{end}, \hat{\lambda}, \hat{\rho}, \sigma) \to \sigma''}$$

where

- $\mathsf{upd}(\lambda, \epsilon) = \lambda$ and
- $\mathsf{upd}(\lambda, \mathsf{proc}\ p\ \mathsf{is}\ S; D_P) = \mathsf{upd}(\lambda[p \mapsto S], D_P)$

$$\frac{(\hat{\lambda}(p), \hat{\lambda}, \hat{\rho}, \sigma) \to \sigma'}{(\mathsf{call}\ \ p, \hat{\lambda}, \hat{\rho}, \sigma) \to \sigma'}$$

Updating the rule for sequential composition:

$$\frac{(S_1, \hat{\lambda}, \hat{\rho}, \sigma) \to \sigma' \quad (S_2, \hat{\lambda}, \hat{\rho}, \sigma') \to \sigma''}{(S_1; S_2, \hat{\lambda}, \hat{\rho}, \sigma) \to \sigma''}$$

Similarly, other rules are adapted in a straightforward manner...

Static binding for procedures: remember the intuition

Example (Static binding for variables and procedures)

$$
\begin{array}{ll}
\text{begin} & \text{var } x := 0; \\
 & \text{var } y := 1 \\
 & \text{proc } p \text{ is } x := x * 2; \\
 & \text{proc } q \text{ is call } p; \\
 & \text{begin var } x := 5; \\
 & \quad \text{proc } p \text{ is } x := x + 1; \\
 & \quad \text{call } q; y := x; \\
 & \text{end}; \\
\text{end} &
\end{array}
$$

We need to:

- have some “memorization” of the current “procedure mapping” that “remembers the current procedure definitions when it has been defined”

$\hookrightarrow$ when we call q we call p and modify x

Semantics with static bindings

$$
\begin{array}{lcll}
\mathbf{Env}_V & = & \mathbf{Var} \stackrel{part.}{\rightarrow} \mathbf{Loc} \ni \rho & \text{Variable environment} \\
\mathbf{Store} & = & \mathbf{Loc} \stackrel{part.}{\rightarrow} \mathbb{Z} \ni \sigma & \text{Store} \\
\mathbf{Env}_P & = & \mathbf{Pname} \stackrel{part.}{\longrightarrow} \mathbf{Stm} \times {\mathbf{Env}_P}^* \times {\mathbf{Env}_V}^* \ni \rho & \text{Procedure environment}
\end{array}
$$

Definition (Updating the procedure environment)

$$\mathsf{upd} : {\mathbf{Env}_P}^* \times {\mathbf{Env}_V}^* \times \mathbf{Env}_P \times \mathbf{Dec}_P \longrightarrow \mathbf{Env}_P$$

- $\mathsf{upd}(\hat{\lambda}_g, \hat{\rho}, \lambda_l, \epsilon) = \lambda_l$, and
- $\mathsf{upd}(\hat{\lambda}_g, \hat{\rho}, \lambda_l, \mathsf{proc}\ p\ \mathsf{is}\ S; D_P)$
  $= \mathsf{upd}(\hat{\lambda}_g, \hat{\rho}, \lambda_l[p \mapsto (S, \hat{\lambda}_g \oplus \lambda_l, \hat{\rho})], D_P)$.

Semantics with static bindings: transition system

Configurations: $(\mathbf{Stm} \times \mathbf{Env}_P{}^* \times \mathbf{Env}_V{}^* \times \mathbf{Store}) \cup \mathbf{Store}$

Transition rules:

$$\frac{(D_V, \hat{\rho}, \emptyset, \sigma) \to_D (\rho_l, \sigma') \quad (S, \hat{\lambda} \oplus \mathsf{upd}(\hat{\lambda}, \hat{\rho} \oplus \rho_l, \emptyset, D_P), \hat{\rho} \oplus \rho_l, \sigma') \to \sigma''}{(\mathsf{begin}\ D_V\ D_P\ \ S\ \mathsf{end}, \hat{\lambda}, \hat{\rho}, \sigma) \to \sigma''}$$

Procedure call:

$$[\mathsf{call}] \quad \frac{(S, \hat{\lambda}', \hat{\rho}', \sigma) \to \sigma''}{(\mathsf{call}\ p, \hat{\lambda}, \hat{\rho}, \sigma) \to \sigma''}$$

where $\hat{\lambda}(p) = (S, \hat{\lambda}', \hat{\rho}')$.

Example dynamic bindings

Example (Program in **Proc**)

begin

D_{V_0} [var $x := 0$;
var $y := 1$;

DP_0 [proc p is $x := x * 2$;
proc q is call p;

S_0 [begin
DV_1 [var $x := 5$;
DP_1 [proc p is $x := x + 1$;
S_1 [call q; $y := x$;
end

end

Derivation tree for dynamic case

$$
\dfrac{\gamma_0 \to (\rho_1, \sigma_1) \qquad \dfrac{\gamma_1 \to (\rho_2, \sigma_2) \qquad \overbrace{(S_1, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \to \sigma_0''}^{T_1}}{(S_0, \hat{\lambda}_1, \rho_1, \sigma_1) \to \sigma_0''}}{(\textbf{begin } D_{V_0}; D_{P_0}; S_0 \textbf{ end}, \hat{\lambda}_0, \hat{\rho}_0, \sigma_0) \to \sigma_0''}
$$

where

$$
\begin{array}{lcl}
\gamma_0 & = & (D_{V_0}, \hat{\rho}_0, \emptyset, \sigma_0) \\
\hat{\lambda}_0 & = & \lambda_0 = \emptyset \\
\hat{\rho}_0 & = & \rho_0 = \emptyset \\
\sigma_0 & = & \emptyset \\
\gamma_1 & = & (D_{V_1}, \hat{\rho}_1, \emptyset, \sigma_1) \\
\rho_1 & = & [x \mapsto l_1, y \mapsto l_2] \\
\sigma_1 & = & [l_1 \mapsto 0, l_2 \mapsto 1] \\
\hat{\lambda}_1 & = & \lambda_1 = [p \mapsto x := x * 2, q \mapsto \text{call } p] (\text{function upd}) \\
\rho_2 & = & [x \mapsto l_3] \\
\sigma_2 & = & [l_1 \mapsto 0, l_2 \mapsto 1, l_3 \mapsto 5] \\
\lambda_2 & = & [p \mapsto x := x + 1]
\end{array}
$$

Derivation tree T_1 for dynamic case

$$
\dfrac{\dfrac{\dfrac{(x := x + 1, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \rightarrow \sigma_3}{(\text{call } p, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \rightarrow \sigma_3}}{(\text{call } q, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \rightarrow \sigma_3} \qquad (y := x, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_3) \rightarrow \sigma_0''}{(S_1, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \rightarrow \sigma_0''}
$$

where

$$
\begin{array}{rcl}
\sigma_3 & = & [l_1 \mapsto 0, l_2 \mapsto 1, l_3 \mapsto 6] \\
\sigma_0'' & = & [l_1 \mapsto 0, l_2 \mapsto 6, l_3 \mapsto 6]
\end{array}
$$

Example of static bindings

The only things that change are:

- function upd, and
- derivation tree T_1.

Changes to function upd

$$
\begin{array}{lcl}
\hat{\lambda}_0 & = & \emptyset = \lambda_0 \\
\hat{\lambda}_{01} & = & [p \mapsto (x := x * 2, \hat{\lambda}_0, \rho_1)] = \lambda_{01} \\
\hat{\lambda}_1 & = & [p \mapsto (x := x * 2, \hat{\lambda}_0, \rho_1), q \mapsto (\text{call } p, \hat{\lambda}_{01}, \rho_1)] = \lambda_1 \\
\lambda_2 & = & [p \mapsto (x := x + 1, \hat{\lambda}_1, \hat{\rho}_1 \oplus \rho_2)]
\end{array}
$$

Derivation tree T_1 for the static case

$$\dfrac{\dfrac{\dfrac{(x := x * 2, \hat{\lambda}_0, \hat{\rho}_1, \sigma_2) \to \sigma_3}{(\text{call } p, \hat{\lambda}_{01}, \hat{\rho}_1, \sigma_2) \to \sigma_3}}{(\text{call } q, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \to \sigma_3} \quad (y := x, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_3) \to \sigma_0''}{(S_0, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \to \sigma_0''}$$

where

$$\begin{array}{rcl} \sigma_3 & = & [l_1 \mapsto 0, l_2 \mapsto 1, l_3 \mapsto 5] \\ \sigma_0'' & = & [l_1 \mapsto 0, l_2 \mapsto 5, l_3 \mapsto 5] \end{array}$$

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Summary

Summary Natural Operational Semantics

Definition of the programming languages **While**, **Block**, **Proc**:

- Syntax (inductive definitions of the syntactic categories)
- Semantics for arithmetical and Boolean expressions
- Semantics for statements
- Termination of programs
- Semantics of blocks (semantics of declaration)
- Semantics of procedures (environment for procedures):
  - dynamic link for variables and procedures
  - static link for variables and procedures (symbol table and a memory)
  - dynamic link for variables and static link for procedures
  - recursive vs non-recursive calls (in the tutorial)