1 Functions and Local Scopes

We extend the language with a new category — declarations ($\mathscr{D}$), which consists of local variable declarations and function declarations. At the expression level we add scope expressions ($\mathscr{S}$), nested scopes and function calls. Finally, the category of programs $\mathscr{P}$ is scope expression:

$\mathscr{S}$	=	$\mathscr{D}^*$ $\mathscr{E}$	— scope expression
$\mathscr{E}$	+=	($\mathscr{S}$)	— nested scope
		$\mathscr{X}$ ($\mathscr{E}^*$)	— function call
$\mathscr{D}$	=	var $\mathscr{X}$	— local variable definition
		fun $\mathscr{X}$ ($\mathscr{X}^*$) { $\mathscr{S}$ }	— function definition
$\mathscr{P}$	=	$\mathscr{S}$	— program

1.1 Concrete Syntax

On the concrete syntax level we stipulate the following conventions:

1. the expression in scope expression is optional; if no expression is explicitly specified then "skip" is assumed;

2. local variable definition has to be terminated by a semicolon, for example

   ```
   var x;
   ```

3. multiple variable names can be specified in a single definition, for example

   ```
   var x, y, z;
   ```

   is equivalent to

   ```
   var x;
   var y;
   var z;
   ```

4. an optional initializer can be specified for a local variable definition; the initializers are reified into sequential assignments, preserving their order, for example

   ```
   var x = 3;
   var y = x + 5, z = x + y;
   ```

   is equivalent to

   ```
   var x;
   var y;
   var z;

   x := 3;
   y := x + 5;
   z := x + y
   ```

5. scope expressions are implicitly assumed in the bodies of loops and branches of conditional expressions;

6. in "repeat" expression the scope of body's immediate definitions is implicitly extended to enclose the whole expression, thus

```
repeat var i; read (i) until x > 0
```

   is equivalent to

```
(var i;
 repeat read (i) until x > 0)
```

7. an implicit scope expression is assumed in the initialization part of "for"-loop; the scope of its immediate definitions is extended to the whole expression as well, thus

```
for var i; i := 0, i < 10, i := i+1 do write (i) od
```

   is equavalent to

```
(var i; for i := 0, i < 10, i := i+1 do write (i) od )
```

1.2 Well-formedness

We assume functions to always return a values; thus, we need a way to materialize a void into some default value "$\perp$". We do this by introducing a new type of attribute — "**Weak** " — and adding the following set of rules to the inference system we used to ensure/restore expression well-formedness (see Fig. 1).

$$\frac{}{\textbf{Weak} \vdash x, \quad x \in \mathscr{X}} \qquad \frac{}{\textbf{Weak} \vdash z, \quad z \in \mathbb{N}}$$

$$\frac{\textbf{Val} \vdash l, \quad \textbf{Val} \vdash r}{\textbf{Weak} \vdash l \oplus r} \qquad \frac{}{\textbf{Weak} \vdash \text{skip}\,; \perp}$$

$$\frac{\textbf{Ref} \vdash l, \quad \textbf{Val} \vdash r}{\textbf{Weak} \vdash l := r} \qquad \frac{}{\textbf{Weak} \vdash \text{read}\ \ (x); \perp}$$

$$\frac{\textbf{Val} \vdash e}{\textbf{Weak} \vdash \text{write}\ \ (e); \perp} \qquad \frac{\textbf{Val} \vdash e, \quad \textbf{Void} \vdash s}{\textbf{Weak} \vdash \text{while } e \text{ do } s \text{ od}; \perp}$$

$$\frac{\textbf{Val} \vdash e, \quad \textbf{Void} \vdash s}{\textbf{Weak} \vdash (\text{repeat } s \text{ until } e); \perp}$$

Figure 1: Well-formedness: additional rules for the old kinds of expressions

We also need to specify the inference rules for the new kinds of expressions (see Fig. 2), and, finally, the rules for definitions (see Fig. 3).

$$\frac{\mathbf{Val} \vdash e_1, ..., \mathbf{Val} \vdash e_k}{\mathbf{Val} \vdash f(e_1, ..., e_k)} \qquad \frac{\mathbf{Val} \vdash e_1, ..., \mathbf{Val} \vdash e_k}{\mathbf{Weak} \vdash f(e_1, ..., e_k)} \qquad \frac{\mathbf{Val} \vdash e_1, ..., \mathbf{Val} \vdash e_k}{\mathbf{Void} \vdash \text{ignore } (f(e_1, ..., e_k))}$$

$$\frac{\vdash^{\mathscr{D}^*} d, \quad a \vdash e}{a \vdash d\ e}, \quad d \in \mathscr{D}^*$$

$$\frac{a \vdash e}{a \vdash \{\ e\ \}}$$

Figure 2: Well-formedness: additional rules for the new kinds of expressions

$$\vdash^{\mathscr{D}^*} \varepsilon \qquad \frac{\vdash^{\mathscr{D}} d, \quad \vdash^{\mathscr{D}^*} ds}{\vdash^{\mathscr{D}^*} d\, ds}$$

$$\vdash^{\mathscr{D}} \text{var } x \qquad \frac{\mathbf{Weak} \vdash e}{\vdash^{\mathscr{D}} \text{fun } f\ (...)\ \ \{\ e\ \}}$$

Figure 3: Well-formedness: additional rules for definitions

The top-level well-formedness condition for the while program p (which is now a scope expression) is

$$\mathbf{Void} \vdash p$$

1.3 Semantics

We now present the big-step operational semantics for functions and scopes. First, we introduce a shortcut for a multiple substitutions into a state: for a lists of variable names $x \in \mathscr{X}^*$ and values $v \in \mathscr{V}^*$ of equal lengths we define

$$\sigma[x_i \leftarrow v_i] = \sigma[x_1 \leftarrow v_1]...[x_k \leftarrow v_k]$$

Then, we add a new kind of value — a functional value:

$$\mathscr{V} += \mathscr{X}^* \mapsto \mathscr{E}$$

The simplest form of semantics for function calls could be as follows: assume we know that f is a function with arguments $a_1, ..., a_k$ and body b; then we evaluate its call $f(e_1, ..., e_k)$ as follows:

$$\frac{c \xrightarrow[*]{e_1 ... e_k} \langle \langle \sigma', \omega' \rangle, v \rangle \quad \langle \sigma'[a_i \leftarrow v_i], \omega' \rangle \xrightarrow{b} c''}{c \xrightarrow{f(e_1, ..., e_k)} c''}$$

Thus, however, would describe dynamic binding for functions, while our goal to have a semantics with static binding.

So, we modify the definition of state as follows:

$$\Sigma = (2^{\mathscr{X}} \times (\mathscr{X} \to \mathscr{V}))^+$$

Now a state is a non-empty list of scopes; in each scope we have a set of scope variables and a local state. The rightmost element of a state corresponds to the global state; all other elements corespond to properly ordered list of enclosing scopes for a given point in a program.

We need to redefine two primitives for states: those for reading and assigning values variables. For reading:

$$((L, s)\,\sigma)\,(x) = \begin{cases} s\,x & , \quad x \in L \\ \sigma(x) & , \quad x \notin L \end{cases}$$

For assigning:

$$((L, s)\,\sigma)\,[x \leftarrow v] = \begin{cases} (L, s\,[x \leftarrow v])\,\sigma & , \quad x \in L \\ (L, s)\,(\sigma\,[x \leftarrow v]) & , \quad x \notin L \end{cases}$$

With these primitives redefined all existing semantic rules can be preserved; however, we need to put additional requirements for some rules (assignment, reading from a variable, reading from a world) to ensure that we do not deal with functional values.

Now we need some new rules describing the semantics of new constructs (definitions and function calls). For scope expressions, the following "structural" rules are sufficient:

$$\frac{\textbf{enterScope}\ c \xRightarrow[\mathscr{D}^*]{d} c' \quad c' \xRightarrow{e} c''}{c \xRightarrow{d\ e} \textbf{leaveScope}\ c''} \qquad [\text{Scope}]$$

$$\frac{c \xRightarrow{e} c'}{c \xRightarrow{(\ e\ )} c'} \qquad [\text{LocalScope}]$$

We describe the primitives **enterScope** / **leaveScope** below; the transition "$\xRightarrow[\mathscr{D}^*]{}$" is, again, described with obvious structural rules:

$$c \xRightarrow[\mathscr{D}^*]{\varepsilon} c \qquad [\text{DefsEmpty}]$$

$$\frac{c \xRightarrow[\mathscr{D}]{d} c' \quad c' \xRightarrow[\mathscr{D}^*]{ds} c''}{c \xRightarrow[\mathscr{D}^*]{d\,ds} c''} \qquad [\text{DefsNonEmpty}]$$

The interesting part is the relation "$\xRightarrow[\mathscr{D}]{}$":

$$c \xrightarrow[\mathscr{D}]{\text{var } x} \textbf{addName}\ x\ 0\ c \qquad [\text{DefVar}]$$

$$c \xrightarrow[\mathscr{D}]{\text{fun } f\ (x_1 ... x_k)\ \{\ e\ \}} \textbf{addName}\ f\ (x_1..x_k\ \mapsto\ e)\ c \qquad [\text{DefFun}]$$

where the primitive "**addName** " is defined as follows:

$$\textbf{addName}\ \ x\ v\ \langle (L,s)\,\sigma,\omega\rangle = \langle (L \cup \{x\}, s[x \leftarrow v])\,\sigma,\omega\rangle$$

and we introduce the following shortcut for adding multiple names at once:

$$\textbf{addName}^*\ \langle x_1, x_1\rangle ... \langle x_k, v_k\rangle\ c = \textbf{addName}\ x_k\ v_k\ (...(\textbf{addName}\ x_1\ v_1\ c)...)$$

Now we define the primitives **enterScope** / **leaveScope** :

$$\begin{array}{rcl} \textbf{enterScope}\ \langle \sigma,\omega\rangle & = & \langle (\varnothing, \Lambda)\,\sigma,\omega\rangle \\ \textbf{leaveScope}\ \langle (L,s)\,\sigma,\omega\rangle & = & \langle \sigma,\omega\rangle \end{array}$$

Finally, we need to define the semantics for function calls:

$$\frac{\begin{array}{c} \sigma f = (\{a_i\}_{i=1}^k\ \mapsto\ e) \\ \langle \sigma,\omega\rangle \xrightarrow{e_1 ... e_k}{}^{*} \left\langle c', \{v_j\}_{j=1}^k \right\rangle \\ \forall j \in \{1..k\}.\ v_j \in \mathbb{Z} \\ \textbf{addName}^*\ \langle a_1, v_1\rangle ... \langle a_k, v_k\rangle\ (\textbf{enterFunction}\ c') \xrightarrow{e} \langle c'', w\rangle \end{array}}{\langle \sigma,\omega\rangle \xrightarrow{f(e_1 ... e_k)} \langle \textbf{leaveFunction}\ c'\ (\textbf{global}\ c''), w\rangle} \qquad [\text{Call}]$$

The primitives "**enterFunction** / **leaveFunction** / **global** " are defined as follows:

$$\begin{array}{rcl} \textbf{enterFunction}\ \langle \sigma,\omega\rangle & = & \langle \textbf{enterScope}\ (\textbf{global}\ \sigma),\omega\rangle \\ \textbf{leaveFunction}\ \langle (L',s')\,\varepsilon,\omega\rangle\ (L,s) & = & \langle (L,s)\,\varepsilon,\omega\rangle \\ \textbf{leaveFunction}\ \langle (L',s')\,\sigma,\omega\rangle\ (L,s) & = & \langle (L',s')\,(\textbf{leaveFunction}\ \sigma\ (L,s)),\omega\rangle,\ \sigma \neq \varepsilon \\ \textbf{global}\ (L,s)\,\varepsilon & = & (L,s) \\ \textbf{global}\ (L,s)\,\sigma & = & \textbf{global}\ \sigma,\ \sigma \neq \varepsilon \end{array}$$