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
		$\mathscr{X}\quad (\ \mathscr{E}^*\ )$	— 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

   ```
   do var i; read (i) while x > 0 od
   ```

   is equivalent to

   ```
   var i;
   do read (i) while x > 0 od
   ```

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).

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

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

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

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

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

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).

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

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

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

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

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

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

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

Figure 3: Well-formedness: additional rules for definitions

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

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:

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

$$\frac{c \xRightarrow{e} c'}{c \xRightarrow{\{\ e\ \}} c'} \qquad [\textsc{LocalScope}]$$

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

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

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

The interesting part is the relation " $\Longrightarrow_{\mathscr{D}}$ ":

$$c \xRightarrow{\texttt{var}\ x}_{\mathscr{D}} \textbf{addName}\ x\ 0\ c \qquad [\textsc{DefVar}]$$

$$c \xRightarrow{\texttt{fun}\ f\ (x_1 \ldots x_k)\ \{\ e\ \}}_{\mathscr{D}} \textbf{addName}\ f\ (x_1 .. x_k\ \mapsto\ e)\ c \qquad [\textsc{DefFun}]$$

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:

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

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:

$$\frac{\begin{array}{c} \sigma f = (a \mapsto e) \\ \langle \sigma, \omega\rangle \xrightarrow{e_1 \ldots e_k}{}_* \langle c', v\rangle \\ v_i \in \mathbb{Z} \\ \textbf{addName}^*\ \langle a_1, v_1\rangle \ldots \langle a_k, v_k\rangle\ (\textbf{enterFunction}\ c') \xrightarrow{e} \langle c'', w\rangle \end{array}}{\langle \sigma, \omega\rangle \xrightarrow{f(e_1 \ldots e_k)} \langle \textbf{leaveFunction}\ c'\ (\textbf{global}\ c''), w\rangle} \quad [\textsc{Call}]$$

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

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

2 Functions and Local Scopes in Stack Machine

To support functions and local scopes the stack machine has to be essentially redesigned.

First, we add a new notion — *location* ($\mathcal{L}oc$) — to the definition of stack machine. A location specifies where a non-stack operand of an instruction resides. For now the three kinds of locations are sufficient:

$$\begin{array}{rl} \textbf{global}\ \mathscr{X} & \text{— global variable} \\ \textbf{local}\ \mathbb{N} & \text{— local variable} \\ \textbf{arg}\ \mathbb{N} & \text{— function argument} \end{array}$$

Thus, now operands for instructions `ST`, `LD` and `LDA` are locations. Moreover, the set of *values* for stack machine now contains references to locations as well as plain integer numbers:

$$\mathscr{V} = \mathbb{Z} \mid \textbf{ref}\ \mathcal{L}oc$$

Next, we need a whole new bunch of instructions:

$$\begin{array}{rl} \texttt{GLOBAL}\ \mathscr{X} & \text{— declaration of global variable} \\ \texttt{CALL}\ \mathscr{X}\ \mathbb{N} & \text{— function call} \\ \texttt{BEGIN}\ \mathscr{X}\ \mathbb{N}\ \mathbb{N} & \text{— begin of function} \\ \texttt{END} & \text{— end of function} \end{array}$$

Next to last, in addition to a regular state we add the notion of *local state*:

$$\Sigma_{loc} = (\mathbb{N} \to \mathscr{V}) \times (\mathbb{N} \to \mathscr{V})$$

Local states keep values of arguments and local variables, indexed by their numbers, respectively.

Finally, we modify the configuration for stack machine:

$$\mathscr{C} = \mathscr{V}^* \times (\Sigma_{loc} \times \mathscr{P})^* \times (\Sigma_{loc} \times \Sigma) \times \mathscr{W}$$

In addition to a regular stack of values, global state and a world now the configurations contains two more items:

- a *control stack*, which is a stack of pairs of local state and programs, which keeps track of return points;
- a local state, which keeps a current local state.

For extended state we need to refedine the primitives for reading

$$\begin{array}{llcl} \langle\langle a, l\rangle, g\rangle & [\textbf{local}\ n] & = & l(n) \\ \langle\langle a, l\rangle, g\rangle & [\textbf{arg}\ n] & = & a(n) \\ \langle\langle a, l\rangle, g\rangle & [\textbf{global}\ x] & = & g(x) \end{array}$$

and the assignment

$$\begin{array}{llcl} \langle\langle a, l\rangle, g\rangle & [\textbf{local}\ n \leftarrow v] & = & \langle\langle a, l[i \leftarrow v]\rangle, g\rangle \\ \langle\langle a, l\rangle, g\rangle & [\textbf{arg}\ n \leftarrow v] & = & \langle\langle a[i \leftarrow v], l\rangle, g\rangle \\ \langle\langle a, l\rangle, g\rangle & [\textbf{global}\ x \leftarrow v] & = & \langle\langle a, l\rangle, g[x \leftarrow v]\rangle \end{array}$$

Now we need to specify the operational semantics for the stack machine (see Fig. 4 – Fig. 7). The primitive **createLocal** is defined as follows:

$$\textbf{createLocal}\ s\ n_a\ n_l = \langle s[n_a...], \langle [i \in [0..n_a - 1] \mapsto s[n_a - i - 1]], [i \in [0..n_l - 1] \mapsto 0]\rangle\rangle\}$$

$$P \vdash c \xrightarrow{\varepsilon}_{\mathscr{S}M} c \qquad [\textsc{Stop}_{SM}]$$

$$\frac{P \vdash \langle (x \oplus y)s, s_c, \sigma, \omega\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash \langle yxs, s_c, \sigma, \omega\rangle \xrightarrow{[\texttt{BINOP}\ \otimes]p}_{\mathscr{S}M} c'} \qquad [\textsc{Binop}_{SM}]$$

$$\frac{P \vdash \langle zs, s_c, \sigma, \omega\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash \langle s, s_c, \sigma, \omega\rangle \xrightarrow{[\texttt{CONST}\ z]p}_{\mathscr{S}M} c'} \qquad [\textsc{Const}_{SM}]$$

$$\frac{P \vdash \langle z, \omega'\rangle = \mathbf{read}\ \omega, \langle zs, s_c, \sigma, \omega'\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash \langle s, s_c, \sigma, \omega\rangle \xrightarrow{\texttt{READ}\ p}_{\mathscr{S}M} c'} \qquad [\textsc{Read}_{SM}]$$

$$\frac{P \vdash \langle s, s_c, \sigma, \mathbf{write}\ z\,\omega\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash \langle zs, s_c, \sigma, \omega\rangle \xrightarrow{\texttt{WRITE}\ p}_{\mathscr{S}M} c'} \qquad [\textsc{Write}_{SM}]$$

$$\frac{P \vdash \langle s, s_c, \sigma, \omega\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash \langle xs, s_c, \sigma, \omega\rangle \xrightarrow{[\texttt{DROP}]p}_{\mathscr{S}M} c'} \qquad [\textsc{Drop}_{SM}]$$

Figure 4: Stack machine: basic rules

$$\frac{P \vdash \langle [\sigma(x)]s, s_c, \sigma, \omega\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash \langle s, s_c, \sigma, \omega\rangle \xrightarrow{[\texttt{LD}\ x]p}_{\mathscr{S}M} c'} \qquad [\textsc{LD}_{SM}]$$

$$\frac{P \vdash \langle [\mathbf{ref}\ x]s, s_c, \sigma, \omega\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash \langle s, s_c, \sigma, \omega\rangle \xrightarrow{[\texttt{LDA}\ x]p}_{\mathscr{S}M} c'} \qquad [\textsc{LDA}_{SM}]$$

$$\frac{P \vdash \langle vs, s_c, \sigma[x \leftarrow v], \omega\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash \langle v[\mathbf{ref}\ x]s, s_c, \sigma, \omega\rangle \xrightarrow{[\texttt{STI}]p}_{\mathscr{S}M} c'} \qquad [\textsc{STI}_{SM}]$$

$$\frac{\langle zs, s_c, \sigma[x \leftarrow z], \omega\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{\langle zs, s_c, \sigma, \omega\rangle \xrightarrow{[\texttt{ST}\ x]p}_{\mathscr{S}M} c'} \qquad [\textsc{ST}_{SM}]$$

Figure 5: Stack machine: state operations

$$\frac{P \vdash c \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash c \xrightarrow{[\texttt{LABEL}\ l]p}_{\mathscr{S}M} c'} \quad [\textsc{Label}_{SM}]$$

$$\frac{P \vdash c \xrightarrow{P[l]}_{\mathscr{S}M} c'}{P \vdash c \xrightarrow{[\texttt{JMP}\ l]p}_{\mathscr{S}M} c'} \quad [\textsc{Jmp}_{SM}]$$

$$\frac{z \neq 0, \quad P \vdash \langle s, s_c, \sigma, \omega\rangle \xrightarrow{P[l]}_{\mathscr{S}M} c'}{P \vdash \langle zs, s_c, \sigma, \omega\rangle \xrightarrow{[\texttt{CJMP}_{nz}\ l]p}_{\mathscr{S}M} c'} \quad [\textsc{CJmp}^{+}_{nz\,SM}]$$

$$\frac{z = 0, \quad P \vdash \langle s, s_c, \sigma, \omega\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash \langle zs, s_c, \sigma, \omega\rangle \xrightarrow{[\texttt{CJMP}_{nz}\ l]p}_{\mathscr{S}M} c'} \quad [\textsc{CJmp}^{-}_{nz\,SM}]$$

$$\frac{z = 0, \quad P \vdash \langle s, s_c, \sigma, \omega\rangle \xrightarrow{P[l]}_{\mathscr{S}M} c'}{P \vdash \langle zs, s_c, \sigma, \omega\rangle \xrightarrow{[\texttt{CJMP}_{z}\ l]p}_{\mathscr{S}M} c'} \quad [\textsc{CJmp}^{+}_{z\,SM}]$$

$$\frac{z \neq 0, \quad P \vdash \langle s, s_c, \sigma, \omega\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash \langle zs, s_c, \sigma, \omega\rangle \xrightarrow{[\texttt{CJMP}_{z}\ l]p}_{\mathscr{S}M} c'} \quad [\textsc{CJmp}^{-}_{z\,SM}]$$

Figure 6: Stack machine: control flow instructions

$$P \vdash \langle s, \varepsilon, \sigma, \omega\rangle \xrightarrow{[\texttt{END}]p}_{\mathscr{S}M} \langle s, \varepsilon, \sigma, \omega\rangle \quad [\textsc{EndStop}_{SM}]$$

$$\frac{P \vdash \langle s, s_c, \langle \sigma_l, \sigma\rangle, \omega\rangle \xrightarrow{q}_{\mathscr{S}M} c'}{P \vdash \langle s, \langle \sigma_l, q\rangle s_c, \langle \_, \sigma\rangle, \omega\rangle \xrightarrow{[\texttt{END}]p}_{\mathscr{S}M} c'} \quad [\textsc{End}_{SM}]$$

$$\frac{\langle s', \sigma_l\rangle = \textbf{createLocal}\ s\ n_a\ n_l \quad P \vdash \langle s', s_c, \langle \sigma_l, \sigma\rangle, \omega\rangle \xrightarrow{p}_{\mathscr{S}M} c'}{P \vdash \langle s, s_c, \langle \_, \sigma\rangle, \omega\rangle \xrightarrow{[\texttt{BEGIN}\ \_\ n_a\ n_l]p}_{\mathscr{S}M} c'} \quad [\textsc{Begin}_{SM}]$$

$$\frac{P \vdash \langle s, \langle \sigma_l, p\rangle s_c, \langle \sigma_l, \sigma\rangle, \omega\rangle \xrightarrow{P[f]}_{\mathscr{S}M} c'}{P \vdash \langle s, s_c, \langle \sigma_l, \sigma\rangle, \omega\rangle \xrightarrow{[\texttt{CALL}\ f\ \_]p}_{\mathscr{S}M} c'} \quad [\textsc{Call}_{SM}]$$

Figure 7: Stack machine: functions, call, return