1 Language and semantics

A language is a collection of programs. A program is an abstract syntax tree (AST), which describes the hierarchy of constructs. An abstract syntax of a programming language describes the format of abstract syntax trees of programs in this language. Thus, a language is a set of objects, each of which can be constructively manipulated.

The semantics of a language $\mathscr{L}$ is a total map

$$[\![\bullet]\!]_{\mathscr{L}} : \mathscr{L} \to \mathscr{D}$$

where $\mathscr{D}$ is some semantic domain. The choice of the domain is at our command; for example, for Turing-complete languages $\mathscr{D}$ can be the set of all partially-recursive (computable) functions.

2 Interpreters

In reality, the semantics often is described using interpreters:

$$\mathit{eval} : \mathscr{L} \to \text{Input} \to \text{Output}$$

where Input and Output are sets of (all possible) inputs and outputs for the programs in the language $\mathscr{L}$. We claim *eval* to possess the following property

$$\forall p \in \mathscr{L}, \forall x \in \text{Input} : [\![p]\!]_{\mathscr{L}}\, x = \mathit{eval}\ p\, x$$

In other words, an interpreter takes a program and its input as arguments, and returns what the program would return, being run on that argument. The equality in the definitional property of an interpreter has to be read "if the right hand side is defined, then the left hand side is defined, too, and their values coinside", and vice-versa.

Why interpreters are so important? Because they can be written as programs in a meta-lanaguge, or a language of implementation. For example, if we take $\lambda^a\mathcal{M}^a$ as a language of implementation, then an interpreter of a language $\mathscr{L}$ is some $\lambda^a\mathcal{M}^a$ program *eval*, such that

$$\forall p \in \mathscr{L}, \forall x \in \text{Input} : [\![p]\!]_{\mathscr{L}}\, x = [\![\mathit{eval}]\!]_{\lambda^a\mathcal{M}^a}\ p\, x$$

How to define $[\![\bullet]\!]_{\lambda^a\mathcal{M}^a}$? We can write an interpreter in some other language. Thus, a tower of meta-languages and interpreters comes into consideration. When to stop? When the meta-language is simple enough for intuitive understanding (in reality: some math-based frameworks like operational, denotational or game semantics, etc.)

Pragmatically: if you have a good implementation of a good programming language you trust, you can write interpreters of other languages.

3 Compilers

A compiler is just a language transformer

$$comp : \mathscr{L} \to \mathscr{M}$$

for two languages $\mathscr{L}$ and $\mathscr{M}$; we expect a compiler to be total and to possess the following property:

$$\forall p \in \mathscr{L} \;\; [\![p]\!]_{\mathscr{L}} = [\![comp\ p]\!]_{\mathscr{M}}$$

Again, the equality in this definition is understood functionally. The property itself is called a complete (or full) correctness. In reality compilers are partially correct, which means, that the domain of compiled programs can be wider.

And, again, we expect compilers to be defined in terms of some implementation language. Thus, a compiler is a program (in, say, $\lambda^a\mathcal{M}^a$), such, that its semantics in $\lambda^a\mathcal{M}^a$ possesses the following property (fill the rest yourself).

4 The first example: the language of expressions

Abstract syntax:

$$\begin{array}{rcll} \mathscr{X} & = & \{x, y, z, \dots\} & \text{(variables)} \\ \otimes & = & \{+, -, \times, /, \%, <, \leq, >, \geq, =, \neq, \vee, \wedge\} & \text{(binary operators)} \\ \mathscr{E} & = & \mathscr{X} & \text{(expressions)} \\ & & \mathbb{N} & \\ & & \mathscr{E} \otimes \mathscr{E} & \end{array}$$

Semantics of expressions:

- state $\sigma : \mathscr{X} \to \mathbb{Z}$ assigns values to (some) variables;
- semantics $[\![\bullet]\!]_{\mathscr{E}}$ assigns each to expression a partial map $\Sigma \to \mathbb{Z}$, where Σ is the set of all states.

Empty state Λ: undefined for any variable.
Big-step operational semantics is defined via a ternary relation

$$\Rightarrow_{\mathscr{E}} \subseteq \Sigma \times \mathscr{E} \times \mathbb{Z}$$

An expression "$\sigma \overset{e}{\underset{\mathscr{E}}{\Rightarrow}} n$" is informally interpreted as "an evaluation of an expression e in a state σ delivers a value n".

$$\frac{n \in \mathbb{N}}{\sigma \underset{\mathscr{E}}{\overset{n}{\Longrightarrow}} n} \quad [\text{Const}]$$

$$\frac{x \in \mathscr{X}}{\sigma \underset{\mathscr{E}}{\overset{x}{\Longrightarrow}} \sigma\, x} \quad [\text{Var}]$$

$$\frac{\sigma \underset{\mathscr{E}}{\overset{l}{\Longrightarrow}} x,\ \sigma \underset{\mathscr{E}}{\overset{r}{\Longrightarrow}} y}{\sigma \underset{\mathscr{E}}{\overset{l \otimes r}{\Longrightarrow}} x \oplus y} \quad [\text{Binop}]$$

$\otimes$	$\oplus$ in $\lambda^a \mathcal{M}^a$
$+$	$+$
$-$	$-$
$\times$	$*$
$/$	$/$
$\%$	$\%$
$<$	$<$
$>$	$>$
$\leq$	$<=$
$\geq$	$>=$
$=$	$=$
$\neq$	$!=$
$\wedge$	$\&\&$
$\vee$	$!!$

Important observations:

1. "$\Rightarrow_{\mathscr{E}}$" is defined compositionally: the meaning of an expression is defined in terms of meanings of its proper subexpressions.
2. "$\Rightarrow_{\mathscr{E}}$" is total, since it takes into account all possible ways to deconstruct any expression.
3. "$\Rightarrow_{\mathscr{E}}$" is deterministic: there is no way to assign different meanings to the same expression, since we deconstruct each expression unambiguously.
4. "$\otimes$" is an element of language syntax, while "$\oplus$" is its interpretation in the meta-language of semanic description (simpler: in the language of interpreter implementation).
5. This concrete semantics is strict: for a binary operator both its arguments are evaluated unconditionally; thus, for example, "$1 \vee x$" is undefined in empty state.

Having "$\Rightarrow_{\mathscr{E}}$", the "$[\![\bullet]\!]_{\mathscr{E}}$" can be defined in obvious way:

$$\frac{\sigma \underset{\mathscr{E}}{\overset{e}{\Longrightarrow}} n}{[\![e]\!]_{\mathscr{E}} \sigma = n}$$

1 Statements

More interesting language — a language of simple statements:

$$\begin{array}{rcl} \mathscr{S} & = & \text{skip} \\ & & \mathscr{X} := \mathscr{E} \\ & & \text{read}\ (\mathscr{X}) \\ & & \text{write}\ (\mathscr{E}) \\ & & \mathscr{S}_1; \mathscr{S}_2 \end{array}$$

Here $\mathscr{E}, \mathscr{X}$ stand for the sets of expressions and variables, as in the previous lecture. Informally, the language allows to write a striaght-line programs which transform an input stream of integers into output stream of integers.

Again, we define the semantics for this language

$$[\![\bullet]\!]_{\mathscr{S}} : \mathscr{S} \mapsto \mathbb{Z}^* \to \mathbb{Z}^*$$

with the semantic domain of partial functions from integer streams to integer streams. Again, we will use big-step operational semantics: we define a ternary relation "$\Rightarrow_{\mathscr{S}}$"

$$\Rightarrow_{\mathscr{S}} \subseteq \mathscr{C} \times \mathscr{S} \times \mathscr{C}$$

where $\mathscr{C}$ — a set of possible configurations during a program execution. We will write $c_1 \overset{S}{\underset{\mathscr{S}}{\Rightarrow}} c_2$ instead of $(c_1, S, c_2) \in \Rightarrow_{\mathscr{S}}$ and informally interpret the former as "the execution of a statement S in a configuration c_1 completes with the configuration c_2".

The set of all configuration is defined as

$$\begin{array}{rcl} \mathscr{C} & = & \Sigma \times \mathscr{W} \\ \mathscr{W} & = & \mathbb{Z}^* \times \mathbb{Z}^* \end{array}$$

where $\mathscr{W}$ — a set of worlds, each of which escapsulates some input-output stream pair. For simplicity, we define the following operations for worlds:

$$\begin{array}{rcl} \textbf{read}\ \langle xi, o\rangle & = & \langle x, \langle i, o\rangle\rangle \\ \textbf{write}\ x\ \langle i, o\rangle & = & \langle i, ox\rangle \\ \textbf{out}\ \langle i, o\rangle & = & o \end{array}$$

The relation "$\Rightarrow_{\mathscr{S}}$" is defined by the following deductive system (see Fig. 1). The first three rules are axioms as they do not have any premises. Note, according to these rules sometimes a program cannot do a step in a given configuration: a value of an expression can be undefined in a given state in rules Assign and Write, and there can be no input value in rule Read. This style of a semantics description is called big-step operational semantics, since the results of a computation are immediately observable at the right hand side of "$\Rightarrow$" and, thus,

$$c \xrightarrow[\mathscr{S}]{\text{skip}} c \qquad [\text{Skip}]$$

$$\langle \sigma, \omega \rangle \xrightarrow[\mathscr{S}]{\text{x} := e} \langle \sigma[x \leftarrow [\![e]\!]_{\mathscr{E}}\, \sigma], \omega \rangle \qquad [\text{Assign}]$$

$$\frac{\langle z, \omega' \rangle = \textbf{read}\ \omega}{\langle \sigma, \omega \rangle \xrightarrow[\mathscr{S}]{\text{read}\ \ (x)} \langle \sigma[x \leftarrow z], \omega' \rangle} \qquad [\text{Read}]$$

$$\langle \sigma, \omega \rangle \xrightarrow[\mathscr{S}]{\text{write}\ \ (e)} \langle \sigma, \textbf{write}\ ([\![e]\!]_{\mathscr{E}}\, \sigma)\, \omega \rangle \qquad [\text{Write}]$$

$$\frac{c_1 \xrightarrow[\mathscr{S}]{S_1} c' \qquad c' \xrightarrow[\mathscr{S}]{S_2} c_2}{c_1 \xrightarrow[\mathscr{S}]{S_1 ; S_2} c_2} \qquad [\text{Seq}]$$

Figure 1: Big-step operational semantics for statements

the computation is performed in a single "big" step. And, again, this style of a semantic description can be used to easily implement a reference interpreter.

With the relation "$\Rightarrow_{\mathscr{S}}$" defined we can abbreviate the "surface" semantics for the language of statements:

$$\frac{\langle \Lambda, \langle i, \varepsilon \rangle \rangle \xrightarrow[\mathscr{S}]{S} \langle \sigma, \omega \rangle}{[\![S]\!]_{\mathscr{S}} i = \textbf{out}\ \omega}$$

2 Stack Machine

Stack machine is a simple abstract computational device, which can be used as a convenient model to constructively describe the compilation process.

In short, stack machine operates on the same configurations, as the language of statements, plus a stack of integers. The computation, performed by the stack machine, is controlled by a program, which is described as follows:

$$\begin{array}{rcl} \mathscr{I} & = & \text{BINOP} \otimes \\ & & \text{CONST}\ \mathbb{N} \\ & & \text{READ} \\ & & \text{WRITE} \\ & & \text{LD}\ \mathscr{X} \\ & & \text{ST}\ \mathscr{X} \\ \mathscr{P} & = & \varepsilon \\ & & \mathscr{I}\,\mathscr{P} \end{array}$$

Here the syntax category $\mathscr{I}$ stands for instructions, $\mathscr{P}$ — for programs; thus, a program is a finite string of instructions.

The semantics of stack machine program can be described, again, in the form of big-step operational semantics. This time the set of stack machine configurations is

$$\mathscr{C}_{SM} = \mathbb{Z}^* \times \mathscr{C}$$

where the first component is a stack, and the second — a configuration as in the semantics of statement language. The rules are shown on Fig. 2; note, now we have one axiom and six inference rules (one per instruction).

As for the statement, with the aid of the relation "$\Rightarrow_{\mathscr{SM}}$" we can define the surface semantics of stack machine:

$$\frac{\langle \varepsilon, \langle \Lambda, \langle i, \varepsilon \rangle \rangle \rangle \overset{p}{\Longrightarrow}_{\mathscr{SM}} \langle s, \langle \sigma, \omega \rangle \rangle}{[\![p]\!]_{\mathscr{SM}}\, i = \mathbf{out}\ \omega}$$

3 A Compiler for the Stack Machine

A compiler of the statement language into the stack machine is a total mapping

$$[\![\bullet]\!]^{comp} : \mathscr{S} \mapsto \mathscr{P}$$

We can describe the compiler in the form of denotational semantics for the source language. In fact, we can treat the compiler as a static semantics, which maps each program into its stack machine equivalent.

As the source language consists of two syntactic categories (expressions and statments), the compiler has to be "bootstrapped" from the compiler for expressions $[\![\bullet]\!]^{comp}_{\mathscr{E}}$:

$$\begin{aligned}
[\![x]\!]^{comp}_{\mathscr{E}} &= [\mathrm{LD}\ x] \\
[\![n]\!]^{comp}_{\mathscr{E}} &= [\mathrm{CONST}\ n] \\
[\![A \otimes B]\!]^{comp}_{\mathscr{E}} &= [\![A]\!]^{comp}_{\mathscr{E}} [\![B]\!]^{comp}_{\mathscr{E}} [\mathrm{BINOP}\ \otimes]
\end{aligned}$$

And now the main dish:

$$\begin{aligned}
[\![x{:=}e]\!]^{comp} &= [\![e]\!]^{comp}_{\mathscr{E}}\, [\mathrm{ST}\ x] \\
[\![\mathrm{read}\ \ (x)]\!]^{comp} &= [\mathrm{READ}][\mathrm{ST}\ x] \\
[\![\mathrm{write}\ \ (e)]\!]^{comp} &= [\![e]\!]^{comp}_{\mathscr{E}}\, [\mathrm{WRITE}] \\
[\![S_1; S_2]\!]^{comp} &= [\![S_1]\!]^{comp} [\![S_2]\!]^{comp}
\end{aligned}$$

$$c \xrightarrow[\mathscr{S}\,\mathscr{M}]{\varepsilon} c \qquad [\mathrm{Stop}_{SM}]$$

$$\frac{\langle (x \oplus y)s, c\rangle \xrightarrow[\mathscr{S}\,\mathscr{M}]{p} c'}{\langle yxs, c\rangle \xrightarrow[\mathscr{S}\,\mathscr{M}]{[\mathrm{BINOP}\otimes]p} c'} \qquad [\mathrm{Binop}_{SM}]$$

$$\frac{\langle zs, c\rangle \xrightarrow[\mathscr{S}\,\mathscr{M}]{p} c'}{\langle s, c\rangle \xrightarrow[\mathscr{S}\,\mathscr{M}]{[\mathrm{CONST}\,z]p} c'} \qquad [\mathrm{Const}_{SM}]$$

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

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

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

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

Figure 2: Big-step operational semantics for stack machine

1 Structural Control Flow

We add a few structural control flow constructs to the language:

$$\begin{aligned} \mathscr{S} \quad += \quad & \text{if } \mathscr{E} \text{ then } \mathscr{S} \text{ else } \mathscr{S} \\ & \text{while } \mathscr{E} \text{ do } \mathscr{S} \end{aligned}$$

The big-step operational semantics is straightforward and is shown on Fig. 1.

In the concrete syntax for the constructs we add the closing keywords " fi " and "od" as follows:

$$\begin{aligned} & \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi} \\ & \text{while } e \text{ do } s \text{ od} \end{aligned}$$

$$\frac{\sigma \underset{\mathscr{E}}{\Longrightarrow} n \neq 0 \qquad \langle \sigma, w \rangle \underset{\mathscr{S}}{\overset{S_1}{\Longrightarrow}} c'}{\langle \sigma, w \rangle \underset{\mathscr{S}}{\overset{\text{if } e \text{ then } S_1 \text{ else } S_2}{\Longrightarrow}} c'} \quad [\text{If-True}]$$

$$\frac{\sigma \underset{\mathscr{E}}{\overset{e}{\Longrightarrow}} 0 \qquad \langle \sigma, w \rangle \underset{\mathscr{S}}{\overset{S_2}{\Longrightarrow}} c'}{\langle \sigma, w \rangle \underset{\mathscr{S}}{\overset{\text{if } e \text{ then } S_1 \text{ else } S_2}{\Longrightarrow}} c'} \quad [\text{If-False}]$$

$$\frac{\sigma \underset{\mathscr{E}}{\overset{e}{\Longrightarrow}} n \neq 0 \qquad \langle \sigma, w \rangle \underset{\mathscr{S}}{\overset{S}{\Longrightarrow}} c' \qquad c' \underset{\mathscr{S}}{\overset{\text{while } e \text{ do } S}{\Longrightarrow}} c''}{\langle \sigma, w \rangle \underset{\mathscr{S}}{\overset{\text{while } e \text{ do } S}{\Longrightarrow}} c''} \quad [\text{While-True}]$$

$$\frac{\sigma \underset{\mathscr{E}}{\overset{e}{\Longrightarrow}} 0}{\langle \sigma, w \rangle \underset{\mathscr{S}}{\overset{\text{while } e \text{ do } S}{\Longrightarrow}} \langle \sigma, w \rangle} \quad [\text{While-False}]$$

Figure 1: Big-step operational semantics for control flow statements

2 Syntax Extensions

With the structural control flow constructs already implemented, it is rather simple to "saturate" the language with more elaborated control constructs, using the method of syntactic extension. Namely, we may introduce the following constructs

```
  if e1 then s1
elif e2 then s2
...
elif ek then sk
[ else sk+1 ]
  fi
```

and

```
for s1, e, s2 do s3 od
```

only at the syntactic level, directly parsing these constructs into the original abstract syntax tree, using the following conversions:

```
  if e1 then s1                    if e1 then s1
elif e2 then s2                  else if e2 then s2
...                                ...
elif ek then sk        ⇝         else if ek then sk
else sk+1                        else sk+1
fi                                    fi
                                      ...
                                      fi
```

```
  if e1 then s1                    if e1 then s1
elif e2 then s2                  else if e2 then s2
...                                ...
elif ek then sk        ⇝         else if ek then sk
fi                               else skip
                                      fi
                                      ...
                                      fi
```

```
for s1, e, s2 do s3 od     ⇝     s1;
                                 while e do
                                   s3;
                                   s2
                                 od
```

The advantage of syntax extension method is that it makes it possible to add certain constructs with almost zero cost — indeed, no steps have to be made in order to implement the extended constructs (besides parsing). Note, the semantics of extended constructs is provided for free as well (which is not always desirable). Another potential problem with syntax extensions is that

they can easily provide unreasonable results. For example, one may be tempted to implement a post-condition loop using syntax extension:

```
repeat s until e   ⇝   s;
                       while e == 0 do
                         s
                       od
```

However, for nested repeat constructs the size of extended program is exponential w.r.t. the nesting depth, which makes the whole idea unreasonable.

1 Extended Stack Machine

In order to compile a language with structural control flow constructs into a program for the stack machine the latter has to be extended. First, we introduce a set of label names

$$\mathscr{L} = \{l_1, l_2, \dots\}$$

Then, we add three extra control flow instructions:

$$\begin{array}{rcl} \mathscr{I} & += & \text{LABEL}\,\mathscr{L} \\ & & \text{JMP}\,\mathscr{L} \\ & & \text{CJMP}_x\,\mathscr{L}, \text{ where } x \in \{\text{nz}, \text{z}\} \end{array}$$

In order to give the semantics to these instructions, we need to extend the syntactic form of rules, used in the description of big-step operational smeantics. Instead of the rules in the form

$$\frac{c \xRightarrow[\mathscr{S}\mathscr{M}]{p} c'}{c' \xRightarrow[\mathscr{S}\mathscr{M}]{p'} c''}$$

we use the following form

$$\frac{\Gamma' \vdash c \xRightarrow[\mathscr{S}\mathscr{M}]{p} c'}{\Gamma \vdash c' \xRightarrow[\mathscr{S}\mathscr{M}]{p'} c''}$$

where Γ, Γ' — environments. The structure of environments can be different in different cases; for now environment is just a program. Informally, the semantics of control flow instructions can not be described in terms of just a current instruction and current configuration — we need to take the whole program into account. Thus, the enclosing program is used as an environment.

Additionally, for a program P and a label l we define a subprogram $P[l]$, such that P is uniquely represented as $p'[\text{LABEL}\,l]P[l]$. In other words $P[l]$ is a unique suffix of P, immediately following the label l (if there are multiple (or no) occurrences of label l in P, then $P[l]$ is undefined).

All existing rules have to be rewritten — we need to formally add a $P \vdash \dots$ part everywhere. For the new instructions the rules are given on Fig. 1.

Finally, the top-level semantics for the extended stack machine can be redefined as follows:

$$\frac{p \vdash \langle \varepsilon, \langle \Lambda, \langle i, \varepsilon \rangle \rangle \rangle \xRightarrow[\mathscr{S}\mathscr{M}]{p} \langle s, \langle \sigma, \omega \rangle \rangle}{[\![p]\!]_{\mathscr{S}\mathscr{M}}\, i = \textbf{out}\ \omega}$$

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

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

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

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

$$\frac{z = 0, \quad P \vdash \langle s, \theta \rangle \xrightarrow[\mathscr{S}\mathscr{M}]{P[l]} c'}{P \vdash \langle zs, \theta \rangle \xrightarrow[\mathscr{S}\mathscr{M}]{[\text{CJMP}_{z}\ l]p} c'} \quad [\text{CJMP}^{+}_{z\,SM}]$$

$$\frac{z \neq 0, \quad P \vdash \langle s, \theta \rangle \xrightarrow[\mathscr{S}\mathscr{M}]{p} c'}{P \vdash \langle zs, \theta \rangle \xrightarrow[\mathscr{S}\mathscr{M}]{[\text{CJMP}_{z}\ l]p} c'} \quad [\text{CJMP}^{-}_{z\,SM}]$$

Figure 1: Big-step operational semantics for extended stack machine

2 A Compiler for the Stack Machine

A compiler for the language with structural control flow into the stack machine can be given in the form of static semantics. Similarly to the big-step operational semantics, the compiler also operates on environment. For now, the environment allows us to generate fresh labels. Thus, a compiler specification for statements has the shape

$$[\![p]\!]^{comp}_{\mathscr{S}}\,\Gamma = \langle c, \Gamma' \rangle$$

where p is a source program, Γ, Γ' — some environments, c — generated program for the stack machine. As we can see, the environment changes during the code generation, hence auxilliary semantic primitive $[\![\bullet]\!]^{comp}_{\mathscr{S}}$. We need one primitive to operate on environments which allocates a number of fresh labels and returns a new environment:

labels Γ

The number of labels allocated is determined by context.

We give an example of compiler specification rule for the while-loop:

$$\frac{\langle l_e, l_s, \Gamma' \rangle = \textbf{labels}\ \Gamma, \quad [\![s]\!]^{comp}_{\mathscr{S}}\,\Gamma' = \langle c_s, \Gamma'' \rangle}{\begin{array}{lcl} [\![\text{while } e \text{ do } s \text{ od}]\!]^{comp}_{\mathscr{S}}\,\Gamma & = & \langle \quad \text{JMP } l_e \\ & & \quad \text{LABEL } l_s \\ & & \quad c_s \\ & & \quad \text{LABEL } l_e \\ & & \quad [\![e]\!]^{comp}_{\mathscr{E}} \\ & & \quad \text{CJMP}_{nz}\ l_s, \quad \Gamma'' \ \rangle \end{array}}$$

Note, the compiler for expressions is not changed and completely reused.

Finally, the top-level compiler for the whole program can be defined as follows:

$$\frac{[\![p]\!]^{comp}_{\mathscr{S}}\,\Gamma_0 = \langle c, \_ \rangle}{[\![p]\!]^{comp} = c}$$

where Γ_0 — empty environment.

1 Contorol Flow Expressions

The separation of control flow constructs into a distinctive category of statements is the easiest way to introduce them into a language. However, this to some extent sacrifices expressivity of the language for the ease of implementation: we can write

```
if c then x := 1 else x := 2 fi
```

but we cannot write

```
x := if c then 1 else 2 fi
```

et cetera. This lack of the expressivity is compensated in some languages by extending the varirty of expressions (for example, with ternary conditional expression in C/C++, etc.)

Meanwhile implementing a "rich" assortment of expressions, including those for control flow constructs, is not a big deal. As a result the language becomes not only more expressive, but also more scalable as adding new features becomes easier.

First, we join all constructs from statement category into expressions:

$$
\begin{array}{rcl}
\mathscr{E} & = & \mathscr{X} \\
 & & \mathbb{N} \\
 & & \mathscr{E} \otimes \mathscr{E} \\
 & & \text{skip} \\
 & & \mathscr{E} := \mathscr{E} \\
 & & \text{read } (\mathscr{X}) \\
 & & \text{write } (\mathscr{E}) \\
 & & \mathscr{E}\,;\mathscr{E} \\
 & & \text{if } \mathscr{E} \text{ then } \mathscr{E} \text{ else } \mathscr{E} \\
 & & \text{while } \mathscr{E} \text{ do } \mathscr{E} \\
 & & \text{repeat } \mathscr{E} \text{ until } \mathscr{E} \\
 & & \text{ref } \mathscr{X} \\
 & & \text{ignore } \mathscr{E}
\end{array}
$$

The majority of of definitions left intact (but all statements are replaced into expressions); we added introduced two additional constructs (ignore / ref). These constructs will not have representation in concrete syntax; instead, they will be inferred.

The motivation for introducing these two additional constructs is as follows. With given abstract syntax we can, in theory, represent programs like

```
x + y := 3
```

or

```
x := while c do read (x) od
```

Both these examples are ill-formed: "x + y" does not designate a mutable reference to assign to, and while-loop does not evaluate any value to assign. Additionally, in the following example

$$\mathbf{Ref} \vdash \text{ref } x \qquad \mathbf{Val} \vdash x \qquad \mathbf{Void} \vdash \text{ignore } x \qquad x \in \mathscr{X}$$

$$\mathbf{Val} \vdash z \qquad \mathbf{Void} \vdash \text{ignore } z \qquad z \in \mathbb{N}$$

$$\frac{\mathbf{Val} \vdash l, \quad \mathbf{Val} \vdash r}{\mathbf{Val} \vdash l \oplus r} \qquad \frac{\mathbf{Val} \vdash l, \quad \mathbf{Val} \vdash r}{\mathbf{Void} \vdash \text{ignore } l \oplus r}$$

$$\mathbf{Void} \vdash \text{skip}$$

$$\frac{\mathbf{Ref} \vdash l, \quad \mathbf{Val} \vdash r}{\mathbf{Val} \vdash l := r} \qquad \frac{\mathbf{Ref} \vdash l, \quad \mathbf{Val} \vdash r}{\mathbf{Void} \vdash \text{ignore } (l := r)}$$

$$\mathbf{Void} \vdash \text{read } (x)$$

$$\frac{\mathbf{Val} \vdash e}{\mathbf{Void} \vdash \text{write } (e)}$$

$$\frac{\mathbf{Void} \vdash s_1, \quad a \vdash s_2}{a \vdash s_1 \,;\, s_2} \qquad \frac{\mathbf{Val} \vdash e, \quad a \vdash s_1, \quad a \vdash s_2}{a \vdash \text{if } e \text{ then } s_1 \text{ else } s_2} \qquad \frac{\mathbf{Val} \vdash e, \quad \mathbf{Void} \vdash s}{\mathbf{Void} \vdash \text{while } e \text{ do } s}$$

$$\frac{\mathbf{Val} \vdash e, \quad \mathbf{Void} \vdash s}{\mathbf{Void} \vdash \text{repeat } s \text{ until } e}$$

Figure 1: Well-formed Expressions

```
x := x
```

the syntactic role of "x" in left and right side of assignment is different: while the left-side designates a reference to assign to, the right-side designates a dereferenced value.

We can rule out such ill-formed expressions by a mean of their static semantics (see Fig. 1). At the same time this static semantics can be used to infer the placements of "ignore" and "ref" constructs (provided that these placements exist). For example, for an incomplete abstract syntax

```
x := y
```

the following complete well-formed AST can be inferred:

```
ref (x) := y
```

et cetera.

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

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

and from now on we will consider only well-formed programs.

1.1 Concrete syntax

We also need to stipulate the details of concrete syntax:

- we treat assignment (:=) and sequencing (;) as right-associative binary operators;
- the precedence level of assignment is less than that for "!!", and the precedence level of sequencing is less than that for assignment;
- in the repeat-until construct "until" has a higher precedence than ";".

These rules can be demonstrated by the following examples:

syntactic form	meaning
x := y := 3	x := (y := 3)
x := y; y := z	(x := y); (y := z)
repeat read (x) until x != 0; write (x)	(repeat read (x) until x != 0); write (x))

1.2 Semantics

The semantics for the language is given in the form of big-step operational semantics. Note, since we have only one syntactic category in principle we do not need different type of arrows; we however define an additional arrow "$\Rightarrow_*$" to denote the semantics of a list of expressions evaluated one after another.

Another observation concerns the type of arrows. In previous case we had two arrows (one for expressions and another for statements) of different types. Now, however, we have a single syntactic category, which means that constructs which used to be expressions now have side effects, and constructs used to be statements now have values. Having said that, we denote the arrows as the following relations:

$$\begin{array}{rcl} \Rightarrow & \subseteq & \mathscr{C} \times \mathscr{E} \times (\mathscr{C} \times \mathscr{V}) \\ \Rightarrow^* & \subseteq & \mathscr{C} \times \mathscr{E}^* \times (\mathscr{C} \times \mathscr{V}^*) \end{array}$$

where $\mathscr{V}$ is the set of values:

$$\mathscr{V} = \mathbb{Z} \mid \textbf{ref}\ \mathscr{X} \mid \bot$$

Note, we need to extend the values from plain integer numbers, adding a "no-value" ($\bot$) and a reference to a variable (**ref**).

First we define an auxilliary relation "$\Rightarrow_*$":

The semantics for expressions is presented on Fig. 2.

$$c \overset{\varepsilon}{\Longrightarrow}_{*} \langle c, \varepsilon \rangle \qquad [\mathrm{Expr}^{*}_{\varepsilon}]$$

$$\frac{c \overset{e}{\Longrightarrow} \langle c', v \rangle, \quad c' \overset{\omega}{\Longrightarrow}_{*} \langle c'', \psi \rangle}{c \overset{e\omega}{\Longrightarrow}_{*} \langle c'', v\psi \rangle} \qquad [\mathrm{Expr}^{*}]$$

$$c \xrightarrow{z} \langle c, z \rangle \qquad [\text{Const}]$$

$$\langle \sigma, \omega \rangle \xrightarrow{x} \langle \langle \sigma, \omega \rangle, \sigma x \rangle \qquad [\text{Var}]$$

$$c \xrightarrow{\text{ref } x} \langle c, \mathbf{ref}\, x \rangle \qquad [\text{Ref}]$$

$$\frac{c \xrightarrow{lr}{}^{*} \langle c', wv \rangle}{c \xrightarrow{l \oplus r} \langle c', w \otimes v \rangle} \qquad [\text{Binop}]$$

$$c \xrightarrow{\text{skip}} \langle c, \bot \rangle \qquad [\text{Skip}]$$

$$\frac{c \xrightarrow{lr}{}^{*} \langle \langle \sigma, \omega \rangle, [\mathbf{ref}\, x][v] \rangle}{c \xrightarrow{l := r} \langle \langle \sigma[x \leftarrow v], \omega \rangle, v \rangle} \qquad [\text{Assign}]$$

$$\frac{\langle z, \omega' \rangle = \mathbf{read}\ \omega}{\langle \sigma, \omega \rangle \xrightarrow{\text{read } (x)} \langle \langle \sigma[x \leftarrow z], \omega' \rangle, \bot \rangle} \qquad [\text{Read}]$$

$$\frac{\langle \sigma, \omega \rangle \xrightarrow{e} \langle \langle \sigma', \omega' \rangle, v \rangle}{\langle \sigma, \omega \rangle \xrightarrow{\text{write } (e)} \langle \langle \sigma', \mathbf{write}\, v\, \omega' \rangle, \bot \rangle} \qquad [\text{Write}]$$

$$\frac{c_1 \xrightarrow{S_1} \langle c', v \rangle, \quad c' \xrightarrow{S_2} c_2}{c_1 \xrightarrow{S_1 ; S_2} c_2} \qquad [\text{Seq}]$$

$$\frac{c \xrightarrow{e} \langle c', n \rangle, \quad n \neq 0, \quad c' \xrightarrow{S_1} c''}{c \xrightarrow{\text{if } e \text{ then } S_1 \text{ else } S_2} c''} \qquad [\text{If-True}]$$

$$\frac{c \xrightarrow{e} \langle c', 0 \rangle, \quad c' \xrightarrow{S_2} c''}{c \xrightarrow{\text{if } e \text{ then } S_1 \text{ else } S_2} c''} \qquad [\text{If-False}]$$

$$\frac{c \xrightarrow{e} \langle c', n \rangle, \quad n \neq 0, \quad c' \xrightarrow{S} \langle c'', v \rangle, \quad c'' \xrightarrow{\text{while } e \text{ do } S} c'''}{c \xrightarrow{\text{while } e \text{ do } S} c'''} \qquad [\text{While-True}]$$

$$\frac{c \xrightarrow{e} \langle c', 0 \rangle}{c \xrightarrow{\text{while } e \text{ do } S} \langle c', \bot \rangle} \qquad [\text{While-False}]$$

$$\frac{c \xrightarrow{S\,;\ \text{while } e == 0 \text{ do } S} c'}{c \xrightarrow{\text{repeat } S \text{ until } e} c'} \qquad [\text{Repeat}]$$

Figure 2: Big-step Operational Semantics for Expressions

1.3 Stack Machine

Surprisingly (or, rather unsurprisingly) the stack machine has to be improved only a little bit. Namely, we add the following instructions:

$\mathscr{I}$	$+=$	LDA $\mathscr{X}$	loading an address of a variable
		STI	storing by indirect address
		DROP	discard the top of the stack

The form of operational semantics is left unchanged; the semantics of additional instructions is as follows:

$$\frac{P \vdash \langle [\mathbf{ref}\, x]s, \theta\rangle \xrightarrow[\mathscr{S}\mathscr{M}]{p} c'}{P \vdash \langle s, \theta\rangle \xrightarrow[\mathscr{S}\mathscr{M}]{[\mathrm{LDA}\, x]p} c'} \quad [\mathrm{LDA}_{SM}]$$

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

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

$$\frac{P \vdash \langle s, \theta\rangle \xrightarrow[\mathscr{S}\mathscr{M}]{p} c'}{P \vdash \langle xs, \theta\rangle \xrightarrow[\mathscr{S}\mathscr{M}]{[\mathrm{DROP}]p} c'} \quad [\mathrm{DROP}_{SM}]$$

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).

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

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

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

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

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

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:

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

1 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:

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

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

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

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

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

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

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

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

Figure 1: Stack machine: basic rules

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

Now we need to specify the operational semantics for the stack machine (see Fig. 1 – Fig. 4). The primitive **createLocal** is defined as follows:

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

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

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

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

$$\frac{\langle zs, s_c, \sigma[x \leftarrow z], \omega\rangle \xRightarrow{p}_{\mathscr{S}M} c'}{\langle zs, s_c, \sigma, \omega\rangle \xRightarrow{[\mathrm{ST}\,x]p}_{\mathscr{S}M} c'} \quad [\mathrm{ST}_{SM}]$$

Figure 2: Stack machine: state operations

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

$$\frac{P \vdash c \xrightarrow[\mathscr{SM}]{P[l]} c'}{P \vdash c \xrightarrow[\mathscr{SM}]{[\text{JMP } l]p} c'} \quad [\text{JMP}_{SM}]$$

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

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

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

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

Figure 3: Stack machine: control flow instructions

$$P \vdash \langle s, \varepsilon, \sigma, \omega \rangle \xRightarrow[\mathscr{S}M]{[\mathrm{END}]p} \langle s, \varepsilon, \sigma, \omega \rangle \qquad [\mathrm{EndStop}_{SM}]$$

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

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

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

Figure 4: Stack machine: functions, call, return

1 Arrays

Arrays are added at the expression level by means of the following extension:

$$\begin{array}{lll} \mathscr{E} & += & [\mathscr{E}^*] \\ & & \mathscr{E}\,[\mathscr{E}] \\ & & \mathscr{E}\ .\ \text{length} \\ & & \text{elemRef}\,\mathscr{E}\,[\mathscr{E}] \end{array}$$

Additional well-formedness rules for the new constructs:

$$\frac{\mathbf{Val} \vdash e_1 \ldots \mathbf{Val} \vdash e_k}{\mathbf{Val} \vdash [e_1, \ldots, e_k]} \qquad \frac{\mathbf{Val} \vdash e_1 \ldots \mathbf{Val} \vdash e_k}{\mathbf{Weak} \vdash [e_1, \ldots, e_k]} \qquad \frac{\mathbf{Val} \vdash e_1 \ldots \mathbf{Val} \vdash e_k}{\mathbf{Void} \vdash \text{ignore}\ \ [e_1, \ldots, e_k]}$$

$$\frac{\mathbf{Val} \vdash e \quad \mathbf{Val} \vdash i}{\mathbf{Val} \vdash e\,[\,i\,]} \qquad \frac{\mathbf{Val} \vdash e \quad \mathbf{Val} \vdash i}{\mathbf{Weak} \vdash e\,[\,i\,]} \qquad \frac{\mathbf{Val} \vdash e \quad \mathbf{Val} \vdash i}{\mathbf{Void} \vdash \text{ignore}\ e\,[\,i\,]}$$

$$\frac{\mathbf{Val} \vdash e \quad \mathbf{Val} \vdash i}{\mathbf{Ref} \vdash \text{elemRef}\,e\,[\,i\,]}$$

2 Operational Semantics

An array can be represented as a pair: the length of the array and a mapping from indices to elements. If we denote X the set of elements then the set of all arrays $\mathscr{A}(\mathscr{X})$ can be defined as follows:

$$\mathscr{A}(X) = \mathbb{N} \times (\mathbb{N} \to X)$$

For an array $\langle n, f \rangle$ we assume $\texttt{dom}\, f = [0\,..\,n-1]$. An element selection function:

$$\bullet[\bullet] : \mathscr{A}(X) \to \mathbb{N} \to X$$

$$\langle n, f \rangle\,[i] = \begin{cases} f(i) & , \quad i < n \\ \bot & , \quad \text{otherwise} \end{cases}$$

We represent arrays by references. Thus, we introduce a (linearly) ordered set of locations

$$\mathscr{L} = \{l_0, l_1, \ldots\}$$

Now, the set of all values the programs operate on can be described as follows:

$$\mathscr{V} = \mathbb{Z} \mid \mathscr{L}$$

To access arrays, we introduce an abstraction of memory:

$$\mathscr{M} = \mathscr{L} \to \mathscr{A}\,(\mathscr{V})$$

We now add two more components to the configurations: a memory function μ and the first free memory location l_m, and define the following primitive

$$\mathbf{mem}\ \langle \sigma, \omega, \mu, l_m \rangle = \mu$$

which gives a memory function from a configuration.

The definition of state does not change, hence all existing rules are preserved (modulo adding additinal components to configurations) The rules for the new kinds of expressions are as follows:

$$\frac{c \xrightarrow[\mathscr{E}^*]{e_0, \ldots, e_k} \langle \langle \sigma, \omega, \mu, l \rangle, v_1, \ldots, v_k \rangle}{c \xrightarrow[\mathscr{E}]{[e_0, \ldots, e_k]} \langle \langle \sigma, \omega, \mu[l \leftarrow \langle k+1, i \mapsto v_i \rangle], l+1 \rangle, l \rangle} \quad [\text{Array}]$$

$$\frac{c \xrightarrow[\mathscr{E}^*]{ei} \langle c', lv \rangle \quad l \in \mathscr{L} \quad v \in \mathbb{Z}}{c \xrightarrow[\mathscr{E}]{e\,[\,i\,]} \langle c', ((\mathbf{mem}\ c')(l))[i] \rangle} \quad [\text{ArrayElem}]$$

$$\frac{c \xrightarrow[\mathscr{E}^*]{ei} \langle c', lv \rangle \quad l \in \mathscr{L} \quad v \in \mathbb{Z}}{c \xrightarrow[\mathscr{E}]{\text{elemRef}\, e\,[\,i\,]} \langle c', \mathbf{elemRef}\ l\ v \rangle} \quad [\text{ArrayElemRef}]$$

$$\frac{c \xrightarrow[\mathscr{E}]{e} \langle c', l \rangle \quad l \in \mathscr{L}}{c \xrightarrow[\mathscr{E}]{e\ .\ \text{length}} \langle c', \mathbf{fst}\ (\mathbf{mem}\ c')(l) \rangle} \quad [\text{ArrayLength}]$$

We also need one additional rule for assignment:

$$\frac{c \xrightarrow[\mathscr{E}^*]{lr} \langle \langle \sigma, \omega, \mu, l \rangle, [\mathbf{elemRef}\ a\ i]v \rangle \quad a \in \mathscr{L}}{c \xrightarrow[\mathscr{E}]{l := r} \langle \langle \sigma, \omega, \mu[a \leftarrow \langle \mathbf{fst}\, \mu(a), (\mathbf{snd}\, \mu(a))[i \leftarrow v] \rangle], l \rangle, v \rangle} \quad [\text{AssignArray}]$$

3 Stack Machine

In stack machine we add the following new instructions:

$$\begin{array}{lll} \mathscr{I} & += & \text{ARRAY}\ \mathbb{N} \\ & & \text{ELEM} \\ & & \text{STA} \end{array}$$

We also add memory function and current location components to the configuration; as state components are preserved, all rules are preserved as well. The new rules are:

$$\frac{P \vdash \langle s[n,..], s_c, \sigma, \omega, \mu[l \leftarrow \langle n, i \mapsto s[n-i-1]\rangle], l+1\rangle \xRightarrow[\mathscr{S}\mathscr{M}]{p} c}{P \vdash \langle s, s_c, \sigma, \omega, \mu, l\rangle \xRightarrow[\mathscr{S}\mathscr{M}]{[\text{ARRAY } n]p} c} \quad [\text{ARRAY}_{\mathscr{S}\mathscr{M}}]$$

$$\frac{P \vdash \langle [(\mu(a))[i]]s, s_c, \sigma, \omega, \mu, l\rangle \xRightarrow[\mathscr{S}\mathscr{M}]{p} c}{P \vdash \langle ias, s_c, \sigma, \omega, \mu, l\rangle \xRightarrow[\mathscr{S}\mathscr{M}]{[\text{ELEM}]p} c} \quad [\text{ELEM}_{\mathscr{S}\mathscr{M}}]$$

$$\frac{P \vdash \langle vs, s_c, \sigma, \omega, \mu[a \leftarrow \langle \textbf{fst } (\mu(a)), (\textbf{snd } (\mu(a)))[i \leftarrow v]\rangle], l\rangle \xRightarrow[\mathscr{S}\mathscr{M}]{p} c}{P \vdash \langle vias, s_c, \sigma, \omega, \mu, l\rangle \xRightarrow[\mathscr{S}\mathscr{M}]{[\text{STA}]p} c} \quad [\text{STA}_{\mathscr{S}\mathscr{M}}]$$

1 Pattern Matching

A new syntacic category: patterns $\mathscr{P}$:

$$\begin{array}{rcll} \mathscr{P} & = & \mathbb{N} & \text{— number} \\ & & \_ & \text{— wildcard} \\ & & [\mathscr{P}^*] & \text{— array} \\ & & \mathscr{C}\,\mathscr{P}^* & \text{— S-expression} \\ & & \mathscr{X}@\mathscr{P} & \text{— named pattern} \end{array}$$

Concrete syntax mainly repeats the abstract; in S-expression patterns extra round brackets are used to delimit the constructor's name from arguments (if any); arguments of array/S-expression patterns are delimited by commas, and extra round brackets can be used to group subpatterns. Additionally, one derived form is used: an identifier x is treated as a pattern $x@\_$.

Pattern matching expression:

$$\mathscr{E} += \text{case } \mathscr{E} \text{ of } (\mathscr{P} \times \mathscr{E})^+ \text{ esac}$$

In a concrete syntax branches of case expression are delimited by "|", and in each branch " $\to$ " is used to delimit pattern from expression.

Well-formedness of case expressions is established in an obvious manner:

$$\frac{e : \mathbf{Val} \quad e_i : a}{\text{case } e \text{ of } p_1 \to e_1 \ldots p_k \to e_k \text{ esac} : a}$$

2 Operational Semantics

There are two aspects that have to be covered in semantic description of pattern matching:

- the criterion for a scrutinee to be matched by a pattern;
- the descipline of binding support.

The latter aspect is covered by a desugaring. First, we define a mapping

$$\beta : \mathscr{P} \to \mathscr{E} \to \mathscr{X} \to \mathscr{E}$$

in a following manner:

$$\begin{array}{rcl} \beta\, n\, e & = & \lambda\_.\bot \\ \beta\, \_\, e & = & \lambda\_.\bot \\ \beta\, ([p_0 \ldots p_k])\, e & = & \\ \beta\, (C\, p_0 \ldots p_k)\, e & = & (\beta\, p_0\, e[0])[\beta\, p_1\, e[1]] \ldots [\beta\, p_k\, e[k]] \\ \beta\, (x@p)\, e & = & (\beta\, p\, e)[x \leftarrow e] \end{array}$$

This function determines a proper subvalue of an expression bound by an identifier in a pattern. For example, for a pattern $[x, C(\_, y)]$ and scrutenee s the value of $\beta\,[x, C(\_, y)]\,s$ can be described by the following table:

$$\begin{array}{lcl} x & \to & s[0] \\ y & \to & s[1][1] \end{array}$$

Then, given a pattern-matching expression case e of ... we, first, bind the expression e to a fresh variable, say, s:

var $s = e$;
case s of ...

Then, we transform each branch $p \to e$ into the following:

$$\begin{array}{l} p \to \text{var } b_1 = \beta\, p\, s\, b_1\,, \\ \qquad\qquad \ldots \\ \qquad\quad b_k = \beta\, p\, s\, b_k\,; \\ \qquad\quad e \end{array}$$

where $b_1, \ldots, b_k$ are all bindings in p.

Now, for determining the descipline of matching we need an extra relation

$$match \subseteq \mathscr{P} \times \mathscr{V}$$

between patterns and values. We define it in a following way:

$$match(\_, v)$$

$$match(n, n)$$

$$\frac{match(p_i, v_i)}{match([p_1, \ldots, p_k], [v_1, \ldots, v_k])}$$

$$\frac{match(p_i, v_i)}{match(C\, p_1, \ldots, p_k, C\, v_1, \ldots, v_k)}$$

$$\frac{match(p, v)}{match(x@p, v)}$$

Finally, the operational semantics of pattern-matching expression can be given by the following rules:

$$c \stackrel{e}{\Longrightarrow} \langle c', v \rangle$$

$$\frac{v \vdash c' \xrightarrow{(p_1, e_1) \ldots (p_k, e_k)}_{\mathscr{P}} c''}{c \xRightarrow{\text{case } e \text{ of } p_1 \to e_1 \ \ldots \ p_k \to e_k \text{ esac}} c''}$$

where an additional transition " $\longrightarrow_{\mathscr{P}}$ " is defined as follows:

$$\frac{match(p, v) \quad c \stackrel{e}{\Longrightarrow} c'}{v \vdash c \xrightarrow{(p,e)ps}_{\mathscr{P}} c'}$$

$$\frac{\neg match(p, v) \quad v \vdash c \xrightarrow{ps}_{\mathscr{P}} c'}{v \vdash c \xrightarrow{(p,e)ps}_{\mathscr{P}} c'}$$