1 Structural Control Flow

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

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

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

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

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

$$\frac{\sigma \xRightarrow{e}_{\mathscr{E}} n \neq 0 \qquad \langle \sigma, w \rangle \xRightarrow{S_1}_{\mathscr{S}} c'}{\langle \sigma, w \rangle \xRightarrow{\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}_{\mathscr{S}} c'} \quad [\textsc{If-True}]$$

$$\frac{\sigma \xRightarrow{e}_{\mathscr{E}} 0 \qquad \langle \sigma, w \rangle \xRightarrow{S_2}_{\mathscr{S}} c'}{\langle \sigma, w \rangle \xRightarrow{\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}_{\mathscr{S}} c'} \quad [\textsc{If-False}]$$

$$\frac{\sigma \xRightarrow{e}_{\mathscr{E}} n \neq 0 \qquad \langle \sigma, w \rangle \xRightarrow{S}_{\mathscr{S}} c' \qquad c' \xRightarrow{\texttt{while}\ e\ \texttt{do}\ S}_{\mathscr{S}} c''}{\langle \sigma, w \rangle \xRightarrow{\texttt{while}\ e\ \texttt{do}\ S}_{\mathscr{S}} c''} \quad [\textsc{While-True}]$$

$$\frac{\sigma \xRightarrow{e}_{\mathscr{E}} 0}{\langle \sigma, w \rangle \xRightarrow{\texttt{while}\ e\ \texttt{do}\ S}_{\mathscr{S}} \langle \sigma, w \rangle} \quad [\textsc{While-False}]$$

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

$$\begin{aligned} & \texttt{if}\ e_1\ \texttt{then}\ s_1 \\ & \texttt{elif}\ e_2\ \texttt{then}\ s_2 \\ & \ldots \end{aligned}$$

```
elif e_k then s_k
[ else s_{k+1} ]
  fi
```

and

```
for s_1 , e , s_2 do s_3 od
```

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

```
  if e_1 then s_1                    if e_1 then s_1
elif e_2 then s_2                  else if e_2 then s_2
...                                ...
elif e_k then s_k        ⇝         else if e_k then s_k
else s_{k+1}                       else s_{k+1}
fi                                      fi
                                        ...
                                        fi
```

```
  if e_1 then s_1                    if e_1 then s_1
elif e_2 then s_2                  else if e_2 then s_2
...                                ...
elif e_k then s_k        ⇝         else if e_k then s_k
fi                                 else skip
                                        fi
                                        ...
                                        fi
```

```
for s_1 , e , s_2 do s_3 od        ⇝    s_1 ;
                                        while e do
                                          s_3 ;
                                          s_2
                                        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:

do s **while** e **od**	$\rightsquigarrow$	s; **while** e == 0 **do** s **od**

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

3 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:

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

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

$$\frac{c \overset{p}{\Longrightarrow}_{\mathscr{I}\mathscr{M}} c'}{c' \overset{p'}{\Longrightarrow}_{\mathscr{I}\mathscr{M}} c''}$$

we use the following form

$$\frac{\Gamma' \vdash c \overset{p}{\Longrightarrow}_{\mathscr{I}\mathscr{M}} c'}{\Gamma \vdash c' \overset{p'}{\Longrightarrow}_{\mathscr{I}\mathscr{M}} c''}$$

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 $p'[\texttt{LABEL}\,l]P[l]$. 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).

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

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

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

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

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

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

Figure 2: Big-step operational semantics for extended stack machine

All existing rules have to be rewritten — we need to formally add a $P \vdash \ldots$ part everywhere. For the new instructions the rules are given in Fig. 2.

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

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

4 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 gener-

ation, 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:

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

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.