1 Structural Control Flow

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

$$\begin{array}{rcl}\mathscr{S} & += & \texttt{if}\,\mathscr{E}\,\texttt{then}\,\mathscr{S}\,\texttt{else}\,\mathscr{S}\\ & & \texttt{while}\,\mathscr{E}\,\texttt{do}\,\mathscr{S}\end{array}$$

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:

$$\texttt{if}\;e\;\texttt{then}\;s_1\;\texttt{else}\;s_2\;\texttt{fi}$$

$$\texttt{while}\;e\;\texttt{do}\;s\;\texttt{od}$$

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

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

$$\frac{\sigma \xRightarrow{e}_{\mathscr{E}} n \neq 0 \qquad \langle\sigma, w\rangle \xRightarrow{S}_{\mathscr{S}} c' \qquad c' \xRightarrow{\texttt{while}\;e\;\texttt{do}\;S}_{\mathscr{S}} c''}{\langle\sigma, w\rangle \xRightarrow{\texttt{while}\;e\;\texttt{do}\;S}_{\mathscr{S}} c''}$$

$$[\textsc{While-True}]$$

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

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

```
  if $e_1$ then $s_1$
elif $e_2$ then $s_2$
...
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 $s_k$   $\leadsto$  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 $s_k$   $\leadsto$  else if $e_k$ then $s_k$
fi                                  else skip
                                         fi
                                         ...
                                         fi
```

```
for $s_1$ , $e$ , $s_2$ do $s_3$ od      $\leadsto$      $s_1$ ;
                                                  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:

```
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.