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:

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

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

```
x := x
```

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

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

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

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

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

$$\mathbf{Void} \vdash \texttt{read}\, (x)$$

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

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

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

Figure 1: Well-formed Expressions

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 "$\Rightarrow_*$":

$$c \xRightarrow{\varepsilon}_* \langle c, \varepsilon \rangle \qquad [\textsc{Expr}^*_\varepsilon]$$

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

The semantics for expressions is presented on Fig. 2.

$$c \xRightarrow{z} \langle c, z\rangle \qquad [\textsc{Const}]$$

$$\langle \sigma, \omega\rangle \xRightarrow{x} \langle\langle \sigma, \omega\rangle, \sigma x\rangle \qquad [\textsc{Var}]$$

$$c \xRightarrow{\texttt{ref}\, x} \langle c, \mathbf{ref}\, x\rangle \qquad [\textsc{Ref}]$$

$$\frac{c \xRightarrow{lr}_* \langle c', wv\rangle}{c \xRightarrow{l \oplus r} \langle c', w \otimes v\rangle} \qquad [\textsc{Binop}]$$

$$c \xRightarrow{\texttt{skip}} \langle c, \bot\rangle \qquad [\textsc{Skip}]$$

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

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

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

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

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

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

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

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

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

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:

$$
\begin{array}{rcll}
\mathscr{I} & += & \texttt{LDA}\ \mathscr{X} & \text{loading an address of a variable} \\
 & & \texttt{STI} & \text{storing by indirect address} \\
 & & \texttt{DROP} & \text{discard the top of the stack}
\end{array}
$$

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

$$\dfrac{P \vdash \langle [\mathbf{ref}\,x]s,\,\theta\rangle \xRightarrow{p}_{\mathscr{SM}} c'}{P \vdash \langle s,\,\theta\rangle \xRightarrow{[\texttt{LDA}\,x]p}_{\mathscr{SM}} c'} \qquad [\text{LDA}_{SM}]$$

$$\dfrac{P \vdash \langle vs,\,\langle \sigma[x \leftarrow v],\,\omega\rangle\rangle \xRightarrow{p}_{\mathscr{SM}} c'}{P \vdash \langle v[\mathbf{ref}\,x]s,\,\langle \sigma,\,\omega\rangle\rangle \xRightarrow{[\texttt{STI}]p}_{\mathscr{SM}} c'} \qquad [\text{STI}_{SM}]$$

$$\dfrac{\langle zs,\,\langle \sigma[x \leftarrow z],\,\omega\rangle\rangle \xRightarrow{p}_{\mathscr{SM}} c'}{\langle zs,\,\langle \sigma,\,\omega\rangle\rangle \xRightarrow{[\texttt{ST}\,x]p}_{\mathscr{SM}} c'} \qquad [\text{ST}_{SM}]$$

$$\dfrac{P \vdash \langle s,\,\theta\rangle \xRightarrow{p}_{\mathscr{SM}} c'}{P \vdash \langle xs,\,\theta\rangle \xRightarrow{[\texttt{DROP}]p}_{\mathscr{SM}} c'} \qquad [\text{DROP}_{SM}]$$