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*:

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

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

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

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

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

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

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

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

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

$\otimes$	$\oplus$ 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, "$1 \vee \mathtt{x}$" is undefined in empty state.

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

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