Today we introduce a very simple imperative language, IMP, along with two systems of rules for evaluation called *small-step* and *big-step* semantics. These both fall under the general style called *structural operational semantics* (SOS). We will also discuss why both the big-step and small-step approaches can be useful.

1 The IMP Language

1.1 Syntax of IMP

There are three types of expressions in IMP:

- *arithmetic expressions AExp* (elements are denoted $a, a_0, a_1, \ldots$)
- *Boolean expressions BExp* (elements are denoted $b, b_0, b_1, \ldots$)
- *commands Com* (elements are denoted $c, c_0, c_1, \ldots$)

A *program* in the IMP language is a command in *Com*.

Let Var be a countable set of variables. Elements of Var are denoted $x, x_0, x_1 \ldots$. Let $n, n_0, n_1, \ldots$ denote integers (elements of $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$). Let $\overline{n}$ be an integer constant symbol representing the number n. The BNF grammar for IMP is

$$
\begin{array}{rrcl}
AExp: & a & ::= & \overline{n} \mid x \mid a_0 \oplus a_1 \\
BExp: & b & ::= & \mathsf{true} \mid \mathsf{false} \mid a_0 \odot a_1 \mid b_0 \oslash b_1 \mid \neg b \\
Com: & c & ::= & \mathsf{skip} \mid x := a \mid c_0 \,;\, c_1 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c \\
 & \oplus & ::= & + \mid * \mid - \\
 & \odot & ::= & \leq \mid = \\
 & \oslash & ::= & \vee \mid \wedge
\end{array}
$$

1.2 Stores and Configurations

A *store* (also known as a *state*) is a function $\sigma : \mathsf{Var} \to \mathbb{Z}$ that assigns an integer to each variable. Stores are denoted $\sigma, \sigma_1, \tau, \ldots$ and the set of all stores is denoted Σ.

A *configuration* is a pair $\langle c, \sigma \rangle$, where $c \in Com$ is a command and σ is a store. Intuitively, the configuration $\langle c, \sigma \rangle$ represents an instantaneous snapshot of reality during a computation, in which σ represents the current values of the variables and c represents the next command to be executed.

2 Small-Step Semantics

Small-step semantics specifies the operation of a program one step at a time. There is a set of rules that we continue to apply to configurations until reaching a final configuration $\langle \mathsf{skip}, \sigma \rangle$ (if ever). We write $\langle c, \sigma \rangle \xrightarrow{1} \langle c', \sigma' \rangle$ to indicate that the configuration $\langle c, \sigma \rangle$ reduces to $\langle c', \sigma' \rangle$ in one step, and we write $\langle c, \sigma \rangle \to \langle c', \sigma' \rangle$ to indicate that $\langle c, \sigma \rangle$ reduces to $\langle c', \sigma' \rangle$ in zero or more steps. Thus $\langle c, \sigma \rangle \to \langle c', \sigma' \rangle$ iff there is a $k \geq 0$ and configurations $\langle c_0, \sigma_0 \rangle, \ldots, \langle c_k, \sigma_k \rangle$ such that $\langle c, \sigma \rangle = \langle c_0, \sigma_0 \rangle$, $\langle c', \sigma' \rangle = \langle c_k, \sigma_k \rangle$, and $\langle c_i, \sigma_i \rangle \xrightarrow{1} \langle c_{i+1}, \sigma_{i+1} \rangle$ for $0 \leq i \leq k-1$.

To be completely proper, we will define auxiliary small-step operators $\rightarrow_a$ and $\rightarrow_b$ for arithmetic and Boolean expressions, respectively, as well as $\rightarrow$ for commands[1]. The types of these operators are

$$\begin{array}{rcl} \rightarrow_a & : & (AExp \times \Sigma) \rightarrow AExp \\ \rightarrow_b & : & (BExp \times \Sigma) \rightarrow BExp \\ \rightarrow & : & (Com \times \Sigma) \rightarrow (Com \times \Sigma) \end{array}$$

Intuitively, $\langle a, \sigma\rangle \rightarrow_a \overline{n}$ if the expression a evaluates to the constant $\overline{n}$ for the integer n in state σ.

We now present the small-step rules for evaluation in IMP. Just as with the λ-calculus, evaluation is defined by a set of inference rules which inductively define relations consisting of acceptable computation steps.

2.1 Arithmetic and Boolean Expressions

- Variables:

$$\frac{}{\langle x, \sigma\rangle \xrightarrow{1}_a \overline{\sigma(x)}}$$

- Arithmetic:

$$\frac{}{\langle \overline{n}_1 \oplus \overline{n}_2, \sigma\rangle \xrightarrow{1}_a \overline{n}_3} \text{ (if } n_3 = n_1 \oplus n_2) \qquad \frac{\langle a_1, \sigma\rangle \xrightarrow{1}_a a_1'}{\langle a_1 \oplus a_2, \sigma\rangle \xrightarrow{1}_a a_1' \oplus a_2} \qquad \frac{\langle a_2, \sigma\rangle \xrightarrow{1}_a a_2'}{\langle \overline{n}_1 \oplus a_2, \sigma\rangle \xrightarrow{1}_a \overline{n}_1 \oplus a_2'}$$

One subtle point: the $\oplus$ appearing in the expression $\overline{n}_1 \oplus \overline{n}_2$ represents the operation symbol in the IMP language, which is a syntactic object; whereas the $\oplus$ appearing in the expression $n_1 \oplus n_2$ represents the actual operation in $\mathbb{Z}$, which is a semantic object. These are two different things, just as $\overline{n}$ and n are two different things and true and *true* are two different things. In this case, at the risk of confusion, we have used the same notation $\oplus$ for both of them.

The rules for Booleans and comparison operators are similar. We leave them as exercises.

2.2 Commands

Let $\sigma[n/x]$ denote the store that is identical to σ except possibly for the value of x, which is n. That is,

$$\sigma[n/x](y) \triangleq \begin{cases} \sigma(y) & \text{if } y \neq x, \\ n & \text{if } y = x. \end{cases}$$

- Assignment:

$$\frac{}{\langle x := \overline{n}, \sigma\rangle \xrightarrow{1} \langle \mathsf{skip}, \sigma[n/x]\rangle} \qquad \frac{\langle a, \sigma\rangle \xrightarrow{1}_a a'}{\langle x := a, \sigma\rangle \xrightarrow{1} \langle x := a', \sigma\rangle}$$

- Sequence:

$$\frac{\langle c_0, \sigma\rangle \xrightarrow{1} \langle c_0', \sigma'\rangle}{\langle c_0 \,;\, c_1, \sigma\rangle \xrightarrow{1} \langle c_0' \,;\, c_1, \sigma'\rangle} \qquad \frac{}{\langle \mathsf{skip} \,;\, c_1, \sigma\rangle \xrightarrow{1} \langle c_1, \sigma\rangle}$$

[1] Winskel [1] uses $\rightarrow_1$ instead of $\xrightarrow{1}$.

- Conditional:

$$\frac{\langle b,\, \sigma\rangle \xrightarrow{1}_b b'}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle \xrightarrow{1} \langle \mathsf{if}\ b'\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle}$$

$$\frac{}{\langle \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle \xrightarrow{1} \langle c_0,\, \sigma\rangle} \qquad \frac{}{\langle \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle \xrightarrow{1} \langle c_1,\, \sigma\rangle}$$

- While statement:

$$\frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma\rangle \xrightarrow{1} \langle \mathsf{if}\ b\ \mathsf{then}\ (c\ ;\ \mathsf{while}\ b\ \mathsf{do}\ c)\ \mathsf{else}\ \mathsf{skip},\, \sigma\rangle}$$

There is no rule for skip, since $\langle \mathsf{skip},\, \sigma\rangle$ is a final configuration.

3 Big-Step Semantics

As an alternative to small-step operational semantics, which specifies the operation of the program one step at a time, we now consider *big-step operational semantics*, in which we specify the entire transition from a configuration (an ⟨expression, state⟩ pair) to a final value. This relation is denoted $\Downarrow$. For arithmetic expressions, the final value is an integer; for Boolean expressions, it is a Boolean truth value *true* or *false*; and for commands, it is a final state. Thus

$$\begin{aligned} \Downarrow_a &: (AExp \times \Sigma) \to \mathbb{Z} \\ \Downarrow_b &: (BExp \times \Sigma) \to \mathbb{2} \\ \Downarrow &: (Com \times \Sigma) \to \Sigma \end{aligned}$$

Here $\mathbb{2}$ represents the two-element Boolean algebra consisting of the two truth values $\{true, false\}$ with the usual Boolean operations $\wedge, \vee, \neg$. Then

- $\langle c,\, \sigma\rangle \Downarrow \sigma'$ says that σ' is the store of the final configuration, starting in configuration $\langle c,\, \sigma\rangle$;
- $\langle a,\, \sigma\rangle \Downarrow_a n$ says that $n \in \mathbb{Z}$ is the integer value of arithmetic expression a evaluated in state σ; and
- $\langle b,\, \sigma\rangle \Downarrow_b t$ says that $t \in \mathbb{2}$ is the truth value of Boolean expression b evaluated in state σ.

3.1 Arithmetic and Boolean Expressions

The big-step rules for arithmetic and Boolean expressions are straightforward. The key when writing big-step rules is to think about how a recursive interpreter would evaluate the expression in question. The rules for arithmetic expressions are:

- Constants:

$$\frac{}{\langle \overline{n},\, \sigma\rangle \Downarrow_a n}$$

- Variables:

$$\frac{}{\langle x,\, \sigma\rangle \Downarrow_a \sigma(x)}$$

- Operations:

$$\frac{\langle a_0,\, \sigma\rangle \Downarrow_a n_0 \qquad \langle a_1,\, \sigma\rangle \Downarrow_a n_1}{\langle a_0 \oplus a_1,\, \sigma\rangle \Downarrow_a n_0 \oplus n_1}$$

The rules for evaluating Boolean expressions and comparison operators are similar.

3.2 Commands

- Skip:

$$\frac{}{\langle \mathsf{skip},\, \sigma\rangle \Downarrow \sigma}$$

- Assignments:

$$\frac{\langle a,\, \sigma\rangle \Downarrow n}{\langle x := a,\, \sigma\rangle \Downarrow \sigma[n/x]}$$

- Sequences:

$$\frac{\langle c_0,\, \sigma\rangle \Downarrow \sigma' \quad \langle c_1,\, \sigma'\rangle \Downarrow \sigma''}{\langle c_0; c_1,\, \sigma\rangle \Downarrow \sigma''}$$

- Conditionals:

$$\frac{\langle b,\, \sigma\rangle \Downarrow \mathit{true} \quad \langle c_0,\, \sigma\rangle \Downarrow \sigma'}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle \Downarrow \sigma'} \qquad \frac{\langle b,\, \sigma\rangle \Downarrow \mathit{false} \quad \langle c_1,\, \sigma\rangle \Downarrow \sigma'}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1,\, \sigma\rangle \Downarrow \sigma'}$$

- While statements:

$$\frac{\langle b,\, \sigma\rangle \Downarrow \mathit{false}}{\langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma\rangle \Downarrow \sigma} \qquad \frac{\langle b,\, \sigma\rangle \Downarrow \mathit{true} \quad \langle c,\, \sigma\rangle \Downarrow \sigma' \quad \langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma'\rangle \Downarrow \sigma''}{\langle \mathsf{while}\ b\ \mathsf{do}\ c,\, \sigma\rangle \Downarrow \sigma''}$$

4 Agreement of Big-Step and Small-Step SOS

If the big-step and small-step semantics both describe the same language, we would expect them to agree. In particular, the relations $\to$ and $\Downarrow$ both capture the idea of a complete evaluation. We would expect that if $\langle c,\, \sigma\rangle$ is a configuration that evaluates in the small-step semantics to $\langle \mathsf{skip},\, \sigma'\rangle$, then σ' should also be the result of the big-step evaluation, and vice-versa. Formally,

$$\langle c,\, \sigma\rangle \to \langle \mathsf{skip},\, \sigma'\rangle \iff \langle c,\, \sigma\rangle \Downarrow \sigma'$$

It is possible to prove this assertion by induction.

Note that this statement about the agreement of big-step and small-step semantics has nothing to say about the agreement of nonterminating computations. This is because big-step semantics cannot talk directly about nontermination. If $\langle c,\, \sigma\rangle$ does not terminate, then there is no σ' such that $\langle c,\, \sigma\rangle \Downarrow \sigma'$.

5 Comparison of Big-Step vs. Small-Step SOS

Small-step semantics can model more complex features such as nonterminating programs and concurrency. However, in many cases it involves unnecessary extra work.

If we do not care about modeling nonterminating computations, it is often easier to reason in terms of big-step semantics. Moreover, big-step semantics more closely models an actual recursive interpreter. However, because evaluation skips over intermediate steps, all programs without final configurations (infinite loops, errors, stuck configurations) are indistinguishable.

References

[1] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.