Comp Sci 704
Fall 2025

Lecture 4: The Imp Language

Ethan Cecchetti

We have so far only been talking about basic mathematical principles and using a simple language of arithmetic programs as an example. That language, however, is far too restricted to talk about the full range of possible programs and computations. We now introduce a more realistic, but still small and simple imperative language that includes not only arithmetic expressions, but also variables and control flow constructs like if and while.

1 A Simple Imperative Language

The language we will look at is Imp. Its syntax is a bit more complicated, consisting of three different parts: arithmetic expressions, boolean expressions, and commands. The BNF grammar for Imp is as follows.

$$\begin{array}{rrcl}
\textbf{AExp}: & a & ::= & x \mid \overline{n} \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2 \\
\textbf{BExp}: & b & ::= & \mathsf{true} \mid \mathsf{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b \\
\textbf{Com}: & c & ::= & \mathsf{skip} \mid x := a \mid c_1 \,;\, c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c
\end{array}$$

As in Arith, the syntax $\overline{n} + \overline{m}$ denotes the syntactic expression with three symbols, $\overline{n}$, $+$, and $\overline{m}$, not the number that is the sum of n and m.

2 Small-Step Operational Semantics

We might hope to define the small-step operational semantics for Imp just as we did for Arith. However, one major feature gets in the way: variables in our expressions. What should $x + 2$ step to?

To address this concern, we go back to the underlying goal of a small-step operational semantics: to describe how a program executes as an abstract machine. In a machine with variables, we often have some sort of memory to record the values of those variables. We can do the same thing here, and define the current state of the abstract machine—called a *configuration*—to include two parts.

- The *expression* to evaluate.
- a *store* (also known as an environment, state, or valuation), which maps variables to integers. Note that not all variables need to be mapped at a given time. The store is often denoted by σ.

More formally, we let **Store** $\triangleq$ **Var** $\rightharpoonup$ **Int** by a partial function from variables to integer symbols, and a configuration is then an **AExp** (or **BExp** or **Com**) and a **Store**.

We will denote configurations using angle brackets. For instance, $\langle (x+2) * (y+3), \sigma \rangle$ is the configuration where $(x+2) * (y+3)$ is an arithmetic expression and σ is the store.

We could just define the semantics of arithmetic expressions this way, but Imp has three different types of syntax! That means we need three sets of operational semantic rules, one for each. The types of these relations are as follows

$$\begin{array}{rcl}
\longrightarrow_a & \subseteq & (\textbf{AExp} \times \textbf{Store}) \times \textbf{AExp} \\
\longrightarrow_b & \subseteq & (\textbf{BExp} \times \textbf{Store}) \times \textbf{BExp} \\
\longrightarrow_c & \subseteq & (\textbf{Com} \times \textbf{Store}) \times (\textbf{Com} \times \textbf{Store})
\end{array}$$

As before, we write $\langle c, \sigma \rangle \longrightarrow_c \langle c', \sigma' \rangle$ to indicate that a configuration $\langle c, \sigma \rangle$ reduces to configuration $\langle c', \sigma' \rangle$ in a single step. Note that the steps for expressions (both arithmetic and boolean) do not contain a store on the right side. That is because the only way to assign variables is in commands, so expressions cannot change the store and there is no reason to track an "output" store of those steps.

We now present the small-step semantic rules for Imp. For simplicity (to avoid the need to worry about free variables), we will assume our stores are *total* functions from the space of variables to integers (**Store** =

Var $\to$ **Int**). We could define a default value (say $\overline{0}$), but it won't be necessary as long as we assume some mapping exists.

Arithmetic Expressions. These rules are identical to the rules for Arith, except there is now a rule to handle variables.

$$\frac{\sigma(x) = \overline{n}}{\langle x, \sigma\rangle \longrightarrow_a \overline{n}} \qquad \frac{\otimes \in \{+, *, -\} \quad \langle a_1, \sigma\rangle \longrightarrow_a a_1'}{\langle a_1 \otimes a_2, \sigma\rangle \longrightarrow_a a_1' \otimes a_2} \qquad \frac{\otimes \in \{+, *, -\} \quad \langle a_2, \sigma\rangle \longrightarrow_a a_2'}{\langle \overline{n} \otimes a_2, \sigma\rangle \longrightarrow_a \overline{n} \otimes a_2'} \qquad \frac{\otimes \in \{+, *, -\} \quad m = n_1 \otimes n_2 \text{ (mathematically)}}{\langle \overline{n}_1 \otimes \overline{n}_2, \sigma\rangle \longrightarrow_a \overline{m}}$$

Boolean Expressions. These rules are similar and we leave them as an exercise. Note that they are defined in terms of the $\longrightarrow_a$ rules, but not vice-versa.

Commands. We again use $\sigma[x \mapsto n]$ to indicate the function that maps x to n but otherwise behaves exactly as σ.

$$[\text{AssignN}]\ \frac{}{\langle x := \overline{n}, \sigma\rangle \longrightarrow_c \langle \mathsf{skip}, \sigma[x \mapsto \overline{n}]\rangle} \qquad [\text{AssignA}]\ \frac{\langle a, \sigma\rangle \longrightarrow_a a'}{\langle x := a, \sigma\rangle \longrightarrow_c \langle x := a', \sigma\rangle}$$

$$[\text{SeqC}]\ \frac{\langle c_1, \sigma\rangle \longrightarrow_c \langle c_1', \sigma'\rangle}{\langle c_1\ ;\ c_2, \sigma\rangle \longrightarrow_c \langle c_1'\ ;\ c_2, \sigma'\rangle} \qquad [\text{SeqSkip}]\ \frac{}{\langle \mathsf{skip}\ ;\ c, \sigma\rangle \longrightarrow_c \langle c, \sigma\rangle}$$

$$[\text{IfB}]\ \frac{\langle b, \sigma\rangle \longrightarrow_b b'}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \longrightarrow_c \langle \mathsf{if}\ b'\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle}$$

$$[\text{IfT}]\ \frac{}{\langle \mathsf{if\ true\ then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \longrightarrow_c \langle c_1, \sigma\rangle} \qquad [\text{IfF}]\ \frac{}{\langle \mathsf{if\ false\ then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \longrightarrow_c \langle c_2, \sigma\rangle}$$

$$[\text{While}]\ \frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle \longrightarrow_c \langle \mathsf{if}\ b\ \mathsf{then}\ (c\ ;\ \mathsf{while}\ b\ \mathsf{do}\ c)\ \mathsf{else\ skip}, \sigma\rangle}$$

Notice that AssignN, which should modify the current store, uses the notation $\sigma[x \mapsto \overline{n}]$. That notation denotes the (partial) function that behaves exactly like σ except on x, where it returns $\overline{n}$. It doesn't matter if σ is defined on x or not. If $\sigma(x)$ was previously defined, it is replaced, and if it was previously undefined, it is added. That is, $f = \sigma[x \mapsto \overline{n}]$ if

$$f(y) = \begin{cases} \overline{n} & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

There is no rule for skip because the configuration $\langle \mathsf{skip}, \sigma\rangle$ is irreducible—it indicates the (sub)program has terminated. In all other cases, there is exactly one rule that applies. Notably, that makes $\longrightarrow_c$ a partial function with type

$$\longrightarrow_c\ : (\mathbf{Com} \times \mathbf{Store}) \rightharpoonup (\mathbf{Com} \times \mathbf{Store}).$$

Similarly, $\longrightarrow_a$ and $\longrightarrow_b$ are also partial functions.

We can again define a *multi-step* rule that for zero or more steps of a configuration.

$$\frac{}{\langle c, \sigma\rangle \longrightarrow_c^* \langle c, \sigma\rangle} \qquad \frac{\langle c, \sigma\rangle \longrightarrow_c \langle c', \sigma'\rangle \qquad \langle c', \sigma'\rangle \longrightarrow_c^* \langle c'', \sigma''\rangle}{\langle c, \sigma\rangle \longrightarrow_c^* \langle c'', \sigma''\rangle}$$

Importantly, this multi-step relation talks about stepping the entire state of the abstract machine—both the command and store—through any finite amount of execution.

These rules tell us all we need to know to run Imp programs.

Unlike Arith programs, not all Imp programs terminate! The simplest nonterminating program one can write in Imp is the infinite loop: `while true do skip`. Indeed, while Arith programs could only represent arithmetic computations, Imp is *Turing complete*, which means it can represent any computation a Turing machine—or most any programming language—can compute.

3 Other Semantics

Some of these operations, particularly the small-step rules for arithmetic and boolean expressions, may seem somewhat tedious. Those expressions always terminate, so why do we have to move them one step at a time to see what happens? There should be a way to "jump" all the way to the end in one "big" step. Indeed there is, and we will see how to define such a big-step semantics for both expressions and all of Imp next time.