1 Intro to semantics

What is the meaning of a program? When we write a program, we use a sequence of characters to represent
the program. But this syntax is just how we represent the program: it is not what the program means.

Maybe we could define the meaning of a program to be whatever happens when we execute the pro-
gram (perhaps using an interpreter, or by compiling it first). But we can have bugs in interpreters and
compilers! That is, an interpreter or compiler may not accurately reflect the meaning of a program. So we
must look elsewhere for a definition of what a program means.

One place to look for the meaning of a program is in the language specification manual. Such manuals
typically give an informal description of the language constructs.

Another option is to give a formal, mathematical definition of the language semantics. A formal math-
ematical definition can have the following advantages over an informal description: less ambiguous; more
concise; and, most importantly, the ability to write mathematical proofs for program properties that we’re
interested in. The drawback of formal semantics is that they can lead to fairly complex mathematical mod-
els, especially if one attempts to describe all details in a full-featured modern language.

There are three main approaches to specify the semantics of programming languages:

- operational semantics: describes how a program would execute on an abstract machine;
- denotational semantics: models programs as mathematical functions;
- axiomatic semantics: defines program behavior in terms of the logical formulae that are satisfied
before and after a program;

Each of these approaches has different advantages and disadvantages in terms of how mathematically
sophisticated they are, how easy it is to use them in proofs, or how easy it is to implement an interpreter or
compiler based on them.

2 A simple language of arithmetic expressions

To understand some of the key concepts of semantics, we will start with a very simple language of integer
arithmetic expressions, with assignment. A program in this language is an expression; executing a program
means evaluating the expression to an integer.

To describe the structure of this language we will use the following domains:

\[
x, y, z \in \text{Var} \\
n, m \in \text{Int} \\
e \in \text{Exp}
\]

\text{Var} is the set of program variables (e.g., foo, bar, baz, i, etc.). \text{Int} is the set of constant integers (e.g., 42,
-40, 7). \text{Exp} is the domain of expressions, which we specify using a BNF (Backus-Naur Form) grammar:

\[
e ::= x | n | e_1 + e_2 | e_1 \times e_2 | x := e_1; e_2
\]

Informally, the expression \( x := e_1; e_2 \) means that \( x \) is assigned the value of \( e_1 \) before evaluating \( e_2 \). The
result of the entire expression is that of \( e_2 \).
This grammar specifies the syntax for the language. An immediate problem here is that the grammar is ambiguous. Consider the expression $1 + 2 \times 3$. One can build two abstract syntax trees:

```
+  *
/ \  / \  
1 * + 3
/ \ / \  
2 3 1 2
```

There are several ways to deal with this problem. One is to rewrite the grammar for the same language to make it unambiguous. But that makes the grammar more complex, and harder to understand. Another possibility is to extend the syntax to require parentheses around all expressions:

$$x \mid n \mid (e_1 + e_2) \mid (e_1 \times e_2) \mid x := e_1; e_2$$

However, this also leads to unnecessary clutter and complexity.

Instead, we separate the “concrete syntax” of the language (which specifies how to unambiguously parse a string into program phrases) from the “abstract syntax” of the language (which describes, possibly ambiguously, the structure of program phrases). In this course we will use the abstract syntax and assume that the abstract syntax tree is known. When writing expressions, we will occasionally use parenthesis to indicate the structure of the abstract syntax tree, but the parentheses are not part of the language itself. (For details on parsing, grammars, and ambiguity elimination, see or take the compiler course Computer Science 153.)

### 2.1 Representing expressions

Note that the syntactic structure of expressions in this language can be compactly expressed in Ocaml using datatypes:

```ocaml
type exp = Var of string | Int of int |
          Add of exp * exp | Mul of exp * exp |
          Asgn of string * exp * exp
```

This closely matches the BNF grammar above. The abstract syntax tree of an expression can be obtained by applying the datatype constructors in each case. For instance, the AST of expression $2 \times (\text{foo} + 1)$ is:

```
Mul(Int(2), Add(Var("foo"), Int(1)))
```

In Ocaml, parentheses can be dropped when there is one single argument, so the above expression can be written as:

```
Mul Int 2, Add Var "foo", Int 1)
```

In a language like Java, expressing this structure would require a more complex declaration consisting of a class hierarchy:

```java
abstract class Expr { } 
class Var extends Expr { String name; .. }
class Int extends Expr { int val; ... }
class Add extends Expr { Expr expr1, expr2; ... }
class Mul extends Expr { Expr expr1, expr2; ... }
class Asgn extends Expr { String var, Expr expr1, expr2; .. }
```

### 3 Small-step operational semantics

At this point we have defined the syntax of our simple arithmetic language. We have some informal, intuitive notion of what programs in this language mean. For example, the program $7 + (4 \times 2)$ should equal 15, and the program $i := 6 + 1; 2 \times 3 \times i$ should equal 42.

We would like now to define formal semantics for this language.
Operational semantics describe how a program would execute on an abstract machine. A small-step operational semantics describe how such an execution in terms of successive reductions of an expression, until we reach a number, which represents the result of the computation.

The state of the abstract machine is usually referred to as a configuration, and for our language it must include two pieces of information:

- a store (aka environment or state), which assigns integer values to variables. During program execution, we will refer to the store to determine the values associated with variables, and also update the store to reflect assignment of new values to variables.
- the expression left to evaluate.

Thus, the domain of stores is functions from \( \text{Var} \) to \( \text{Int} \) (written \( \text{Var} \rightarrow \text{Int} \)), and the domain of configurations is pairs of expressions and stores.

\[
\text{Config} = \text{Exp} \times \text{Store} \\
\text{Store} = \text{Var} \rightarrow \text{Int}
\]

We will denote configurations using angle brackets. For instance, \( \langle (\text{foo} + 2) \times (\text{bar} + 1), \sigma \rangle \), where \( \sigma \) is a store and \( (\text{foo} + 2) \times (\text{bar} + 1) \) is an expression that uses two variables, foo and bar.

The small-step operational semantics for our language is a relation \( \xrightarrow{\rightarrow} \subseteq \text{Config} \times \text{Config} \) that describes how one configuration transitions to a new configuration. That is, the relation \( \xrightarrow{\rightarrow} \) shows us how to evaluate programs, one step at a time. We use infix notation for the relation \( \xrightarrow{\rightarrow} \). That is, given any two configurations \( \langle e_1, \sigma_1 \rangle \) and \( \langle e_2, \sigma_2 \rangle \), if \( (\langle e_1, \sigma_1 \rangle, \langle e_2, \sigma_2 \rangle) \) is in the relation \( \xrightarrow{\rightarrow} \), then we write \( \langle e_1, \sigma_1 \rangle \xrightarrow{\rightarrow} \langle e_2, \sigma_2 \rangle \).

For example, we have \( \langle (4 + 2) \times y, \sigma \rangle \xrightarrow{\rightarrow} \langle 6 \times y, \sigma \rangle \). That is, we can evaluate the configuration \( \langle (4 + 2) \times y, \sigma \rangle \) by one step, to get the configuration \( \langle 6 \times y, \sigma \rangle \).

Now defining the semantics of the language boils down to defining the relation \( \xrightarrow{\rightarrow} \) that describes the transitions between machine configurations.

One issue here is that the domain of integers is infinite, and so is the domain of expressions. Therefore, there is an infinite number of possible machine configurations, and an infinite number of possible one-step transitions. We need to use a finite description for the infinite set of transitions.

We can compactly describe the transition function \( \xrightarrow{\rightarrow} \) using inference rules:

\[
\text{VAR} \quad \langle x, \sigma \rangle \xrightarrow{\rightarrow} \langle n, \sigma \rangle \quad \text{where} \quad n = \sigma(x)
\]

\[
\text{LADD} \quad \langle e_1, \sigma \rangle \xrightarrow{\rightarrow} \langle e'_1, \sigma' \rangle \quad \langle e_1 + e_2, \sigma \rangle \xrightarrow{\rightarrow} \langle e'_1 + e_2, \sigma' \rangle
\]

\[
\text{RADD} \quad \langle e_2, \sigma \rangle \xrightarrow{\rightarrow} \langle e'_2, \sigma' \rangle \quad \langle n + e_2, \sigma \rangle \xrightarrow{\rightarrow} \langle n + e'_2, \sigma' \rangle
\]

\[
\text{ADD} \quad \langle n + m, \sigma \rangle \xrightarrow{\rightarrow} \langle p, \sigma \rangle \quad \text{where} \quad p \text{ is the sum of } n \text{ and } m
\]

\[
\text{LMUL} \quad \langle e_1, \sigma \rangle \xrightarrow{\rightarrow} \langle e'_1, \sigma' \rangle \quad \langle e_1 \times e_2, \sigma \rangle \xrightarrow{\rightarrow} \langle e'_1 \times e_2, \sigma' \rangle
\]

\[
\text{RMUL} \quad \langle e_2, \sigma \rangle \xrightarrow{\rightarrow} \langle e'_2, \sigma' \rangle \quad \langle n \times e_2, \sigma \rangle \xrightarrow{\rightarrow} \langle n \times e'_2, \sigma' \rangle
\]

\[
\text{MUL} \quad \langle n \times m, \sigma \rangle \xrightarrow{\rightarrow} \langle p, \sigma \rangle \quad \text{where} \quad p \text{ is the product of } n \text{ and } m
\]
Lecture 1: Intro to semantics; Small-step semantics

Hence, \(e\) rules. The rules without premises are axioms; and the rules with premises are inductive rules. The fact below the line holds. The fact(s) above the line are called premises; the fact below the line is called the conclusion. The rules without premises are axioms; and the rules with premises are inductive rules.

Also, we use the notation \(\sigma[x \mapsto n]\) for a store that maps the variable \(x\) to integer \(n\), and maps every other variable to whatever \(\sigma\) maps it to. More explicitly, we if \(f\) is the function \(\sigma[x \mapsto n]\), then we have

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

4 Using the Semantic Rules

Let’s see how we can use these rules. Suppose we want to evaluate expression \((\text{foo} + 2) \times (\text{bar} + 1)\) in a store \(\sigma\) where \(\sigma(\text{foo}) = 4\) and \(\sigma(\text{bar}) = 3\). That is, we want to find the transition for configuration \(\langle(\text{foo} + 2) \times (\text{bar} + 1), \sigma\rangle\). For this, we look for a rule with this form of a configuration in the conclusion. By inspecting the rules, we find that the only matching rule is \(\text{LMUL}\), where \(e_1 = \text{foo} + 2\), \(e_2 = \text{bar} + 1\), but \(e'_1\) is not yet known. We can instantiate the rule \(\text{LMUL}\), replacing the metavariables \(e_1\) and \(e_2\) with appropriate expressions.

\[
\text{LMUL} \quad \frac{\langle \text{foo} + 2, \sigma \rangle \longrightarrow \langle e'_1, \sigma \rangle}{\langle (\text{foo} + 2) \times (\text{bar} + 1), \sigma \rangle \longrightarrow \langle e'_1 \times (\text{bar} + 1), \sigma \rangle}
\]

Now we need to show that the premise actually holds and find out what \(e'_1\) is. We look for a rule whose conclusion matches \(\langle \text{foo} + 2, \sigma \rangle \longrightarrow \langle e'_1, \sigma \rangle\). We find that \(\text{LADD}\) is the only matching rule:

\[
\text{LADD} \quad \frac{\langle \text{foo}, \sigma \rangle \longrightarrow \langle e''_1, \sigma \rangle}{\langle \text{foo} + 2, \sigma \rangle \longrightarrow \langle e''_1 + 2, \sigma \rangle}
\]

where \(e'_1 = e''_1 + 2\). We repeat this reasoning for \(\langle \text{foo}, \sigma \rangle \longrightarrow \langle e''_1, \sigma \rangle\), and we find that the only applicable rule is the axiom \(\text{VAR}\):

\[
\text{VAR} \quad \frac{\langle \text{foo}, \sigma \rangle \longrightarrow \langle 4, \sigma \rangle}{\langle 4, \sigma \rangle}
\]

because we have \(\sigma(\text{foo}) = 4\). Since this is an axiom and has no premises, there is nothing left to prove. Hence, \(e''_1 = 4\) and \(e'_1 = 4 + 2\). We can put together the above pieces and build the following proof:

\[
\begin{align*}
\text{VAR} & \quad \frac{\langle \text{foo}, \sigma \rangle \longrightarrow \langle 4, \sigma \rangle}{\langle 4, \sigma \rangle} \\
\text{LADD} & \quad \frac{\langle \text{foo}, \sigma \rangle \longrightarrow \langle 4, \sigma \rangle}{\langle \text{foo} + 2, \sigma \rangle \longrightarrow \langle 4 + 2, \sigma \rangle} \\
\text{LMUL} & \quad \frac{\langle \text{foo} + 2, \sigma \rangle \longrightarrow \langle 4 + 2, \sigma \rangle}{\langle (\text{foo} + 2) \times (\text{bar} + 1), \sigma \rangle \longrightarrow \langle (4 + 2) \times (\text{bar} + 1), \sigma \rangle}
\end{align*}
\]

This proves that, given our inference rules, the one-step transition \(\langle (\text{foo} + 2) \times (\text{bar} + 1), \sigma \rangle \longrightarrow \langle (4 + 2) \times (\text{bar} + 1), \sigma \rangle\) is possible. The above proof structure is called a “proof tree” or “derivation”. It is important to keep in mind that proof trees must be finite for the conclusion to be valid.

We can use a similar reasoning to find out the next evaluation step:

\[
\begin{align*}
\text{ADD} & \quad \frac{\langle 4 + 2, \sigma \rangle \longrightarrow \langle 6, \sigma \rangle}{\langle 6, \sigma \rangle} \\
\text{LMUL} & \quad \frac{\langle 4 + 2, \sigma \rangle \longrightarrow \langle 6, \sigma \rangle}{\langle (4 + 2) \times (\text{bar} + 1), \sigma \rangle \longrightarrow \langle 6 \times (\text{bar} + 1), \sigma \rangle}
\end{align*}
\]
And we can continue this process. At the end, we can put together all of these transitions, to get a view of the entire computation:

\[
\langle (\text{foo} + 2) \times (\text{bar} + 1), \sigma \rangle \rightarrow \langle (4 + 2) \times (\text{bar} + 1), \sigma \rangle \\
\rightarrow \langle 6 \times (\text{bar} + 1), \sigma \rangle \\
\rightarrow \langle 6 \times (3 + 1), \sigma \rangle \\
\rightarrow \langle 6 \times 4, \sigma \rangle \\
\rightarrow \langle 24, \sigma \rangle
\]

The result of the computation is a number, 24. The machine configuration that contains the final result is the point where the evaluation stops; they are called final configurations. For our language of expressions, the final configurations are of the form \(\langle n, \sigma \rangle\) where \(n\) is a number and \(\sigma\) is a store.

We write \(\rightarrow^*\) for the reflexive transitive closure of the relation \(\rightarrow\). That is, if \(\langle e, \sigma \rangle \rightarrow^* \langle e', \sigma' \rangle\), then using zero or more steps, we can evaluate the configuration \(\langle e, \sigma \rangle\) to the configuration \(\langle e', \sigma' \rangle\). Thus, we can write

\[
\langle (\text{foo} + 2) \times (\text{bar} + 1), \sigma \rangle \rightarrow^* \langle 24, \sigma \rangle.
\]