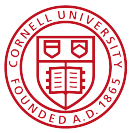
CS 4110 – Programming Languages and Logics
Lecture #2: Introduction to Semantics



What is the meaning of a program? When we write a program, we represent it using sequences of characters. But these strings are just *concrete syntax*—they do not tell us what the program actually means. It is tempting to define meaning by executing programs—either using an interpreter or a compiler. But interpreters and compilers often have bugs! We could look in a specification manual. But such manuals typically only offer an informal description of language constructs.

A better way to define meaning is to develop a formal, mathematical definition of the semantics of the language. This approach is unambiguous, concise, and—most importantly—it makes it possible to develop rigorous proofs about properties of interest. The main drawback is that the semantics itself can be quite complicated, especially if one attempts to model all of the features of a full-blown modern programming language.

There are three pedigreed ways of defining the meaning, or *semantics*, of a language:

- *Operational semantics* defines meaning in terms of execution on an abstract machine.
- *Denotational semantics* defines meaning in terms of mathematical objects such as functions.
- *Axiomatic semantics* defines meaning in terms of logical formulas satisfied during execution.

Each of these approaches has advantages and disadvantages in terms of how mathematically sophisticated they are, how easy they are to use in proofs, and how easy it is to use them to implement an interpreter or compiler. We will discuss these tradeoffs later in this course.

1 Arithmetic Expressions

To understand some of the key concepts of semantics, let us consider a very simple language of integer arithmetic expressions with variable assignment. A program in this language is an expression; executing a program means evaluating the expression to an integer. To describe the syntactic structure of this language we will use variables that range over the following domains:

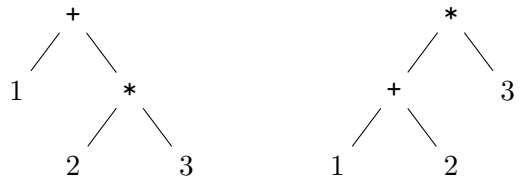
$$\begin{array}{rcl} x, y, z & \in & \textbf{Var} \\ n, m & \in & \textbf{Int} \\ e & \in & \textbf{Exp} \end{array}$$

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

$$\begin{array}{rcl} e & ::= & x \\ & \mid & n \\ & \mid & e_1 \texttt{ + } e_2 \\ & \mid & e_1 \texttt{ * } e_2 \\ & \mid & x \texttt{ := } e_1 \texttt{ ; } e_2 \end{array}$$

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 the value described by e_2.

This grammar specifies the syntax for the language. An immediate problem here is that the grammar is ambiguous. Consider the expression $1 + 2 * 3$. One can build two abstract syntax trees:



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 addition and multiplication expressions:

$$
\begin{aligned}
e ::= \; & x \\
\mid \; & n \\
\mid \; & (e_1 + e_2) \\
\mid \; & (e_1 * e_2) \\
\mid \; & x := e_1 \; ; \; e_2
\end{aligned}
$$

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 its "abstract syntax" (which describes, possibly ambiguously, the structure of program phrases). In this course we will 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 CS 4120.)

1.1 Representing Expressions

The syntactic structure of expressions in this language can be compactly expressed in OCaml using datatypes:

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

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

We could express the same structure in a language like Java using a class hierarchy, although it would be a little more complicated:

```
abstract class Expr { }
class Var extends Expr { String name; .. }
class Int extends Expr { int val; ... }
class Add extends Expr { Expr exp1, exp2; ... }
class Mul extends Expr { Expr exp1, exp2; ... }
class Assgn extends Expr { String var, Expr exp1, exp2; .. }
```

2 Operational semantics

We have an intuitive notion of what expressions mean. For example, the $7 + (4 * 2)$ evaluates to 15, and $i := 6 + 1\,;\; 2 * 3 * i$ evaluates to 42. In this section, we will formalize this intuition precisely.

An *operational semantics* describes how a program executes on an abstract machine. A *small-step* operational semantics describes how such an execution proceeds in terms of successive reductions—here, of an expression—until we reach a value that represents the result of the computation. The state of the abstract machine is often referred to as a *configuration*. For our language a configuration must include two pieces of information:

- a *store* (also known as environment or state), which maps 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 reect assignment of new values to variables,
- the *expression* to evaluate.

We will represent stores as partial functions from **Var** to **Int** and configurations as pairs of expressions and stores:

$$\begin{aligned} \mathbf{Store} &\triangleq \mathbf{Var} \rightharpoonup \mathbf{Int} \\ \mathbf{Config} &\triangleq \mathbf{Store} \times \mathbf{Exp} \end{aligned}$$

We will denote configurations using angle brackets. For instance, $\langle \sigma, (\mathit{foo} + 2) * (\mathit{bar} + 2) \rangle$ is a configuration where σ is a store and $(\mathit{foo} + 2) * (\mathit{bar} + 2)$ is an expression that uses two variables, *foo* and bar. The small-step operational semantics for our language is a relation $\rightarrow \subseteq \mathbf{Config} \times \mathbf{Config}$ that describes how one configuration transitions to a new configuration. That is, the relation $\rightarrow$ shows us how to evaluate programs one step at a time. We use infix notation for the relation $\rightarrow$. That is, given any two configurations $\langle \sigma_1, e_1 \rangle$ and $\langle \sigma_2, e_2 \rangle$, if $(\langle e_1, \sigma_1 \rangle, \langle e_2, \sigma_2 \rangle)$ is in the relation $\rightarrow$, then we write $\langle \sigma_1, e_1 \rangle \rightarrow \langle \sigma_2, e_2 \rangle$. For example, we have $\langle \sigma, (4 + 2) * y \rangle \rightarrow \langle \sigma, 6 * y \rangle$. That is, we can evaluate the configuration $\langle \sigma, (4 + 2) * y \rangle$ one step to get the configuration $\langle \sigma, 6 * y \rangle$.

Using this approach, defining the semantics of the language boils down to to defining the relation $\rightarrow$ that describes the transitions between configurations.

One issue here is that the domain of integers is infinite, as is the domain of expressions. Therefore, there is an infinite number of possible machine configurations, and an infinite number of possible single-step transitions. We need a finite way of describing an infinite set of possible transitions. We can compactly describe $\rightarrow$ using inference rules:

$$\frac{n = \sigma(x)}{\langle \sigma, x \rangle \to \langle \sigma, n \rangle}\ \textsc{Var}$$

$$\frac{\langle \sigma, e_1 \rangle \to \langle \sigma', e_1' \rangle}{\langle \sigma, e_1 + e_2 \rangle \to \langle \sigma', e_1' + e_2 \rangle}\ \textsc{LAdd} \qquad \frac{\langle \sigma, e_2 \rangle \to \langle \sigma', e_2' \rangle}{\langle \sigma, n + e_2 \rangle \to \langle \sigma', n + e_2' \rangle}\ \textsc{RAdd} \qquad \frac{p = m + n}{\langle \sigma, n + m \rangle \to \langle \sigma, p \rangle}\ \textsc{Add}$$

$$\frac{\langle \sigma, e_1 \rangle \to \langle \sigma', e_1' \rangle}{\langle \sigma, e_1 * e_2 \rangle \to \langle \sigma', e_1' * e_2 \rangle}\ \textsc{LMul} \qquad \frac{\langle \sigma, e_2 \rangle \to \langle \sigma', e_2' \rangle}{\langle \sigma, n * e_2 \rangle \to \langle \sigma', n * e_2' \rangle}\ \textsc{RMul} \qquad \frac{p = m \times n}{\langle \sigma, m * n \rangle \to \langle \sigma, p \rangle}\ \textsc{Mul}$$

$$\frac{\langle \sigma, e_1 \rangle \to \langle \sigma', e_1' \rangle}{\langle \sigma, x := e_1 \,;\, e_2 \rangle \to \langle \sigma', x := e_1' \,;\, e_2 \rangle}\ \textsc{Assgn1} \qquad \frac{\sigma' = \sigma[x \mapsto n]}{\langle \sigma, x := n \,;\, e_2 \rangle \to \langle \sigma', e_2 \rangle}\ \textsc{Assgn}$$

The meaning of an inference rule is that if the facts above the line holds, then the fact below the line holds. The fact 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. We use the notation $\sigma[x \mapsto n]$ for the store that maps the variable x to integer n, and maps every other variable to whatever σ maps it to. More explicitly, if f is the function $\sigma[x \mapsto n]$, then we have

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

3 Using the Semantics

Now let's see how we can use these rules. Suppose we want to evaluate the expression $(foo + 2) * (bar + 1)$ with a store σ where $\sigma(foo) = 4$ and $\sigma(bar) = 3$. That is, we want to find the transition for the configuration $\langle \sigma, (foo + 2) * (bar + 1) \rangle$. 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 rule that matches the form of our configuration is LMul, where $e_1 = foo + 2$ and $e_2 = bar + 1$ but e_1' is not yet known. We can instantiate LMul, replacing the metavariables e_1 and e_2 with appropriate expressions.

$$\frac{\langle \sigma, foo + 2 \rangle \to \langle e_1', \sigma \rangle}{\langle \sigma, (foo + 2) * (bar + 1) \rangle \to \langle \sigma, e_1' * (bar + 1) \rangle}\ \textsc{LMul}$$

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 $\langle \sigma, foo + 2 \rangle \to \langle e_1', \sigma \rangle$. We find that LAdd is the only matching rule:

$$\frac{\langle \sigma, foo \rangle \to \langle \sigma, e_1'' \rangle}{\langle \sigma, foo + 2 \rangle \to \langle \sigma, e_1'' + 2 \rangle}\ \textsc{LAdd}$$

We repeat this reasoning for $\langle \sigma, foo \rangle \to \langle \sigma, e_1'' \rangle$ and find that the only applicable rule is the axiom Var:

$$\frac{\sigma(foo) = 4}{\langle \sigma, foo \rangle \to \langle \sigma, 4 \rangle}\ \textsc{Var}$$

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:

$$\dfrac{\dfrac{\dfrac{\sigma(foo) = 4}{\langle \sigma, foo \rangle \to \langle \sigma, 4 \rangle}\ \textsc{Var}}{\langle \sigma, foo + 2 \rangle \to \langle \sigma, 4 + 2 \rangle}\ \textsc{LAdd}}{\langle \sigma, (foo + 2) * (bar + 1) \rangle \to \langle \sigma, (4 + 2) * (bar + 1) \rangle}\ \textsc{LMul}$$

This proves that, given our inference rules, the one-step transition

$$\langle \sigma, (foo + 2) * (bar + 1) \rangle \to \langle \sigma, (4 + 2) * (bar + 1) \rangle$$

is derivable. The structure above 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:

$$\dfrac{\dfrac{6 = 4 + 2}{\langle \sigma, 4 + 2 \rangle \to \langle \sigma, 6 \rangle}\ \textsc{Add}}{\langle \sigma, (4 + 2) * (bar + 1) \rangle \to \langle \sigma, 6 * (bar + 1) \rangle}\ \textsc{LMul}$$

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:

$$\begin{aligned} \langle \sigma, (foo + 2) * (bar + 1) \rangle &\to \langle \sigma, (4 + 2) * (bar + 1) \rangle \\ &\to \langle \sigma, 6 * (bar + 1) \rangle \\ &\to \langle \sigma, 6 * (3 + 1) \rangle \\ &\to \langle \sigma, 6 * 4 \rangle \\ &\to \langle \sigma, 24 \rangle \end{aligned}$$

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 $\langle \sigma, n \rangle$.

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

$$\langle \sigma, (foo + 2) * (bar + 1) \rangle \to^* \langle \sigma, 24 \rangle$$