Introduction

What is a programming language? Let's take the example of arithmetic. We're all used to expressions like $1 + 6 * 3/2$. I want you to think about this as a *program* in a *programming language*. Here, a program is a single *expression* that *evaluates* to a *value* (e.g. the number 10 above). To describe the programming language of arithmetic, we need to talk about its *syntax*, the structure of an arithmetic expression, and its *semantics*, how to evaluate an arithmetic expression.

Syntax

To describe a language's syntactic structure, we often use a grammar, or a set of recursive rules that define allowable shapes for syntactic forms. For example in English, a general grammar rule is subject-verb-object (e.g. "I ate a carrot", not "I a carrot ate"). Here's the grammar for our arithmetic language:

$$\begin{array}{rrl}
\textsf{Integer} & n ::= & \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \\
\\
\textsf{Binop} & \oplus ::= & + \mid - \mid * \mid / \\
\\
\textsf{Expression} & e ::= & n \\
& \mid & e_1 \oplus e_2
\end{array}$$

This is a context-free grammar (see CS 103 Context-Free Grammars to refresh on what "context-free" means) that defines three "kinds" of syntax: integers, binary operators, and expressions. For each kind of syntax, the grammar describes the structure of the syntax in terms of "terminals" (e.g. the number 0, the symbol $*$) and in terms of "nonterminals", indicated by variables (e.g. e_1). The variable name used indicates which syntax kind is being referenced, e.g. e_1 is an expression and n is a number. Syntax can be recursive, as an expression is either a binary operation of two sub-expressions or a number.

You can think about a grammar in two ways: from one direction, a grammar says how to decompose a piece of syntax into its parts. For example, $1 + 6 * 3/2 = e_1 \oplus e_2$ where $e_1 = 1, e_2 = 6 * 3/2, \oplus = +$. Grammars are often used to define *parsers* that map strings into *syntax trees*. One issue with our grammar above is that it is ambiguous—while we know the PEMDAS rule when parsing arithmetic in our heads, that isn't explicitly encoded in the grammar. Parsing and ambiguities are covered in much greater detail in CS 143 (see Introduction to Parsing).

Another way to think about grammars is as a generator—the grammar above inductively defines the set of all arithmetic expressions. Although a given expression is finite, there are an infinite number of expressions. In language theory (which you may recall from CS 103), a language is modeled as a set of strings. Hence, the grammar is a generative definition of the "language" of arithmetic.

The last thing to note is that this method for describing the grammar of a language (using BNF-ish notation) is our first example of *metalanguage*, or a language for describing languages. A grammar is not "the" way to describe syntax, simply a conventional way. An important part of the theory of programming languages is developing a common notation for describing the structure and semantics of programs.

Semantics

While syntax defines the *structure* of a program, we still need to define rules for what to actually do it. Intuitively, we want something that tells us how to get from the expression $1 + 6 * 3/2$ to the expression 10. When we have operations like $6 * 3$, those are *reducible expressions* (or a "redex"), and we want to continually reduce until there are none left. We call this the *semantics* of the programming language. As with syntax, there are many ways to define semantics, and we will choose the conventionally-used *small step operational semantics*. Here's what that looks like:

$$\frac{}{n \ \mathsf{val}} \ (\text{D-Num}) \quad \frac{e_1 \mapsto e_1'}{e_1 \oplus e_2 \mapsto e_1' \oplus e_2} \ (\text{D-Binop}_1) \quad \frac{e_1 \ \mathsf{val} \quad e_2 \mapsto e_2'}{e_1 \oplus e_2 \mapsto e_1 \oplus e_2'} \ (\text{D-Binop}_2) \quad \frac{n_3 = n_1 \bar{\oplus} n_2}{n_1 \oplus n_2 \mapsto n_3} \ (\text{D-Binop}_3)$$

Logic essentials

To understand what's going on here, we need to talk about logic. If you don't remember the essentials of logic from your discrete math course (most importantly quantification, implication, and relations), I highly recommend you review CS 103 Propositional Logic, First Order Logic, and Binary Relations. The first logic tool we need is a *judgment*, or a unary relation. Judgments are relations that assert facts. Above, we use two judgments:

1. e val is a unary relation that reads "e is a value",
2. $e_1 \mapsto e_2$ is a binary relation that reads "e_1 small steps to e_2".

The goal of a system of semantics for our arithmetic language is to define a path from expressions to values. Given an expression like $1 + 6 * 3/2$, we should be able to take small steps until reaching a value (we will formalize this idea of "making progress" next week). The way we define this path from expressions to values is by defining *inference rules* (the things that look like big fractions). An inference rule is exactly the same as an implication arrow, just written vertically.

$$\left(\frac{e_1 \mapsto e_1'}{e_1 \oplus e_2 \mapsto e_1' \oplus e_2}\right) \equiv (e_1 \mapsto e_1' \implies e_1 \oplus e_2 \mapsto e_1' \oplus e_2)$$

This rule reads "when e_1 steps to e_1', then $e_1 \oplus e_2$ steps to $e_1' \oplus e_2$". Note that these logical statements reference variables (e_1 and e_2) that aren't actually defined anywhere. Conventionally, these inferences rules are *implicitly quantified* over all such variables. We could write this explicitly:

$$\forall e_1 . \forall e_2 . \left(\frac{e_1 \mapsto e_1'}{e_1 \oplus e_2 \mapsto e_1' \oplus e_2}\right)$$

Moreover, this isn't actually quantified over all possible objects in the universe, it's quantified over all possible *expressions*, indicated by the name of the variable (the same convention we used in the grammar). Now let's go back and read the rules more closely.

Semantic proofs

$$\frac{}{n \ \mathsf{val}} \ (\text{D-Num}) \quad \frac{e_1 \mapsto e_1'}{e_1 \oplus e_2 \mapsto e_1' \oplus e_2} \ (\text{D-Binop}_1) \quad \frac{e_1 \ \mathsf{val} \quad e_2 \mapsto e_2'}{e_1 \oplus e_2 \mapsto e_1 \oplus e_2'} \ (\text{D-Binop}_2) \quad \frac{n_3 = n_1 \bar{\oplus} n_2}{n_1 \oplus n_2 \mapsto n_3} \ (\text{D-Binop}_3)$$

The first rule says "unconditionally, any number n is a value." The string in parentheses "D-Num" is the name of the rule. This rule collectively defines the terminal state for our programming language—all expressions should become values.

The next three rules define how to evaluate composite expressions with binary operators. The first rule says "when the left expression steps to another expression, then the whole expression steps to a new expression with the left expression changed." The second rule says the same thing, but for the right expression when the left expression is a value.

> Note: multiple conditions above the bar are separated by spaces, and are implicitly and-ed together. So D-Binop_2 says "when e_1 val AND $e_2 \mapsto e_2'$ then the thing below the bar is true.

The last rule says that when the left and right hand side are both numbers, then they step to whatever the result of their operation is. We're using the bar $\bar{\oplus}$ to indicate that the actual operation performed occurs in a different "language" than the source (see the section below for further detail).

We use this funky bar syntax because it simplifies the creation of visual inductive proofs. For example, let's prove that the expression $1 + 6 * 3/2$ evaluates to 10:

$$\dfrac{\dfrac{}{1\ \mathsf{val}}\ (\text{D-Num}) \qquad \dfrac{\dfrac{\dfrac{}{18 = 6 \mathbin{\bar{*}} 3}\ (\text{arithmetic})}{6 * 3 \mapsto 18}\ (\text{D-Binop}_3)}{6 * 3/2 \mapsto 18/2}\ (\text{D-Binop}_1)}{1 + 6 * 3/2 \mapsto 1 + 18/2}\ (\text{D-Binop}_2)$$

This is a logical proof of the statement $1 + 6 * 3/2 \mapsto 1 + 18/2$. To prove it, we start by citing the corresponding rule (D-Binop_2) and prove the two premises of the rule, recursively proving until we reach unconditionally true judgments (e.g. $1\ \mathsf{val}$ or $18 = 6 \mathbin{\bar{*}} 3$. As an exercise, try formulating and proving the next two steps in the evaluation.

Metalanguage

Again, we have defined another set of tools in our metalanguage toolbox. The judgments ($\mathsf{val}, \mapsto$) and the inference rules (horziontal bars) along with our usual set of first-order logic primitives (variables, foralls, existentials) define the language with which we will describe the semantics of other languages. I want to drive this point home because in the discussion of programming languages, it can be quite easy to accidentally mix up multiple languages with similar syntax.

For example, in the rule:

$$\dfrac{n_3 = n_1 \mathbin{\bar{\oplus}} n_2}{n_1 \oplus n_2 \mapsto n_3}\ (\text{D-Binop}_3)$$

We have the predicate $n_3 = n_1 \mathbin{\bar{\oplus}} n_2$. However, we never defined a notion of "equality", or the meaning of any of the arithmetic operators. We were somehow appealing to an *external theory* of numbers and arithmetic to define the individual operators, and our language was simply the *glue* that allowed us to combine expressions of multiple operators together.

Moreover, this means that the equals and $\bar{\oplus}$ operators are not part of the syntax of the programming language we defined, but rather of an external language we used to understand $2 = 1 + 1$. I avoided explicitly defining this language or its semantics today so as to not overcomplicate things, but it's important to understand at least that it is a *different* language than our own.

Peano numerals are an example of a way to potentially define a theory of arithmetic *internal* to our language rather than *external*. You'll see more about Peano numerals on the homework!