Lecture 9: Languages and Interpreters

In 6.009 and its prerequisites, we spend a lot of time getting comfortable with the notations and behavior of particular programming languages. Such comfort is the first stage of enlightenment in computer science. The next stage is developing your own languages and implementing them in other languages! This lecture introduces that concept, including via writing *interpreters* that execute programs in new languages. Most working programmers don't exercise these skills very regularly, but they are essential for many kinds of more advanced development. In any case, the subject provides a good excuse to practice our skills in such topics as recursion and object-oriented design.

This lecture is meant partly to prepare you for 6.009's Lab 8. We will work with a fairly different type of language in lecture, and it will be your job to generalize the lessons to Lab 8's language. This week's tutorial will also be very helpful for the lab, as it goes over some details more specific to the language from the lab.

Warm-Up: A Language of Arithmetic Expressions

You might remember the concept of a *grammar* from a problem on Quiz 1. If not, we'll explain it from scratch. Here's a grammar to define the first, simple language that we implement in this lecture. We write $x$ to stand for a variable, any sequence of one or more letters; and $n$ to stand for a constant, any sequence of one or more digits. We won't be too precise about grammar notation; in 6.009, we just use it as a hint to drive writing of Python code.

$$
\begin{align*}
e &::= e_+ \\
e_+ &::= e_x | e_+ "+" e_x | e_+ "-" e_x \\
e_x &::= e_c | e_x "*" e_c \\
e_c &::= x | n | "(" e_+ ")"
\end{align*}
$$

Each identifier to the left of a “ ::=" defines a category of syntax. To the rights of “ ::="s appear rules explaining the options within the category. We can derive membership of strings in the category by repeatedly *rewriting* with the rules. Here's an example:

$$
\begin{align*}
e \\
e_+
\end{align*}
$$
Notice how the rules for $e_+$ and $e_-$ include a certain kind of asymmetry. It's important that we did things this way, to encode *left associativity*. Also, splitting expressions into several layered categories implements *order of operations*. Try experimenting with changes to the grammar to see how we end up implementing different rules than the usual ones for arithmetic!

Our first implementation task in class is to build a *parser* for this language. The job of the parser is to translate strings into *syntax trees*. We will represent syntax trees using classes, just as in the prior lecture about symbolic algebra. We also apply a pretty dumb, slow parsing style, which takes the grammar quite literally as a code outline. You will learn and implement a better parsing style in Lab 8!

The next implementation task is to build an *interpreter* for the language, which computes the value of an expression, given values for the available variables. We build the interpreter as a method in each of the classes standing for different nodes in syntax trees.

This two-step style turns out to be quite general, as we will see in applying it to further language features.

**Adding Commands**

Let's turn our language into more of a programming language, with commands and control flow. Keep the prior grammar and add/replace these categories:

\[
\begin{align*}
e &::= e_m \\
e_m &::= e_+ | e_+ "=" e_+ | e_+ "<" e_+ \\
c &::= x "=" e | c ";" c | "if" e "{" c "}" "else" "{" c "}" |
\end{align*}
\]

These programming features should be familiar from Python, though we use slightly different concrete syntax.
It turns out that our two-step approach from the last section is fairly easily adapted to this new language, including for running programs that change variable values as they go.

Adding Dictionaries

To make the language more interesting, we can extend some categories with operations similar to Python's on dictionaries.

\[
e_c ::= \ldots \mid \{\}\mid x \cdot x
\]

\[
c ::= \ldots \mid x \cdot x= e
\]

In order, we have empty dictionaries, looking up a field in a dictionary, and assigning to a field in a dictionary.

Some Simple Program Analysis

Sometimes Python programming can be aggravating because programs with obvious bugs are accepted, but we only discover the bugs when running code in certain scenarios.

For instance, we might mention a variable that hasn't been assigned yet. In the final part of lecture, our first exercise is to write some code that can guarantee that a program in our small language never reads a variable that hasn't been assigned yet.

Deeper bugs are possible, however. We might get confused about which variables hold numbers vs. dictionaries, or we might get confused about which fields are present in a dictionary. Our last exercise is to write a simple static type checker that can rule out these bugs, too.

A word of warning: each of our simple program analyses is necessarily incomplete. In follow-on classes like 6.006, you will see how it is actually impossible to implement a fully precise analysis for a language like ours with loops! For the curious, the relevant keyword to look up online is “undecidability.”