CSE 341, Winter 2008, Lecture 1 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

A course titled, "Programming Languages" can mean many different things. For us, it means the opportunity to learn the basics of a couple different *programming paradigms* (fundamental ways to describe and structure a computation) and lots of different *fundamental concepts* (ideas that appear in one form or another in almost every programming language). Many people would say this course will "teach" the 3 languages ML, Scheme, and Ruby, but that is fairly misleading. We will use these languages to learn the paradigms and concepts because they are well-suited to do so. That said, being able to learn new languages and recognize the similarities and differences across languages is an important goal.

The course syllabus, challenge-problem policy, and academic-integrity policy have important information not worth repeating here. The syllabus also has a more complete description of the course goals and advice for being successful — read it.

Most of the course will use *functional programming* (both ML and Scheme are functional languages), which emphasizes immutable data (no assignment statements) and functions, especially functions that take and return other functions. As we will discuss near the end of the course, functional programming does some things exactly the opposite of object-oriented programming but also has many similarities. The conventional thing to do in a first lecture is to motivate the course, which in this case would explain why you should learn functional programming and more generally why it is worth learning different languages, paradigms, and language concepts. We will *delay* this discussion until Lecture 6. It's simply too important to cover when most students are more concerned with getting a sense of what the work in the course will be like, and it's an easier discussion to have after we have built up a few lectures of shared terminology and experience. Motivation does matter; let's take a "raincheck" with the promise that it will be well worth it.

So let's just start "learning ML" but in a way that teaches core programming-languages concepts rather than just "getting down some code that works." Therefore, pay extremely careful attention to the words used to describe the very, very simple code in today's lecture. We are building a foundation that we will expand very quickly over the next few lectures.

An ML program is a sequence of *bindings*. Each binding gets *type-checked* and then (assuming it type-checks) *evaluated*. What type (if any) a binding has depends on a *context*, which is roughly the types of the preceding bindings in the file. How a binding is evaluated depends on an *environment*, which is roughly the values of the preceding bindings in the file.

The only kind of binding we'll consider in this lecture is a *variable binding*, which in ML has this *syntax*:

```
val x = e;
```

Here, `val` is a keyword, `x` can be any variable, and `e` can be any *expression*. We will learn many ways to write expressions. The semicolon is optional in a file, but necessary in the *read-eval-print loop* to let the *interpreter* know that you are done typing the binding.

We now know a variable binding's syntax, but we still need to know how it type-checks and evaluates. Mostly this depends on the expression `e`. To type-check a variable binding, we use the "current context" (the types of preceding bindings) to type-check `e` (which will depend on what kind of expression it is) and produce a "new context" that is the current context except with `x` having type `t` where `t` is the type of `e`. Evaluation is analogous: To evaluate a variable binding, we use the "current environment" (the values of preceding bindings) to evaluate `e` (which will depend on what kind of expression it is) and produce a "new environment" that is the current context except with `x` having the value `v` where `v` is the result of evaluating `e`.

A *value* is an expression that "has no more computation to do", i.e., there is no way to simplify it. As we'll see soon, 17 is a value, but 8+9 is not. All values are expressions, not all expressions are values.

This whole description of what ML programs mean (bindings, expressions, types, values, contexts, environments) may seem awfully theoretical or esoteric, but it's exactly the foundation we need to give precise and concise definitions for several different kinds of expressions. Without further ado, here are several kinds of expressions:

- Integer constants: Syntax is a sequence of digits. Type-checking is type `int` in any context. Evaluation is to itself in any environment (it's a value).
- Addition: Syntax is `e1+e2` where `e1` and `e2` are expressions. Type-checking is type `int` but only if `e1` and `e2` have type `int` (using the same context). Evaluation is to evaluate `e1` to `v1` and `e2` to `v2` (using the same environment) and then produce the sum of `v1` and `v2`.
- Variables: Syntax is a sequence of letters, underscores, etc. Type-checking is to look up the variable in the context and use that type. Evaluation is to look up the variable in the environment and use that value.
- Conditionals: Syntax is `if e1 then e2 else e3` where `e1`, `e2`, and `e3` are expressions. Type-checking under a context is to type-check `e1`, `e2`, and `e3` under the same context. `e1` must have type `bool`. `e2` and `e3` must have the same type, call it `t`. Then the type of the whole expression is `t`. Evaluation under ane environment is to evaluate `e1` under the same environment. If the result is `true`, the result of evaluating `e2` under the same environment is the overall result. If the result is `false`, the result of evaluating `e3` under the same environment is the overall result.
- Boolean constants: ...
- Less-than comparison: ...

When using the read-eval-print loop, it's very convenient to add a sequence of bindings from a file. `use "foo.sml"` does just that. Its type is `unit` and its result is `()` (the only value of type `unit`), but its effect is to extend the context and environment with all the bindings in the file `"foo.sml"`.

Bindings are *immutable*. Given `val x = 8+9;` we produce an environment where `x` maps to 17. In this environment, `x` will *always* map to 17; there is no "assignment statement" in ML for changing what `x` maps to. That is very useful if you are using `x`. You *can* have another binding later, say `val x = 19;`, but that just creates a *different environment* where the later binding for `x` *shadows* the earlier one. This distinction will be extremely important when we define functions that use variables.

While we haven't even learned enough of ML yet to really think of it as a programming language (need one more lecture for that at least), we have enough to list the essential "pieces" necessary for defining and learning a programming language:

- Syntax: How do you write the various parts of the language?
- Semantics: What do the various language features mean? For example, how are expressions evaluated?
- Idioms: What are the common approaches to using the language features to express computations?
- Libraries: What has already been written for you? How do you do things you couldn't do without library support (like access files)?
- Tools: What is available for manipulating programs in the language (compilers, read-eval-print loops, debuggers, ...)