CSE341, Fall 2011, Lecture 1 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

The course syllabus, challenge-problem policy, and academic-integrity policy have important information not repeated here. The syllabus also describes the course goals and advice for being successful — read it.

A course titled, "Programming Languages" can mean many different things. For us, it means the opportunity to learn the *fundamental concepts* that appear in one form or another in almost every programming language. We will also get some sense of how these concepts "fit together" to provide what programmers need in a language. And we will use different languages to see how they can take complementary approaches to representing these concepts. All of this is intended to make you a better software developer, in any language.

Many people would say this course "teaches" the 3 languages ML, Racket, and Ruby, but that is fairly misleading. We will use these languages to learn various paradigms and concepts because they are well-suited to do so. If our goal were just to make you as productive as possible in these three languages, the course material would be very different. That said, being able to learn new languages and recognize the similarities and differences across languages is an important goal.

Most of the course will use *functional programming* (both ML and Racket are functional languages), which emphasizes immutable data (no assignment statements) and functions, especially functions that take and return other functions. As we will discuss later in the course, functional programming does some things exactly the opposite of object-oriented programming but also has many similarities. Functional programming is not only a very powerful and elegant approach, but learning it helps you better understand all styles of programming.

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 "rain-check" 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. Do not *yet* try to relate what you see back to what you already know in other languages as that is likely to lead to struggle.

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 *static environment*,[1] which is roughly the types of the preceding bindings in the file. How a binding is evaluated depends on a *dynamic environment*, which is roughly the values of the preceding bindings in the file. When we just say *environment*, we usually mean *dynamic environment*. Sometimes *context* is used as a synonym for *static environment*.

There are several kinds of bindings, but the only one 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

[1] The word *static* here has a tenuous connection to its use in Java/C/C++, but too tenuous to be worth explaining at this point.

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 (how to write it), but we still need to know its *semantics* (how it type-checks and evaluates). Mostly this depends on the expression `e`. To type-check a variable binding, we use the "current static environment" (the types of preceding bindings) to type-check `e` (which will depend on what kind of expression it is) and produce a "new static environment" that is the current static environment 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 dynamic environment" (the values of preceding bindings) to evaluate `e` (which will depend on what kind of expression it is) and produce a "new dynamic environment" that is the current environment 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 described more generally below, `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, 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: a sequence of digits
  - Type-checking: type `int` in any static environment
  - Evaluation: to itself in any dynamic environment (it is a value)
- Addition:
  - Syntax: `e1+e2` where `e1` and `e2` are expressions
  - Type-checking: type `int` but only if `e1` and `e2` have type `int` in the same static environment, else does not type-check
  - Evaluation: evaluate `e1` to `v1` and `e2` to `v2` in the same dynamic environment and then produce the sum of `v1` and `v2`
- Variables:
  - Syntax: a sequence of letters, underscores, etc.
  - Type-checking: look up the variable in the current static environment and use that type
  - Evaluation: look up the variable in the current dynamic environment and use that value
- Conditionals:
  - Syntax is `if e1 then e2 else e3` where `e1`, `e2`, and `e3` are expressions
  - Type-checking: using the current static environment, a conditional type-checks only if (a) `e1` has type `bool` and (b) `e2` and `e3` have the same type. The type of the whole expression is the type of `e2` and `e3`.
  - Evaluation: under the current dynamic environment, evaluate `e1`. If the result is `true`, the result of evaluating `e2` under the current dynamic environment is the overall result. If the result is `false`, the result of evaluating `e3` under the current dynamic environment is the overall result.
- Boolean constants:
  - Syntax: either `true` or `false`

  - Type-checking: type `bool` in any static environment
  - Evaluation: to itself in any dynamic environment (it is a value)

- Less-than comparison: ...
  - Syntax: `e1 < e2` where `e1` and `e2` are expressions
  - Type-checking: type `bool` but only if `e1` and `e2` have type `int` in the same static environment, else does not type-check
  - Evaluation: evaluate `e1` to `v1` and `e2` to `v2` in the same dynamic environment and then produce `true` if `v1` is less than `v2` and `false` otherwise

Whenever you learn a new construct in a programming language, you should ask these three questions: What is the syntax? What are the type-checking rules? What are the evaluation rules?

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 include all the bindings in the file `"foo.sml"`.

Bindings are *immutable*. Given `val x = 8+9;` we produce a dynamic 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 ML yet to really think of it as a programming language (we need at least one more lecture for that), we can still 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, ...)