CSE341, Fall 2011, Lecture 7.5 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.

We now fulfill our promise to motivate the topics covered in this course now that we have enough shared experience to have a productive conversation.

We can break the question of, "what is this course good for" into four parts:

1. Why should we learn programming languages other than popular industry ones like Java, C, C++, Perl, etc.?
2. Why should we learn fundamental concepts that appear in most programming languages, rather than just learning particular languages?
3. Why should we focus on languages that encourage (mostly) *functional programming* (i.e., that discourage mutation, encourage recursion, and encourage functions that take and return other functions)?
4. Why are we using SML, Racket, and Ruby as complements to your Java experience (remembering that the course is more about the concepts than the languages)?

The answers are numerous. Some may resonate with you more than others. The following discussion is necessarily incomplete.

Why Different Languages

One good analogy is with automobiles. There will never be a best programming language much as there will never be a best car. Different cars serve different purposes: some go fast, some can go off-road, some are safer, some have room for a large family, etc. Yet there are remarkable similarities and while drivers or auto mechanics may have preferences and specialties, they learn enough fundamental principles to work with new kinds of cars easily. Still, it can be uncomfortable to switch cars, as anyone who has ever had trouble finding the windshield wipers in a friend's car can attest.

When learning automotive principles, it is probably helpful to start with simple and elegant cars where each piece has a clear and simple purpose rather than an "industrial-strength" car with features that have been added over many years.

Are All Languages The Same?

You should also know that in a very precise sense all programming languages are equally powerful: If you need to write a program that takes some input X and produces output Y, there is *some* way to do it Java or ML or Perl or a ridiculous language where you have only 3 variables and 1 while loop. The equal expressiveness is basically the Church-Turing Thesis, a core topic in courses on the theory of computation such as CSE431. But just because there is always *some way* to implement your program, does not mean it is easy, clear, or robust. A related concept is the, "Turing tarpit" where a language's powerful features are lauded for what they can do, but it remains difficult to use them (like moving in a tarpit). Arguing over "which language is better" in terms of which has more useful features is also often unilluminating since there is always some way to get the job done in your favorite language.

The similarities among languages we focus on in this course have more to do with fundamental language concepts, such as variables, abstraction, support for one-of types, recursive definitions, etc. We won't repeat those here because that would require repeating most of the course!

So are all languages really the same and picking one just a matter of personal preference? No, that would take things too far. A good analogy might be different human cultures: On many levels, "people are people" and there are universal experiences in the human condition. Yet there are also fascinating differences among cultures and communities: their customs, their language, their values, and so on. In fact, one of the best ways to learn more about your own culture (maybe Java or C) is to immerse yourself in other cultures (maybe ML). In fact, you may bring experiences "back home" that make you a better and happier person.

In terms of programming languages, in addition to syntactic differences, often the most important differences is that what is "primitive" and really easy in one language is awkward in another. For example, returning a pair in Java is annoying. Conversely, setting up the equivalent of subclasses in ML is not very pleasant.

Reality and Why We Ignore Most of It

It is also fair to point out that when choosing a language for a software project, whether the language is an elegant design that is easy to learn and useful for writing correct, concise code is only one consideration. In the real world, it also matters what libraries are available, what your boss wants, and whether you can hire enough developers to do the task. We have the luxury in the classroom of ignoring these issues to focus on the fundamental truths underlying programming languages. Doing so is important, so it is fine to ignore other important real-world issues.

Why is precisely defining the semantics and idioms of a programming language so important? Because there is no other way to reason about what your software is doing: If you do not know the language definition you are stuck with vague notions about "what this code might mean." This is a horrible recipe for software development. Only with semantics can we resolve issues like whether a library user or a library implementor is at fault for a bug.

More generally, much of software development is about designing interfaces and explaining as precisely as possible how they should be used. A programming language is one such interface: It takes a *program* and returns an *answer*. So it is a really good example of an interface needing a precise definition.

While it is unlikely you will be involved in designing a new general-purpose programming language like Java, ML, or C++, it is surprisingly likely that you will end up designing a smaller new language for some specific project. This happens all the time — whenever some application wants a way for users to extend its functionality. Editors (like emacs), game engines (like Quake), CAD tools (like AutoCAD), desktop software (like Microsoft Office), and web browsers are all examples (corresponding languages include elisp, JavaScript, QuakeC, etc.). Seeing a range of programming languages and understanding their essential design is invaluable.

Finally, there is a place at universities to learn beautiful works of art that teach us about the universe and enrich us as people. Elegant programming languages are such works of art, just like the play Hamlet. Educated citizens should know SML and Hamlet – even though they both have strange syntax, are not the most modern and popular languages / plays (or movies), and may not help you get a summer internship.

Why Functional Languages

Having covered just a few reasons to study programming languages in general, we can focus on why to focus on a functional languages like ML. The main reason is that it has many features that encourage a programming style that is invaluable for writing correct, elegant, and efficient software. It develops a way of thinking about computation that will make you a better programmer even in other languages. Specific examples are what the course notes are all about, so rather then repeat topics like function closures and deep pattern-matching here, instead let's purposely "brag" about the important role functional languages have played in the past and are likely to play in the future.

One sometimes hears functional languages dismissed as "slow, worthless, beautiful things you have to learn in school." However, they tend to teach exactly the language constructs and concepts that are useful but "ahead of their time." Students in programming-languages courses learned about garbage collection (not having to manage memory manually), generics (like Java's `List<T>` type), universal data representations (like XML), function closures (as in Python, Ruby, and JavaScript), type inference (C#), etc. many years before they were in widely popular languages. One way to think about it is that functional programming has not "conquered" the programming world, but *many* of its features have been "assimilated" and are now widely promoted without functional languages getting much credit. Here are some examples:

- The difference between C# 2.0 and C# 3.0 is largely support for functional-programming features and other ML-like conveniences (e.g., type inference)
- Java currently has a preliminary proposal for closures in a future version.
- Now that desktop computers are getting parallel processors, more software and languages will encourage not mutating data, since this makes it much more difficult to do things in parallel.

So it is reasonable to think other ideas like pattern-matching, currying, or hygienic macros might also eventually achieve assimilation.

We should also mention MapReduce, the Google system for large-scale fault-tolerant data processing on computer clusters, and the open-source variant Hadoop. These systems are now used throughout big (and small) business and increasingly much of science to analyze massive data sets. To quote the introduction of the original 2004 paper describing MapReduce (http://labs.google.com/papers/mapreduce.html), "we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages." This is probably the most successful use of the core concept of using higher-order functions to separate the traversal of data from how to process data elements.

Are Functional Languages Used for Anything?

Functional languages are useful for much more than teaching students and influencing more popular languages. While the amount of software written in functional languages, remains a small percentage of all the software out there, it is certainly more than zero! Moreover, functional programming has seen a real surge in interest and adoption in the last few years. Here we just sketch some high-points.

Admittedly, SML as a particular language, has not seen much "real world" use in the last few years (it is older), and Racket certainly has many interesting projects doing real things like web servers, but it remains in a "niche" community with a large focus on education. Ruby (which is more OO than functional, though the lines are always fuzzy) is a very popular language in large part because of frameworks for writing web applications (and this course will not go anywhere near such features). But there are many other functional languages, OO+functional languages, and projects using functional ideas where there is a lot of current investment: real companies building real products. Here we organize brief highlights by language, listed alphabetically and omitting many wonderful languages and projects:[1]

- Erlang, http://www.erlang.org: Erlang is a functional language originally developed for telecommunications infrastructure. It is well-suited for *distributed programs*, programs running on many computers that may be physically separate and may fail. It has enjoyed popularity for various programs running on the web, notably the chat program in Facebook.
- F#, http://tryfsharp.org: F# is a dialect of ML (i.e., it is a lot like ML with different syntax, extra features, and a few restrictions) that runs on Microsoft's .Net platform and is part of Visual Studio 2010. It is *fully interoperable* with other .Net languages like C# and Visual Basic, so an application can be written in multiple languages, calling libraries written in others. A typical choice is to program the core algorithms of an application in F# while leaving the graphical interface in C#. For what it is worth, here is one "case study" published by Microsoft:
  http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?casestudyid=4000006794

[1]It should also be clear that the goal is to provide interesting links without implying any endorsement of any product or company.

- Haskell, http://www.haskell.org: Haskell is a cutting-edge functional language enjoying increased popularity (I heard recently that new versions of the Glasgow Haskell compiler get downloaded about 200,000 times). It has higher-order functions and pattern-matching like ML, but it is also substantially different: It is *pure* (the only mutation is in an outer layer kept separate using things called monads), it is *lazy* (we will discuss lazy evaluation a bit in the context of Racket), and it has *type classes* (which help make code more reusable). As for industrial use, see http://haskell.org/haskellwiki/Haskell_in_industry, which lists almost 50 companies, big and small, that have reported on their use of Haskell.

- OCaml, http://caml.inria.fr: OCaml is a dialect of ML almost as old as SML and much older than F#. Like SML, it has been used for many, many research projects and compilers as well as useful open-source and commercial programs. Probably the largest but by no means only commercial user is a New York finance company, as described in this recent article: http://queue.acm.org/detail.cfm?id=2038036

- Scala, http://www.scala-lang.org: Scala is a general-purpose language that is enjoying a lot of increased popularity. It is fully interoperable with Java and claims increased productivity, largely by adding to Java functional programming as well as using functional programming to make concurrent programming easier. Particularly well-known users are Twitter, LinkedIn, and FourSquare.

Another good general place to read about cutting-edge commercial use of functional programming is the web-site for the annual conference on, well, Commercial Users of Functional Programming, http://cufp.org. See in particular abstracts and videos from previous years' events.

While the information above focused on industrial use outside of the programming-languages community, functional languages remain exceedingly well-suited to writing tools related to languages themselves, such as compilers, interpreters, theorem provers, code-analysis tools, etc.

Static vs. Dynamic and Object-Oriented vs. Functional

To focus a bit more on the actual organization and topics in this course, we can ask why we are learning SML, Racket, and Ruby. They all have higher-order functions. Plus, each has several features that help us learn essential language concepts:

- ML has parametric polymorphism, which is complementary to OO-style subtyping, a rich module system for abstract types, and rich pattern-matching. OCaml and F# would work just about as well.

- Scheme has dynamic typing, hygienic macros, fascinating control operators (though we may skip this), and a minimalist design.

- Ruby has classes but not types, a more complete commitment to OO than Java, and mixins.

Were the course longer we would also investigate Haskell (a pure, lazy functional language with type classes and monads) and perhaps Prolog (a logic language with unification and backtracking).

We will also compare and contrast functional and object-oriented programming styles (two major high-level language paradigms), as well as dynamic versus static typing. These are orthogonal issues and after this course you will have seen one language with each combination:

	dynamically typed	statically typed
functional	Scheme	SML
object-oriented	Ruby	Java

The benefits and limitations of static typing are a “big idea” in the course, but we have not gotten there yet.