A Concepts-Focused Introduction to Functional Programming
Using Standard ML

Motivation

Dan Grossman

DRAFT of December 10, 2010

There are many ways to motivate the study of programming languages in general and functional programming in particular. The arguments below are the author’s personal take on the issue. The author has also helped write the document, “Why Undergraduates Should Learn the Principles of Programming Languages,” currently available at http://www.cs.pomona.edu/kim/why.pdf. That document is more general in focus and aimed at faculty. The notes below are aimed at second-year undergraduate students just learning functional programming. While one might naturally motivate material at the beginning (“lecture 1”), it actually works better to wait until students have a little idea what functional programming actually is.

Looking at the course notes for, “A Concepts-Focused Introduction to Functional Programming Using Standard ML,” we can ask, “What is learning this material good for?” Here are three more specific questions:

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)?

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

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. An elegantly designed automobile is, to some, a work of beauty, just like Hamlet is an elegant archetype of a tragic play. Elegant designs — in engineering and English and programming languages — are worth studying because they make you a better person, whether or not you put them on your resume.

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 only have 3 variables and 1 while loop. This equality is often referred to as the “Turing tarpit” because Alan Turing first advanced the thesis that (roughly speaking) all languages were equally powerful and “tarpit” because a tarpit is somewhere you get stuck (in this case, making arguments that some language is great because it can do everything you would want it to).

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 write correct, concise code in 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. Moreover, learning how to describe what a program means is very important — there is often no other way to resolve an argument about whether a library user or library implementor made a mistake, for example.

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.

Though new general-purpose languages do not achieve wide popularity very often (perhaps once a decade), it does happen. Students who took studied programming languages in the early 1990s were probably better prepared to learn Java after it was invented than students who studied only C.

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 type), universal data representations (like XML), function closures (as in Python, Ruby, and JavaScript), 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. So it is reasonable to think ideas like pattern-matching or currying might be next.

In fact, in recent years, there has been a surprising amount of widespread interest in core ideas of functional programming. Microsoft recently announced it would fully support the language F# on its .NET platform. F# is a dialect of ML. Similarly, 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. 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. An extreme example is Google’s MapReduce, which is revolutionizing data-center computing. MapReduce did not exist until the early 2000s and is essentially restricted higher-order functional programming built on top of a fault-tolerant distributed-computing infrastructure.

Beyond these big recent moves, there is actual use of functional languages outside of courses. Several small companies (and groups in large companies) consider it their “secret to success.” It is also common to use languages like ML in research projects, even in areas of computer science other than programming languages. A good place to look for examples is the annual meeting called, “Commerical Users of Functional Programming,” http://cufp.org. Presentations from previous conferences show a wide variety of functional languages used in many settings. To be honest, Standard ML is not a frequent language-of-choice. But other functional languages, including Clojure, Erlang, F#, Haskell, Lisp, Objective Caml, and Scheme are much easier to learn after you have experience in a functional language. The course notes could be easily adapted to most of these languages, although some of them have less support for pattern-matching.

The biggest thing missing from these notes, but which would likely be included in a full course on programming languages, is experience with one or more dynamically typed languages. Since Java and ML are both statically typed, they form a useful contrast for functional programming and object-oriented programming. Whether a language is dynamically typed or not is an entirely orthogonal dimension: there are dynamically typed functional languages (e.g., Scheme) and dynamically typed object-oriented languages (e.g., Smalltalk and Ruby). Hopefully an introduction to functional programming is only the beginning of your programming-languages study!