1 Course motivation

Now that we have enough shared experience to have a productive conversation, we fulfill our promise to motivate this course. Here are four questions you might have about what this course is “good for”:

- Why should we learn programming languages other than popular industry ones like Java, C, C++, Perl, etc.?
- Why should we learn fundamental concepts of programming languages, rather than particular languages?
- 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)?
- Why are we using SML and Ruby?

To answer these questions, we make several observations below. Some of those observations might resonate more with you than others.

Observation 1: Programming languages are like cars, cultures, and works of art.

Consider the automobile. Much as there will never be a “best” car, there will never be a best programming language. 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 technical similarities between almost all cars. So drivers generally manage to drive new cars without tremendous difficulty. It can nonetheless be uncomfortable to switch cars—as anyone who has ever had trouble finding the windshield wipers in a friend’s car can attest. Likewise, auto mechanics might have preferences and specialties, but they learn enough fundamental principles to work with new kinds of cars easily. 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.

Consider 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). You will bring experiences “back home” that make you a better and happier person.

Finally, consider art. There is a place at universities to learn beautiful works of art. Elegant programming languages are such works of art, just like Shakespeare’s Hamlet. Educated citizens should know SML and Hamlet. True, both have strange syntax, are neither especially modern nor popular, and they aren’t likely to land you a summer job. But they teach us about the universe and enrich us as people.

Observation 2: Programming languages are tools.

If you need to write a program that takes some input X and produces output Y, there exists some way to do it in Java or ML or Perl or a ridiculous language where you have only 3 variables and 1 while loop. That equal expressiveness is basically Turing completeness, a core topic in courses on the theory of computation such as CSci 3313. So it’s not useful to argue about which language is more powerful. Arguing about which language is “better” in terms of useful features is also often unilluminating, since there always exists some way to get the job done in your favorite language.

But just because there is some way to implement a program does not mean that way is easy, clear, or robust. Besides syntactic differences (which are boring yet frequently debated), often the most important difference is
that what is “primitive” and really easy in one language can be awkward in another. For example, returning a tuple in Java is annoying. Conversely, setting up the equivalent of subclasses in ML is not very pleasant. Choosing the right tool for the job requires knowing which tools are good for which tasks.

**Observation 3: Semantics do matter.**

When choosing a language for a software development project in the real world, there are many important criteria: what libraries are available, what your boss wants, and whether you can hire enough competent developers to do the task. But in the classroom we have the luxury of ignoring these issues to focus on the fundamental ideas underlying programming languages. In this class, we specifically focus on semantics (and idioms).

Why is precisely defining the semantics of a programming language so important? Because there is no better way to reason about what software does: 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. 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 a result. So it is a really good example of an interface needing a precise definition. Libraries are another such interface: users and implementors must agree on the semantics of the library. Only with semantics can we resolve issues like whether a library user or a library implementor is at fault for a bug.

It’s unlikely you will be involved in designing a new general-purpose programming language like Java, ML, or C++. But it’s surprisingly likely that you will end up designing a smaller domain specific language for some specific project. This happens 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; their corresponding languages include elisp, JavaScript, QuakeC, etc. So learning a variety of programming languages and understanding their design and semantics will inform you when you are called upon to design some small language.

**Observation 4: Functional languages predict the future.**

ML 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. Unfortunately, functional languages are sometimes dismissed as “slow, worthless, beautiful things you have to learn in school.”

On the other hand, function languages have an excellent track record of incorporating exactly the language constructs and concepts that are useful but ahead of their time. Students of functional programming 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 adopted in mainstream 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 two 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 8 will have higher-order and anonymous functions.
- Google’s MapReduce paradigm for large-scale fault-tolerant data processing on computer clusters (and its open-source variant Hadoop) is “inspired by the map and reduce primitives present in Lisp and many other functional languages.”

So it is reasonable to expect that other ideas currently incubating in functional languages will one day make their way into mainstream languages. In fact, now that desktop computers have multicore parallel processors,
software and languages are likely to encourage immutability, because mutation makes parallel computation more difficult.

**Observation 5: Functional languages are actually used in the real world.**

Functional languages are useful for much more than teaching students and influencing mainstream languages. Although the amount of software written in functional languages remains a small percentage of all the software out there, it is certainly more than zero! Real companies are building real products with functional languages. Here are some highlights, listed alphabetically and omitting many excellent languages and projects:

- **Erlang, [http://www.erlang.org](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 app in Facebook.

- **F#, [http://tryfsharp.org](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 one languages and use 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](http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?casestudyid=4000006794)

- **Haskell, [http://www.haskell.org](http://www.haskell.org)**: Haskell is a cutting-edge functional language enjoying increased popularity. 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 a feature called monads), it is lazy (or at least non-strict, meaning function arguments are not evaluated until needed), and it has type classes (which help make code more reusable). As for industrial use, see [http://haskell.org/haskellwiki/Haskell_in_industry](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](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 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](http://queue.acm.org/detail.cfm?id=2038036)

- **Scala, [http://www.scala-lang.org](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 functional programming feature to Java 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 real-world use of functional programming is the website for the annual conference on Commercial Users of Functional Programming, [http://cufp.org](http://cufp.org).

**Observation 6: SML and Ruby are exemplars.**

Why pick SML and Ruby as particular objects of study?

- SML has pattern matching, type inference, a module system for abstract types (which we’ll see soon), and parametric polymorphism, which is complementary to OO-style subtyping. We could also use either OCaml or F#.

- Ruby has OO features that go beyond Java, but also includes some functional features.

---

2It should also be clear that the goal is to provide interesting links without implying any endorsement of any product or company.
Were the course longer we would also investigate Racket (which has dynamic typing, hygienic macros, fascinating control operators, and a minimalist design), Haskell (a pure, lazy functional language with type classes and monads) and Prolog (a logic programming language with unification and backtracking).

As the semester progresses, we will compare and contrast functional and object-oriented programming styles, as well as dynamic typing versus static typing (is type checking performed at compile time or at run time?). These are orthogonal issues. After this course you will have seen almost every combination:

<table>
<thead>
<tr>
<th>functional</th>
<th>dynamically typed</th>
<th>statically typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>object-oriented</td>
<td>Ruby</td>
<td>SML</td>
</tr>
</tbody>
</table>

Racket and Scheme are examples of languages that would fit in the functional, dynamically typed paradigm.

2 Infix operators

In the definition of lists,

```
datatype 'a list = nil | :: of 'a * 'a list
```

the list cons operator :: is written infix as h::t rather than prefix as ::(h,t). This creates pleasant syntax, but you might be wondering why :: is special. Why can’t you make your own infix operators?

In fact, you can. For example, you could make your own plus:

```
fun plus (x,y) = x+y
infix plus
val seven = 3 plus 4
```

This is a silly example; plus would only obfuscate your code.

So why can we use :: as infix? Exactly because the ML standard library includes the following code:

```
datatype 'a list = nil | :: of 'a * 'a list
infix ::
```

And that’s all there is to it—there’s nothing magical about the cons operator. The same is true of append operator @.

After an operator is declared as infix, if you want to use it as a normal identifier again, put keyword op in front of it. For example, here’s a better way to declare your own plus:

```
val plus = op +
infix plus
val seven = 3 plus 4
```

3 Type constructors

What is list? It is not a type; you cannot write a function the REPL will agree has type list->int, for example. But you can easily write functions that have type int list -> int or string list -> string, etc. And we’ve seen many examples of polymorphic types involving list:
It turns out that `list` is actually a type constructor, something that makes a type out of another type. (Don't confuse this with a datatype constructor, which makes a value of a datatype out of values of other types.) You could even think of `list` as a function that takes a type and returns a type. So `list` isn't itself a type, but `int list` is a type, `'a list` is a type, `(int->int) list` is a type, etc.

Similarly, `option` is a type constructor. So `int option`, `'a option`, etc. are types, but `option` is not itself a type: you can never write an expression that the REPL will agree has type `option`.

It would be a poor language design if programmers couldn't create their own new type constructors. After all, if a feature is useful for built-in features like lists, it is useful for programmer-defined features. Indeed, we can define our own type constructors in ML: we just use a `datatype` binding, explicitly giving one or more type variables, which can be used in the types of the datatype constructors. (Reread that sentence carefully: it contains several precise uses of the words “type” and “constructor” that you need to understand.)

Example 1. Stacks implemented with lists.

```
datatype 'a stack = S of 'a list
val empty_stack = S []
fun push (x, S s) = S (x::s)
fun top (S []) = raise Empty
  | top (S (x::s)) = x
fun pop (S []) = raise Empty
  | pop (S (_::s)) = S s
```

As usual, `stack` is not a type, but `t stack` is a type for any type `t`—that is, we can instantiate type variable `'a` with any type we like. Our use of datatype constructor `S` here is rather artificial; we don't really need an each-of type to represent stacks-as-lists. But ML's only mechanism for type constructors is datatypes, so we have to invent a datatype constructor to represent “the stack”. The usual convention in this scenario is to choose a very short datatype constructor name, as we did with `S`.

Example 2. Maps implemented with lists. A map is a data structure that associates values with keys. Maps are also known as dictionaries. Hash tables are but one way to implement maps. Note: don't confuse the map data structure we're discussing here with the higher-order function named maps.

```
datatype (''k,'v) map = M of (''k * 'v) list
exception NotFound;
val empty_map = M []
fun put (k, v, M m) = M ((k,v)::m)
fun exists (k, M []) = false
  | exists (k, M ((k', v)::m)) = if k=k' then true else exists(k, M m)
fun get (k, M []) = raise NotFound
  | get (k, M ((k',v)::m)) = if k=k' then v else get(k, M m)
```

This example shows how to create a type constructor that uses multiple type variables—that is, a type constructor that takes in multiple types and produces a single new type. As usual, `map` is not a type, but `(t1,t2) map` is a type for any types `t1` and `t2`. To map integers to strings, for example, we could use an `(int,string) map`. Note that we made type variable `'''k` for map keys an equality type variable by using two single-quote characters in its name. We did that because our `get` and `exists` functions need to compare
keys for equality. But since we never need to compare values for equality, we use a standard (non-equality) type variable \( v \) for values.

**Type constructors that aren’t datatypes.** There are a few type constructors in ML that aren’t actually datatypes. For example, \( \rightarrow \) is a type constructor that takes two types \( t_1 \) and \( t_2 \) and produces the function type \( t_1 \rightarrow t_2 \). Note that \( \rightarrow \) itself is not a type. Similarly, \( * \) is a type constructor that takes two types \( t_1 \) and \( t_2 \) and produces the tuple type \( t_1 * t_2 \). Again, note that \( * \) itself is not a type.