CSE 341, Spring 2008, Lecture 6 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.

This lecture covers two topics:

- It completes our investigation of pattern-matching by giving the full definition of pattern-matching, which allows for nested patterns (patterns inside patterns). It shows several examples where nested pattern-matching is particularly elegant.
- It motivates why one should study programming languages, particularly functional programming languages. Most courses have this discussion in the first lecture, but it can make much more sense after having seen some basic functional programming.

So far, we have said all patterns look like `C1` or `C2(x1,...,xn)` or `{f1=x1,...,fn=xn}` or `(x1,...,xn)` where `C1` and `C2` are constructors and the `fi` are field names. These *are* all patterns, and we have learned *when* they match a value and *what* bindings they introduce when the match is successful. However, it turns out that where we have been putting variables in our patterns we can also put other patterns, which leads to an elegant recursive definition of pattern-matching. Here are some parts of the recursive definition:

- A variable pattern (`x`) matches any value `v` and introduces one binding (from `x` to `v`).
- A wildcard pattern (`_`) matches any value `v` and introduces no bindings.
- The pattern `C` matches the value `C`, if `C` is a constructor that carries no data.
- The pattern `C p` where `C` is a constructor and `p` is a pattern matches a value of the form `C v` (notice the constructors are the same) if `p` matches `v` (i.e., the nested pattern matches the carried value).
- The pattern `(p1,p2)` matches a pair of values `(v1,v2)` if `p1` matches `v1` and `p2` matches `v2`.
- ...

This recursive definition extends our previous understanding in two interesting ways. First, for a constructor `C` that carries multiple arguments, we do not have to write patterns like `C(x1,...,xn)` though we often do. We could also write `C x`; this would bind `x` to the tuple that the value `C(v1,...,vn)` carries. What is really going on is that all constructors take 0 or 1 arguments, but the 1 argument can itself be a tuple. So `C(x1,...,xn)` is really a nested pattern where the `(x1,...,xn)` part is just a pattern that matches all tuples with n parts.

Second, and more importantly, we can use nested patterns instead of nested case expressions when we want to match only values that have a certain "shape." For example, the pattern `x::(y::z)` would match all lists that have at least 2 elements, whereas the pattern `x::[]` would match all lists that have exactly one element. Both examples use one constructor pattern inside another one.

The nested patterns can be for different types. For example, the pattern `(_,(_,x),_)::tl` would match a non-empty list where each element was a triple where the second element was a pair. It would bind the list's head's middle element's second element to `x` and the list's tail to `tl`.

We considered several examples where nested patterns lead to elegant code, including "zipping" or "unzipping" multiple lists, seeing if a list is sorted, and coding a table of outputs by matching against a tuple of inputs. This silly example of the last idiom computes the sign that would result from multiplying two numbers (without actually doing the multiplication).

```
datatype sign = P | N | Z

fun multsign (x1,x2) =
  let fun sign x = if x=0 then Z else if x>0 then P else N
  in
```

```
      case (sign x1,sign x2) of
          (Z,_) => Z
        | (_,Z) => Z
        | (P,P) => P
        | (N,N) => P
        | _     => N
  end
```

Ending a case-expression with the pattern `_`, which matches everything, is somewhat controversial style. The type-checker certainly won't complain about an inexhaustive match, which can be bad if you actually did forget some possibly that you are accidentially covering with your "default case." So more careful coding of the example above would replace the last case with two cases with patterns `(P,N)` and `(N,P)`, but the code as written above is also okay.

Returning to how general pattern-matching is, it turns out that every val-binding and function argument is really a pattern. So the syntax is actually `val p = e` and `fun f p = e` where `p` is a pattern. In fact, the textbook and many ML programmers use a form of function-binding where you can have multiple cases by repeating the function name and using a different pattern:

```
fun f p1 = e1
|   f p2 = e2
...
|   f pn = en
```

This is just syntactic sugar for:

```
fun f x =
   case x of
     p1 => e1
   | p2 => e2
...
   | pn => en
```

Your instructor happens not to be a big fan of the repeated function-name style, but you are welcome to use it if you are.

Turning our attention to course motivation, we can break the question of, "what is this course good for" into 3 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)?

A 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.

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 341 of ignoring these issues to focus on the fundamental truths underlying programming langauges. 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 browers 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 341 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 341 spends most of the course using functional languages, particularly ML and Scheme. The main reason is that they have many features that are invaluable for writing correct, elegant, and efficient software — and they encourage a way of thinking about computation that will make you a better programmer even in other languages. But since these features are the subject of many other lectures, today is a chance to "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" when they tend to teach exactly the language constructs and concepts that are useful but "ahead of their time." Students in 341 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 and Ruby), 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 we will learn like pattern-matching or currying or multimethods might be next.

In fact, in just the last year, there has been a surprising (even to your instructor) 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 *lot* like SML. 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). 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 5 years ago and is essentially functional programming along the lines we will learn next week 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 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. For example, the Macah compiler project in the UW CSE department uses Caml (a dialect of ML) even though much of the project's focus is computer architecture.

Some links to this "real-world functional programming" are on the course web-site, but this is just a small sample.