CSE 341, Spring 2011, Lecture 25 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 short lecture compares functional programming to object-oriented programming. There are actually many similarities. For example, a closure is like an object with one method ("call me") where the environment is like a list of private fields. There are also essential differences, such as OO's built-in support for late-binding, which we could code up in Scheme. Today, though, we focus less on these core issues of semantics and more on how *stylistically* functional and object-oriented programming often take exactly the opposite approach — and which is "better" often depends on how you expect software to evolve or be extended over time.

The canonical example we will consider is a little "expression language." In this language we have a number of "kinds of expressions" (integer constants, negations, additions, multiplications, etc.), and a number of "things to do with expressions" (evaluate them, convert them to strings, see if they have a 0 in them, etc.). This general set-up is common: we have data variants and we have operations over the data. To write all the code we want to write, we need to fill out this conceptual two-dimensional grid:

	Int	Negate	Add	Mult
eval				
toString				
hasZero				

In general, we can think of having one column for each data variant and one row for each operation. Often multiple grid squares can be implemented in the same way with some helper function or method in a superclass, but fundamentally we have to "do something" for each grid square. Now we can contrast the styles encouraged by functional programming and object-oriented programming:

- In functional programming, we tend to put all the code for a row together (in a function that has one branch for each data variant). An ML-style datatype definition declares the data variants and we get an inexhaustive pattern-match error/warning if some function does not support all the columns.

- In object-oriented programming, we tend to put all the code for a column together (in a class that has one method for each operation). A Java-style abstract class (or interface) definition declares the operations and we get an error if some non-abstract subclass does not support all the rows.

In this sense, how you represent the grid in your program (by rows or columns) is a matter of taste. Which seems most natural may depend on what kind of program you are writing.

However, if we expect our code to be extended in certain ways over time, then "which way we organize things" becomes more than a matter of taste:

- Suppose we expect to add new data variants (e.g., exponentiation operations) in the future. Then object-oriented programming is more convenient because we can add a "new column" without changing any existing code. To add a new column in our ML approach, we have to change our datatype binding and all our old functions. As a partial antidote, after adding a new constructor, the type-checker's inexhaustive-match warnings provide a nice to-do list (unless we used wildcard patterns in our initial program).

- Suppose we expect to add new operations (e.g., sum all the ints in an expression) in the future. Then functional programming is more convenient because we can add a "new row" without changing any existing code. To add a new row in our OO approach, we have to change our abstract class (or interface) definition and all our old classes. As a partial antidote, after adding a new abstract method, the type-checker provides a nice to-do list of all the classes that need new methods (unless we used run-time exceptions instead of abstract methods).

The problem, alas, is that the future is hard to predict, so we may not know in advance whether we will want new data variants or new operations. Moreover, maybe we will want both.

- If we write our original OO code in the natural way and then try to add a new row without changing existing code, it doesn't really work. You will end up with a lot of downcasts because the existing types expect something without the new operations, but we need to assume we have the new operations.
- If we write our original functional code in the natural way and then try to add a new column without changinge existing code, it also doesn't work. ML's datatype bindings do not let us "cast" something else, so we will end having to define a whole new datatype with new constructors.

The code accompanying this lecture shows what "goes wrong" in both cases. Particularly notice that in the ML code we cannot write down an expression where he have an old data variant (like an addition) that has "under it" a new data variant (like exponentiation).

However, if we write our original code in a way that *plans for extensibility* we can do better, although the code becomes considerably more complicated. An OO programming pattern for making it easy to add new operations without changing existing classes is called the "visitor pattern" (a synonym is "double dispatch"). While we won't go over it, it is well-known and definitely worth learning. Making a class hierarchy "visitable" requires some strange-looking code, but the pattern is always the same and then writing new "visitors" (operations over the data) is reasonably straightforward.

Similarly, we could define ML datatypes that allow for "other unknown possibilities." The "trick" is to use type constructors and a constructor that carries values of type `'a`. However, then our existing operations won't know how to handle this other case. To fix this, clients will have to pass in a function that handles this other case. For programs that don't extend the datatype with new data variants, we can just use a function that raises an exception. This programming pattern is pretty cumbersome and ML programmers do not tend to use it unless extensibility is really important.

There have been some language proposals and (ongoing) research projects to design languages that make both forms of extensibility convenient within the same program. It is probably fair to say that we can do better than today's popular languages but the community has not yet decided on the "best" (or if there is a "best") way.