Functional and Imperative Programming

Languages can be classified in many different ways. This course started by distinguishing between the labels *functional* and *imperative*. In fact, functional languages are a subclass of what are called *declarative* languages.

Declarative languages specify *what* should be done but not necessarily *how* it should be done. They are composed of declarations (statements that hold true at all times);

Imperative languages specify exactly *how* something should be done.

There are some languages that are *purely* declarative—often these languages are database-related. An example can be found in an SQL SELECT statement which looks like

```
SELECT fname,lname from namesdatabase;
```

This perhaps gets a list of (first name, last name) pairs from a database. But note how we haven't said *how* this should be done: the statement is purely declarative.

You'd be forgiven for feeling a little confused with regards ML—there you wrote out a series of things to do in each function so surely you specified both what *and* how? The key is that a functional compiler is free to look at your function and decide to do something completely different at a low level, *so long as it achieves the same result.* The entire function is taken as declarative—if you like, you provided an illustrative example of *what* needs to happen, but the compiler can do something else that gets the same result.

For example, you might specify a function f(x) = x+x+x+x;.A functional compiler would view this as telling it *what* to do by giving *one particular* example of how to do it. At a low level, it may decide instead to do a single multiplication: 4*x.

A purely imperative compiler, however, would follow your instructions to the letter—performing three additions in sequence. So you can consider a imperative compiler to be more dumb—it does *exactly* what you tell it to. This has the advantage that you can easily map your code to what goes on in the machine code.

In principle the functional approach is very nice—you can imagine an inexperienced programmer writing sub-optimal code and the compiler magically improving it. But think what would need to go into such a compiler—all the possible permutations of code and fixes become incredibly complex very fast. So functional languages limit what you can do—specifically with regards functions. Take this piece of imperative code:

```
int globalvar=5;
int multiply(int x) {
  globalvar=globalvar+1;
  return globalvar*x;
};
```

This is a procedure with clear side-effects (globalvar is altered). Ask yourself *what* this function does. What if some other function changed globalvar between calls as well? Hopefully you can see that modelling this function as a black box with a single output can't work. **So pure functional languages do *not* allow side-effects. i.e. they only use proper functions.**

Note, however, that *real* language implementations are not purely functional or imperative. ML contains some imperative features (references produce side-effects, and you have to actually start your program by running a function!) and Java often performs some small optimisations to code during compilation (albeit rarely entire functions). We think of ML as functional and Java as imperative because these are the dominant paradigms employed. But really things are not so black-and-white.