# Introduction to Functional Programming

**John Harrison**

**University of Cambridge**

## Lecture 2

## Basics of ML

Topics covered:

- Versions of ML
- Running ML on Thor
- Interacting with ML
- Higher order functions and currying
- Evaluation order

# The ML family

ML is not a single language. Apart from Standard ML, which we use here, there are many descendants of the original ML used as the metalanguage of Edinburgh LCF, e.g.

- CAML Light — an excellent lightweight system developed at INRIA.
- Objective CAML — a new version of CAML Light including object-oriented features.
- Lazy ML — a version from Gothenburg using *lazy evaluation*.
- Standard ML — an agreed 'standard version'

Standard ML has two parts: the *Core* language and the *Modules*. We will not cover the module system at all.

## Implementations of Standard ML

There are several implementations of (something similar to) Standard ML:

- Standard ML of New Jersey — free but very resource-hungry.
- Moscow ML — free but doesn't have the Modules.
- Poly ML — good, but a commercial product, and doesn't run under Linux
- Harlequin ML — a newer commercial system with integrated development environment.

We'll use Moscow ML. This is a good lightweight implementation based on CAML Light, written by Sergei Romanenko (Moscow) and Peter Sestoft (Copenhagen).

The features we use are general, and can easily be translated to other dialects.

# Starting up ML

Moscow ML is already installed on Thor. In order to start it up, type:

```
/group/clteach/jrh/mosml/`arch`/bin/mosml
```

This will invoke the appropriate version, depending on the machine architecture.

The system should start up and present its prompt.

```
$ /group/clteach/jrh/mosml/`arch`/bin/mosml
Moscow ML version 1.42 (July 1997)
Enter `quit();' to quit.
-
```

If you want to install Moscow ML on your own computer, see the Web page:

```
http://www.dina.kvl.dk/~sestoft/mosml.html
```

## Interacting with ML

We will run ML as an interpreter, rather like a sophisticated calculator. You just type an expression into it, terminated by a semicolon, and it prints the result of evaluating it, e.g.

```
- 10 + 5;
> val it = 15 : int
```

ML not only prints the result, but also infers the *type* of the expression, namely `int`. If ML cannot assign a type to an expression then it is rejected:

```
- 1 + true;
! .... Type clash: ....
```

ML knows the type of the built-in operator `+`, and that is how it makes its type inference.

We'll treat the type system more systematically next time; for the moment, it should be intuitively clear.

## Loading from files

We have just been typing things into ML and thinking about the results. However one doesn't write real programs in this way.

Typically, one writes the expressions and declarations in a file. To try them out as you go, these can be inserted in the ML window using cut and paste.

You can cut and paste using X-windows and similar systems, or an editor like Emacs with multiple buffers.

For larger programs, it's convenient simply to load them from file into the ML session. This can be done using the `use` command in ML, e.g:

```
use "myprog.ml";
```

Programs can also include comments written between `(*` and `*)`. These are ignored by ML, but are useful for people reading the code.

# Using functions

Since ML is a functional language, expressions are allowed to have function type. The ML syntax for a function taking `x` to `t[x]` is `fn x => t[x]`. For example we can define the successor function:

```
- fn x => x + 1;
> val it = fn : int -> int
```

Again, the type of the expression, this time `int -> int`, is inferred and displayed. However the function itself is not printed; the system merely writes `fn`.

Functions are applied by juxtaposition, with or without bracketing:

```
- (fn x => x + 1) 4;
> val it = 5 : int
- (fn x => x + 1)(3);
> val it = 4 : int
```

## Curried functions (1)

Every function in ML takes a single argument.

One way to get the effect of multiple arguments is to use a *pair* for the argument — we'll discuss this next time.

Another way, often more powerful, is to use *Currying*, making the function take its arguments one at a time, e.g.

```
- fn x => (fn y => x + y);
> val it = fn : int -> (int -> int)
```

This function takes the first argument and gives a new function. For example:

```
- (fn x => (fn y => x + y)) 1;
> val it = fn : int -> int
```

is just the successor function again, e.g:

```
- ((fn x => (fn y => x + y)) 1) 6;
> val it = 7 : int
```

# Curried functions (2)

Because curried functions are so common in functional programming, ML fixes the rules of association to avoid the need for many parentheses.

When one writes $E_1\ E_2\ E_3$, ML associates it as $(E_1\ E_2)\ E_3$. For example, all the following are equivalent:

```
- ((fn x => (fn y => x + y))(1))(6);
> val it = 7 : int

- ((fn x => (fn y => x + y)) 1) 6;
> val it = 7 : int

- (fn x => (fn y => x + y)) 1 6;
> val it = 7 : int

- (fn x => fn y => x + y) 1 6;
> val it = 7 : int
```

# Bindings (1)

It is not necessary to evaluate an expresion all in one piece. You can *bind* an expression to a name using `val`:

```
- val successor = fn x => x + 1;
> val successor = fn : int -> int
- successor(successor(successor 0));
> val it = 3 : int
```

Note that this isn't an assignment statement, merely an abbreviation. But bindings can be recursive: just add `rec`:

```
- val rec fact =
    fn n => if n = 0 then 1
            else n * fact(n - 1);
> val fact = fn : int -> int
- fact 3;
> val it = 6 : int
```

# Bindings (2)

When binding functions, one can also use fun.

The following are equivalent:

```
- val successor = fn x => x + 1;

- fun successor x = x + 1;
```

and

```
- val rec fact =
        fn n => if n = 0 then 1
                else n * fact(n - 1);

- fun fact n =
      if n = 0 then 1
      else n * fact(n - 1);
```

Note that bindings with `fun` are *always* recursive.

## Higher order functions

Curried functions are an example of a function that gives another function as a result.

We can also define functions that take other functions as arguments. This one takes a positive integer $n$ and a function $f$ and returns $f^n$, i.e. $f \circ \cdots \circ f$ ($n$ times):

```
- fun funpow n f x =
      if n = 0 then x
      else funpow (n - 1) f (f x);
> val funpow = fn : int -> ('a -> 'a) ->
                              'a -> 'a
- funpow 20 (fn x => 2 * x) 1;
> val it = 1048576 : int
```

# Local declarations

Bindings can be made local using

`let` decs `in` expr `end`

For example:

```
- let fun fact n =
        if n = 0 then 1
        else n * fact(n - 1)
   in fact 6
  end;
```

This binding is now invisible beyond the terminating `end`. Similarly one can make a declaration local to another *declaration* by:

`local` decs `in` decs `end`

# ML's evaluation strategy

Execution of an ML program just means evaluating an expression. You can think of this evaluation as happening by a kind of syntactic unfolding of the program.

- Constants like `1` and `+` evaluate to themselves.
- Evaluation stops immediately at something of the form `fn x => ...` and does not 'look inside' the `...`.
- When evaluating a combination `s t`, then *first* both `s` and `t` are evaluated to `s'` and `t'`. If `s'` is not of the form `fn x => u[x]` then things stop there. Otherwise `u[t']`, the result of replacing the dummy variable `x` by the evaluated form of `t` is evaluated.

It is important to grasp this evaluation strategy in order to understand ML properly.

## Which evaluation strategy

But does evaluation order actually matter? One might guess not. For example:

```
(1 + 2) + (3 + 4) = 3 + (3 + 4)
                  = 3 + 7 = 10
```

and

```
(1 + 2) + (3 + 4) = (1 + 2) + 7
                  = 3 + 7 = 10
```

give the same answer. Does this generalize?

In fact, for 'pure' functional programs, if two different evaluation orders terminate, they give the same answer. (This is roughly the *Church-Rosser theorem*.)

But there are still reasons to care about evaluation order: termination, efficiency, and imperative features.

# Lazy evaluation

ML's evaluation strategy is called *eager* or *call-by-value* because the argument to a function is evaluated even if it never ends up being used.

An alternative would be, when evaluating `(fn x => u[x]) t`, to evaluate `u[t]` without yet evaluating `t`. This is *call-by-name* evaluation.

The advantage is: we avoid evaluating `t` if it isn't actually used, and this evaluation might be very costly or even fail to terminate.

The disadvantage of a naive implementation is that one would re-evaluate `t` if it's used more than once. So one needs a clever sharing implementation — *lazy evaluation.*

Lazy evaluation seems better in principle, and many functional languages, even ML's relative Lazy ML, use it. But it's harder to implement efficiently and doesn't seem to fit with imperative features. Hence ML's choice.

# ML evaluation examples

```
(fn x => (fn y => y + y) x) (2 + 2)
(fn x => (fn y => y + y) x) 4
(fn y => y + y) 4
4 + 4
8

((fn f => fn x => f x) (fn y => y + y))
           (2 + 2)
(fn x => (fn y => y + y) x) (2 + 2)
(fn x => (fn y => y + y) x) 4
(fn y => y + y) 4
4 + 4
8
```

## The conditional

Consider evaluating a factorial (as defined earlier) where the conditional is evaluated eagerly. To evaluate `fact(0)` we unfold it to:

```
if 0 = 0 then 1 else 0 * fact(0 - 1)
```

We need to cut this down to `1`. But under the standard eager rules, we would first evaluate all three subexpressions, including `fact(0 - 1)`. This leads to an infinite loop.

So we can't regard the conditional as an ordinary function of three arguments: it has to have its own 'lazy' reduction strategy.

The test expression is evaluated first, and according to its value, precisely one of the arms, never both,