What is functional programming?

“Functional programming” can mean a few different things:
1. Avoiding mutation in most/all cases (done and ongoing)
2. Using functions as values (this unit)
   ...  
   • Style encouraging recursion and recursive data structures  
   • Style closer to mathematical definitions  
   • Programming idioms using laziness (later topic, briefly)  
   • Anything not OOP or C? (not a good definition)

Not sure a definition of “functional language” exists beyond “makes functional programming easy / the default / required”
   – No clear yes/no for a particular language

First-class functions

• First-class functions: Can use them wherever we use values
  – Functions are values too
  – Arguments, results, parts of tuples, bound to variables, 
  carried by datatype constructors or exceptions, ...

  ```
  fun double x = 2*x
  fun incr x = x+1
  val a_tuple = (double, incr, double(incr 7))
  ```

• Most common use is as an argument / result of another function
  – Other function is called a higher-order function
  – Powerful way to factor out common functionality

Function Closures

• Function closure: Functions can use bindings from outside the 
  function definition (in scope where function is defined)
  – Makes first-class functions much more powerful
  – Will get to this feature in a bit, after simpler examples

• Distinction between terms first-class functions and function 
  closures is not universally understood
  – Important conceptual distinction even if terms get muddled

Onward

The next week:
   – How to use first-class functions and closures
   – The precise semantics
   – Multiple powerful idioms

Functions as arguments

• We can pass one function as an argument to another function
  – Not a new feature, just never thought to do it before

  ```
  fun f (g,...) = ... g (...) ...
  fun h1 ... = ..
  fun h2 ... = ..
  .. f(h1,..) .. f(h2,..) ..
  ```

• Elegant strategy for factoring out common code
  – Replace N similar functions with calls to 1 function where 
  you pass in N different (short) functions as arguments

[See the code file for this lecture]
Example

Can reuse `n_times` rather than defining many similar functions

\[ \text{fun } n\_times(f,n,x) = \]
\[ \text{if } n=0 \]
\[ \text{then } x \]
\[ \text{else } f(n\_times(f,n-1,x)) \]

\[ \text{fun } double\ x = x + x \]
\[ \text{fun } increment\ x = x + 1 \]
\[ \text{val } x1 = n\_times(double,4,7) \]
\[ \text{val } x2 = n\_times(increment,4,7) \]
\[ \text{val } x3 = n\_times([1,2,3,4,5],6) \]

\[ \text{fun } double\_n\_times\ (n,x) = n\_times(double,n,x) \]
\[ \text{fun } nth\_tail\ (n,x) = n\_times(tl,n,x) \]

Relation to types

- Higher-order functions are often so “generic” and “reusable” that they have polymorphic types, i.e., types with type variables

- But there are higher-order functions that are not polymorphic

- And there are non-higher-order (first-order) functions that are polymorphic

- Always a good idea to understand the type of a function, especially a higher-order function

Types for example

\[ \text{fun } n\_times(f,n,x) = \]
\[ \text{if } n=0 \]
\[ \text{then } x \]
\[ \text{else } f(n\_times(f,n-1,x)) \]

\[ \text{val } n\_times : ('a -> 'a) * int * 'a -> 'a \]

- Simpler but less useful: `(int -> int) * int * int -> int`

- Two of our examples instantiated `'a` with `int`

- One of our examples instantiated `'a` with `int list`

- This polymorphism makes `n_times` more useful

- Type is inferred based on how arguments are used (later lecture)

  - Describes which types must be exactly something (e.g., `int`) and which can be anything but the same (e.g., `'a`

Polymorphism and higher-order functions

- Many higher-order functions are polymorphic because they are so reusable that some types, "can be anything"

- But some polymorphic functions are not higher-order

  - Example: `len` : `'a list -> int`

- And some higher-order functions are not polymorphic

  - Example: `times_until_zero` : `(int->int)*int->int`

  \[ \text{fun } times\_until\_zero\ (f,x) = \]
  \[ \text{if } x=0 \]
  \[ \text{then } 0 \]
  \[ \text{else } 1 + times\_until\_zero(f, f x) \]

  Note: Would be better with tail-recursion

Toward anonymous functions

- Definitions unnecessarily at top-level are still poor style:

  \[ \text{fun } trip\ x = 3*x \]
  \[ \text{fun } triple\_n\_times\ (f,x) = n\_times(trip,n,x) \]

- So this is better (but not the best):

  \[ \text{fun } triple\_n\_times\ (f,x) = \]
  \[ \text{let fun } trip\ y = 3*y \]
  \[ \text{in } n\_times(trip,n,x) \]
  \[ \text{end} \]

- And this is even smaller scope

  - It makes sense but looks weird (poor style; see next slide)

  \[ \text{fun } triple\_n\_times\ (f,x) = \]
  \[ \text{n\_times(let fun } trip\ y = 3*y \text{ in } trip end, n, x) \]

Anonymous functions

- This does not work: A function binding is not an expression

  \[ \text{fun } triple\_n\_times\ (f,x) = \]
  \[ \text{n\_times((Fun } trip\ y = 3*y), n, x) \]

- This is the best way we were building up to: an expression form for anonymous functions

  \[ \text{fun } triple\_n\_times\ (f,x) = \]
  \[ \text{n\_times((fn } y \Rightarrow 3*y), n, x) \]

  - Like all expression forms, can appear anywhere

  - Syntax:

    - `fn not Fun`
    - `=> not =`
    - `no function name, just an argument pattern`
Using anonymous functions

- Most common use: Argument to a higher-order function
  - Don’t need a name just to pass a function
- But: Cannot use an anonymous function for a recursive function
  - Because there is no name for making recursive calls
  - If not for recursion, `fun` bindings would be syntactic sugar for `val` bindings and anonymous functions

```ml
fun triple x = 3*x
val triple = fn y => 3*y
```

A style point

Compare:

```ml
if x then true else false
```

With:

```ml
(fn x => f x)
```

So don’t do this:

```ml
n_times((fn y => t1 y),3,xs)
```

When you can do this:

```ml
n_times(t1,3,xs)
```

Map

```ml
fun map (f,xs) =
    case xs of
      [] => []
    | x::xs' => (f x)::(map(f,xs'))

val map : ('a -> 'b) * 'a list -> 'b list
```

Filter

```ml
fun filter (f,xs) =
    case xs of
      [] => []
    | x::xs' => if f x then x::(filter(f,xs')) else filter(f,xs')

val filter : ('a -> bool) * 'a list -> 'a list
```

Generalizing

Our examples of first-class functions so far have all:
- Taken one function as an argument to another function
- Processed a number or a list

But first-class functions are useful anywhere for any kind of data
- Can pass several functions as arguments
- Can put functions in data structures (tuples, lists, etc.)
- Can return results as inputs
- Can write higher-order functions that traverse your own data structures

Useful whenever you want to abstract over “what to compute with”
- No new language features

Returning functions

- Remember: Functions are first-class values
  - For example, can return them from functions
- Silly example:

```ml
fun double_or_triple f =
    if f 7
    then fn x => 2*x
    else fn x => 3*x
```

Has type `int -> bool` -> `(int -> int)`

But the REPL prints `(int -> bool) -> int` -> `int` because it never prints unnecessary parentheses and `t1 -> t2 -> t3 -> t4` means `t1->(t2->(t3->t4))`
Other data structures

- Higher-order functions are not just for numbers and lists
- They work great for common recursive traversals over your own data structures (datatype bindings) too
- Example of a higher-order predicate:
  - Are all constants in an arithmetic expression even numbers?
  - Use a more general function of type
    `(int -> bool) * exp -> bool`
  - And call it with `(fn x => x mod 2 = 0)`