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, …

  \[
  \begin{align*}
  \text{fun } \text{double } x & = 2 \times x \\
  \text{fun } \text{incr } x & = x+1 \\
  \text{val } \text{a_tuple } = (\text{double}, \text{incr}, \text{double}(\text{incr} 7))
  \end{align*}
  \]

- 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

  \[
  \begin{align*}
  \text{fun } f (g,\ldots) & = \ldots g (\ldots) \ldots \\
  \text{fun } h1 & = \ldots \\
  \text{fun } h2 & = \ldots \\
  \ldots f(h1,\ldots) & \ldots f(h2,\ldots) \ldots
  \end{align*}
  \]

- 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 \texttt{n\_times} rather than defining many similar functions

\begin{verbatim}
fun n_times (f,n,x) = 
  if n=0
    then x
  else f (n_times(f,n-1,x))
\end{verbatim}

\begin{verbatim}
fun double x = x + x
fun increment x = x + 1
val x1 = n_times(double,4,7)
val x2 = n_times(increment,4,7)
val x3 = n_times(tl,2,[4,8,12,16])
fun double_n_times (n,x) = n_times(double,n,x)
fun nth_tail (n,x) = n_times(tl,n,x)
\end{verbatim}

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

\begin{verbatim}
fun n_times (f,n,x) = 
  if n=0
    then x
  else f (n_times(f,n-1,x))
\end{verbatim}

- \texttt{val n\_times : ('a -> 'a) * int * 'a -> 'a}
  - Simpler but less useful: (int -> int) * int * int -> int
- Two of our examples \texttt{instantiated} 'a with \texttt{int}
- One of our examples \texttt{instantiated} 'a with \texttt{int list}
- This \texttt{polymorphism} makes \texttt{n\_times} more useful
- Type is \texttt{inferred} based on how arguments are used (later lecture)
  - Describes which types must be exactly something (e.g., \texttt{int}) and which can be anything but the same (e.g., \texttt{'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: \texttt{len : 'a list -> int}
- And some higher-order functions are not polymorphic
  - Example: \texttt{times\_until\_0 : (int\_->\_int) * int\_->\_int}

\begin{verbatim}
fun times_until_zero (f,x) = 
  if x=0 then 0 else 1 + times_until_zero(f, f x)
\end{verbatim}

Note: Would be better with tail-recursion

Anonymous functions

- This does not work: A function \texttt{binding} is not an expression

\begin{verbatim}
fun triple_n_times (f,x) = 
  let fun trip y = 3*y
  in
    n_times(trip,n,x)
  end
\end{verbatim}

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

\begin{verbatim}
fun triple_n_times (f,x) = 
  n_times((fn y => 3*y), n, x)
\end{verbatim}

- Like all expression forms, can appear anywhere
- Syntax:
  - \texttt{fn} not \texttt{fun}
  - \texttt{=>} not \texttt{=}
  - 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

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

A style point

Compare: `if x then true else false` With: `(fn x => f x)`

So don’t do this: `n_times((fn y => tl y),3,xs)`

When you can do this: `n_times(tl,3,xs)`

Map

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

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

Filter

```haskell
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 functions as results
- 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:

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

val double_or_triple = fn f => double_or_triple f
```

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
    
    $$(\text{int} \to \text{bool}) \times \text{exp} \to \text{bool}$$
    
  - And call it with $(\text{fn } x => x \mod 2 = 0)$