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

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

• 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{itemize}
\item Computes $f(f(...f(x)))$ where number of calls is $n$
\end{itemize}

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

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

fun n_times (f,n,x) =  
  if n=0
  then x
  else f (n_times(f,n-1,x))

• 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: \( \text{len} : \ '\text{a list} -> \text{int} \)

• And some higher-order functions are not polymorphic
  – Example: \( \text{times\_until\_0} : (\text{int} -> \text{int}) \ast \text{int} -> \text{int} \)

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

Note: Would be better with tail-recursion
Toward anonymous functions

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

\[
\begin{align*}
\text{fun } & \text{trip } x = 3 \times x \\
\text{fun } & \text{triple_n_times (f,x) = n\_times(trip,n,x)}
\end{align*}
\]

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

\[
\begin{align*}
\text{fun } & \text{triple_n_times (f,x) =}
\text{let fun } & \text{trip y = 3 \times y}
\text{in}
\text{n\_times(trip,n,x)}
\text{end}
\end{align*}
\]

• And this is even smaller scope
  – It makes sense but looks weird (poor style; see next slide)

\[
\begin{align*}
\text{fun } & \text{triple_n_times (f,x) =}
\text{n\_times(let fun } & \text{trip y = 3 \times y in } \text{trip end, n, x)}
\end{align*}
\]
Anonymous functions

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

```plaintext
fun triple_n_times (f,x) = 
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

```plaintext
fun triple_n_times (f,x) = 
n_times( (fn y => 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

```plaintext
fun triple x = 3*x
val triple = fn y => 3*y
```
A style point

Compare:

\[
\text{if } x \text{ then true else false}
\]

With:

\[
(fn \ x \Rightarrow f \ x)
\]

So don’t do this:

\[
n\_\text{times}((fn \ y \Rightarrow \text{tl} \ y), 3, \text{xs})
\]

When you can do this:

\[
n\_\text{times}(\text{tl}, 3, \text{xs})
\]
Map

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

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

Map is, without doubt, in the “higher-order function hall-of-fame”

- The name is standard (for any data structure)
- You use it all the time once you know it: saves a little space, but more importantly, communicates what you are doing
- Similar predefined function: List.map
  - But it uses currying (coming soon)
Filter is also in the hall-of-fame

- So use it whenever your computation is a filter
- Similar predefined function: `List.filter`
  - But it uses currying (coming soon)

```plaintext
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
```

Autumn 2019
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:

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

Has type \((\text{int} \to \text{bool}) \to \text{int} \to \text{int})\)

But the REPL prints \((\text{int} \to \text{bool}) \to \text{int} \to \text{int}\)
because it never prints unnecessary parentheses and
\(t_1 \to t_2 \to t_3 \to t_4\) means \(t_1 \to (t_2 \to (t_3 \to t_4))\)
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
    
    \[(\text{int} \rightarrow \text{bool}) \times \text{exp} \rightarrow \text{bool}\]

  – And call it with \(\text{(fn } x \Rightarrow x \mod 2 = 0)\)