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, …

- 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

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

- 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]
Can reuse n_times rather than defining many similar functions

- Computes f(f(…f(x))) where number of calls is n

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

```latex
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])
```

```latex
fun double_n_times (n, x) = n_times(double,n,x)
fun nth_tail (n, x) = n_times(tl,n,x)
```

Autumn 2019
CSE341: Programming Languages

### 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

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

```latex
fun 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

### Toward anonymous functions

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

```latex
fun trip x = 3*x
fun triple_n_times (f, x) = n_times(trip,n,x)
```

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

```latex
fun triple_n_times (f, x) = 
  let fun trip y = 3*y in
    n_times(trip,n,x)
  end
```

- And this is even smaller scope:

```latex
fun triple_n_times (f, x) = 
  n_times(let fun trip y = 3*y in trip end, n,x)
```

### Anonymous functions

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

```latex
fun triple_n_times (f, x) = 
  n_times(let f x = 3*y in f end, n,x)
```

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

```latex
fun triple_n_times (f, x) = 
  n_times(let trip y = 3*y in trip end, n,x)
```

- Like all expression forms, can appear anywhere
- Syntax:
  - fn not fn
  - no name, just an argument pattern

Note: Would be better with tail-recursion

### 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
- And some higher-order functions are not polymorphic:

```latex
fun times_until_zero (f, x) = 
  if x=0 then 0
  else 1 + times_until_zero(f, f x)
```

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

```latex
fun triple_n_times (f, x) = 
  n_times(let trip y = 3*y in trip end, n,x)
```

- Like all expression forms, can appear anywhere
- Syntax:
  - fn not fn
  - no name, just an argument pattern

Note: Would be better with tail-recursion
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

\[
\begin{align*}
\text{fun triple x} &= 3 \times x \\
\text{val triple} &= \text{fn } y \Rightarrow 3 \times y
\end{align*}
\]

A style point

Compare:

\[
\begin{align*}
\text{if } x \text{ then true else false}
\end{align*}
\]

With:

\[
\begin{align*}
(fn x \Rightarrow f x)
\end{align*}
\]

So don’t do this:

\[
\text{fun times}((f \Rightarrow y) \times t1 \times y, 3, x)
\]

When you can do this:

\[
\text{fun times}(t1, t3, x)
\]

Map

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

val \(\text{map} : (\text{int} \rightarrow \text{bool}) \times \text{int} \rightarrow \text{int} \rightarrow \text{int}\)

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: \(\text{List.map}\)
  - But it uses currying (coming soon)

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:

\[
\begin{align*}
\text{fun \(double_or_triple\) \(f\)} =
  \text{if } f 7 \text{ then fn } x \Rightarrow 2 \times x
  \text{ else fn } x \Rightarrow 3 \times x
\end{align*}
\]

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

But the REPL prints \((\text{int} \rightarrow \text{bool}) \rightarrow \text{int} \rightarrow \text{int}\)
because it never prints unnecessary parentheses and
\(1 \rightarrow t1 \rightarrow t2 \rightarrow t3 \rightarrow t4\) means \(t1 \Rightarrow (t2 \Rightarrow (t3 \Rightarrow t4))\)
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 (data type bindings) too
- Example of a higher-order predicate:
  - Are all constants in an arithmetic expression even numbers?
  - Use a more general function of type
    \( \text{int} \rightarrow \text{bool} \) \* \( \text{exp} \rightarrow \text{bool} \)
  - And call it with \( \text{fn} \ x \Rightarrow x \mod 2 = 0 \)