On to first-class functions

“Functional programming” can mean a few different things:

1. Avoiding mutation in most/all cases (done and ongoing)
2. Using functions as values (the next week)

... Recursion? Mathematical definitions? Not OO? Laziness (later)?

First-class functions

- Functions are (first-class) values: Can use them wherever we use values
  - 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
  - The other function is called a higher-order function
  - Powerful way to factor out common functionality
- 3-ish lectures on how and why to use first-class functions

Example

Can reuse \texttt{n\_times} rather than defining many similar functions
- Computes \( f(f(\ldots f(x))) \) where number of calls is \( n \)

\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,[4,8,12,16,20])
fun double\_n\_times (n,x) = n\_times(double,n,x)
fun nth\_tail (n,x) = n\_times(tl,n,x)
\end{verbatim}

Types

- \texttt{val n\_times : ('a -> 'a) * int * 'a -> 'a}
- Two of our examples instantiated 'a with \texttt{int}
- One of our examples instantiated 'a with \texttt{int list}
- This polymorphism makes \texttt{n\_times} more useful
- Type is \textit{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., '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{length : 'a list \rightarrow int}
- And some higher-order functions are not polymorphic
  - Example: \texttt{times\_til\_0 : (int \rightarrow int) * int \rightarrow int}

\begin{verbatim}
fun times\_til\_0 (f,x) =
  if x=0 then 0 else 1 + times\_til\_0(f, f x)
\end{verbatim}
* Would be better with tail-recursion
Toward anonymous functions

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

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

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

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

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

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

Anonymous functions

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

\[
\begin{align*}
\text{fun } \text{triple}_n \text{times } (f,x) &= \\
&= \text{n_times}((\text{fun } \text{trip } y = 3 \times y), n, x)
\end{align*}
\]

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

\[
\begin{align*}
\text{fun } \text{triple}_n \text{times } (f,x) &= \\
&= \text{n_times}((\text{fn } y => 3 \times y), n, x)
\end{align*}
\]

- Like all expression forms, can appear anywhere
- Syntax:
  - \text{fn} not \text{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, \text{fun} bindings would be syntactic sugar for \text{val} bindings and anonymous functions

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

A style point

Compare:

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

With:

\[
(\text{fn } x => f \ x)
\]

So don't do this:

\[
\text{n_times}((\text{fn } y => \text{tl } y),3,\text{xs})
\]

When you can do this:

\[
\text{n_times}((\text{tl},3,\text{xs})
\]

Map

\[
\begin{align*}
\text{fun } \text{map } (f,xs) &= \\
&= \text{case } xs \text{ of } \\
&\text{[] } => \text{[]} \\
&\text{| x::xs' } => (f \ x)::(\text{map}(f,xs'))
\end{align*}
\]

\[
\text{map} : (\text{'a } => \text{'b}) \times \text{'a list } \rightarrow \text{'b list}
\]

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: \text{List.map}
  - But it uses currying (lecture 9)

Filter

\[
\begin{align*}
\text{fun } \text{filter } (f,xs) &= \\
&= \text{case } xs \text{ of } \\
&\text{[] } => \text{[]} \\
&\text{| x::xs } => \text{if } f \ x \text{ then } x::(\text{filter}(f,\text{rest})) \\
&\text{else } \text{filter}(f,\text{rest})
\end{align*}
\]

\[
\text{filter} : (\text{'a } => \text{bool}) \times \text{'a list } \rightarrow \text{'a list}
\]

Filter is also in the hall-of-fame
- So use it whenever your computation is a filter
- Similar predefined function: \text{List.filter}
  - But it uses currying (lecture 9)
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
```

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
    \((\text{int} \to \text{bool}) \times \text{exp} \to \text{bool}\)

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