Review

- Features learned: functions, tuples, lists, let expressions, options, records, datatypes, case expressions, type synonyms, pattern matching, exceptions, type variables

- Today:
  - First-class functions
  - Higher-order functions
  - Anonymous functions

...functions are FUN!

Functions are values

- Can use them anywhere we use values
  - Arguments, results, parts of tuples, bound to variables, carried by datatype constructors or exceptions...
- First-class citizens of language, afforded all the “rights” of any other values
- Functions can take functions as arguments
- Functions can return functions as results
  - ...so functions are higher-order
- This is not a new language feature; just a consequence of choice we made long ago when we said "functions are values"

Higher-order functions

“A language that doesn’t affect the way you think about programming is not worth knowing.”

– Alan J. Perlis

Example

Can reuse \texttt{n\_times} rather than defining many similar functions
  - Computes \(f(f(...f(x)))\) 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 double\_n\_times \(\(n,x\)\) = \(n\_times\ (double,n,x)\)

(* more examples in code *)
\end{verbatim}

Types

- \(\text{val n\_times : } ('a \to 'a) \times \text{int} \to 'a\)
- Two of our examples instantiated \('a\) with \text{int}
- One of our examples instantiated \('a\) with \text{int list}
- This polymorphism makes \text{n\_times} more useful

- Type is inferred based on how arguments are used (later lecture)
  - Describes which types must be exactly something (e.g. \text{int}) and which can be anything but the same (e.g. \('a\)
Polymorphism and higher-order functions

• Often, higher-order & polymorphic go hand-in-hand
  – Both features enable reusability of code

• But some polymorphic functions are not higher-order
  – Example: `length : 'a list -> int`

• And some higher-order functions are not polymorphic
  – Example: `times_til_0 : (int -> int) * int -> int`

  ```
  fun times_til_0 : (f : int -> int, x : int) =>
  if x = 0 then 0 else 1 + times_til_0(f, f x)
  ```

  (Note: tail recursive version would be more efficient)

Toward anonymous functions

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

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

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

  ```
  fun times_til_0 : (f : int -> int, x : int) =>
  if x = 0 then 0 else 1 + times_til_0(f, f x)
  ```

  ```
  fun triple_n_times : (f : int -> int, n : int, x : int) =>
  let
  trip y = 3 * y
  in
  n_times(trip, n, x)
  ```

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

Anonymous functions

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

  ```
  fun triple_n_times : (f : int -> int, n : int, x : int) =>
  n_times((fn y => 3 * y), n, x)
  ```

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

  ```
  fun triple_n_times : (f : int -> int, n : int, x : int) =>
  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: anonymous function cannot be recursive
  – Exactly because they have no names, so you can’t write the recursive call
  – If not for recursion, `fun` binding would be syntactic sugar for `val` binding + anonymous function:

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

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

\[
\text{fun map } (f, xs) = \\
\text{ case xs of} \\
\quad [] \Rightarrow [] \\
\quad \text{x} :: \text{xs} \Rightarrow (f \text{x}) :: \text{map}(f, \text{xs}')
\]

\[
\text{map } : \ ('a -> 'b) * \ 'a \text{ list} -> \ 'b \text{ list}
\]

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 (later lecture)

---

Filter

\[
\text{fun filter } (f, xs) = \\
\text{ case xs of} \\
\quad [] \Rightarrow [] \\
\quad \text{x} :: \text{xs} \Rightarrow if f \text{x} then \text{x} :: \text{filter}(f, \text{rest}) else \text{filter}(f, \text{rest})
\]

\[
\text{filter } : \ ('a -> \text{bool}) * \ 'a \text{ list} -> \ 'a \text{ list}
\]

Filter is also in the hall-of-fame
- So use it whenever your computation is a filter
- Similar predefined function: List.filter
- Also uses currying

---

Not just for lists...

- Higher-order functions work over any type
- Great for recursive traversals over your own data structures (datatype bindings)
  - E.g: Are all constants in an arithmetic expression even numbers?

\[
\text{fun true_of_all_constants}(f, e) = \\
\text{ case e of} \\
\quad \text{Constant } i \Rightarrow f i \\
\quad \text{Add}(e_1, e_2) \Rightarrow \\
\quad \text{true_of_all_constants}(f, e_1) \land \text{true_of_all_constants}(f, e_2)
\]

\[
\text{fun all_even } e = \\
\text{true_of_all_constants}(\text{fn } x \Rightarrow (x \text{ mod } 2) = 0, e)
\]

---

Returning functions

- Since functions are first-class values, can return them from other functions
- Silly example:

\[
\text{fun double_or_triple } f = \\
\text{ if } f 7 \text{ then } \text{fn x} \Rightarrow 2 \text{x} \\
\text{ else } \text{fn x} \Rightarrow 3 \text{x}
\]

\[
\text{double_or_triple } : \ \text{(int -> bool)} \times \text{(int -> int)} \\
\]

Has type (\text{int -> bool}) \times \text{int -> int}
- But the REPL prints (\text{int -> bool}) \rightarrow \text{int -> int}
  because REPL never prints unnecessary parentheses
  and because -> is right-associative, e.g.,
  \text{t1 -> t2 -> t3 -> t4} means \text{t1 -> (t2 -> (t3 -> t4))}