Review

- Features learned: functions, tuples, lists, let expressions, options, records, datatypes, case expressions, pattern matching, exceptions, type variables, higher-order and anonymous functions, infix operators, type constructors
- Today:
  - Function composition
  - fold
  - Currying (\texttt{mmm})

\textbf{Function Composition}

Combine functions

In math, \( f \circ g(x) = f(g(x)) \)

\begin{verbatim}
fun compose \( f, g \) = \( fn x \Rightarrow f \( g \ x \) \)
\end{verbatim}

- \texttt{Type \(('b -> 'c) * ('a -> 'b) -> ('a -> 'c)\)}
- but the REPL prints something equivalent
- ML standard library provides this as infix operator \( o \)
- That's small-case letter \( O \), as in...
- Example (third version best):

\begin{verbatim}
fun sqrt_of_abs \( i \) = Math.sqrt \( \textit{Real.fromInt} \( \text{abs} \( i \) \) \)
fun sqrt_of_abs \( i \) = (Math.sqrt \( \textit{Real.fromInt} \( \text{abs} \( i \) \) \)
val sqrt_of_abs = Math.sqrt \( \textit{Real.fromInt} \( \text{abs} \( i \) \) \)
\end{verbatim}

Left-to-right vs. right-to-left

In math, function composition is read right-to-left

- "take absolute value; convert to real; take square root"

But programmers often prefer reading left-to-right

- "Pipelines" of functions are common in functional programming
- Pipeline operator is very popular (and predefined) in F#
  - Can define ourselves in ML

\begin{verbatim}
infix |> fun \( x \) |> \( f \) = \( f \ x \)
fun sqrt_of_abs \( i \) =
  \( i \ |> \text{abs} \ |> \text{Real.fromInt} \ |> \text{Math.sqrt} \)
\end{verbatim}

\textbf{Fold}

\begin{verbatim}
fun sqs \( i \) = Math.sqrt \( \textit{Real.fromInt} \( \text{abs} \( i \) \) \)
fun sqs \( i \) = (Math.sqrt \( \textit{Real.fromInt} \( \text{abs} \( i \) \) \)
val sqs \( i \) = Math.sqrt \( \textit{Real.fromInt} \( \text{abs} \( i \) \) \)
\end{verbatim}
Map

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

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

\[
\text{map} \ (\text{fn} \ x \Rightarrow \text{shirt_color}[x], []) \quad \text{bad style!}
\] = [\text{gold}, \text{blue}, \text{red}]

Filter

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

filter : ('a -> bool) * 'a list -> 'a list

\[
\text{filter} \ (\text{is_vulcan}, [], []) = []
\] (\emph{er}, half vulcan)

Another famous function: Fold

foldl
-
  used to iterate over recursive data structures
-
  several synonyms/cousins: \(\text{reduce}, \text{inject}, \text{etc.}\)
-
  Accumulates an answer by repeatedly applying \(f\) to "answer so far"
foldl \((f, acc, [x_1, x_2, x_3])\) computes
\(f(x_3, f(x_2, f(x_1, acc)))\)

fun foldl \((f, acc, xs)\) =
  case xs of
    [] \Rightarrow acc
  | x :: xs' \Rightarrow \text{foldl} (f, f(x, acc), xs')

\[
\text{val foldl} = \text{fn} : ('a * 'b -> 'b) * 'b * 'a list \rightarrow 'b
\]

Foldl example

val l = []
foldl (max_rank, Ensign, l) (* = Captain *)

Another famous function: Fold

foldr
-
  \(\text{folds "from the right"}\)
foldr \((f, acc, [x_1, x_2, x_3])\) computes
\(f(x_1, f(x_2, f(x_3, acc)))\)

fun foldr \((f, acc, xs)\) =
  case xs of
    [] \Rightarrow acc
  | x :: xs' \Rightarrow f(x, foldr (f, acc, xs'))

\[
\text{val foldr} = \text{fn} : ('a * 'b -> 'b) * 'b * 'a list \rightarrow 'b
\]
Examples with fold

Implement so many other functions with fold!

```ocaml
fun rev xs = foldl(op ::, [], xs)
fun length xs = foldl((fn (_, a) => a+1, 0, xs))
fun map(f, xs) = foldr(fn (x, a) => (f x)::a, [], xs)
fun filter(f, xs) = foldr(fn (x, a) => if f x then x::a else a, [], xs))
```

Examples with fold

These are useful and do not use “private data”

```ocaml
fun f1 xs = foldl((fn (x, y) => x+y), 0, xs)
fun f2 xs = foldl((fn (x, y) => x andalso y>=0), true, xs)
```

These are useful and do use “private data”

```ocaml
fun f3 (xs, hi, lo) = foldl((fn (x, y) =>
  x + (if y >= lo andalso y <= hi
      then 1
      else 0))
    0, xs)
fun f4 (g, xs) = foldl((fn (x, y) => x andalso g y), true, xs)
```

The benefits of iterators

• These “iterator-like” functions are not built-in to the language
  – Just a programming pattern
  – Though many languages have built-in support, which often allows
    stopping early without using exceptions

• This pattern separates recursive traversal from data processing
  – Can reuse same traversal for different data processing
  – Can reuse same data processing for different data structures

CURRYING

Currying and Partial Application

• Recall every ML function takes exactly one argument
• Previously encoded n arguments via one n-tuple
• Another way. Take one argument and return a function that takes
  another argument and…
  – Called “currying” after famous logician Haskell Curry

Haskell B. Curry

Curry-Howard isomorphism

Types are logical formulas
Programs are logical proofs

\[ \text{fn } x \mapsto x : \text{a } \rightarrow \text{a} \]
Currying and Partial Application

- Recall every ML function takes exactly one argument
- Example, with full and partial application:
  ```
  val sorted3 = fn x => fn y => fn z =>
      z >= y andalso y >= x
  val true_ans = ((sorted3 7) 9) 11
  val is_non_negative = (sorted3 0) 0
  ```

Syntactic sugar

Currying is much prettier than we have indicated so far
- Can write `e1 e2 e3 e4` in place of `((e1 e2) e3) e4`
- Can write `fun f x y z = e` in place of `fun f x = fn y => fn z => e`

Result is a little shorter and prettier than the tupled version:

```
fun sorted3 x y z =
    z >= y andalso y >= x
val true_ans =
    sorted3(7,9,11)
val is_non_negative =
    sorted3(0,0,x)
```

Return to the fold 😁

In addition to being sufficient multi-argument functions and pretty, currying is useful because partial application is convenient

Example: Often use higher-order functions to create other functions

```
fun fold f acc xs =
    case xs of
    | [] => acc
    | x::xs' => fold f (f(acc,x)) xs'
fun sum xs = fold (fn (x,y) => x+y) 0 xs
val sum_REST = fold (op+) 0
```

The library's way

- So the ML standard library is fond of currying iterators
- So calling them as though arguments are tupled won’t work
- Another example is `List.exists`:

```
fun exists predicate xs =
    case xs of
    | [] => false
    | x::xs' => predicate x orelse exists predicate xs'
val no = exists (fn x => x=7) [4,11,23]
val has_seven = exists (fn x => x=7)
```

Another example

Currying and partial application can be convenient even without higher-order functions

```
fun zip xs ys =
    case (xs,ys) of
    | ([],[]) => []
    | (x::xs',y::ys') => (x,y)::(zip xs' ys')
    | _ => raise Empty
fun range i j =
    if i>j then []
    else i :: range (i+1)
val countup = range 1 (* partial application * )
fun add_number xs = zip (countup (length xs)) xs
```

More combining functions

- What if you want to curry a tupled function or vice-versa?
- What if a function’s arguments are in the wrong order for the partial application you want?

Naturally, it’s easy to write higher-order wrapper functions
- And their types are neat logical formulas

```
fun other_curry1 f = fn x => fn y => f y x
fun other_curry2 f x y = f y x
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
```
The Value Restriction Appears

If you use partial application to create a polymorphic function, it may not work due to the value restriction:

- Warning about “type vars not generalized”
  - And won’t let you call the function

- This should surprise you; you did nothing wrong but you still must change your code

- See the written lecture summary about how to work around this wart (and ignore the issue until it arises)

- The wart is there for good reasons, related to mutation and not breaking the type system