Today's Agenda

- Standard Library Documentation  (for HW3)
- Anonymous Functions
  - "Unnecessary Function Wrapping"
  - Returning Functions
- High-Order Functions
  - Map
  - Filter
  - Fold
- More Practice
  - Tree example
  - Expression example

What is in a Standard Library?

- Things that you simply can't implement on your own.
  - Creating a timer, opening a file, etc.
- Things that are so common a “standardized” version will save you time and effort
  - List.map, string concatenation, etc.
  - A standard library makes writing and reading code easier.
  - Common operations don't have to be implemented, and are immediately recognizable.

Anonymous Functions

- An expression that evaluates to a new function with no name
- Usually used as an argument or returned from a higher-order function
- Almost equivalent to the following:

```ml
let fun name pattern = expression in name end
```

- The difference is that anonymous functions cannot be recursive!

"Unnecessary Function Wrapping"

- When called both functions will evaluate to the same result
- However, one creates an unnecessary function to wrap `tl`
- Compare to:

```ml
if el then true else false vs. el
```

<table>
<thead>
<tr>
<th>Bad Style</th>
<th>Good Style</th>
<th>TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>if x &gt; 0 then true else false</td>
<td>x &gt; 0</td>
<td></td>
</tr>
<tr>
<td>n_times(fn ys =&gt; tl ys), 3, xs)</td>
<td>n_times(tl, 3, xs)</td>
<td></td>
</tr>
</tbody>
</table>
Returning Functions

- Remember - Functions are first-class values
  - We can return them from functions
- Example:
  ```haskell
define double_or_triple f
define fn x =
  if f x
  then 2 * x
  else 3 * x
```
  - Has type (int -> bool) -> (int -> int)
- The REPL will print (int -> bool) -> int -> int because it never prints an unnecessary parenthesis

High-order Hall of Fame

- `fun filter (f, xs) =`  
  ```haskell
  case xs of
  [] => []
  | x::xs' => if f x
  then x::filter(f, xs')
  else filter(f, xs')
  ```

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

Fold

- Fold (synonyms/close relatives reduce, inject, etc.) is another very famous iterator over recursive structures
- Accumulates an answer by repeatedly applying a function \( f \) to the answer so far
  ```haskell
  fun fold (f, acc, xs) =
  case xs of
  [] => acc
  | x::xs' => fold(f, f(acc, x), xs')
  ```

Practice - Tree Example

- `datatype 'a tree = Empty`  
  ```haskell
  | Node of 'a * 'a tree * 'a tree
  ```
- `fun tree_map = fn : ('a * 'b) * 'a tree -> 'b tree`
- `fun tree_all = fn : ('a -> bool) * 'a tree -> bool`

Practice - Expression Example

- `datatype exp = Constant of int`  
  ```haskell
  | Negate of exp
  | Add of exp * exp
  | Multiply of exp * exp
  | Var of string
  ```
- `fun visit_post_order = fn : exp -> exp * exp -> exp`
- `fun simplify_once = fn : exp -> exp`
- `fun simplify = fn : exp -> exp`