# Contents

1 Higher Order Functions (A Course Numbering Problem)  2  
1.1 Currying  2  
1.2 A Map  2  
1.3 Filter? I barely know her!  3  
1.4 Folds  3  
1.5 Compose  3  

2 Haskell Brooks Curry Has Three Programming Languages Named After Him  4  
   Task 1.  5  
   Task 2.  6  
   Task 3.  6  
   Task 4.  6  

3 Folding is Entirely Overpowered  8  
   Task 5.  8  
   Task 6.  8  
   Task 7.  8  
   Task 8.  9  
   Task 9.  9  
   Task 10.  10  

4 Point-Free Programming  11  
   Task 11.  11  
   Task 12.  11  
   Task 13.  11  
   Task 14.  11  
   Task 15.  11  

5 Better\textsuperscript{TM} Function Application  13  
5.1 Apply  13  
   Task 16.  13  
   Task 17.  13  
5.2 Hype for Pipes  13  
   Task 18.  14  
   Task 19.  14  
   Task 20.  14
1 Higher Order Functions (A Course Numbering Problem)

1.1 Currying

Previously, if we’ve wanted a function to take in multiple arguments, we’ve passed in a tuple of those arguments.

With curried functions, we pass in one argument, and the function application evaluates to another function that takes in the remaining arguments.

Example(s):

1.2 A Map

(* map : *)
fun map

Example(s):
1.3 Filter? I barely know her!

(* filter : *)

fun filter

Example(s):

1.4 Folds

(* foldr : *)

fun foldr

(* foldl : *)

fun foldl

Example(s):

1.5 Compose

(* op o : *)

Example(s):
2 Haskell Brooks Curry Has Three Programming Languages Named After Him

We can define functions in SML like this:

```sml
val addOne : int -> int = fn x => 1 + x
```

SML gives us a convenience syntax to do almost exactly the same thing with `fun`:

```sml
fun addOne (x : int) : int = 1 + x
```

Between the two, `fun` is a bit more powerful as it lets us define things recursively.

When we want to write functions that take in “more than one argument,” perhaps TWO arguments, we could do it like this:

```sml
val add : int * int -> int = fn (x, y) => x + y
```

We can write an equivalent function with `fun`. This syntax is usually preferred.

```sml
fun add (x : int, y : int) : int = x + y
```

Now, suppose we had `add` in scope. We notice that `add` takes in two arguments (of type `int`) and returns their sum.

More pedantically, `add` takes in one argument: a single tuple of two `int`s.

Let’s suppose we want to define `addOne` in terms of `add`. We might do the following:

```sml
fun addOne (x : int) : int = add (1, x)
```

Can we do better? Yes, if we change the way we define `add`.

Right now, we have that `add` takes in a single tuple of two `int`s `x` and `y`, and returns their sum `x + y`. This new `add` takes in a single `int` `x`, and returns a function which (takes in a single `int` `y`, and returns `x + y`).

If we had such an `add`, we could do this:

```sml
val addOne : int -> int = add 1
```

Very clean!

How would we define this `add`? Well, it takes in a single int, so let’s start with that:

```sml
fun add (x : int) : ??? = ???
```

We return a value of type `int -> int`. So let’s write that in as the return type:

```sml
fun add (x : int) : int -> int = ???
```
How can we create a function value to return? With fn.

```latex
fun add \ (x : int) \ : int \rightarrow int = fn \ y => ??
```

Now, what should this new function, which we return, return when we give it a y? It should return \(x + y\).

```latex
fun add \ (x : int) \ : int \rightarrow int = fn \ y => x + y
```

This completes the definition of add.

But there is another way to write add. We could have avoided all the fun and done this:

```latex
val add = (fn x => (fn y => x + y))
```

This avoids mixing fun and fn, which may help to avoid confusion.

However, we like having fun. Is there a way we can write functions with fun that return other functions, without using fn?

It turns out there is:

```latex
fun add \ (x : int) \ (y : int) \ : int = x + y
```

Much better! This is the usual, preferred syntax we use for writing curried functions.

What’s a “curried” function? It’s just a function that takes in multiple arguments not by taking in a single tuple of all the multiple arguments, but instead by taking in a single one of those multiple arguments, and returning a function that takes in the rest.

One last quick note: we can rewrite

```latex
t1 \rightarrow (t2 \rightarrow t3)
```

as

```latex
t1 \rightarrow t2 \rightarrow t3
```

and usually, we do. We can do this because \(\rightarrow\) is right-associative.

Note that in conjunction with this, function application is left-associative. So we can rewrite

```latex
val 4 = (add 1) 3
```

as

```latex
val 4 = add 1 3
```

**Task 1.**

Define \texttt{add3 : int \rightarrow int \rightarrow int \rightarrow int}. It should be that

- for all values \(x : \text{int}\),
- for all values \(y : \text{int}\),
- for all values \(z : \text{int}\),
we have that

\[ \text{add3 } x \ y \ z \equiv x + y + z \]

**Constraint:** Do not use the `fn` keyword.

**Task 2.**

Define \( \text{mul3} : \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int} \). It should be that

- for all values \( x : \text{int} \),
- for all values \( y : \text{int} \),
- for all values \( z : \text{int} \),

we have that

\[ \text{mul3 } x \ y \ z \equiv x \times y \times z \]

**Constraint:** Do not use the `fun` keyword.

**Task 3.**

Define \( \text{curry} : ('a \times 'b \rightarrow 'c) \rightarrow ('a \rightarrow 'b \rightarrow 'c) \). It should be that

- for all types \( t_1 \),
- for all types \( t_2 \),
- for all types \( t_3 \),
- for all values \( f : t_1 \times t_2 \rightarrow t_3 \),
- for all values \( y : t_1 \),
- for all values \( z : t_2 \),

we have that

\[ f (x, y) \equiv (\text{curry } f) x y \]

**Task 4.**

Define \( \text{uncurry} : ('a \rightarrow 'b \rightarrow 'c) \rightarrow ('a \times 'b \rightarrow 'c) \). It should be that

- for all types \( t_1 \),
- for all types \( t_2 \),
- for all types \( t_3 \),
- for all values \( f : t_1 \rightarrow t_2 \rightarrow t_3 \),
- for all values \( y : t_1 \),
- for all values \( z : t_2 \),
we have that

\[
f \, x \, y \cong (\text{uncurry } f) \, (x, \, y)
\]
3 Folding is Entirely Overpowered

Recall the definitions of \texttt{foldl} and \texttt{foldr}:

\begin{verbatim}
fun foldl (cmb : 'a * 'b -> 'b) (z : 'b) (L : 'a list) : 'b =
  case L of
    [] => z
  | x :: xs => foldl cmb (cmb (x, z)) xs

fun foldr (cmb : 'a * 'b -> 'b) (z : 'b) (L : 'a list) : 'b =
  case L of
    [] => z
  | x :: xs => cmb (x, foldr cmb z xs)
\end{verbatim}

Consider that \texttt{folding} generalizes the idea behind many of the functions we’ve written in 15-150 up until now: write a base case \((z)\), and then accumulate some value through each recursive call, based on some part of the value we’re applying the function to \((\texttt{cmb})\). Let’s prove how true this is by rewriting some familiar functions using only \texttt{foldl}/\texttt{foldr}!

\textbf{Task 5.}

Consider

\begin{verbatim}
fun sum (L : int list) : int =
  case L of
    [] => 0
  | x :: xs => x + sum xs
\end{verbatim}

Define \texttt{sum cmb} and \texttt{sum z} such that

\[ \texttt{foldr sum cmb sum z} \cong \texttt{foldl sum cmb sum z} \cong \texttt{sum} \]

\textbf{Task 6.}

Consider

\begin{verbatim}
fun length (L : 'a list) : int =
  case L of
    [] => 0
  | _ :: xs => 1 + length xs
\end{verbatim}

Define \texttt{length cmb} and \texttt{length z} such that

\[ \texttt{foldr length cmb length z} \cong \texttt{foldl length cmb length z} \cong \texttt{length} \]

\textbf{Task 7.}

Consider
fun rev (L : 'a list) : 'a list = 
case L of 
  [] => []
  | x :: xs => rev xs @ [x]

Define rev_cmb and rev_z such that

foldl rev_cmb rev_z ≡ rev

Task 8.

Consider

fun map (f : 'a -> 'b) (L : 'a list) : 'b list = 
case L of 
  [] => []
  | x :: xs => f x :: map f xs

Define map_cmb and map_z such that

• for all types t1,
• for all types t2,
• for all values f : t1 -> t2,

we have that

foldr (map_cmb f) map_z ≡ map f

Task 9.

Consider

fun filter (p : 'a -> bool) (L : 'a list) : 'a list = 
case L of 
  [] => []
  | x :: xs =>
    if p x
    then x :: filter p xs
    else filter p xs

Define filter_cmb and filter_z such that

• for all types t,
• for all values p : t -> bool,

we have that

foldr (filter_cmb p) filter_z ≡ filter p
Task 10.

Warning: this is pretty tricky.

Define foldl_cmb and foldl_z such that

- for all types t1,
- for all types t2,
- for all types t3,
- for all values cmb : t1 * t2 -> t2,
- for all values z : t2,
- for all values xs : t1 list,

we have that

\[ \text{foldr} \ (\text{foldl}_c \text{mb} \ cmb) \ \text{foldl}_z \ xs \ z \cong \text{foldl} \ cmb \ z \ xs \]
4 Point-Free Programming

We’ve provided you with a (NOT infixed) curried version of function composition, co:

\[
\text{co} : (\text{'}b \to \text{'}c) \to (\text{'}a \to \text{'}b) \to \text{'}a \to \text{'}c
\]

This may come in handy for some of the trickier functions below. You may also find the built-in
infix composition function \(\circ\) handy, as well as the built-in List library.

It’s fine if the functions you define have more general types than the ones listed below. Ignore the
value restriction if you run into it.

For each of the following tasks, write the function in code/pointfree/pointfree.sml.

**Constraint:** Define the functions without using `fun` or `fn`.

**Task 11.**

```sml
sum_up : int list -> int
REQUIRES: true
ENSURES: sum_up l sums the elements of l
```

**Task 12.**

```sml
sum_with : int -> int list -> int
REQUIRES: true
ENSURES: sum_up n l sums the elements of l, adding n to the sum
```

**Task 13.**

```sml
sum_with' : int * int list -> int
REQUIRES: true
ENSURES: sum_with' (n, l) sums the elements of l, adding n to the sum
```

**Task 14.**

```sml
sum_both : int list -> int list -> int
REQUIRES: true
ENSURES: sum_both 11 12 sums the elements of 11 and 12
```

**Task 15.**
sum_both' : int list * int list -> int
REQUIRES: true
ENSURES: sum_both' (l1,l2) sums the elements of l1 and l2
5 Better\textsuperscript{TM} Function Application

5.1 Apply

Consider the following: You start out with some piece of data $x : \text{t1}$. You first want to transform it into something else using a function $f_1 : \text{t1} \rightarrow \text{t2}$. Then you want to transform that result with a function $f_2 : \text{t2} \rightarrow \text{t3}$. And so on.

An expression like this will do the trick:

$$f_8 \ (f_7 \ (f_6 \ (f_5 \ (f_4 \ (f_3 \ (f_2 \ (f_1 \ x)))))))$$

There’s problems with this, however.

- There’s a lot of parentheses.
- Everything is written “backwards.” That is, the original piece of data that we start with is written after the function that does the first transformation, which is written after the function that does the second transformation, and so on.

Let’s solve the first problem with an infix operator $\$, pronounced “apply.” Such a $\$ would be defined like this:

```plaintext
infixr $
fun L \$ R = ???
```

We can then use it like this:

$f_8 \ F f_7 \ F f_6 \ F f_5 \ F f_4 \ F f_3 \ F f_2 \ F f_1 \ F x$

Because we said $\$ is a right-associative infix operator (hence \texttt{infixr}), everything will be done in the correct order.

Task 16.
What is the type of $\$?

Task 17.
Define $\$.

5.2 Hype for Pipes

We fixed the problem of lots of parentheses. But everything’s still in the wrong order.

Let’s define a new infix operator $\rightarrow$, pronounced “pipe.” We will be able to use it like this:

$x \rightarrow f_1 \rightarrow f_2 \rightarrow f_3 \rightarrow f_4 \rightarrow f_5 \rightarrow f_6 \rightarrow f_7 \rightarrow f_8 \rightarrow f_9$

Such a $\rightarrow$ would be defined like this:

```plaintext
infix $\rightarrow
fun L \rightarrow R = ???
```

Once you figure our the definition of $\triangleright$, feel free to paste it everywhere in your SML files and call all your functions with it. Doesn’t it read nicely?!?

**Task 18.**

Notice that this time, we said *infix*, not *infixr*. This means $\triangleright$ is **left-associative**. Why does this make sense?

**Task 19.**

What is the type of $\triangleright$?

**Task 20.**

Define $\triangleright$. 