Contents

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 Folds

```
(* foldr :                                                    *)
fun foldr
```

```
(* foldl :                                                    *)
fun foldl
```

Example(s):

1.4 Compose

```
(* op o :                                                     *)
```

Example(s):

2 Adding the Curry

In `addmul.sml`, we have defined the following functions:

```
fun incr (y : int) : int = 1 + y

fun addTwo (x : int) (y : int) : int = x + y
```

Task 2.1.

Which of the following are correct ways to write the type of `addTwo`?
Hint: `->` is **right-associative**.

1. `int -> (int -> int)`
2. `(int -> int) -> int`
3. `int -> int -> int`

The following tasks should be done in `addmul.sml`.

Task 2.2.

Define the function

`incr1 : int -> int`

REQUIRES: true

ENSURES: `incr1` $\cong$ `incr`

Constraint: Declare your function using `val`, and use `addTwo`.

Task 2.3. (Recommended)

Define the function

`add3 : int -> int -> int -> int`

REQUIRES: true

ENSURES: `add3 x y z` $\cong$ `x + y + z`

Note that `add3 x y z` $\cong$ `((add3 x) y) z`, because function application is **left-associative**.

Constraint: Declare your function using `fun`.

Task 2.4.

Define the function

```
mul3 : int -> int -> int -> int
```

REQUIRES: true

ENSURES: `mul3 x y z` $\cong$ `x * y * z`

Constraint: Declare your function using `val`.

3 3...2...1... Blast-HOF!

3.1 Zip it, it's time to Filter

Let's practice implementing some more higher order functions!

For each of the following tasks, write the function in `code/implementing-hofs/implementing-hofs.sml`.

Task 3.1. (Recommended)

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

REQUIRES: `p x` is valuable for all `x` in `L`

ENSURES: Returns the list of those `x` in `L` for which `p x` $\Longrightarrow$ `true`

Task 3.2. (Recommended)

Now, implement the function

```
zipWith : ('a * 'b -> 'c) -> 'a list * 'b list -> 'c list
```

REQUIRES: true

ENSURES: `zipWith f (L1,L2)` $\Longrightarrow$ `L'`, where `L'` $\cong$ `map f (ListPair.zip (L1,L2))`

Note that `ListPair.zip (L1,L2) : 'a list * 'b list -> ('a * 'b) list` takes two lists and "zips" them together by pairing them up element by element until either list runs out of elements. Some examples:

```
ListPair.zip ([1,2,3],["a","b","c"])≅ [(1,"a"),(2,"b"),(3,"c")]
ListPair.zip ([1,2,3,4],["a","b"])≅ [(1,"a"),(2,"b")]
```

Constraint: You may not use `map` or `ListPair.zip` in your implementation. Do this recursively!

In the next task, you will be writing the same functions twice: once recursively without any higher-order functions, and once using HOFs.

Task 3.3.

Implement the function

```
mapPartial : ('a -> 'b option) -> 'a list -> 'b list
```

REQUIRES: true

ENSURES: `mapPartial f L` $\Longrightarrow$ `L'` where `L'` contains all elements `y` such that for `x` in `L`, `f x` $\cong$ `SOME y`

Constraint: For `mapPartial`, you should write this function recursively, without using any built-in HOFs. For `mapPartial'`, you should use higher-order functions, but you may not use `filter` or `map` in your solution.

4 Point-Free Programming

You might find the built-in infix composition function `o` 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`.

First, we are going to implement some functions without the constraint of being point-free (to better prepare you for the point-free versions). We will, however, have the following constraint:

Constraint: Define the functions without using `fun`.

Task 4.1.

```
sum_with_lambda : int -> int list -> int
```

REQUIRES: true

ENSURES: `sum_with_lambda n l` sums the elements of `l`, adding `n` to the sum

Task 4.2.

```
sum_both_lambda : int list -> int list -> int
```

REQUIRES: true

ENSURES: `sum_both_lambda l1 l2` sums the elements of `l1` and `l2`

Now, we are going to implement the same functions except with the following constraint:

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

Task 4.3.

```
sum_with : int -> int list -> int
```

REQUIRES: true

ENSURES: `sum_with n l` sums the elements of `l`, adding `n` to the sum

Task 4.4. (Recommended)

```
sum_with' : int * int list -> int
```

REQUIRES: true

ENSURES: `sum_with' (n,l)` sums the elements of `l`, adding `n` to the sum

Task 4.5.

```
sum_both : int list -> int list -> int
```

REQUIRES: true

ENSURES: `sum_both l1 l2` sums the elements of `l1` and `l2`

Task 4.6.

```
sum_both' : int list * int list -> int
```

REQUIRES: true

ENSURES: `sum_both' (l1,l2)` sums the elements of `l1` and `l2`

5 Don't Nod Off, There's More HoF

Recall the definitions of `foldl` and `foldr`:

```
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)
```

Consider that `fold`ing generalizes the idea behind many of the functions we've written in 15-150 up until now: write a base case (`z`), and then building up a return value by accumulating the result of applying some part of the value to (`cmb`). Let's prove how true this is by rewriting some familiar functions using only `foldl`/`foldr`!

For each of the following tasks, write the function in `code/using-hofs/use-hofs.sml`.

Task 5.1.

Consider

```
fun sum (L : int list) : int =
  case L of
    [] => 0
  | x :: xs => x + sum xs
```

Rewrite `sum` using `foldl`/`foldr`.

Task 5.2. (Recommended)

Consider

```
fun rev ([] : int list) : int list = []
  | rev (x::xs) = (rev xs) @ [x]
```

Rewrite `rev` using `foldl`/`foldr`.

Task 5.3. (Recommended)

Consider

```
fun flatten ([] : int list list) : int list = []
  | flatten (x::xs) = x @ flatten xs
```

Rewrite `flatten` using `foldl`/`foldr`.

Before we move on to more exciting functions we can implement using HOFs, let's take some time to consider how the types of these HOFs (and their inputs) are affected when given an input with a particular type.

Task 5.4.

Consider

```
val boolify = foldl g true [1, 2, 3]
```

What is the type of `g`?

For the following tasks, consider

```
val toStringify = foldr (fn (x, y) => (Int.toString x) ^ y) z
```

Task 5.5.

What is the type of `z`?

Task 5.6.

What is the type of the function `toStringify`?

For the following tasks, consider the datatype `'a idxTree`, defined as

```
datatype 'a idxTree = Empty'
                    | Node' of 'a idxTree * (int * 'a) * 'a idxTree
```

Task 5.7.

Consider

```
val treeify = foldl g (Node'(Empty',(0, 0),Empty')) ["15","1","50"]
```

What is the type of `g`?

Task 5.8.

Now, suppose we had

```
val treeify' = foldl g Empty' [1, 2, 3]
```

Give a type for the function `g` such that the declaration is well-typed.

Task 5.9.

Consider the following code

```
fun inord Empty' = []
  | inord (Node'(L, v, R)) = inord L @ (v :: inord R)

fun treeFold g z T = foldr g z (inord T)
```

What is the type of the function `treeFold`?

Now that we've tried rewriting simple functions using `foldl/foldr` and have considered the types of expressions with HOFs, it's time for us to try something more interesting using all the HoFs we've seen so far!

For each of the following tasks, write the function in `code/using-hofs/use-hofs.sml`.

Task 5.10.

```
maxBy : ('a * 'a -> order) -> 'a list -> 'a option
```

REQUIRES: `cmp` is total

ENSURES:

$$\texttt{maxBy cmp L} \Longrightarrow \begin{cases} \texttt{SOME x} & \text{where } \texttt{x} \text{ is the maximum element in } \texttt{L} \text{ according to } \texttt{cmp} \\ \texttt{NONE} & \text{if the list is empty} \end{cases}$$

Task 5.11.

```
gradebook : int list list -> int list -> int list
```

REQUIRES: For all `S` in `scores`, $|\texttt{S}| = |\texttt{weights}|$

ENSURES: `gradebook scores weights` returns a list `L` of the same length as `scores` such that

$$\texttt{L}[i] \cong \sum_{j=0}^{|\texttt{S}|-1} \texttt{scores}[i][j] * \texttt{weights}[j]$$

Note: Which HoF can we use to combine two lists?

Example:

```
val scores = [[10, 10, 10], [9, 10, 8], [9, 9, 9], [5, 10, 10]]
val weights = [10, 10, 20]
val [400, 350, 360, 350] = gradebook scores weights
```

6 The three best things in life: Money, Pipes, and Curry

The following tasks should be done in `combinators.sml`.

6.1 Apply

Consider the following: You start out with some piece of data `x : t1`. You first want to transform it into something else using a function `f1 : t1 -> t2`. Then you want to transform that result with a function `f2 : t2 -> t3`. And so on.

An expression like this will do the trick:

```
f8 (f7 (f6 (f5 (f4 (f3 (f2 (f1 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:

```
infixr <|
fun L <| R = ???
```

We can then use it like this:

```
f8 <| f7 <| f6 <| f5 <| f4 <| f3 <| f2 <| f1 <| x
```

Because we said `<|` is a **right-associative infix** operator (hence `infixr`), everything will be done in the correct order.

Task 6.1.

What is the type of `<|`?

Task 6.2.

Define `<|`.

Note that in haskell, this operator is the `$` function.

6.2 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 `|>`, pronounced "pipe." We will be able to use it like this:

```
x |> f1 |> f2 |> f3 |> f4 |> f5 |> f6 |> f7 |> f8
```

Such a `|>` would be defined like this:

```
infix |>
fun L |> R = ???
```

Once you figure our the definition of `|>`, feel free to paste it everywhere in your SML files and call all your functions with it. Doesn't it read nicely!?!?

Task 6.3.

Notice that this time, we said `infix`, not `infixr`. This means `|>` is **left-associative**. Why does this make sense?

Task 6.4.

What is the type of `|>`?

Task 6.5.

Define `|>`.

6.3 Curry

Task 6.6. (Recommended)

Define the function

> `curry : ('a * 'b -> 'c) -> ('a -> 'b -> 'c)`
>
> REQUIRES: true
>
> ENSURES: `f (x, y)` ≅ `curry f x y`
>
> Note that we could have written the right hand side as `(curry f) x y`.

Hint: `curry` takes in 3 arguments:

- An uncurried function `f : ('a * 'b -> 'c)`,
- A value `x : 'a`,
- A value `y : 'b`.

Follow the types!

Task 6.7. (Recommended)

Define the function

> `uncurry : ('a -> 'b -> 'c) -> ('a * 'b -> 'c)`
>
> REQUIRES: true
>
> ENSURES: `f x y` ≅ `uncurry f (x,y)`

7 Folding is Entirely Overpowered

Given how general the concept of `fold`ing is, might it be possible to write other HOFs in terms of `fold`? In this task, we'll see that, in fact, it is!

Task 7.1.

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

$$\texttt{foldr (map\_cmb f) map\_z} \cong \texttt{map f}$$

Task 7.2.

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

$$\texttt{foldr (filter\_cmb p) filter\_z} \cong \texttt{filter p}$$

Task 7.3.

Warning: this is quite tricky. Don't worry if you can't get it!

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

$$\texttt{foldr (foldl\_cmb cmb) foldl\_z xs z} \cong \texttt{foldl cmb z xs}$$

Task 7.4.

Explain how you came to your answer to the previous task and why it works.