Anonymity, Polymorphism pt.2, and Higher Order

Anonymous Functions/Unnecessary Function Wrapping

Re-write the following functions as val bindings to anonymous functions:

1. fun double x = x * 2;

   val double = (fn x => x * 2);

2. fun identity x = x

   val identity = (fn x => x);

3. fun apply_to_five f = f 5;

   val apply_to_five = (fn x => x 5);

Re-write the following expressions without unnecessary “wrapping”:

1. if e then true else false → **e**
2. fn x => f x → **f**

Polymorphic Datatypes

Consider the following datatype binding that represents a binary tree:

datatype ('a, 'b) tree = Leaf of 'a | Node of 'b * ('a, 'b) tree * ('a, 'b) tree

- What expressions could this datatype support, and what are their types? List at least 3 here:

  (string,’a) tree [i.e. a leaf with string. For example → Leaf “hi”]

  (bool, string) tree [i.e. a branch with internal node values of bool and children that leaves of type string. For example → Node(“a”, Leaf true, Leaf false)]

  (string, string) tree [i.e. a branch with internal node values of bool and children that leaves of type string. For example → Node(“a”, Leaf “hi”, Leaf “bye”)]

  ...any type ‘a for leaves and any type ‘b for branch values! (as long as they agree)

- What expressions does this datatype **not** support, and what are their types? List at least 3 here:

  Essentially, any type in which either the leaves or branches do not agree. E.g.:

  Node("hi", Leaf true, Leaf "bye")

  Node(1, Leaf false, Leaf "hi")

Higher Order Functions

Write the function definition for the following functions:
(Hint: which of map, filter, and fold could be useful here? Any previous function can be used?)

1. double_all which has type **fn : int list -> int list**. This takes an int list and returns an int list whose elements are twice the original.

   fun double_all xs = map((fn x => x * 2), xs)

2. Write a function join with type **'a list list -> 'a list** using foldr which returns the concatenation of each element in its argument.

   fun join xss = fold((fn (acc, x) => x @ acc), [], xss)

   or…. (closer to standard library)

   fun join xss = foldr((fn (acc, x) => x @ acc), [], xss)
   fun join xss = foldl((fn (acc, x) => acc @ x), [], xss)

3. count_zeros which has type **fn : int list -> int.** This takes an int list and returns the number of times "0" appears.

   fun count_zeros xs = fold((fn (acc,x) => if x=0 then acc+1 else acc), 0, xs)
   fun count_zeros xs = sum(map((fn (x) => if x=0 then 1 else 0), xs))
   fun count_zeros xs = length(filter((fn (x) => x=0), xs))

4. Consider the following definitions (from HW1):

   ```
   type date = int * int * int
   fun day (d : date) = #1 d
   fun month (d : date) = #2 d
   fun year (d : date) = #3 d
   ```

   Write a function number_in_month whose type is **fn : ('a * ''b * 'c) list * ''b -> bool**. This takes a list of dates and a month and returns the number of dates that are in the given month. (hint: which of map, filter, and fold could be useful here?)

```
fun is_in_month((_,m,_), month) = (m = month);

fun number_in_month(dates, month) =
  let
    fun check_date d = is_in_month(d, month)
  in
    length(List.filter check_date dates)
  end

Or…

fun number_in_month(dates, month) =
    length(filter((fn (_,m,_) => m = month), dates))
```

5. Write a function flat_map which has type **fn : ('a -> 'b list) * 'a list -> 'b list**. This function should take a function as its first argument which maps elements of the second argument to lists, and then flat_map should return the concatenation of those lists. (hint: does this sound familiar?)

```
fun flat_map (f, xs) =
  case xs of
     [] => []
   | x::xs' => (f x) @ flat_map (f, xs')
```