F28PL - Programming Languages

Question 2

Tommy Lamb

H00217505

1. (Reasonable) The Bubblesort and Quicksort algorithms all have type `int list -> int list`.
    - Implement Bubblesort in ML.

```
fun bubbleelement [] = []
  | bubbleelement (head::[]) = [head]
  | bubbleelement (head::second::tail) =
    if head>second then
      second::(bubbleelement (head::tail))
    else
      head::(bubbleelement (second::tail));

val bubbleelement = fn: int list -> int list
```

    - This function recursively iterates once over a list, and 'bubbles' the largest value to the last position in the list. List element reordering is achieved by using concatenation to effectively reconstruct the list after hitting the base case as control moves up the stack.

```
open List;
fun bubblesort [] = []
  | bubblesort (head::tail) =
    let
      val bubbledlist = bubbleelement(head::tail)
    in
      bubblesort(take(bubbledlist,(length(bubbledlist)-1)))@[last(bubbledlist)]
    end;

val bubblesort = fn: int list -> int list
```

    - Acting recursively this function applies `bubbleelement` (above) to the argument list *N* times. For each recursive call the last element is essentially dropped from the list, to be concatenated to the end of the result of sorting the first *N-1* elements. It is this function which implements the bubble sort, with the previous function being a nothing more than a helper function. The List structure is required in order to use the inbuilt `take` function.

- Implement Quicksort in ML

```
fun partition pivot [] = ([], [], [])
  | partition pivot (head::tail) =
    let
          val (less, equal, greater) = partition pivot tail
    in
          if head<pivot then
                (head::less, equal, greater)
          else if head>pivot then
                (less, equal, head::greater)
          else
                (less, head::equal, greater)
    end;

val partition = fn: int -> int list -> int list * int list * int list
```

  - Given a list and a pivot value, this function recursively partitions said list into three sub-lists of values greater than (`greater`), equal to (`equal`), and less than (`less`) the pivot value. These three sub-lists are returned in a tuple of form `(less, equal, greater)`.

```
fun quicksort [] = []
  | quicksort (head::tail) =
    let
          val (less, equal, greater) = partition head (head::tail)
    in
          (quicksort(less))@equal@(quicksort(greater))
    end;

val quicksort = fn: int list -> int list
```

  - Much like with the `bubblesort` function above, this function first calls the helper `partition` function to partition the list, before recursively sorting both the `greater` and `less` lists. The result of the recursive calls are then concatenated together to form the final list.

2. (Hard)Write a pair of functions `tofun : int -> ('a->'a)` and `fromfun : ('a->'a) -> int` such that `(fromfun o tofun) n` evaluates to `n` for every `n:int`. Do not use exceptions.

```
fun toFun int =
   let
         val filestream = (TextIO.openOut "F28PL - QUESTION Z - DATA")
   in
         (fn x => x) before (TextIO.output(filestream, Int.toString int) before
TextIO.closeOut filestream)
   end;

val toFun = fn: int -> 'a -> 'a
```

   - While very much not in the spirit of the question, this and the following function satisfy the question specification perfectly. This `toFun` function writes a string representation of the `int` argument to a file named "F28PL - QUESTION Z - DATA" in the current working directory for later retrieval by the `fromFun` function, and returns the function `fn x=> x` to satisfy the type requirements. It should be noted that the file only holds a single `int` at any time, that of the most recent call to `toFun`, and that it will not work on Windows. The functions work correctly on Linux, and presumably via SSH as well, however Poly/ML running on Windows itself has insufficient permissions to create the necessary file. You are free to delete this file at your pleasure.

```
fun fromFun (f:'a->'a) =
   let
         val filestream = (TextIO.openIn "F28PL - QUESTION Z - DATA")
   in
         valOf(Int.fromString (TextIO.inputAll filestream)) before TextIO.closeIn
filestream
   end;

val fromFun = fn: ('a -> 'a) -> int
```

   - By using a type declaration this function takes in a function of type `'a->'a` as specified, before completely ignoring it and retrieving the `int` from the file saved to by `toFun`. This function will raise an exception if the file is not present, or if whatever is in the file cannot be parsed to `int`.

3. (Infernal) Write a function of type `(('a -> 'b) -> 'b) -> 'a`.

   - I am unable to answer this question fully. The closest answer I have achieved is:

```
fun Q3 (f:('a -> 'b) -> 'b) = raise Match;

val Q3 = fn: (('a -> 'b) -> 'b) -> 'c
```

4. (Very hard) Implement the Tower of Hanoi as a function of type `unit -> (int list*int list*int list)`.

```
fun moveTower 0 (pegA, pegB, pegC) = (drop (pegA,1),(hd pegA)::pegB, pegC)
  | moveTower x (pegA, pegB, pegC) =
    let
          val (pegA1, pegC1, pegB1) = moveTower (x-1) (pegA, pegC, pegB)
          val (pegA2, pegB2, pegC2) = moveTower 0 (pegA1, pegB1, pegC1)
          val (pegC3, pegB3, pegA3) = moveTower (x-1) (pegC2, pegB2, pegA2)
    in
          (pegA3, pegB3, pegC3)
    end;

val moveTower = fn:   int -> 'a list * 'a list * 'a list -> 'a list * 'a list * 'a list
```

- This function recursively moves the first x+1 number of elements from the list pegA to the head of list pegB, using list pegC as a go-between. The reason for the (perhaps excessive) use of local declarations is to format the results from each recursive call before passing them to the next recursive call; since the order of arguments is not consistent between each call, neither is the order of results. The use of numbers in the local declarations is a potentially pointless attempt to enforce order of execution, as well as avoiding naming confusion. The result is a tuple of lists representing the three pegs after the move has been completed.

```
fun towerOfHanoi [] = ([],[],[])
  | towerOfHanoi initialTower =
          moveTower ((length initialTower)-1) (initialTower, [], []);

val towerOfHanoi = fn: 'a list -> 'a list * 'a list * 'a list
```

- This wrapper class for moveTower just takes in a list representing the first peg, and then calls moveTower to move the elements of this list to the empty list representing the second peg, pegB. The pattern matching is used to prevent the computer system crashing when the function is called on the empty list.

```
fun hanoiWrap () = towerOfHanoi [0,1,2,3,4,5,6,7,8];

val hanoiWrap = fn: unit -> int list * int list * int list
```

- This second wrapper function is just used to give a function of the form specified in the question, taking in unit and running towerOfHanoi on an arbitrary initial list.

5. (Very hard) What does this program calculate, and how?
`f = lambda x: [[y for j, y in enumerate(set(x)) if (i >> j) & 1] for i in range(2**len(set(x)))]`

   - Given an `iterable` object - x - this program calculates the powerset of x, represented as a list of lists. That is the set of all 2^n possible subsets of the set of elements contained in x. Each subset is calculated as a list of items *y* such that the floored value of $\frac{i}{2^j}$ is odd, where *y* is an element in the set of x, j is the index of element *y* in *x*, and i is an integer in the range from 0 to 2^n. The value n is taken to be the length of the set of elements in x.

     The bitwise shift operator `i>>j` is equivalent to $floor\left(\frac{i}{2^j}\right)$, with the bitwise AND operation `(i >> j) & 1` returning 1 if `(i >> j)` is odd, and 0 if even. This is interpreted by Python as either true (1) or false (0) in the `if` statement, and for every value of j that the bitwise AND operation holds true, y is added to the current subset. This calculation and evaluation is repeated for all values of i.

     As an example, `f` applied to the list `[1,2,3]` returns
     `[ [], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3] ]`