<u>F28PL Coursework ML1 (ML questions). Deadline 26 Oct 2018</u>

- Code must be valid PolyML (not SML).
- Your answer should be a plaintext file with extension .ml and be a literate program in the sense discussed in lectures.
- You can't use library functions if they make the question trivial (e.g. ListPair).
- You can write your own helper functions, if convenient.
- A model answer is at the end of the question set.
- The essay question is worth 20 points. All other questions are worth 10.
- Code should be clearly-written and laid out, and should include a brief explanation in English explaining the design of your code.
- Your answer must take the form of plaintext including the program and a nontrivial collection of tests, which can be cut-and-pasted by your marker into the command line, to test that it works.
- Consistent with the principle that *code is written for humans to read* in the first instance, and for computers to execute only in the second instance, marks will be awarded for *style and clarity.*
- You may use functions defined in answers to previous questions, in later questions. If you do, duplicate the relevant code at the start of the answer and state where it came from, so that each answer is self-contained.
- Use PolyML, not SMLNJ (`rlwrap poly`, not `rlwrap sml`).

1. **Complex number arithmetic**

The **complex numbers** are explained here (and elsewhere):
http://www.mathsisfun.com/algebra/complex-number-multiply.html
Represent a complex integer as an element of the datatype

```
datatype cint = CI of int * int.
```

(So `CI(4,5)` represents 4+5i.)

Implement functions `cadd` and `cmult` of type `cint * cint -> cint` representing complex integer addition and multiplication.

For instance,

```
cadd(CI(1,0),CI(0,1))
```

should compute

```
CI(1,1).
```

Here's a hint for Question 1. Consider:
Question. *Given*

```
datatype myInt = MI of int
```

write a function

```
myAdd : myInt * myInt -> myInt
```

which calculates addition. For example `myAdd(MI 1,MI 1)` should compute `MI 2`.
Answer.
We use pattern-matching as follows:

```
fun myAdd (MI x,MI y) = MI (x+y);
```

Follow with three tests including code and statement of expected result.

2. **Sequence arithmetic**

An **integer sequence** is an element of

```
   type intseq = int list.
```

(So `intseq` is a type alias for a list of integers.)

Implement recursive functions `seqadd` and `seqmult` of type `intseq * intseq -> intseq` that implement pointwise addition and multiplication of integer sequences.

For instance

```
   seqadd([1,2,3],[~1,2,2])
```

should compute

```
   [0,4,5]
```

Please note:

1. *Don't* write error-handling code to handle the cases that sequences have different lengths.
2. *Don't* worry too much if your functions are reported to have type `intseq * intseq -> int list` or `int list * int list -> int list`. Behaviour of type aliases can be hard to control.
3. *Do* worry if your function has type `intseq -> intseq -> intseq`. That is an error.

3. **Matrices**

Matrix addition and multiplication are described here:

- addition: http://www.mathsisfun.com/algebra/matrix-introduction.html
- Multiplication (dot product): http://www.mathsisfun.com/algebra/matrix-multiplying.html

Represent integer matrices as the datatype `intmatrix = IM of intseq list`.

So a matrix is a column of rows of integers.

Write functions

1. `ismatrix : intmatrix -> bool`
   This should test whether a list of lists of integers represents a matrix (so the length of each row should be equal).
2. `matrixshape : intmatrix -> (int * int)`
   This should return a pair that is the number of columns, and the number of rows, in that order.
3. `matrixadd : intmatrix * intmatrix -> intmatrix`
   Matrix addition, which is simply pointwise addition. You may find your previous answers useful.
4. `matrixmult : intmatrix * intmatrix -> intmatrix`
   Similarly for matrix multiplication.

Please note:

1. *Don't* write error-handling code for malformed input, e.g. a column of rows of integers of different lengths, or an attempt to sum matrices of different shapes.
2. The question is ambiguous whether the 0x0 empty matrix `[]` is a matrix (and how about 0xn or nx0?). Flag the ambiguity, make a design decision, and state and justify it. Always consider the edge cases.
3. A "vector" `[1,2,3]` is not a matrix and should raise a type error if fed e.g. to `ismatrix`. But `[[1,2,3]]` and `[[1],[2],[3]]` are matrices.
4. You aren't allowed to use library functions like map or `List.all`. However, defining these is rather easy, for example:

```
  fun mymap f [] = ...
    | mymap f (h::t) = ...
```

4. **Essay-style question**

Write an essay on the ML type system. Be clear, to-the-point, and concise. Convince your marker that you understand:

- Ad-hoc and parametric polymorphism.
- Function types.
- List types and tuple types (and their differences).
- Equality types.
- ML patterns and pattern-matching.

Include short code-fragments (as I do when lecturing) to illustrate your observations.

5. **Bonus question (this question is marked)**

- Write a pair of functions of types
  ```
  ((‘a * ‘b) -> ‘c) -> (‘a -> (‘b -> ‘c))
  ```
  and
  ```
  (‘a -> ‘b -> ‘c) -> ((‘a * ‘b) -> ‘c)
  ```
  and explain why this was a cool question.

6. **Seriously cool bonus question (this question is marked)**

- Write a pair of functions of types
  ```
  int -> (‘a -> ‘a) -> ‘a -> ‘a
  ```
  and
  ```
  ((int -> int) -> int -> ‘a) -> ‘a
  ```
  (Hint: search for “Church numerals”.)

7. **Unmarked question**

- Implement the Tower of Hanoi as a function of type
  ```
  unit -> (int list*int list*int list)
  ```
- Implement Bubblesort and Quicksort in ML.

Model question M:

Write a function

   `sumf : ‘a list -> (‘a -> int) -> int`

that inputs a list `l` and a function `f : ‘a -> int` and outputs

  the sum of `f` applied to all the elements of `l`

(so `sumf [1,2,3] (fn x => x*x)` calculates 1*1+2*2+3*3 = 21).

Model answer:

```
(******************************)
(* Start of answer to Question M
Write a function
   sumf : ‘a list -> (‘a -> int) -> int
that inputs a list l and a function f : int -> int and outputs
   the sum of f applied to all the elements of l
(so sumf [1,2,3] (fn x => x*x) calculates 1*1+2*2+3*3 = 14).
*)

fun sumf [] f = 0
(* If the list is empty then the sum of the empty list is 0 *)
  | sumf (h::t) f = (f h)+(sumf t f);
(* Otherwise calculate (f h) and proceed recursively *)

(* Test 1 (should return 14): *)
sumf [1,2,3] (fn x => x*x);

(* Test 2 (should return 0): *)
sumf [1,2,-3] (fn x => x);

(* Test 3; sum squares of a list of lists (should return true) *)
sumf [[1,2,3],[4,5,6],[7,8,9]] (fn l => sumf l (fn x => x*x))
=
sumf [1,2,3,4,5,6,7,8,9] (fn x => x*x);

(* End of answer to Question M *)
(******************************)
```

Assuming 10 points are awarded, this answer gets:

- 4 points for being a correct, well-structured program,
- 3 points for a clear explanation, and
- 3 points for exhaustive testing.

Your marker is particularly impressed that the third and final test demonstrates understanding of polymorphism.