F28PL OCaml Coursework. Deadline Friday 25 Oct 2019

Fork this project into your own namespace:
https://gitlab-student.macs.hw.ac.uk/f28pl-2019-20/f28pl-2019-20-ocaml-coursework

- The submission of coursework code will be done by you pushing your code to the GitLab server. Only code that has been pushed to your fork of the **f28pl-2019-20-ocaml-coursework** before the deadline will be marked. (We are not using Vision for coursework submission)
- Code must be valid OCaml.
- You can't use library functions if they make the question trivial (e.g. List).
- You can write your own helper functions, if convenient.
- 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 the completed functions, and for some questions you are also required to write some tests in the ***test/*** directories.
- 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.*
- A model answer is in the ***model-answer*** directory.
- You may use functions defined in answers to previous questions, in later questions.
- Use an OCaml interpreter when developing your solution (**ocaml** or **utop**), use the **dune** tool to run the tests against your code, and use **git** to push your commits to the GitLab repository (or use IDE support for OCaml and git if you are more comfortable with that).

Marking scheme

The essay question (the ***ocaml-essay*** directory) is worth 20 points. All other marked questions are worth 10:

Question 1 – complex numbers	10 marks
Question 2 – sequence arithmetic	10 marks
Question 3 - matrices	10 marks
Question 4 – essay question	20 marks
Question 5 – interesting functions	10 marks
Question 6 – Church numerals	10 marks
Question 7 – sorting *(optional)*	0 marks
Total	**Out of 70**

1. Complex number arithmetic

Code in GitLab: *f28pl-2019-20-ocaml-coursework/complex-numbers*

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

```
type complex_number = CI of int*int ;;
```

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

Implement functions `cadd` and `cmult` of type:

```
complex_number -> complex_number -> complex_number
```

representing complex integer addition and multiplication.

For instance,

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

should compute

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

To get all the marks, you must also write the unfinished tests in
test/ComplexNumbersTests.ml

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

```
type 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:*

```
let myAdd (MI x) (MI y) = MI (x+y) ;;
```

2. Sequence arithmetic

Code on GitLab: *f28pl-2019-20-ocaml-coursework/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]
```

To get full marks, you should write the unfinished tests in **test/SequenceArithmeticTests.ml**

Please note:

1. *Don't* write error-handling code to handle the cases that sequences have different lengths, you can assume that the two input lists are of equal length.

3. Matrices

Code on GitLab: *f28pl-2019-20-ocaml-coursework/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

```
type 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.

To get all the marks, you should write the unfinished tests in **test/MatricesTests.ml**

Please note:

1. To kee your code simpler for the `matrixshape`, `matrixadd` and `matrixmult`, 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. Read the tests in the ***test/*** directory to understand the expected output of the matrix add and matrix multiply functions for the `[[]]` and `[]` matrices.

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`.

4. Essay-style question

Code on GitLab: *f28pl-2019-20-ocaml-coursework/ocaml-essay*

Write an essay on OCaml. Be clear, to-the-point, and concise. You should write this essay as an Ocaml file in ***lib/Essay.ml***, using comments to describe your understanding with code segments to demonstrate each one.

Convince your marker that you understand:

- Function type signatures.
- Polymorphism.
- List types and tuple types (and their differences).
- OCaml pattern-matching on values (e.g. integers) and structures (e.g. lists).
- Named and anonymous functions.
- Recursive functions.
- Unit and property based tests.

Include short code-fragments (as I do when lecturing) to illustrate your observations. Use extensive commentary as OCaml comments above each code segment that demonstrates the concept that the comment is describing. For example:

```
(* The code demonstrates how to define an integer value *)
let x : int = 34 ;;
```

You should also add unit tests and property based tests, with comments on what they're aiming to verify, to the ***test/EssayTests.ml*** file.

5. Bonus question (this question is marked)

Code on GitLab: *f28pl-2019-20-ocaml-coursework/functions*

- Implement a pair of functions of types

```
(('a * 'b) -> 'c) -> 'a -> 'b -> 'c
```

and

```
('a -> 'b -> 'c) -> ('a * 'b) -> 'c
```

and explain what these functions do.

To get all the marks, you should write the unfinished tests in **test/FunctionsTests.ml**

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

Code on GitLab: *f28pl-2019-20-ocaml-coursework/church-numerals*

First, add the following to the top of the lib/ChurchNumerals.ml file:

```
type church_numeral = (int -> int) -> int -> int ;;
```

- Implement a pair of functions of types

  ```
  i2c : int -> church_numeral
  ```

  and

  ```
  c2i : chuch_numeral -> int
  ```

  Add the type signatures, then implement the two functions.

  The `i2c` function takes an integer and returns a Church encoding for that integer. E.g.

  For integer 0:
  ```
  i2c 0 f x = x
  ```

  For 1:
  ```
  i2c 1 f x = f x
  ```

  This continues:

  ```
  i2c 2 f x = f (f x)
  i2c 3 f x = f (f (f x))
  i2c 4 f x = f (f (f (f x)))
  ```

  The `c2i` function takes a function and returns an integer. The function that it takes itself takes an (int -> int) function, an integer, and returns an integer.

The idea of c2i is that it takes a church numeral, and applies it to a function that increments an integer by 1, and a 0 integer. The church numeral function will keep applying incrementing function according to how ever many times it is recursively applied, according to the definition above. Here's a start:

```
let c2i : church_numeral -> int =
    fun church_numeral_f -> <complete yourself> ;;
```

Applying i2c to an integer, then applying c2i to that should return the initial integer, e.g.

c2i (i2c 0)) = 0
c2i (i2c 5)) = 5

(Hint: search for "Church numerals".)

Complete the tests in **test/ChurchNumeralsTests.ml**

7. Unmarked question

Code on GitLab: *f28pl-2019-20-ocaml-coursework/sorting*

- Implement Bubblesort and Quicksort in ML.

```
bubble_sort : 'a list -> 'a list
quick_sort : 'a list -> 'a list
```

Write unit tests with some handwritten input/output tests, and property based tests to check that they return the same output list for a randomly generated list of integers. These should be added in ***test/SortingTests.ml***

Model Question/Answer

Write a function

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

that inputs a list and a function `'a -> int` and outputs the sum of `f` applied to all the elements of the list. So:

```
sumf [1,2,3] (fun x -> x*x)
```

Calculates: 1*1+2*2+3*3 = 21

Model answer:

https://gitlab-student.macs.hw.ac.uk/f28pl-2019-20/f28pl-2019-20-ocaml-coursework/blob/master/model-answer/model_answer.ml

Assuming 10 points are awarded, answers to the above questions will in general get:

- 4 points for being a correct, well-structured program,
- 3 points for a clear explanation, and
- 3 points for including well written tests.