

## F28PL Coursework 2, OCaml

(deadline: 27th November 2020)

Fork this project

<https://gitlab-student.macs.hw.ac.uk/f28pl-2020-21/f28pl-2020-21-ocaml-coursework>

into your own namespace.

- 1) Submit by pushing your code to the GitLab server. Only code that has been pushed to **your fork** of the **f28pl-2020-21-ocaml-coursework** before the deadline will be marked.  
We are not using Vision for coursework submission.
- 2) Code must be valid OCaml.
- 3) You cannot use library functions if they make the question trivial.
- 4) You can write your own helper functions, if convenient.
- 5) Code should be clearly written and laid out and should include a brief explanation in English explaining the design of your code.
- 6) 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.
- 7) 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*.
- 8) A model answer is in the **model-answer** directory.
- 9) You may use functions defined in answers to previous questions, in later questions.
- 10) 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).
- 11) Your marker will expect you to deliver code that compiles. **Code that cannot even compile, may score zero marks.**
- 12) If you are in difficulty with this coursework then be in touch with the lecturer ASAP to discuss getting additional support. We can only help you if you ask.

## Marking scheme

The marking scheme below is indicative and may be subject to change. As well as marks for individual questions, there are 10 bonus marks available for writing useful test cases and writing clear well-documented code.

Question 1 – complex numbers	10 marks
Question 2 – sequence arithmetic	10 marks
Question 3 – matrices	20 marks
Question 4 – essay question	20 marks
Question 5 – isomorphic functions	10 marks
Question 6 – Church numerals	10 marks
Question 7 – sums	10 marks
Testing, clarity, and style	10 marks
<b>Total</b>	<b>Out of 100</b>

# 1. Complex numbers

Code in GitLab: [f28pl-2020-21-ocaml-coursework/1-complex-numbers](https://gitlab.com/f28pl-2020-21-ocaml-coursework/1-complex-numbers)

The **complex numbers** are explained here (and elsewhere):

<http://www.mathsisfun.com/algebra/complex-number-multiply.html>

In this question you will represent complex numbers as pairs of integers:

```
type complex_number = int * int
```

So  $(4, 5)$  represents  $4+5j$ .

Implement functions `complex_add` and `complex_mult` of type:

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

representing complex integer addition and multiplication. For instance,

```
complex_add (1, 0) (0, 1)
```

should compute:

```
(1, 1)
```

For full marks, you must also write the unfinished tests in **test/ComplexNumbersTests.ml**

## 2. Sequence arithmetic

Code on GitLab: [f28pl-2020-21-ocaml-coursework/2-sequence-arithmetic](https://gitlab.com/f28pl-2020-21-ocaml-coursework/2-sequence-arithmetic)

An **integer sequence** has the following type

```
type int_seq = int list
```

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

Implement recursive functions `seq_add` and `seq_mult` of type:

```
int_seq -> int_seq -> int_seq
```

that implement pointwise addition and multiplication of integer sequences.

For instance:

```
seq_add [1; 2; 3] [-1; 2; 2]
```

should compute

```
[0; 4; 5]
```

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

Please note:

Do *not* write error-handling code to for the cases that sequences have different lengths  
— you may assume that the two input lists have equal length.

### 3. Matrices

Code on GitLab: [f28pl-2020-21-ocaml-coursework/3-matrices](https://gitlab.com/f28pl-2020-21-ocaml-coursework/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>

An integer matrix has the following type

```
type int_matrix = int_seq list
```

So a matrix is a column of rows of integers.

Write functions

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

For full marks, you should write the unfinished tests in **test/MatricesTests.ml**

Please note:

1. To keep your code simpler for the `matrix_shape`, `matrix_add` and `matrix_mult`, functions do not 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 `is_matrix`. But `[[1; 2; 3]]` and `[[1]; [2]; [3]]` are matrices.
4. Hint: you may find `List.map` and `List.all` helpful and you may use these functions.

## 4. Essay question

Code on GitLab: [f28pl-2020-21-ocaml-coursework/4-essay-question](https://gitlab.com/f28pl-2020-21-ocaml-coursework/4-essay-question)

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

Your essay must discuss the following seven concepts:

1. Function type signatures.
2. Polymorphism.
3. List types and tuple types, and their differences.
4. OCaml pattern-matching on base types (e.g. booleans, integers, and strings), tuples, lists, and algebraic data types.
5. Named and anonymous functions (i.e. lambda expressions).
6. Recursive functions, tail recursion, and accumulating parameters.
7. Unit and property-based tests.

Include short code-fragments (as in the lectures) to illustrate your observations. Use extensive commentary as OCaml comments above each code segment that demonstrates the concept you are discussing. For example:

```
(* The following 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 aim to check, to the **test/EssayTests.ml** file.

## 5. Isomorphic functions

Code on GitLab: [f28pl-2020-21-ocaml-coursework/5-isomorphic-functions](https://gitlab.com/f28pl-2020-21-ocaml-coursework/5-isomorphic-functions)

- Implement a pair of functions of types

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

and

```
uncurry : ('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. Church numerals

Code on GitLab: [f28pl-2020-21-ocaml-coursework/6-Church-numerals](https://gitlab.com/f28pl-2020-21-ocaml-coursework/6-Church-numerals)

Church numerals are a way of encoding natural numbers as functions introduced by Alonzo Church (inventor of the lambda calculus and Alan Turing's PhD supervisor). For the purpose of this question, define the type of Church numerals as follows:

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

- Implement a pair of functions of types

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

and

```
c2i : church_numeral -> int
```

where the `i2c` function takes a non-negative integer `i` and returns the *Church numeral* for `i` and the `c2i` function does the converse.

For 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 two arguments: the increment function that adds one to an integer, and the integer 0. The church numeral



function will increment according to however many times  $\mathbb{f}$  is recursively applied in the definition above.

Applying `i2c` to an integer, then applying `c2i` to the result should give back the original integer, for instance:

`c2i (i2c 0) = 0`

`c2i (i2c 5) = 5`

(Hint: use the type signatures to guide you)

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

## 7. Sums

Code on GitLab: [f28pl-2020-21-ocaml-coursework/7-sums](https://gitlab.com/f28pl-2020-21-ocaml-coursework/7-sums)

At school we are taught the algebra of sums. The goal of this question is to write a program for deciding whether two sum expressions are equivalent. The way we do that on paper is to apply the following laws to sub expressions:

(Left identity)	$0+a = a$
(Right identity)	$a+0 = a$
(Associativity)	$(a+b)+c = a+(b+c)$
(Commutativity)	$a+b = b+a$

For instance, we can show that  $((x+0)+w)+((0+y)+z) = w+(x+(y+z))$  by applying each law once:

```
((x+0)+w)+((0+y)+z)
= (left identity applied to 0+y)
((x+0)+w)+(y+z)
= (right identity applied to x+0)
(x+w)+(y+z)
= (commutativity applied to x+w)
(w+x)+(y+z)
= (associativity applied to the whole expression)
w+(x+(y+z))
```

Your main task is to write comparison functions for deciding whether two sum expressions are equal: a) according to the first three laws, and then b) according to all four laws. Rather than actually applying the laws step-by-step, your code should implement a simple algorithm, described in **lib/Sums.ml** that flattens each sum expression into a standard form making them easy to compare. You should also write a pretty-printer and evaluator for expressions and rewrite your code more concisely by defining a fold operator for expressions.

The full details are given in the comments in **lib/Sums.ml**.

## Model question

Write a function

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

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

```
sum_with [1; 2; 3] (fun x -> x * x) = 1*1 + 2*2 + 3*3 = 14
```

## Model answer

[https://gitlab-student.macs.hw.ac.uk/f28pl-2020-21/f28pl-2020-21-ocaml-coursework/blob/master/model-answer/model\\_answer.ml](https://gitlab-student.macs.hw.ac.uk/f28pl-2020-21/f28pl-2020-21-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.