1 Syntax.

Question 1.1 *Find two ways to parenthesis the* ***add_two*** *function so that it do what you want: add two to an integer* ***g****.*

```
let f a b = a + b

let add_two g = if g = 0 then f 2 0 else f 2 + 1 g - 1
```

Question 1.2 *Find syntax errors in the following function, which goes through two lists of integers down to their last elements, to sum these last two elements and return this amount. There are four errors to correct.*

```
let sums_last l1 l2 = match l1 with
| [] -> match l2 with
        | [] -> 0
        | x2 :: [] -> x2
        | hd2 :: tl2 -> sums_last [] tl2
| x1 :: [] -> match l2 with
        | [] -> x1
        | x2 :: [] -> x1 + x2
        | hd2 @ tl2 -> sums_last [ x1 ] tl2
| hd1 :: tl1 -> sums_last tl1 l2
```

Question 1.3 *Find syntax errors in the following code. There are three errors.*

```
let x a b = a +. b


let f y z = let v = y + 5
if z > y then Printf . printf "%d" z ; z
else x v 0
```

Question 1.4 *For each occurrence of the variables u z and y, indicate where their linker, that is, the declaration that corresponds to this occurrence. To do this, indicate in line comment where the variable, the number of the line of its linker appears.*

```
1  let y = 5.3 ;;
2  let u z =
3  z -. y ;;
```

```
4  let y =
5  let y =
6  y +. 2.5
7  in let z = u y
8  in y *. z
9  in let u = 2.0 in
10  y +. y +. u ;;
```

2 Typing.

Question 2.1 *Give the type of each of the variables of the function and the object returned. Justify your answer.*

```
let f1 (x , y ) z =
        let g a b = a < b in
         if z then g x else g y ;;
```

Question 2.2 *In the following code, what the* ***list_sum*** *and* ***list_or*** *functions do ? What are their types? Is the code of lines 4-6 correctly typed? What is the type of* ***a*** *at each step ? Same questions for lines 8-10.*

```
1  let list_sum p = List . fold_left ( fun x y -> if p y then x + 1 else x ) 0 ;;
2  let list_or = List . fold_left ( fun x y -> x || y ) false ;;
3
4  let a = [] ;;
5  list_sum a ;;
6  list_or a ;;
7
8  let a = ref [] ;;
9  list_sum ! a ;;
10  list_or ! a ;;
```

Question 2.3 *The following type is defined :*

```
type 'a binary_tree = Leaf
```

```
| Node of 'a * 'a binary_tree * 'a binary_tree ;;
```

*Write a **tree_map** function that takes as input a binary tree represented by this type and a function **f**, and which returns the tree obtained by transforming each of the values carried at each node, by **f**. Write a **forall_tree** function that takes as input a binary tree and a predicate (function of type **'a ->bool**), and returns **true** if all labels verify the predicate. What are the types of these two functions ?*

3 Linear search for an optimal path.

In this section we propose to implement an algorithm for finding a path of greater sum in a grid of integers, of linear complexity in the number of boxes of this grid. The next grid with 4 rows and 3 columns is the example grid :

-2	7	6
0	2	-1
1	-3	3
4	-4	5

More details on the paths will be given in the following. Let's look for the moment at integer grids, these are represented in OCaml by the following type :

```
type grid = int list list
```

A grid is therefore represented by a list of lists, where each list represents a column of the grid. For example, the grid :

0	2
1	3

is represented by [[0; 1]; [2; 3]] of type grid.

Question 3.1 *Define an element **g_example**: **grid** representing the example grid.*

Question 3.2 *Define the **height** function: **grid -> int** which, for a given grid **g** inargument, returns the length of the first list of **g**, or an exception if **g** is empty. By example we must have **height g_example = 4.***

*An element **g** is said to be well formed if **g** is not the empty list and if each list that makes up **g** has the same non-zero size.*

Question 3.3 *Define the **wf_grid_exn** function: **grid -> unit** that returns an exception if and only if the given argument is not a well-formed grid*

It is now assumed that we only work with well-formed grids.A path in a grid starts with a box in the first column and then continues with a box in the second column, and so on to get to a box in the last column.

In addition, if the path passes through box ***i*** of column ***j***, it can only continue to column ***j + 1*** through boxes in rows ***i – 1*** *, i or* ***i + 1***. For example, in the example grid, a path containing box *2* must continue with either *6, −1, or 3.*

We further consider that **the first line is below the last line and that the last line is above the first line –** you can imagine the grid like a cylinder. The value of a path is defined in the most natural way possible: it is the sum of the boxes through which it passes.

Question 3.4 *Comment the maximum number that can be obtained by summing the boxes of a valid path in the example grid.*

Question 3.5 *Write a function* ***rotate_up: 'a list -> 'a list*** *which, applied to a non-empty list* $[i_0;i_1; ...; i_n]$ *gives the list* $[i_1; ...; i_n;i_0]$.

Question 3.6 *Write a function* ***rotate_down: 'a list -> 'a list*** *which, applied to a non-empty list* $[i_0;i_1; ...; i_n]$ *gives the list* $[i_n;i_0;i_1; ...; i_{n-1}]$.

Given a grid $[i_0; ..; i_n]$, a partial path in this grid is a path in $[i_0; ...; i_k]$, for a certain **k ≤ n**. In other words, it is a path that begins with the first column but doesn't need to go to the last column.

Question 3.7 Using **rotate_up** and **rotate_down**, define the **best_option** function: **int list -> int list**. *Consider any column* ***j > 0****. This function takes as an argument a list whose position* ***i*** *indicates the largest value of a partial path that ends at the* ***i*** *line of colone* ***j – 1****. It must calculate a list whose position* ***i*** *indicates the largest value of a path that ends in column* ***j –1*** *and that can be extended through line* ***i*** *of colone* ***j****.*[1]

Question 3.8 *Using* ***best_option****, define a* ***sum*** *function:* ***grid -> int list*** *which, given a grid, returns a list whose position* ***i*** *contains the largest value of a path that ends at line* ***i****.*

Question 3.9 *Write a function* ***max_list****:* ***int list -> int*** *that returns the largest element in a list, or an exception if the list is empty.*

Question 3.10 *Write a* ***solve*** *function:* ***grid -> int*** *which, given a grid, Returns the maximum value of a path in this grid.*

---

1 : Note that this calculation does not depend on the values in column j

4 Fibonacci and memorialization.

Question 4.1 *Coding **fibo**: **int -> int** in a recursive and naïve way, i.e. by directly transcribing the following mathematical definition: recall that **$F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$**. Failwith will be used in case of negative argument.*

Question 4.2 *Copy and rename **fibo** to **fibocount**: **int -> int**, as well as calls recursive of course, and add a line at the beginning of the function to increment a counter global **count** (which you will have taken care to declare beforehand as a **ref**). This counter allows you to track the number of calls to **fibocount**. Having initialized **count** to **0**, then executed **fibocount n** for different values of **n**, and repeated the experiment a few times, make a remarks about the **Count**'s growth law as a function of **n**.*

Question 4.3 *Report/Declare a table **f** of size **1024**, initialized to **–1** everywhere. Copy and rename **fibocount** to **memofibocount** by replacing all recursive calls to **fibocount i**, with a call to a **fibocheck i** function.*

*Your **fibocheck i** checks the table to see if **f.(i)** has already been calculated. If **f.(i)=-1** this is not the case, and so **fibocheck** calls **memofibocount i** to get, this value, and then puts it in **f.(i)**. In all cases, **fibocheck i** ends up returning the **f.(i)**.*

Be careful, memofibocount and fibocheck are therefore mutually recursive.

Question 4.4 *Having initialized **count** to **0**, then executed **memofibocount n** for different values of **n<1024**, and repeated the experiment a few times, comment on the growth law of **count** depending of **n**.Give a short explanation of the difference in behaviour compared to **4.1**.*

5 Turtle.

We will program a turtle in a very simple programming language, represented in OCAML by the type:

```
type command = Up | Down | Left | Right | Seq of command list
```

It is assumed that the turtle starts in coordinates (0, 0), where the first coordinate is that of the abscissa and the second that of the ordinate. The terrain is square, with the lower left square having coordinates (−100, −100), and the upper right square being coordinates (100, 100).

Question 5.1 *Write a function **evalpos c: command -> int * int** which, given a set of commands, calculates the final coordinate of the turtle. Throw an exception if the turtle leaves the field.*

Question 5.2 *A version 2 of the turtle has the possibility to repeat **n** times a order. Modify the code of the **command** type and **evalpos** to take this evolution into account.*

Question 5.3 *A version 3 of the turtle has the possibility to repeat an order infinitly. Give the modifications to be made to the* ***command*** *type to take this evolution into account. Write a* ***safety c*** *function:****command -> bool*** *function that returns* ***true*** *if and only if the turtle will never leave the field during this infinite execution.*[1]