INFFPR01-D - Functional programming

Exam procedure

- The students begin with the theoretical part.
- The theoretical part lasts 30 minutes.
- The students deliver their answers for the theoretical part on the official school paper when the time expires.
- After the theoretical part is over, the students can start with the practical part.
- It is not possible to use any help other than what can be found in the exam. If a student is caught using other material, the student's exam is immediately stopped and he/she will be reported to the exam commission.
- The final grade is computed as `score * 10 / 15`.

Theoretical part

Instructions

- For this part you are not allowed to use any notes. The relevant formulas are given in the exam if needed.
- Your answers must be written on the official school paper. Everything that is not on the paper will not be graded.
- Each question awards you with 1 point.

Beta-reduction rules:

Variables:

$$x \to x$$

Function application:

$$(\text{fun } x \to t)\ u \to t[x \mapsto u]$$

Application

$$\text{Given } t \to t' \text{ and } u \to u' \text{ and } t'\ u' \to v$$
$$t\ u \to v$$

Exercise 1:

Given the following untyped lambda-calculus expression:

$$(\text{fun } x\ y \to y)\ (\text{fun } x \to y\ x)$$

replace the requested terms with the elements from the expression in the following lambda-calculus rule that evaluates it:

$$(\text{fun } x \to t)\ u \to t[x \mapsto u]$$

$$\begin{cases} x = \ldots \\ t = \ldots \\ u = \ldots \\ t[x \mapsto u] = \ldots \end{cases}$$

Exercise 2:

Given the following untyped lambda-calculus expression:

$$(\text{fun } x\ y \to t\ x)\ (((\text{fun } x\ y \to y\ x)\ (\text{fun } x \to x))\ (\text{fun } x \to x))$$

replace the requested terms with the elements from the expression in the following lambda-calculus rule that evaluates it:

$$\text{Given } t \to t' \text{ and } u \to u' \text{ and } t'\ u' \to v$$
$$t\ u \to v$$

$$\begin{cases} t = \ldots \\ u = \ldots \\ t' = \ldots \\ u' = \ldots \\ v = \ldots \end{cases}$$

Exercise 3:

Complete the missing types (denoted with ___) in the following code:

```
let foo (x : int -> string) (y : int) : string = x y
let (f : ___) = foo(fun (x : int) -> string x)
```

Exercise 4:

Complete the missing types (denoted with ___) in the following code. The dots denote missing code implementation **that is omitted for brevity and you do not have to complete**:

```
let map (f : 'a -> 'b) -> (l : List<'a>) : List<'b> = ...
let (x : ___) = map (fun (x : int) -> (string x) + "1")
```

Exercise 5:

Complete the missing types (denoted with ___) in the following code. The dots denote missing code implementation **that is omitted for brevity and you do not have to complete**:

```
let curry (f : 'a * 'b -> 'c) : 'a -> 'b -> 'c = ...
let add (x : int, y : int) : int = ...
let (t : ___) = curry add 5
```

Practical part

Instructions

- In this part you must use the provided exam template on your laptop.
- You can only use the data structures that are defined in the exam templates.
- You cannot use the course materials during the exam.
- You cannot have anything open but the provided exam template, including its source files. This means that catching you with anything else open but the exam template will be reported as fraud/irregularity in the exam, including (but not limited to) other instances of the IDE or different files other than those in the exam template.
- You cannot use ANY imperative statement except printing to the standard output. Imperative constructs include (but are not limited to) variables, loops, classes, records with mutable fields.
- You have to decide whether a function is recursive or not, it will not be specified in the signature. This is because some exercises might be solved with or without recursion.
- You are not allowed to use library functions that provide an immediate answer to the question. For instance, if a question asks the implementation of `map2` you are not allowed to simply call the function `List.map2`, which is already built in the F# standard library.
- When you want to submit your exam, call a teacher who will check your final grade and possibly your work. This can happen only once, meaning that the grade you are assigned after calling a teacher is final.
- The automatic checker will tell you at any time the current score and whether it is correct or not. The score that you see is not final, you can fix your exercises as much as you want until you call for a teacher.
- Teachers have veto power over the result of the checker, meaning that if they spot anything wrong in your submission that the checker was not able to verify (for instance the use of imperative code or library functions that are not allowed), they can adjust the grade accordingly.
- You are not authorized to alter the behaviour of the checker or the `main` function of the program when submitting the exam, meaning that the provided code must be exactly as given.

Exercise 1:

Implement a function

```
let rectangle (width : int) (height : int) : string = ...
```

that returns a rectangle of asterisks
Example: calling `rectangle 4 6` returns **in the terminal**:

```
****
****
****
****
****
****
```

Exercise 2:

Implement a function

```
let suffixes (l : List<'a>) : List<List<'a>> = ...
```

that returns all the possible suffixes of a list. A suffix of a list is a subset of its elements where a number of the elements has been removed from the front of the list. The whole list is also considered a suffix of the list itself.
Example: `suffixes [5; 2; 1; 5] = [[5; 2; 1; 5]; [2; 1; 5]; [1; 5]; [5]; []]`

Exercise 3:

Implement a function

```
let nearBy (mapObject : MapObject) (objects : List<MapObject>) (radius : float) :
List<MapObject> = ...
```

For this exercise you can use the method `Distance` defined in `Point`. `MapObject` contains a field of type `Point`. Consider a list of map objects whose type is defined in the template below and a single map object of the same type.

The function `nearBy` finds all map objects within a specified `radius` containing the same `Symbol`.

Exercise 4:

Implement a function

```
let maxBy (projection :  'a -> 'b) (l :  List<'a>) :  Option<'b> = ...
```

that, given a projection function that transforms the element of a list, computes the maximum value among the transformed elements. The function returns `None` if the list is empty.
Example: `maxBy (fun (x,y) -> y) [(1,2);(3,4);(-1,-1)] = Some 4`

Exercise 5:

Implement a function

```
let allPaths (tree :  Tree<'a>) :  List<List<'a>> = ...
```

Given the tree data structure defined in the template below, `allPaths` compute all possible paths starting from the root of the tree. Remember that the path of a tree is a sequence of adjacent nodes starting from the root and ending up in a leaf.