Ocaml

Functors - 1

Summary: The main theme of this module is related to the functor in ocaml.

Version: 1.00

Contents

Chapter I

General rules

- Your project must be realized in a virtual machine.
- Your virtual machine must have all the necessary software to complete your project. These softwares must be configured and installed.
- You can choose the operating system to use for your virtual machine.
- You must be able to use your virtual machine from a cluster computer.
- You must use a shared folder between your virtual machine and your host machine.
- During your evaluations you will use this folder to share with your repository.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter II

Ocaml piscine, general rules

- Every output goes to the standard output, and will be ended by a newline, unless specified otherwise.
- The imposed filenames must be followed to the letter, as well as class names, function names and method names, etc.
- Unless otherwise explicitly stated, the keywords `open`, `for` and `while` are forbidden. Their use will be flagged as cheating, no questions asked.
- Turn-in directories are `ex00/`, `ex01/`, . . . , `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.
- Since you are allowed to use the `OCaml` syntaxes you learned about since the beginning of the piscine, you are not allowed to use any additional syntaxes, modules and libraries unless explicitly stated otherwise.
- The exercices must be done in order. The graduation will stop at the first failed exercice. Yes, the old school way.
- Read each exercise FULLY before starting it! Really, do it.
- The compiler to use is `ocamlopt`. When you are required to turn in a function, you must also include anything necessary to compile a full executable. That executable should display some tests that prove that you've done the exercise correctly.
- Remember that the special token `";;"` is only used to end an expression in the interpreter. Thus, it must never appear in any file you turn in. Regardless, the interpreter is a powerfull ally, learn to use it at its best as soon as possible!
- The subject can be modified up to 4 hours before the final turn-in time.
- In case you're wondering, no coding style is enforced during the `OCaml` piscine. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code he or she can't grade. As usual, big functions are a weak style.
- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are given a certain amount of freedom in how you choose to do the

exercises. However, some piscine day might explicitly cancel this rule, and you will have to respect directions and outputs perfectly.

- Only the requested files must be turned in and thus present on the repository during the peer-evaluation.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

Chapter III

Exercise 00: The Set module and the Set.Make functor

Exercise 00
Exercise 00: The Set module and the Set.Make functor
Turn-in directory : *ex*00/
Files to turn in : `ex00.ml`
Allowed functions : `The Set module`

OCaml's STD lib provides a Set module. This module hides an implementation of sets based on trees. As a consequence, such an efficient implementation of sets needs ordered elements to build the inner tree. The implementation of a set is completly dependant of the type of its elements, that's why OCaml's sets are generated by a functor. The `Set` module exposes 3 things:

OrderedType : The signature of the parameter of the functor.

S : The signature of the actual generated set.

Make : The actual functor to create a set from an ordered type.

Copy the following lines into the file `"ex00.ml"`:

```
let () =
  let set = List.fold_right StringSet.add [ "foo"; "bar"; "baz"; "qux" ] StringSet.empty in
  StringSet.iter print_endline set;
  print_endline (StringSet.fold ( ^ ) set "")
```

Complete the file `"ex00.ml"` in order to achieve the following output:

```
$> ocamlopt ex00.ml && ./a.out
bar
baz
foo
qux
quxfoobazbar
$>
```

Chapter IV

Exercise 01: The Hashtbl module and the Hashtbl.Make functor

	Exercise 01
Exercise 01: The Hashtbl module and the Hashtbl.Make functor	
Turn-in directory : *ex*01/	
Files to turn in : `ex01.ml`	
Allowed functions : `The Hashtbl module, String.length and String.get`	

OCaml's STD lib also provides a hash table. As you can read in the documentation, this module exposes a lot things, including a functorial interface. This functorial interface is composed of several things, but for this exercice, let's focus on: HashedType, S and Make. Copy the following lines into the file `"ex01.ml"`:

```
let () =
  let ht = StringHashtbl.create 5 in
  let values = [ "Hello"; "world"; "42"; "Ocaml"; "H" ] in
  let pairs = List.map (fun s -> (s, String.length s)) values in
  List.iter (fun (k,v) -> StringHashtbl.add ht k v) pairs;
  StringHashtbl.iter (fun k v -> Printf.printf "k = \"%s\", v = %d\n" k v) ht
```

Complete the file `"ex01.ml"` in order to achieve the following output:

```
$> ocamlopt ex01.ml && ./a.out
k = "Ocaml", v = 5
k = "Hello", v = 5
k = "42", v = 2
k = "H", v = 1
k = "world", v = 5
$>
```

`The order of your output might differ from above according to your hash function. A dummy hash function such as length won't be accepted, write a true one that is known.`

Chapter V

Exercise 02: Projections

	Exercise 02
Exercise 02: Projections	
Turn-in directory : *ex*02/	
Files to turn in : `ex02.ml`	
Allowed functions : `Stdlib.fst and Stdlb.snd`	

Enough with OCaml's STD lib, you got it now. It's time to create your first own functor. Your first two functors actually... The goal of this exercice is to write the two functors `MakeFst` and `MakeSnd` and their signature `MAKEPROJECTION` to allow the following code to compile:

```
module type PAIR = sig val pair : (int * int) end
module type VAL = sig val x : int end

(* FIX ME !!! *)

module Pair : PAIR = struct let pair = ( 21, 42 ) end

module Fst : VAL = MakeFst (Pair)
module Snd : VAL = MakeSnd (Pair)


let () = Printf.printf "Fst.x = %d, Snd.x = %d\n" Fst.x Snd.x
```

And to output:

```
$> ocamlopt ex02.ml && ./a.out
Fst.x = 21, Snd.x = 42
$>
```

Chapter VI

Exercise 03: Fixed point

	Exercise 03
Exercise 03: Fixed point	
Turn-in directory : *ex*03/	
Files to turn in : `ex03.ml`	
Allowed functions : `The Stdlib module`	

As `OCaml` lacks fixed point numbers, you're going to add them yourself today. I'd recommend this article from `Berkeley` as a start. If it's good for them, it's good for you. If you have no idea what `Berkeley` is, read this section of their wikipedia page.

Write in the file `"ex03.ml"` a functor `Make` implementing the functor signature `MAKE`, that takes as input modules implementing the signature `FRACTIONNAL_BITS` and outputs modules that implement the signature `FIXED`. The signature `FIXED` is defined as follows:

```
module type FIXED = sig
  type t
  val of_float  : float -> t
  val of_int    : int   -> t
  val to_float  : t     -> float
  val to_int    : t     -> int
  val to_string : t     -> string
  val zero : t
  val one  : t
  val succ : t -> t
  val pred : t -> t
  val min  : t -> t -> t
  val max  : t -> t -> t
  val gth  : t -> t -> bool
  val lth  : t -> t -> bool
  val gte  : t -> t -> bool
  val lte  : t -> t -> bool
  val eqp  : t -> t -> bool  (** physical equality   *)
  val eqs  : t -> t -> bool  (** structural equality *)
  val add  : t -> t -> t
  val sub  : t -> t -> t
  val mul  : t -> t -> t
  val div  : t -> t -> t
  val foreach : t -> t -> (t -> unit) -> unit
end
```

Add the following code to your file `"ex03.ml"`:

```
module Fixed4 : FIXED = Make (struct let bits = 4 end)
module Fixed8 : FIXED = Make (struct let bits = 8 end)

let () =
  let x8 = Fixed8.of_float 21.10 in
  let y8 = Fixed8.of_float 21.32 in
  let r8 = Fixed8.add x8 y8 in
  print_endline (Fixed8.to_string r8);
  Fixed4.foreach (Fixed4.zero) (Fixed4.one) (fun f -> print_endline (Fixed4.to_string f))
```

The output must be:

```
$> ocamlopt ex03.ml && ./a.out
42.421875
0.
0.0625
0.125
0.1875
0.25
0.3125
0.375
0.4375
0.5
0.5625
0.625
0.6875
0.75
0.8125
0.875
0.9375
1.
$>
```

You MUST also provide some additionnal test code to proove that EVERY requested functions in the signature FIXED work as intended. This will be checked during peer-evaluation.

Chapter VII

Exercise 04: Evalexpr is so easy it hurts

	Exercise 04
Exercise 04: Evalexpr is so easy it hurts	
Turn-in directory : *ex*04/	
Files to turn in : `ex04.ml`	
Allowed functions : `The Stdlib module`	

OCaml is very well suited to write expressions evaluation functions. Really. You don't believe me ? You're going to write a functorial evalexpr and you'll be astonished to see for yourself how easy and straight forward it is.

The aim is to write a functor able to generate evalexprs according to an arithmetic module. Such arithmetic modules must implement the following signature:

```
module type VAL =
sig
  type t
  val add : t -> t -> t
  val mul : t -> t -> t
end
```

As you can tell from the above signature, we'll limit our expressions to literals, sums and products for the sake of brevity.

Evalexpr modules must implement an EVALEXPR signature up to you, but it must possess:

- An abstract type `t` defining the type of literals.
- A non abstract type `expr` to represent the expressions the evalexpr is able to evaluate : sums, products and literals.

- A function `eval` that take an expression of type `expr` as a parameter and returns a result literal of type `t`.

Now you have the input and ouput signatures of your functor, write the signature `MAKEEVALEXPR` of said functor. Of course the next step is to write the functor `MakeEvalExpr` implementing the signature `MAKEEVALEXPR`.

Add the following code to your file `"ex04.ml"`. Indeed, the compilation fails. Add the 6 required constraint sharing in that code (or 7 if you missed the one on the functor's signature...). It's sligthly less obvious than in the video, but still easy nonetheless.

```
module IntVal : VAL =
struct
  type t = int
  let add = ( + )
  let mul = ( * )
end

module FloatVal : VAL =
struct
  type t = float
  let add = ( +. )
  let mul = ( *. )
end

module StringVal : VAL =
struct
  type t = string
  let add s1 s2 = if (String.length s1) > (String.length s2) then s1 else s2
  let mul = ( ^ )
end

module IntEvalExpr    : EVALEXPR = MakeEvalExpr (IntVal)
module FloatEvalExpr  : EVALEXPR = MakeEvalExpr (FloatVal)
module StringEvalExpr : EVALEXPR = MakeEvalExpr (StringVal)

let ie = IntEvalExpr.Add (IntEvalExpr.Value 40, IntEvalExpr.Value 2)
let fe = FloatEvalExpr.Add (FloatEvalExpr.Value 41.5, FloatEvalExpr.Value 0.92)
let se = StringEvalExpr.Mul (StringEvalExpr.Value "very ",
                             (StringEvalExpr.Add (StringEvalExpr.Value "very long",
                                                  StringEvalExpr.Value "short")))

let () = Printf.printf "Res = %d\n" (IntEvalExpr.eval ie)
let () = Printf.printf "Res = %f\n" (FloatEvalExpr.eval fe)
let () = Printf.printf "Res = %s\n" (StringEvalExpr.eval se)
```

As a final step, and as a proof of your absolute victory upon functors, use destructive substitution in the constraints sharing of the functor's signature, and on the `IntEvalExpr`, `FloatEvalExpr` and `StringEvalExpr` modules signature bindings.

If you did well, the output must be:

```
$> ocamlopt ex04.ml && ./a.out
Res = 42
Res = 42.420000
Res = very very long
$>
```

That's all folks !!!

This exercise is not mandatory.

Chapter VIII

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.

> The evaluation process will happen on the computer of the evaluated group.