



# OCaml

## Monoids and Monads - 3 (also known as The Day the student stood still)

*Summary: The main theme of this module is to introduce the Monoids and the Monads. This is hard but not as hard as it seems.*

*Version: 1.00*

# Contents

<b>I</b>	<b>Foreword</b>	<b>2</b>
<b>II</b>	<b>General rules</b>	<b>3</b>
<b>III</b>	<b>Ocaml piscine, general rules</b>	<b>4</b>
<b>IV</b>	<b>Day-specific rules</b>	<b>6</b>
<b>V</b>	<b>Exercise 00: All Along the Watchtower</b>	<b>7</b>
<b>VI</b>	<b>Exercise 01: The "Alan Parson's Project"</b>	<b>8</b>
<b>VII</b>	<b>Exercise 02: These aren't the functoids you're looking for</b>	<b>10</b>
<b>VIII</b>	<b>Exercise 03: Try. Or at least try to try. Or die trying.</b>	<b>12</b>
<b>IX</b>	<b>Exercise 04: Game, Set and Match.</b>	<b>13</b>
<b>X</b>	<b>Submission and peer-evaluation</b>	<b>15</b>

# Chapter I

## Foreword

Gentlemen, Ladies, you are the top 1 percent of all OCaml programmers from 42. The elite. The best of the best. We'll make you better ... You might say we'll make you the best of the best of the best. Those of you who can't cut it to graduation will still be the best of the best. But you will simply be the rest of the best of the best, not the best of the best of the best, like the best of you will be.

As programmers in the Almighty 42 School, you are no doubt used to code with the best. But those coders, however good, are not the best of the best. They are only the rest of the best. And while the rest of the best are good, they're obviously not as good as you, the best of the best. Even the worst among you here at 42, or the worst of the rest of the best of the best, are better than even the best of the rest of the best.

Many of you are probably wondering who the best coder here is. That plaque back there is where we list the Top Coders for each class, or the best of the best of the best ... of the best. There can only be one Top Coder per class, so don't be hard on yourselves if you only wind up being the rest of the best of the best ... of the best. You'll still be better than those I mentioned before who couldn't cut it—the rest of the best of the best—and, of course, better than the other coders who weren't even admitted to 42 in the first place—the rest of the best.

So, to recap: There's the best of the best of the best of the best, or 42. Then there's the rest of the best of the best of the best, which is everyone who makes it to graduation and who was previously only the best of the best. Then there are those who couldn't make it to graduation—the worst of the best of the best, who are still better than the best of the rest in the World. Simple.

I don't imagine you have any questions, so I won't even ask. Good luck, ladies and gentlemen. I'll see you in the cluster. Class dismissed.

# Chapter II

## 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 III

## 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 exercises must be done in order. The graduation will stop at the first failed exercise. 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 powerful 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 explicitly 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 IV

## Day-specific rules

- Some themes of this day can be hard to understand. Feel free to practice as much as you can. They will all be used wisely and frequently during your OCaml developer's life.
- You are in a functional programming pool, so your coding style **MUST** be functional (Except for the side effects for the input/output). I insist, your code **MUST** be functional, otherwise you'll have a tedious defence session.
- For each exercise of the day, you must provide sufficient material for testing during the defence session. **Every functionality that can't be tested won't be graded!**
- YOU MUST WATCH This video : [Bryan Beckman - Don't fear the Monad](#)

# Chapter V

## Exercise 00: All Along the Watchtower

	Exercise : 00
Exercise 00: All Along the Watchtower	
Turn-in directory : <i>ex00/</i>	
Files to turn in : *.ml, Makefile	
Forbidden functions : None	

In this exercise you will implement a basic monoid named `Watchtower`. This monoid is an implementation of the Brian Beckman concept of clock-monoid.

Your monoid will contain :

- A type `hour` as an alias of type `int`
- A `zero`
- An `add` rule to add hours according to the concept of a 12 hours clock
- A `sub` rule to sub hours according to the concept of a 12 hours clock

Your monoid will have the following signature:


```
module Watchtower :  
sig  
  type hour = int  
  val zero : hour  
  val add : hour -> hour -> hour  
  val sub : hour -> hour -> hour  
end
```

You will provide sufficient testing for your defence session.



# Chapter VI

## Exercise 01: The "Alan Parson's Project"

	Exercise : 01
Exercise 01: The "Alan Parson's Project"	
Turn-in directory : <i>ex01/</i>	
Files to turn in : *.ml, Makefile	
Forbidden functions : None	

In this exercise you will implement a basic monoid named **App**.

This monoid is an implementation of a project manager.

Your monoid will contain :

- A type `project` as a product type of a string, a string as a status (fail or succeed) and an integer as grade
- A zero which is two empty strings and a 0
- An `combine` rule to combine two projects resulting in a new project with the strings concatenated and the average of grades, if the average is above 80 the status is "succeed", else it's "failed"
- A `fail` rule to create a new project from the project as a parameter with grade equal to 0 and status as "failed"
- A `success` rule to create a new project from the the project as a parameter with grade 80 and status as "succeed"
- Also you will provide a `print_proj` function in your main for testing purpose typed as `App.project -> unit`


You will provide sufficient testing for your defence session.

Your monoid will have the following signature:

```
module App :  
  sig  
    type project = string * string * int  
    val zero : project  
    val combine : project -> project -> project  
    val fail : project -> project  
    val success : project -> project  
  end
```

# Chapter VII

## Exercise 02: These aren't the functors you're looking for

	Exercise : 02
Exercise 02: These aren't the functors you're looking for	
Turn-in directory : <i>ex02/</i>	
Files to turn in : *.ml, Makefile	
Forbidden functions : None	

In this exercise you will implement some arithmetic monoids modules INT and FLOAT to use them in a functor and in various abstract calculation functions.

Your INT and FLOAT modules will contain :

- A type named `element` as an alias of the obvious matching type
- A zero for the addition and subtraction rule named `zero1`
- A zero for the multiplication and division rule named `zero2`
- An `add` and a `sub` rules to add and subtract 2 elements of type `element`
- A `mul` and `div` rules to multiply and divide 2 elements of type `element`

After that you will implement a `Calc` functor which takes a module of type `MONOID` as parameter and the following functions :

- An `add` function which use the `add` of the `Monoid`
- A `sub` function which use the `sub` of the `Monoid`
- A `mul` function which use the `mul` of the `Monoid`
- A `div` function which use the `div` of the `Monoid`
- A `power` function which calculate the power of a parameter `x` by a second parameter of type `int` and always positiv

- A fact function which calculate the factorial of a parameter of type element

You will provide sufficient testing for your defence session.

Your monoids will have the following signature:

```
module type MONOID =
  sig
    type element
    val zero1 : element
    val zero2 : element
    val mul   : element -> element -> element
    val add   : element -> element -> element
    val div   : element -> element -> element
    val sub   : element -> element -> element
  end
```

Your Calc functor will have the following signature :

```
module Calc :
  functor (M : MONOID) ->
  sig
    val add : M.element -> M.element -> M.element
    val sub : M.element -> M.element -> M.element
    val mul : M.element -> M.element -> M.element
    val div : M.element -> M.element -> M.element
    val power : M.element -> int -> M.element
    val fact : M.element -> M.element
  end
```

Below is an basic (ang ugly as hell!) way to check your monoid and your functor.


```
module Calc_int = Calc(INT)
module Calc_float = Calc(FLOAT)

let () =
  print_endline (string_of_int (Calc_int.power 3 3));
  print_endline (string_of_float (Calc_float.power 3.0 3));
  print_endline (string_of_int (Calc_int.mul (Calc_int.add 20 1) 2));
  print_endline (string_of_float (Calc_float.mul (Calc_float.add 20.0 1.0) 2.0))
```

Obviously, there's a lot more to test...

## Chapter VIII

### Exercise 03: Try. Or at least try to try. Or die trying.

	Exercise : 03
Try or try not; there's no try. No, wait...	
Turn-in directory : <i>ex03/</i>	
Files to turn in : *.ml, Makefile	
Forbidden functions : None	

In this exercise you will implement a monad named **Try**, to provide a more functional and elegant way to handle exceptions. An instance of **Try** can be either of:


- **Success** of 'a
- **Failure** of *exn*

Your monad module will implement the following functions:

- **return**: 'a -> 'a Try.t  
Creates a **Success** which contains your value.
- **bind**: 'a Try.t -> ('a -> 'b Try.t) -> 'b Try.t  
Applies a function to your monad, converting it to a **Failure** if your function argument raises an exception. Your function is only applied if your monad is a **Success**.
- **recover**: 'a Try.t -> (exn -> 'a Try.t) -> 'a Try.t  
If your monad is a **Failure**, applies the function to it.
- **filter**: 'a Try.t -> ('a -> bool) -> 'a Try.t  
Converts your monad to a **Failure** if your monad is a **Success** that does not satisfy the predicate given in argument.
- **flatten**: 'a Try.t Try.t -> 'a Try.t  
Flattens a nested Try into a simple Try. Note that a **Success** of **Failure** is a **Failure**.

# Chapter IX

## Exercise 04: Game, Set and Match.

	Exercise : 04
Somebody set up us the bomb.	
Turn-in directory : <i>ex04/</i>	
Files to turn in : *.ml, Makefile	
Forbidden functions : None	

In this exercise you will implement a monad named **Set**, to implement a set. You are free to choose whatever internal implementation you want for your sets, but your module will provide at least the following functions:

- **return:** `'a -> 'a Set.t`  
Creates a singleton containing the value given in argument.
- **bind:** `'a Set.t -> ('a -> 'b Set.t) -> 'b Set.t`  
Applies the function to every element in the Set and returns a new set.
- **union:** `'a Set.t -> 'a Set.t -> 'a Set.t` Returns a new set containing the union of the two sets given in argument.
- **inter:** `'a Set.t -> 'a Set.t -> 'a Set.t`  
Returns a new set containing the intersection of the two sets given in argument.
- **diff:** `'a Set.t -> 'a Set.t -> 'a Set.t` Returns a new set containing the difference between the two sets given in argument.
- **filter:** `'a Set.t -> ('a -> bool) -> 'a Set.t`  
Returns a new set containing only the elements that satisfy the predicate given in argument.
- **foreach:** `'a Set.t -> ('a -> unit) -> unit`  
Executes the function given in argument on every element in the Set.
- **for\_all:** `'a Set.t -> ('a -> bool) -> bool`  
Returns true if all the elements in the set satisfy the predicate given in argument, false otherwise.

- `exists: 'a Set.t -> ('a -> bool) -> bool` Returns true if at least one element in the set satisfies the predicate given in argument, false otherwise.

You will provide sufficient testing for your defence session.



This exercise is not mandatory.

# Chapter X

## 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.