

# OCaml

Recursion and higher-order functions - 0

Summary: In this second OC aml module, you will discover the recurion and the higher-order functions..

Version: 1.00

# Contents

1	General Tules	4
II	Ocaml piscine, general rules	3
III	Day-specific rules	5
IV	Foreword	7
V	Exercise 00: Eat, sleep, X, repeat.	8
VI	Exercise 01: Say what again?	9
VII	Exercise 02: On that day, mankind received a grim reminder.	10
VIII	Exercise 03: Let's do some weird stuff.	12
IX	Exercise 04: Fibonacci!	14
$\mathbf{X}$	Exercise 05: Bazinga!	16
XI	Exercise 06: It goes round and round	18
XII	Exercise 07:until it stops.	20
XIII	Exercise 08: In Too Deep	22
XIV	Exercise 09: It's a pie machine. Chickens go in, pies come out.	24
XV	Submission and peer-evaluation	25

# 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

## Day-specific rules

- Unless otherwise specified, you are **NOT** required to implement your functions with tail recursion.
- But if your function has to be implemented with tail recursion, it obviously means that it has some performance requirements. As such, functions which run slower than O(n) (linear time) will get no points.
- Some of the exercises involve heavy calculations, which means it's okay if some of your functions are slow.
- For the same reason, you cannot have points deducted if your code causes a stack overflow.
- However, any infinite recursion means no points for the exercise.
- Any use of while and/or for is cheating, no questions asked.
- Unless otherwise specified, the exercises you turn in must fit in **ONE** (1) top-level let definition. Use nested definitions and be clever.
- Today's exercises make you write one (or several) functions, but you are required to turn in a **full** working program for each evercise. That means each file you turn in must include one let definition for the exercise you're solving and a let () definition to define a full program with sufficient examples to prove that you have solved the exercise correctly. The examples I'm giving you in the subject usually aren't sufficient. If there's no example to prove a feature is working, the feature is considered non-functional.
- Unless otherwise specified, you cannot use any function from the standard library to solve your exercise. However, you are free to use whatever function you want and see fit to use in the let () definition for your examples. As long as you don't have to link your exercise with a third-party library, use anything you want.
- As stated in the general rules, you cannot use the OCaml structures you haven't yet seen in the videos. Just to make it clear, Any use of the keywords match and with is forbidden and will be considered cheating. No questions asked.
- Though they are also functions, all operators are allowed. Operators are surrounded with parentheses in the Pervasives module's documentation.

# Chapter IV

# Foreword

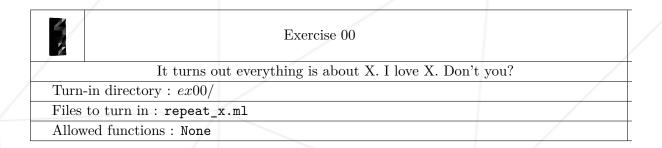
Look at the picture before reading this title.



Please note that "I got stuck in an infinite loop while reading the foreword" is not a valid excuse not to do the exercises.

# Chapter V

# Exercise 00: Eat, sleep, X, repeat.



You will write a function named repeat\_x, which takes an int argument named n and returns a string containing the character 'x' repeated n times.

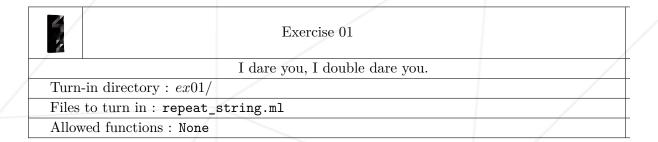
Obviously, your function's type will be int -> string.

If the argument given to the function is negative, the function must return "Error".

```
# repeat_x (-1);;
- : string = "Error"
# repeat_x 0;;
- : string = ""
# repeat_x 1;;
- : string = "x"
# repeat_x 2;;
- : string = "xx"
# repeat_x 5;;
- : string = "xxxxx"
```

# Chapter VI

# Exercise 01: Say what again?



You will write a function named repeat\_string which takes two arguments:

- A string named str
- An integer named n

The function will, of course, return str repeated n times. It must be possible to omit str, and if you do so your function must behave like repeat\_x as stated in the previous exercise. Your function's type will be ?str:string -> int -> string.

If the argument given to the function is negative, the function must behave like repeat\_x as stated in the previous exercise.

```
# repeat_string (-1);;
-: string = "Error"
# repeat_string 0;;
-: string = ""
# repeat_string ~str:"Toto" 1;;
-: string = "Toto"
# repeat_string 2;;
-: string = "xx"
# repeat_string ~str:"a" 5;;
-: string = "aaaaa"
# repeat_string ~str:"what" 3;;
-: string = "whatwhatwhat"
```

# Chapter VII

# Exercise 02: On that day, mankind received a grim reminder.

19	Exercise 02	
A tr	ribute to a person named Ackermann. Not the one from Attack	k on Titan, nor the
	greatest hacker of all times, sadly.	/
Turn-	in directory: $ex02/$	/
Files 1	to turn in : ackermann.ml	/
Allow	ed functions: None	/

You will write a function named ackermann, which will be an implementation of the Ackermann function. The Ackermann function is defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0\\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0\\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

If any argument given to the function is negative, the function must return -1. Obviously, your function's type will be int -> int -> int.Don't forget to look up who Wilhelm Ackermann is and why this function is important in the history of computer science.

```
# ackermann (-1) 7;;
    -: int = -1
# ackermann 0 0;;
    -: int = 1
# ackermann 2 3;;
    -: int = 9
# ackermann 4 1;; (* This may take a while. Don't worry. *)
    -: int = 65533
```



This function is very heavy to compute and will cause a stack overflow if given unreasonable input. Remember, this is expected.

# Chapter VIII

# Exercise 03: Let's do some weird stuff.



#### Exercise 03

A tribute to Mr. Takeuchi. Not the one from Type-MOON, sadly.

Turn-in directory : ex03/

Files to turn in : tak.ml

Allowed functions: None

You will write a function named tak, which will be an implementation of the Tak function.

The Tak function is defined as follows:

$$tak(x,y,z) = \begin{cases} tak(tak(x-1,y,z), tak(y-1,z,x), tak(z-1,x,y)) & \text{if } y < x \\ z & \text{otherwise} \end{cases}$$

Obviously, your function's type will be int -> int -> int -> int. Don't forget to look up who Takeuchi Ikuo is and why this function is important in the history of computer science.



There are different definitions of the tak function, because it has evolved over the years. Note that the definition given in the subject is the only implementation that will be deemed correct.

Recursion and higher-order functions -  $\boldsymbol{0}$ 

OCaml

```
# tak 1 2 3;;
-: int = 3
# tak 5 23 7;;
-: int = 7
# tak 9 1 0;;
-: int = 1
# tak 1 1 1;;
-: int = 1
# tak 0 42 0;;
-: int = 0
# tak 23498 98734 98776;;
-: int = 98776
```

# Chapter IX

## Exercise 04: Fibonacci!

Exercise 04	
A tribute to Fibonacci. Not the pasta, sadly.	
Turn-in directory : $ex04/$	
Files to turn in : fibonacci.ml	/
Allowed functions : None	/

You will write a function named fibonacci, which will be an implementation of the Fibonacci sequence. However, your implementation will be tail-recursive. No tail recursion, no points. The Fibonacci sequence is defined as follows:

$$F(n) = \begin{cases} 0 & \text{if } n = 0\\ 1 & \text{if } n = 1\\ F(n-2) + F(n-1) & \text{if } n > 1 \end{cases}$$

If given a negative argument, your function will return -1. Obviously, your function's type will be int -> int. It's okay not to know about Fibonacci, but you should at least look up why this sequence is important in the history and mathematics and art, and what kind of number it generates...and what the fuck rabbits have to do with all that. Also, don't forget to watch some raving rabbids for a few minutes. That might help you heal your poor brain, badly wounded by imperative languages.



### Example:

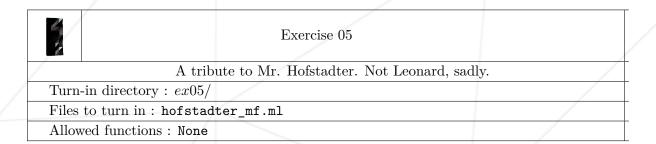
```
# fibonacci (-42);;
- : int = -1
# fibonacci 1;;
- : int = 1
# fibonacci 3;;
- : int = 2
# fibonacci 6;;
- : int = 8
```



Remember that if your function uses more than one top-level let definition, you will get no points from this exercise. Also, the function must be tail recursive and it must run in linear time. If your exercise doesn't comply to these restrictions, it's a failure.

# Chapter X

# Exercise 05: Bazinga!



You will write two functions named hfs\_f and hfs\_m, which will be an implementation of the Hofstadter Female and Male sequences. The Hofstadter Female and Male sequences are defined as follows:

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \\ n - M(F(n-1)) & \text{if } n > 0 \end{cases} \qquad M(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - F(M(n-1)) & \text{if } n > 0 \end{cases}$$

If given a negative argument, your functions must return -1. In case you could not guess, the type of your functions will be int -> int. Of course, don't forget to look up who Douglas Hofstadter is, and watch an episode of The Big Bang Theory if you never have.

```
# hfs_m 0;;
-: int = 0
# hfs_f 0;;
-: int = 1
# hfs_m 4;;
-: int = 2
# hfs_f 4;;
-: int = 3
```



Obviously, each sequence must be implemented only once. Implementing them more than once means you have failed the exercise.



This exercise is not mandatory.

# Chapter XI

# Exercise 06: It goes round and round...

1	Exercise 06	
	You can try to deliver cool lines while in the Lotus Blossom	position, but
Turn-	in directory: ex06/	/
Files	to turn in: iter.ml	/
Allow	ed functions : None	/

You will write a function which takes three arguments: a function of type int -> int, a start argument and a number of iterations. This function is really simple:

$$iter(f, x, n) = \begin{cases} x & \text{if } n = 0\\ f(x) & \text{if } n = 1\\ f(f(x)) & \text{if } n = 2\\ \dots & \text{and so on.} \end{cases}$$

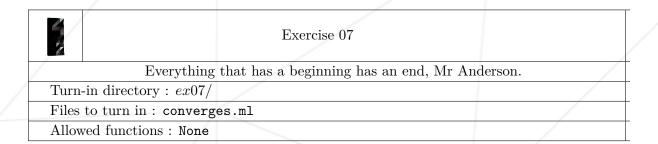
This time I'm not giving you the exact function definition; I could, but then the exercise would be too easy and you haven't been thinking hard enough today.

If n is negative, your function will return -1. Its type will be (int -> int) -> int -> int -> int.

```
# iter (fun x -> x * x) 2 4;;
- : int = 65536
# iter (fun x -> x * 2) 2 4;;
- : int = 32
```

# Chapter XII

# Exercise 07: ...until it stops.



You will write a function named converges which takes the same arguments as iter but returns true if the function reaches a fixed point in the number of iterations given to converges and false otherwise. Its type will be ('a -> 'a) -> 'a -> int -> bool.

A fixed point is an x for which x = f(x). For example, let's say  $f(x) = x^2$ .  $f(1) = 1^2 = 1$ , which means f has a fixed point at 1. If you want more information, well, look it up.

#### Example:

```
# converges (( * ) 2) 2 5;;
- : bool = false
# converges (fun x -> x / 2) 2 3;;
- : bool = true
# converges (fun x -> x / 2) 2 2;;
- : bool = true
```



What is 'a? Wait for Victor to explain that to you tomorrow. Remember to prenounce "alpha", not "quote a". Nobody would understand and you would sound stupid.



This exercise is not mandatory.

# Chapter XIII

## Exercise 08: In Too Deep

		Exercise 08	
	A	tribute to a sum. Not Sum 41, sadly.	
Turn-	in directory: $ex08/$		
Files	to turn in : ft_sum.	nl	/
Allow	red functions : None		/

Starting from now we're going to do some maths, for a change. You've seen some math before in your life, right? We're going to do a summation function, named  $ft_sum$ , which works like  $\Sigma$ . If you don't know what the big scary E-like symbol is, it's called a sigma and you can look it up. Your function will select the following arguments:

- 1. An expression to add: since its value usually depends on the index, that means it will be a function taking the index as parameter,
- 2. The index's lower bound of summation,
- 3. The index's upper bound of summation.

Your function's type will be: (int -> float) -> int -> int -> float, and it will be tail-recursive. No tail recursion, no points. For example, the following expression:

$$\sum_{i=1}^{10} i^2$$

Will be computed using your function as:

```
# ft_sum (fun i -> float_of_int (i * i)) 1 10;;
- : float = 385.
```

If the upper bound is less than the lower bound, ft sum must return nan.



This exercise is not mandatory.

# Chapter XIV

# Exercise 09: It's a pie machine. Chickens go in, pies come out.

1	Exercise 09	
	I don't want to be a pie! I don't like gravy.	/
Turn-in directory : $ex09/$		/
Files	to turn in : leibniz_pi.ml	/
Allov	ved functions: atan, float_of_int	/

Now that you can do a summation, we're going to use your skills to compute  $\pi$ . To do that we'll be using Leibniz's formula, which is fairly easy to understand:

$$\pi = 4 \times \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}$$

Okay, I know it's scary, but it's actually **not** difficult. Now just do what I do: hold on tight and pretend it's a plan.

Of course you can't really go to infinity, only Chuck Norris can — and he did twice. That means we'll stop when we'll reach a minimal delta. A delta is a gap between your computed value and  $\pi$ 's real value. To compute your delta, the reference value to use is:  $\pi = 4 \times \arctan 1$ .

Your function will return the number of iterations needed to reach a minimum delta, which will be given to your function as argument. If the given delta is negative, your function will return -1. Its type will be float -> int, and it will be named leibniz\_pi. Your function must be tail-recursive. No tail recursion, no points.

Phew! That's a wrap. You can grab a drink and chill.



This exercise is not mandatory.

# Chapter XV

# 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.