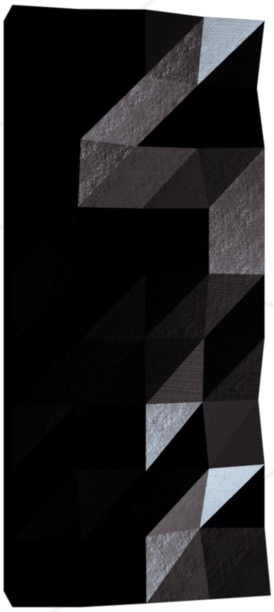

OCaml Module 01

Recursion and higher-order functions

Summary: In this module, we will go deeper into writing recursive function. We will discover the very important notion of tail recursive functions, and you will implement some of the most important functions of computer science history!

Contents

Chapter I

OCaml modules, 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.
- You are only allowed to use the `OCaml` syntaxes you learned about since the OCaml module 00 up this current module or project. You are not allowed to use any additional syntax, modules and libraries unless explicitly stated otherwise.
- The assignments must be done in order. The graduation will stop at the first failed assignment.
- 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!
- No coding style is enforced during the `OCaml` piscine. You can use any style you like, no restrictions. Keep in mind 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 complete the assignments. However, some OCaml modules 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 II

Module specific instructions

- Unless specified otherwise, you are **NOT** required to implement your functions with tail recursion. Each assignment will make it clear.
- Some of the assignments involve some heavy calculations. Therefore it's fine if some of your functions are slow as long as they finish in a matter of minutes.
- Stack overflows and infinite recursion are forbidden.
- Any use of `while` and/or `for` is considered cheating.
- Unless otherwise specified, the exercises you turn in must fit in **ONE (1)** top-level `let` definition. Use nested `let` definitions as you see fit.
- Today's assignments make you write one (or several) functions, but you are required to turn in a **full** working program for each assignment. That means each file you turn in must include one `let` definition for the exercise you're solving and a `let ()` definition as an entry point 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 explicitely, you cannot use any function from the standard library to solve an assignment. 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 from the next OCaml modules and related videos. In particular, the keywords `match` and `with` are forbidden for now and will be considered cheating in this module.
- All operators arythmetic operators are allowed. Operators are surrounded with parentheses in the Pervasives module's documentation.

Chapter III

Foreword

Look at the picture before reading this title.

Chapter IV

Exercise 00: Eat, sleep, x, repeat.

	Exercise 00
It turns out everything is about x	
Turn-in directory : *ex*00/	
Files to turn in : `repeat_x.ml`	
Allowed functions : `None`	

Write a function named `repeat_x`, which takes an `int` argument named `n` and returns a string containing the character `'x'` repeated `n` times.

Your function's type must be `int -> string`.

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

Example:

```
# 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 V

Exercise 01: Say what again?

	Exercise 01
I dare you, I double dare you.	
Turn-in directory : *ex*01/	
Files to turn in : `repeat_string.ml`	
Allowed functions : `None`	

Write a function named `repeat_string` which takes two arguments:

- A string named `str`
- An integer named `n`

The function must 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.

Example:

```
# 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 VI

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

	Exercise 02
A tribute to a person named Ackermann. Not the one from Attack on Titan, nor the greatest hacker of all times, sadly.	
Turn-in directory : *ex*02/	
Files to turn in : `ackermann.ml`	
Allowed functions : `None`	

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`. 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.

Example:

```
# 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. This is expected.

Chapter VII

Exercise 03

	Exercise 03
A tribute to Mr. Takeuchi. Not the one from Type-MOON, sadly.	
Turn-in directory : *ex*03/	
Files to turn in : `tak.ml`	
Allowed functions : `None`	

Write a function named `tak` defined as follows:

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

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. Please use the one defined above.

Example:

```
# 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 VIII

Exercise 04: BWAAAAAAAAAAH!

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

Write a function named `fibonacci`, which will be an implementation of the Fibonacci sequence.

Your implementation must be tail-recursive.

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`. 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 heck rabbits have to do with all that.

Example:

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

Chapter IX

Exercise 05: Bazinga!

	Exercise 05
A tribute to Mr. Hofstadter. Not Leonard, sadly.	
Turn-in directory : *ex*05/	
Files to turn in : `hofstadter_mf.ml`	
Allowed functions : `None`	

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:

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

Your functions must me mutually recursive. If given a negative argument, your functions must return `-1`. The type of your functions must 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.

Example:

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

Chapter X

Exercise 06: It goes round and round...

	Exercise 06
You can try to deliver cool lines while in the Lotus Blossom position, but...	
Turn-in directory : *ex*06/	
Files to turn in : `iter.ml`	
Allowed 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:

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

This time I'm not giving you the exact function definition; I could, but then the exercise would be too easy.

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

Example:

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

Chapter XI

Exercise 07: ...until it stops.

	Exercise 07
Everything that has a beginning has an end, Mr Anderson.	
Turn-in directory : *ex*07/	
Files to turn in : `converges.ml`	
Allowed functions : `None`	

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 type 'a ("alpha")? Well Victor will explain that to you tomorrow. For today, just remember to pronounce "alpha", not "quote a".`

Chapter XII

Exercise 08: In Too Deep

	Exercise 08
A tribute to a sum. Not Sum 41, sadly.	
Turn-in directory : *ex*08/	
Files to turn in : `ft_sum.ml`	
Allowed functions : `None`	

Ok, let's do some maths now for a change, ok? We're going to write a summation function named `ft_sum`, which works like Σ. 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 take 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 must be: `(int -> float) -> int -> int -> float`, and it must be **tail-recursive**. 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`.

Chapter XIII

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

	Exercise 09
I don't want to be a pie! I don't like gravy.	
Turn-in directory : *ex*09/	
Files to turn in : `leibniz_pi.ml`	
Allowed functions : `atan, float_of_int`	

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

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

Of course you can't really count 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 π'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.

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