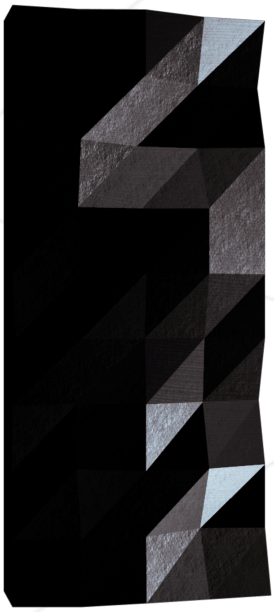


Ocaml piscine - D00

Basic syntaxes and semantics

Staff 42 bocal@staff.42.fr

Abstract: This document is the subject for day 00 of 42's Ocaml piscine.

Contents

Chapter I

Foreword

Black metal is an extreme subgenre and subculture of heavy metal music. Common traits include fast tempos, a shrieking vocal style, highly or heavily distorted guitars played with tremolo picking, raw (lo-fi) recording, unconventional song structures and an emphasis on atmosphere. Artists often appear in corpse paint and adopt pseudonyms.

During the 1980s, several thrash and death metal bands formed a prototype for black metal. This so-called first wave included bands such as Venom, Bathory, Mercyful Fate, Hellhammer and Celtic Frost. A second wave arose in the early 1990s, spearheaded by Norwegian bands such as Mayhem, Darkthrone, Burzum, Immortal and Emperor. The early Norwegian black metal scene developed the style of their forebears into a distinct genre. Norwegian-inspired black metal scenes emerged throughout Europe and North America, although some other scenes developed their own styles independently.

Figure I.1: Abbath from the Norwegian black metal band Immortal

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 keyword `open` is forbidden. its 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 additionnal 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 exercice right.
- 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. Anyway, the interpreter is a powerfull ally, learn to use it at its best as soon as possible !
- The subject can be modified up to 4h 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 she or he can't grade. As usual, big fonctions is a weak style.

- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are afforded a certain amount of freedom in how you choose to do the exercises. Anyway, 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

For this day, you must respect directions and outputs perfectly. A single character mismatch means that the exercice is wrong, althought you are still free to wrap theses outputs as you wish. For instance, the first exercice of the days expects the words "positive" or "negatives" followed by a new line. You made three tests to prove your work:

This output is right:

```
$> ocamlopt ft_test_sign.ml
$> ./a.out
positive
positive
negative
$>
```

This output is also right:

```
$> ocamlopt ft_test_sign.ml
$> ./a.out
Test with [42]: positive
Test with [0]: positive
Test with [-42]: negative
$>
```

This output is WRONG:

```
$> ocamlopt ft_test_sign.ml
$> ./a.out
positive positive negative
$>
```

This output is also WRONG:

```
$> ocamlopt ft_test_sign.ml
$> ./a.out
Test with [42]: [positive]
Test with [0]: [positive]
Test with [-42]: [negative]
$>
```

Chapter IV

Exercise 00: ft_test_sign

	Exercise 00
Exercise 00: ft_test_sign	
Turn-in directory : *ex*00/	
Files to turn in : `ft_test_sign.ml`	
Allowed functions : `print_endline`	
Remarks : `n/a`	

Write a function `ft_test_sign` of type `int -> unit` that displays `"positive"` or `"negative"` followed by a new line, depending on the sign of the parameter. `0` is always considered positive.

Exemples in the interpreter:

```
# ft_test_sign 42;;
positive
- : unit = ()
# ft_test_sign 0;;
positive
- : unit = ()
# ft_test_sign (-42);;
negative
- : unit = ()
#
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter V

Exercise 01: ft_countdown

	Exercise 01
Exercise 01: ft_countdown	
Turn-in directory : *ex*01/	
Files to turn in : `ft_countdown.ml`	
Allowed functions : `print_int and print_char`	
Remarks : `n/a`	

Write a function `ft_countdown` of type `int -> unit` that displays a countdown from the parameter's value down to `0` and a new line after each value. If the value is negative, just display `0` and a new line.

Exemples in the interpreter:

```
# ft_countdown 3;;
3
2
1
0
- : unit = ()
# ft_countdown 0;;
0
- : unit = ()
# ft_countdown (-1);;
0
- : unit = ()
#
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter VI

Exercise 02: ft_power

	Exercise 02
Exercise 02: ft_power	
Turn-in directory : *ex*02/	
Files to turn in : `ft_power.ml`	
Allowed functions : `Nothing`	
Remarks : `n/a`	

Write a function `ft_power` of type `int -> int -> int` that returns first parameter power the second parameter. Both parameters will always be positives or equal to `0`, but both will never be equal to `0` at the same time.

Exemples in the interpreter:

```
# ft_power 2 4;;
- : int = 16
# ft_power 3 0;;
- : int = 1
# ft_power 0 5;;
- : int = 0
#
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter VII

Exercise 03: ft_print_alphabet

	Exercise 03
Exercise 03: ft_print_alphabet	
Turn-in directory : *ex*03/	
Files to turn in : `ft_print_alphabet.ml`	
Allowed functions : `char_of_int, int_of_char and print_char`	
Remarks : `n/a`	

Write a function `ft_print_alphabet` of type `unit -> unit` that displays the alphabet on a single line followed by a new line.

Exemple in the interpreter:

```
# ft_print_alphabet ();;
abcdefghijklmnopqrstuvwxyz
- : unit = ()
#
```

Be sure to provide a tests suite to prove that your funciton works as intended during peer-evaluation ("tests suite" might be a little bit over-powered word for this exercice).

Chapter VIII

Exercise 04: ft_print_comb

	Exercise 04
Exercise 04: ft_print_comb	
Turn-in directory : $ex04/$	
Files to turn in : `ft_print_comb.ml`	
Allowed functions : `print_int and print_string`	
Remarks : `n/a`	

Write a function `ft_print_comb` of type `unit -> unit` that displays in ascending order all the different combinaisons of 3 digits, each digit different from the 2 others, and the 3 digits also in ascending order. Each combinaison is separated from the next one by a comma and a space. Finish your display by a new line.

You must have something that starts an finishes that way :

```
# ft_print_comb ();;
012, 013, 014, 015, 016, 017, 018, 019, 023, <more numbers>, 789
- : unit = ()
#
```

As additonnal informations, `987` is not part of the sequence because `789` is already part of it. Also note that for instance, `999` is not part of the sequence because the 3 digits are not different from the 2 others.

Displaying the right answer in a big string without actually computing it will be treated as cheating during the peer-evaluation.

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation (“tests suite” might be a little bit over-powered word for this exercice).

Chapter IX

Exercise 05: ft_print_rev

	Exercise 05
Exercise 05: ft_print_rev	
Turn-in directory : *ex*05/	
Files to turn in : `ft_print_rev.ml`	
Allowed functions : `print_char, String.get and String.length`	
Remarks : `n/a`	

Write a function `ft_print_rev` of type `string -> unit` that prints its `string` parameter in reverse order, one character at a time, ending with a new line.

Exemple in the interpreter :

```
# ft_print_rev "Hello world !";;
! dlrow olleH
- : unit = ()
# ft_print_rev "";;

- : unit = ()
#
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter X

Exercise 06: ft_string_all

	Exercise 06
Exercise 06: ft_string_all	
Turn-in directory : *ex*06/	
Files to turn in : `ft_string_all.ml`	
Allowed functions : `String.get and String.length`	
Remarks : `n/a`	

Write a function `ft_string_all` of type `(char -> bool) -> string -> bool`. To help you get on tracks, the first parameter, of type `char -> bool`, is a function. As this function returns a `bool`, it can therefore be refered as a "predicate" function.

So, the function `ft_string_all` takes a predicate function and a string as parameters, and applies each character of the string to the predicate function. If the predicate is `true` for every character of the string, `ft_string_all` returns `true`. Otherwise, if the predicate function is `false` for at least one character, `ft_string_all` returns `false`.

Exemples in the interpreter :

```
# let is_digit c = c >= '0' && c <= '9';;
val is_digit : char -> bool = <fun>
# ft_string_all is_digit "0123456789";;
- : bool = true
# ft_string_all is_digit "012EAS67B9";;
- : bool = false
#
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter XI

Exercise 07: ft_is_palindrome

	Exercise 07
Exercise 07: ft_is_palindrome	
Turn-in directory : *ex*07/	
Files to turn in : `ft_is_palindrome.ml`	
Allowed functions : `String.get and String.length`	
Remarks : `n/a`	

Write a function `ft_is_palindrome` of type `string -> bool` that returns `true` if the string parameter is a palindrome character by character, `false` otherwise. If you intend to use your previous function `ft_string_all`, please embed its code in the file `ft_is_palindrome.ml` as well. The empty string is a palindrome.

Exemples in the interpreter:

```
# ft_is_palindrome "radar";;
- : bool = true
# ft_is_palindrome "madam";;
- : bool = true
# ft_is_palindrome "car";;
- : bool = false
# ft_is_palindrome "";;
- : bool = true
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter XII

Exercise 08: ft_rot_n

	Exercise 08
Exercise 08: ft_rot_n	
Turn-in directory : *ex*08/	
Files to turn in : `ft_rot_n.ml`	
Allowed functions : `char_of_int, int_of_char and String.map`	
Remarks : `n/a`	

Write a function `ft_rot_n` of type `int -> string -> string`. Let `n` the first parameter and `str` the second parameter, `ft_rot_n` rotates each lower case and upper case alphabetical characters of `str` of `n` in acsending order. The value `n` will always be positive.

Exemples in the interpreter:

```
# ft_rot_n 1 "abcdefghijklmnopqrstuvwxyz";;
- : string = "bcdefghijklmnopqrstuvwxyza"
# ft_rot_n 13 "abcdefghijklmnopqrstuvwxyz";;
- : string = "nopqrstuvwxyzabcdefghijklm"
# ft_rot_n 42 "0123456789";;
- : string = "0123456789"
# ft_rot_n 2 "OI2EAS67B9";;
- : string = "QK2GCU67D9"
# ft_rot_n 0 "Damned !";;
- : string = "Damned !"
# ft_rot_n 42 "";;
- : string = ""
# ft_rot_n 1 "NBzlk qnbjr !";;
- : string = "OCaml rocks !"
```

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation.

Chapter XIII

Exercise 09: ft_print_comb2

	Exercise 09
Exercise 09: ft_print_comb2	
Turn-in directory : *ex*09/	
Files to turn in : `ft_print_comb2.ml`	
Allowed functions : `print_char, print_int`	
Remarks : `n/a`	

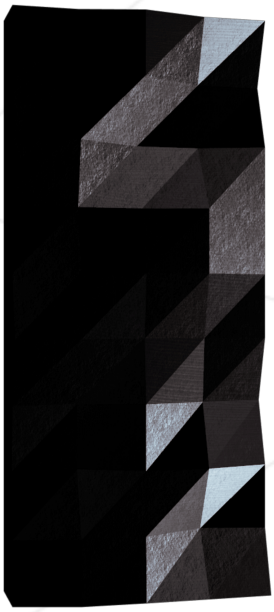
Write a function `ft_print_comb2` of type `unit -> unit` that displays each unique combinaisons of two numbers, each one between `00` and `99`, in ascending order. Each combinaison is separated from the next one by a comma and a space. Finish your display by a new line.

You must have something that starts an finishes that way :

```
# ft_print_comb2 ();;
00 01, 00 02, 00 03, 00 04, 00 05, <more numbers>, 00 99, 01 02, <more numbers>, 97 99, 98 99
- : unit = ()
#
```

Displaying the right answer in a big string without actually computing it will be treated as cheating during the peer-evaluation.

Be sure to provide a tests suite to prove that your function works as intended during peer-evaluation (“tests suite” might be a little bit over-powered word for this exercice).



d01

Recursion and higher-order functions

42 staff pedago@staff.42.fr
nate alafouas@student.42.fr

Abstract: *This is the subject for d01 of the OCaml pool.*

Contents

Chapter I

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 II

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.

- Do use your brains. **PLEASE**. For your own sake, think before you make an idiot out of yourself on the forum.

Chapter III

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. You will be bitchslapped if you try to use that excuse, because you tried to be funny. And you are not.

Chapter IV

Exercise 00: Eat, sleep, X, repeat.

	Exercise 00
It turns out everything is about X. I love X. Don't you?	
Turn-in directory : *ex*00/	
Files to turn in : `repeat_x.ml`	
Allowed functions : `None`	
Remarks : `n/a`	

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"`.

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`	
Remarks : `n/a`	

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.

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 : $ex02/$	
Files to turn in : `ackermann.ml`	
Allowed functions : `None`	
Remarks : `n/a`	

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.

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.  Remember, this is expected.
```

Chapter VII

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

	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`	
Remarks : `n/a`	

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.`

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 : $ex04/$	
Files to turn in : `fibonacci.ml`	
Allowed functions : `None`	
Remarks : `n/a`	

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 IX

Exercise 05: Bazinga!

	Exercise 05
A tribute to Mr. Hofstadter. Not Leonard, sadly.	
Turn-in directory : $ex05/$	
Files to turn in : `hofstadter_mf.ml`	
Allowed functions : `None`	
Remarks : `n/a`	

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.

Example:

```
# 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.

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 : $ex06/$	
Files to turn in : `iter.ml`	
Allowed functions : `None`	
Remarks : `n/a`	

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 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`.

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`	
Remarks : `n/a`	

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.

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`	
Remarks : `n/a`	

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 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 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`.

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 : $ex09/$	
Files to turn in : `leibniz_pi.ml`	
Allowed functions : `atan, float_of_int`	
Remarks : `n/a`	

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:

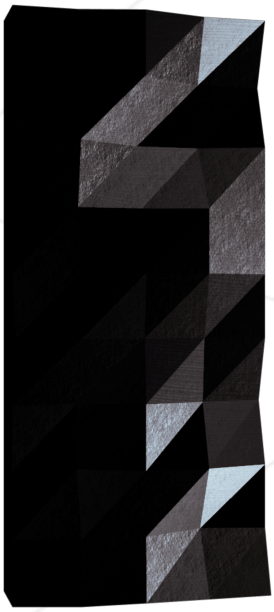
$$\pi = 4 \times \sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}$$

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 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. No tail recursion, no points.

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



OCaml Pool - d02

Pattern Matching and Data Types

42 pedago pedago@staff.42.fr
kashim vbazenne@student.42.fr

Abstract: This is the subject for d02 of the OCaml pool. The main theme of this day is the pattern matching usages and the manipulation of the many constructed types available in OCaml.

Contents

Chapter I

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 additionnal 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 exercice right.
- 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. Anyway, the interpreter is a powerfull ally, learn to use it at its best as soon as possible !
- The subject can be modified up to 4h 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 she or he can't grade. As usual, big fonctions is a weak style.

- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are afforded a certain amount of freedom in how you choose to do the exercises. Anyway, 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 II

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 the pool. The part about basic genetics stuff is not as hard as it seems. Calm down and take a deep breath. Really.

- You are in a functional programming pool, so your coding style MUST be functionnal (Except for the side effects for the input/output). I insist, your code MUST be functionnal, otherwise you'll have a tedious defence session.

- For **EVERY** exercices, you `MUST` provide a full program that runs enough tests to prove that your work is done.

- From excercice 05, you must embed the code of each previous exercices into the next one, i.e., ex06 embeds ex05, ex07 embeds ex05 and ex06, and so on.

Chapter III

Foreword

Here is what wikipedia has to say about the DNA :

Deoxyribonucleic acid is a molecule that encodes the genetic instructions used in the development and functioning of all known living organisms and many viruses. DNA is a nucleic acid; alongside proteins and carbohydrates, nucleic acids compose the three major macromolecules essential for all known forms of life. Most DNA molecules consist of two biopolymer strands coiled around each other to form a double helix. The two DNA strands are known as polynucleotides since they are composed of simpler units called nucleotides. Each nucleotide is composed of a nitrogen-containing nucleobase—either guanine (G), adenine (A), thymine (T), or cytosine (C)—as well as a monosaccharide sugar called deoxyribose and a phosphate group. The nucleotides are joined to one another in a chain by covalent bonds between the sugar of one nucleotide and the phosphate of the next, resulting in an alternating sugar-phosphate backbone. According to base pairing rules (A with T and C with G), hydrogen bonds bind the nitrogenous bases of the two separate polynucleotide strands to make double-stranded DNA. DNA was first discovered by **James Watson** and **Francis Crick**, using experimental data collected by Rosalind Franklin and Maurice Wilkins. The structure of DNA of all species comprises two helical chains each coiled round the same axis, and each with a pitch of 34 ångströms (3.4 nanometres) and a radius of 10 ångströms (1.0 nanometres).

Chapter IV

Exercise 00: Do you even compress bro?

	Exercise 00
Exercise 00: Do you even compress bro?	
Turn-in directory : *ex*00/	
Files to turn in : `encode.ml`	
Allowed functions : `Pervasives module.`	
Remarks : `n/a`	

The Run-length encoding is a very simple form of data compression algorithm. Consecutive elements are stored as single data element and the number of times it repeats. For instance, the string "aaabbb" can be stored as "3a3b".

Write a function *encode* that encode a list of elements to a list of tuples containing the element and the number of times it repeats. The function must be typed as :

```
val encode : 'a list -> (int * 'a) list
```

In case of an empty list as parameter, the function should return an empty list too.

Chapter V

Exercise 01: Crossover

	Exercise 01
Exercise 01: Crossover	
Turn-in directory : $ex01/$	
Files to turn in : `crossover.ml`	
Allowed functions : `Pervasives module`	
Remarks : `n/a`	

Write a function `crossover` that takes two lists as parameters and returns the list of all the common elements between the two lists. The function must be typed as :

```
val crossover : 'a list -> 'a list -> 'a list
```

In case of an empty list as one of the parameters, the function should return an empty list too. But it's obvious isn't it? We don't have to handle duplicates in lists.

Chapter VI

Exercise 02: Fifty Strings of Gray

	Exercise 02
Exercise 02: Fifty Strings of Gray	
Turn-in directory : $ex02/$	
Files to turn in : `gray.ml`	
Allowed functions : `Pervasives except the @ operator and String module`	
Remarks : `n/a`	

The sequence of Gray is a sequence of possible combinations of bits ordered so that when you want to go from one of the element to the following, you only have to shift one bit. It's a way of having a constant time of computing when changing values so that there's no intermediate state that can crash a program. If you have a 2-bits standard set sequence for example : `00 01 10 11`

Assume you are in the state 01, if you want to switch to the next state, you have to change the last bit to 0 and the first one to 1. There could be an intermediate state where the set of bits is 00 before being 10. And that's wrong.

The gray sequence of a set of two bits is as follows : `00 01 11 10`. That way when you pass from 01 to 11 you only have to shift one bit.

Write a function that takes an int n as parameter and write all the strings of the Gray sequence of size n, in the correct order on the standard output, finished by a newline.

```
# gray 1
0 1
- : unit = ()
# gray 2
00 01 11 10
- : unit = ()
# gray 3
000 001 011 010 110 111 101 100
- : unit = ()
```

Chapter VII

Exercise 03: One and one and one is three

	Exercise 03
Exercise 03: One and one and one is three	
Turn-in directory : *ex*03/	
Files to turn in : `sequence.ml`	
Allowed functions : `Pervasives module`	
Remarks : `n/a`	

Assume the following sequence:

```
1
11
21
1211
111221
312211
13112221
...
```

Just like in exercise 00, this sequence generate the element *n* from the element *n-1* and consists of enumerating the count of the numbers found in *n-1*.

1. The first element is 1 so there is one 1 so the second element is 11.
2. The second element is 11 so there are two 1 so the third element is 21.
3. The third element is 21 so there is one 2 and one 1 so the fourth element is 1211.
4. And so on...

Write a function *sequence* that takes an int *n* as parameters and returns the n^{th} element of that sequence as a string. The function must be typed as: `val sequence :`

`int -> string`. In case of an invalid parameter, the function should return an empty string.

Chapter VIII

Exercise 04: DNA -> Nucleotides

	Exercise 04
Exercise 04: DNA -> Nucleotides	
Turn-in directory : *ex*04/	
Files to turn in : `nucleotides.ml`	
Allowed functions : `Pervasives module`	
Remarks : `n/a`	

The very beginning of the dna takes places in a structure consisting of a Phosphate acid linked to a Deoxyribose, itself linked with a nucleobase. A list of many structures is called an helix and two of them makes a DNA sample. Helix: P - D - Base, P - D - Base, ...

- Create the type `phosphate` which is an alias for the string type.
- Create the type `deoxyribose` which is also an alias for the string type.
- Create the variant type `nucleobase`. Its constructors are `A`, `T`, `C`, `G` and `None`.
- Write the *nucleotide* type that contains 3 elements: one `phosphate`, one `deoxyribose` and one `nucleobase`. The kind of type *nucleotide* is is up to you, a record or a tuple will do the trick.
- Write a function `generate_nucleotide` that returns a nucleotide from a given nucleobase passed as a `char`. The function must be typed as `val generate_nucleotide : char -> nucleotide`. Set the `phosphate` value to `"phosphate"` and the `deoxyribose` value to `"deoxyribose"`.

Chapter IX

Exercise 05: DNA -> Helix

	Exercise 05
Exercise 05: DNA -> Helix	
Turn-in directory : *ex*05/	
Files to turn in : `helix.ml`	
Allowed functions : `String concatenation operator, Pervasives module and Random module`	
Remarks : `n/a`	

As seen previously, two helixes can combine to create a DNA structure. As you will see in this exercise, rules are applied when there is a combination. The link of that combination occures where the bases are located: P - D - Base <=> Base - D - P, P - D - Base <=> Base - D - P, ...

- Write an *helix* type that is a list of elements of type *nucleotide*.
- Write a function `generate_helix` that takes an int n as a parameter and construct a random sequence of nucleotides as a list of size n. The function must be typed a : `val generate_helix : int -> helix`.
- Write a function helix_to_string that convert a list of nucleotides as helix type resulting from the previous function to a string of nucleobases. The function must be typed as: `val helix_to_string : helix -> string`.
- Write a function complementary_helix that takes an helix as a parameter and generate the corresponding helix according of the Nucleobase pairing rules that follows :
  - A (Adenine) can be associated with T.
  - T (Thymine) can be associated with A.
  - C (Cytosine) can be associated with G.
  - G (Guanine) can be associated with C.

  The function must be typed as: `val complementary_helix : helix -> helix`

Chapter X

Exercise 06: DNA -> Messenger RNA

	Exercise 06
Exercise 06: DNA -> Messenger RNA	
Turn-in directory : $ex06/$	
Files to turn in : `rna.ml`	
Allowed functions : `Pervasives module`	
Remarks : `n/a`	

A Messenger RNA is a molecule involved in the process of synthesizing proteins. The main aim of the RNA is to create a complementary working copy of a DNA Helix. It's really clever since it prevents the DNA of being altered and allows multiple copies so that the process can be really fast. It was first introduced by scientists Jacques Monod and Francois Jacob.

- Write a type `rna` as a list of elements of type `nucleobase`.
- Write a function `generate_rna` that creates an element of type `rna` from an element of type `helix` according to the following rules :
    - During the creation, the rna is just like a complementary helix except that the `T` nucleobase is switched to `U` nucleobase (Uracil). Modify your type `nucleobase` accordingly. (I told you to read the whole subject before starting...)
    - The list of nucleobases of the rna is the list of nucleobases that are complementary with the original helix's nucleobase (except for the first rule).

  For instance, the sequence of `nucleobase` "ATCGA" will produce a [U;A;G;C;U] rna. The function must be typed as: `val generate_rna: helix -> rna`.

Chapter XI

Exercise 07: DNA -> Ribosome

	Exercise 07
Exercise 07: DNA -> Ribosome	
Turn-in directory : *ex*07/	
Files to turn in : `ribosome.ml`	
Allowed functions : `Pervasives module`	
Remarks : `n/a`	

The ribosome is a large and complex molecular machine found within all living cell. It's main purpose is to create proteins from a Messenger RNA by combining amino acids together. Proteins are essential to all living organisms, Humans included.

- Write a function *generate_bases_triplets* that creates a list of triplets of elements of type *nucleobase* from an element of type *rna* according to the following rule: if the number of nucleobases of the list is not a multiple of 3, it ignores the last incomplete triplet. The function must be typed as : `generate_bases_triplets : rna -> (nucleobase * nucleobase * nucleobase) list`.

- Write a `protein` type that consists of a list of `aminoacid`, and of function `string_of_protein` of type `protein -> string`.

- Write a function *decode_arn* of type `rna -> protein` that creates a list of the variant type *aminoacid* from an element of type *rna* according to the following rules:

    - The decode process begins with the first triplet and ends with the first Stop triplet encountered. Obvious isn't it?

    - Here is the matching table of the nucleobases triplet, the corresponding amino acid and the constructor of type *aminoacid* :

        * UAA, UAG, UGA : End of translation -> `Stop`

        * GCA, GCC, GCG, GCU : Alanine -> `Ala`

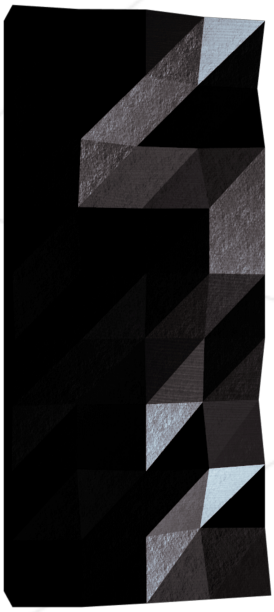
* AGA, AGG, CGA, CGC, CGG, CGU : Arginine -> `Arg`
* AAC, AAU : Asparagine -> `Asn`
* GAC, GAU : Aspartique -> `Asp`
* UGC, UGU : Cysteine -> `Cys`
* CAA, CAG : Glutamine -> `Gln`
* GAA, GAG : Glutamique -> `Glu`
* GGA, GGC, GGG, GGU : Glycine -> `Gly`
* CAC, CAU : Histidine -> `His`
* AUA, AUC, AUU : Isoleucine -> `Ile`
* CUA, CUC, CUG, CUU, UUA, UUG : Leucine -> `Leu`
* AAA, AAG : Lysine -> `Lys`
* AUG : Methionine -> `Met`
* UUC, UUU : Phenylalanine -> `Phe`
* CCC, CCA, CCG, CCU : Proline -> `Pro`
* UCA, UCC, UCG, UCU, AGU, AGC : Serine -> `Ser`
* ACA, ACC, ACG, ACU : Threonine -> `Thr`
* UGG : Tryptophane -> `Trp`
* UAC, UAU : Tyrosine -> `Tyr`
* GUA, GUC, GUG, GUU : Valine -> `Val`

Chapter XII

Exercise 08: DNA -> The Complete Process of Protein Creation

	Exercise 08
Exercise 08: DNA -> The Complete Process of Protein Creation	
Turn-in directory : *ex*08/	
Files to turn in : `life.ml`	
Allowed functions : `Pervasives and String module`	
Remarks : `n/a`	

- Write a function that goes from the generation of an helix of a reasonable length to the creation of the corresponding `protein`. Each step must be displayed clearly on the standard output. this function takes a string as a parameter.



OCaml Pool - d03

Development Environment

42 pedago pedago@staff.42.fr
kashim vbazenne@student.42.fr

Abstract: This is the subject for d03 of the OCaml piscine. The main theme of this day is to introduce the basic development environment of OCaml and the many ways to build a project. It also introduce the notion of tree in OCaml, because trees are good for you.

Contents

Chapter I

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 additionnal 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 exercice right.
- 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. Anyway, the interpreter is a powerfull ally, learn to use it at its best as soon as possible !
- The subject can be modified up to 4h 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 she or he can't grade. As usual, big fonctions is a weak style.

- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are afforded a certain amount of freedom in how you choose to do the exercises. Anyway, 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 II

Day-specific rules

- For each exercise of the day, you must provide sufficient material for testing during the defence session. **Every functionnality that can't be tested won't be graded!**

- During the whole day, we will be using a type tree defined as follows:

```
type 'a tree = Nil | Node of 'a * 'a tree * 'a tree
```

  A basic node can be created like this: `Node (42, Nil, Nil)`.

- During the whole day, you are allowed to use any method you prefer to handle the Makefile. (Makefile, OCamlMakefile, ocamlbuild, ...)

Chapter III

Foreword

Level 42 are an english pop-rock and jazz-funk band well known in the 80's and 90's. The band became famous for his bass player and singer Mark King whose percussive slap technique was known as one of the most impressive bass technique of all time. The most successfull single are:

- Lessons in Love
- Love Games
- The Chinese Way
- The Sun Goes Down (Living It Up)

The name of the band is taken from the book The Hitchhiker's Guide to the Galaxy by Douglas Adams. You can view the band in action by clicking here.

Here is a subjectiv list of non-exhaustiv very good bass players :

- Paul McCartney (The Beatles)
- John Entwistle (The Who)
- John Paul Jones (Led Zeppelin)
- Jaco Pastorius (Weather Report)
- John Deacon (Queen)
- Prince
- Stuart Hamm (Joe Satriani)
- No, not Flea from the Red Hot Chili Peppers!
- And many others...

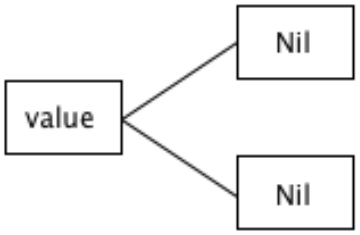
Chapter IV

Exercise 00: Chucalescu - Stupalacci

	Exercise 00
Exercise 00: Chucalescu - Stupalacci	
Turn-in directory : $ex00/$	
Files to turn in : `ft_graphics.ml, Makefile`	
Allowed functions : `open_graph, lineto, moveto, draw_string functions of the Graphics module and Pervasives module`	
Remarks : `n/a`	

- Write a function `draw_square` that takes three ints x, y, size as parameters and draw a square centered in (x, y) on a graphical window. You'll have to use the `Graphics` module, of which the only allowed functions are `lineto`, `moveto` amd the `draw_string` functions.

- Write a function `draw_tree_node` that takes a basic tree node Node(v, Nil, Nil) or Nil as a parameter and draw it according to this example (if you prefer to draw it from up to down it's ok):also, you don't have to handle if the printed value goes out of the square



Chapter V

Exercise 01: Basic Gardening

	Exercise 01
Exercise 01: Basic Gardening	
Turn-in directory : $ex01/$	
Files to turn in : `gardening.ml, Makefile`	
Allowed functions : `Graphics module and Pervasives module`	
Remarks : `n/a`	

The root of a binary tree is the top node. A leaf of a binary tree is a node with no sub-tree. The size of a binary tree is the number of nodes that are defined. The height of a binary tree is the number of relations on the longest downward path between the root and a leaf.

- Write a function `size` that takes a tree as a parameter and returns its size. The function must be typed as :

```
val size : 'a tree -> int
```

- Write a function `height` that takes a tree as a parameter and returns its height. The function must be typed as :

```
val height : 'a tree -> int
```

- Write a function `draw_tree` that takes a tree as a parameter and draws it using the graphics module. We assume that the size of the window is sufficient to draw it, so you don't have to handle it. We'll consider only trees of type `string tree` for this function, so you can draw the values without conversion problems. The function must be typed as :

```
val draw_tree : string tree -> unit
```

Chapter VI

Exercise 02: All Good Ciphers Go To Heaven

	Exercise 02
Exercise 02: All Good Ciphers Go To Heaven	
Turn-in directory : *ex*02/	
Files to turn in : `cipher.ml, uncipher.ml, Makefile`	
Allowed functions : `Pervasives module and the String.map function`	
Remarks : `n/a`	

This Exercise is what people usually called fun, a kind of distraction between two trees. The main purpose of this exercise is to introduce to you the concept of very very very basic ciphering. Of course, needless to say that the `Cryptokit` module of OCaml is strictly forbidden for the entire excercise.

- Write a function and it's opposite `rot42` and `unrot42` that takes a string as parameter and returns the string by 42-rotating all of its char.

- Write a function and it's opposite `caesar` and `uncaesar` that takes a string and an int as parameters and returns the string by rotating all of its char to the right according to the int (just like a rotn actually).

- Write a function `xor` that takes a string and an int `key` as parameters and returns the string by xor-ing all of its char with the `key`. This function is it's own opposite.

- Write a function and it's opposite `ft_crypt` and `ft_uncrypt` that takes a string and a list of functions caesar-like or xor-like as parameters and returns the new string after the application of the functions. Of course, wrap everything in a program to prove that your work works.

```
val ft_crypt : string -> (string -> string) list-> string
val ft_uncrypt : string -> (string -> string) list-> string
```

- Every cipher functions must go in the file `"cipher.ml"` and every uncipher functions must go in the file `"uncipher.ml"`.

Chapter VII

Exercise 03: Gardening Research

	Exercise 03
Exercise 03: Gardening Research	
Turn-in directory : $ex03/$	
Files to turn in : `btree.ml`	
Allowed functions : `Pervasives module and List.hd`	
Remarks : `n/a`	

Ok, enough fun, back to the trees. A binary search tree (BST) is a binary tree where the nodes are sorted in a way that for every node, all the nodes of the left subtree of the root are inferior to the current node and all the nodes of the right subtree of the root are superior to the current node. A perfect btree is a tree where all the leaves have 0 or 2 Sons and all the leaves are at equal distance of the root.

- Write a function `is_bst` that takes a tree as a parameter and returns a boolean. True if the tree is an bst, false otherwise.
- Write a function `is_perfect` that takes a tree as a parameter and returns a boolean. True if the tree is perfect BST, false otherwise.
- Write a function `is_balanced` that takes a tree as a parameter and returns a boolean. True if the tree is a height-balanced BST, false otherwise.
- Write a function `search_bst` that takes a value and a BST as a parameter and returns a boolean. True if the the value is in the tree, false otherwise.
- Write a function `add_bst` that takes a value and a BST as a parameter and returns BST with the new value inserted.
- Write a function `delete_bst` that takes a BST as a parameter and a value and returns a BST with the value deleted.

You may encounter a case in which one of your functions cannot return a value because it wouldn't make sens. In that case, instead of returning a value, you must use the following expression : failwith "your error message here", with your custom error message obvioulsy.

Chapter VIII

Exercise 04: Adelson-Velsky-Landis

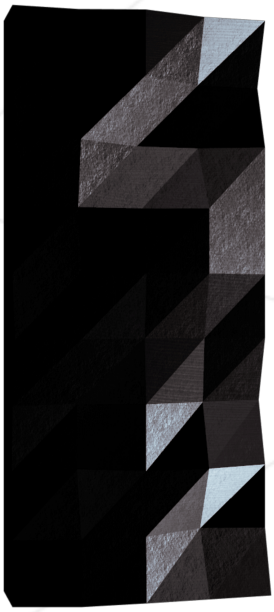
	Exercise 04
Exercise 04: Adelson-Velsky-Landis	
Turn-in directory : *ex*04/	
Files to turn in : `avl.ml`	
Allowed functions : `Pervasives module`	
Remarks : `n/a`	

An AVL(**Georgy Adelson-Velsky and Landis**) tree is a Self-Balancing Binary Search Tree (BST). In an AVL tree, the heights of the two child subtrees of any node differ by at most one. If at any time they differ by more than one, rebalancing is done to restore this property.

- Write a function `insert_avl` that takes a value and an AVL tree as parameters. The function insert the value in the tree, check if the tree is still balanced and rebalance it if necessary by performing one or more rotation. The function returns the newly created AVL.

- Write a function `delete_avl` that takes a value and an AVL tree as parameters. The function delete the value in the tree, check if the tree is still balanced and rebalance it if necessary by performing one or more rotation. The function returns the newly created AVL.

`You may encounter a case in which one of your functions cannot return a value because it wouldn't make sens. In that case, instead of returning a value, you must use the following expression : failwith "your error message here", with your custom error message obvioulsy.`



Ocaml piscine - D04

OCaml's modules language

Staff 42 bocal@staff.42.fr

Abstract: *This document is the subject for day 04 of 42's Ocaml piscine.*

Contents

Chapter I

Foreword

About dildos, according to Wikipedia:

A dildo is a device designed for vaginal or anal penetration, usually solid and phallic in shape. Some expand this definition to include vibrators. Others exclude penis prosthetic aids, which are known as "extensions". Some include penis-shaped items clearly designed with vaginal penetration in mind even if they are not true approximations of a penis. Some people include devices designed for anal penetration (butt plugs) while others do not. These devices are often used by people of all genders and sexual orientations, for masturbation or for other sexual activity.

The etymology of the word dildo is unclear. The Oxford English Dictionary (OED) describes the word as being of "origin unknown". One theory is that it originally referred to the phallus-shaped peg used to lock an oar in position on a dory (small boat). It would be inserted into a hole on the side of the boat, and is very similar in shape to the modern toy. It is possible that the sex toy takes its name from this sailing tool, which also lends its name to the town of Dildo and the nearby Dildo Island in Newfoundland, Canada. Others suggest the word is a corruption of Italian diletto "delight". It has also been noted that the word dildo has similarity to "dill", a pickled cucumber, which is a vegetable that has been used as a natural dildo.

According to the OED, the word's first appearance in English was in Thomas Nashe's The Choice of Valentines or the Merie Ballad of Nash his Dildo (c. 1593). The word also appears in Ben Jonson's 1610 play, The Alchemist. William Shakespeare used the term once in The Winter's Tale, believed to be from 1610 or 1611, but not printed until the First Folio of 1623.

The phrase "Dil Doul", referring to a man's penis, appears in the 17th century folk ballad "The Maids Complaint for want of a Dil Doul". The song was among the many in the library of Samuel Pepys.

In some modern languages, the names for dildo can be more descriptive, creative or subtle—note, for instance, the Spanish consolador "consoler" and Welsh cala goeg "fake penis".

Figure I.1: A pair of dildos

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 additionnal 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 exercice right.
- 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. Anyway, the interpreter is a powerfull ally, learn to use it at its best as soon as possible !
- The subject can be modified up to 4h 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 she or he can't grade. As usual, big fonctions is a weak style.

- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are afforded a certain amount of freedom in how you choose to do the exercises. Anyway, 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: Cards colors

	Exercise 00
Exercise 00: Cards colors	
Turn-in directory : $ex00/$	
Files to turn in : `Color.ml and main.ml`	
Allowed functions : `Nothing`	
Remarks : `n/a`	

Regular play cards fit nicely as a programming topic when dealing with modules and nested modules. Colors, values, cards and decks, all tied together in a smart design.

As a start, we need to represent cards colors, namely `spade`, `heart`, `diamond` and `club`, as an `OCaml` type and instrument that type with relevant values and functions.

Write the file `Color.ml` that respects the following interface:

```
type t = Spade | Heart | Diamond | Club

val all : t list                        (** The list of all values of type t *)

val toString        : t -> string                   (** "S", "H", "D" or "C"  *)
val toStringVerbose : t -> string                   (** "Spade", "Heart", etc *)
```

Provide some tests in the file `main.ml` to prove that your `Color` module works as intended.

Chapter IV

Exercise 01: Cards values

	Exercise 01
Exercise 01: Cards values	
Turn-in directory : $ex01/$	
Files to turn in : `Value.ml and main.ml`	
Allowed functions : `invalid_arg`	
Remarks : `n/a`	

We have colors, now we need values for our cards. Cards values form a total ordered set, we need a type to represent them, and values and functions to instrument that type. The card values of a regular 52 cards deck are 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king and as.

Write the file `Value.ml` that respects the following interface:

```
type t = T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | Jack | Queen | King | As

(** The list of all values of type t                                    *)
val all : t list

(** Interger representation of a card value, from 1 for T2 to 13 for As  *)
val toInt           : t -> int

(** returns "2", ..., "10", "J", "Q", "K" or "A"                        *)
val toString        : t -> string
(** returns "2", ..., "10", "Jack", "Queen", "King" or "As"             *)
val toStringVerbose : t -> string

(** Returns the next value, or calls invalid_arg if argument is As      *)
val next      : t -> t
(** Returns the previous value, or calls invalid_arg if argument is T2  *)
val previous  : t -> t
```

Provide some tests in the file `main.ml` to prove that your `Value` module works as intended.

Chapter V

Exercise 02: Cards

	Exercise 02
Exercise 02: Cards	
Turn-in directory : $ex02/$	
Files to turn in : `Card.ml and main.ml`	
Allowed functions : `invalid_arg, Printf.sprintf and the List module`	
Remarks : `n/a`	

We have colors and values, now we can have cards ! Write the file `Card.ml` that respects the interface below. Several things to note regarding this interface:

- The `Card` module embeds the `Color` and `Value` modules. Just copy your previous code in the corresponding structures.
- The type `Card.t` is abstract. That means you're free to implement it as you want. Choose wisely, some solutions are better than otters. And otters are cute.
- All values' and functions' types and identifiers are self explainatory. Just read and use your brain, no tricks here.
- The function `toString : t -> string` returns strings like: `"2S"`, `"10H"`, `"KD"`, ...
- The function `toStringVerbose : t -> string` returns strings like: `"Card(7, Diamond)"`, `"Card(Jack, Club)"`, `"Card(As, Spade)"`, ...
- The function `compare : t -> t -> int` behaves like the Pervasives `compare` function.
- The functions `max` and `min` return the first parameter if the two cards are equal.
- The function `best : t list -> t` calls `invalid_arg` if the list is empty. If two or more cards are equal in value, return the first one. True coders use `List.fold_left` to do this function.

Provide some tests in the file `main.ml` to prove that your `Card`, `Card.Color` and `Card.Value` modules work as intended.

```
module Color :
sig

    type t = Spade | Heart | Diamond | Club

    val all : t list

    val toString        : t -> string
    val toStringVerbose : t -> string

end


module Value :
sig

    type t = T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | Jack | Queen | King | As

    val all : t list

    val toInt           : t -> int
    val toString        : t -> string
    val toStringVerbose : t -> string

    val next      : t -> t
    val previous  : t -> t

end


type t

val newCard : Value.t -> Color.t -> t

val allSpades   : t list
val allHearts   : t list
val allDiamonds : t list
val allClubs    : t list
val all         : t list

val getValue : t -> Value.t
val getColor : t -> Color.t

val toString        : t -> string
val toStringVerbose : t -> string

val compare : t -> t -> int
val max     : t -> t -> t
val min     : t -> t -> t
val best    : t list -> t

val isOf      : t -> Color.t -> bool
val isSpade   : t -> bool
val isHeart   : t -> bool
val isDiamond : t -> bool
val isClub    : t -> bool
```

Chapter VI

Exercise 03: Deck

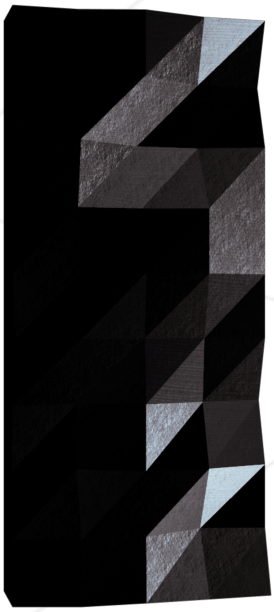
	Exercise 03
Exercise 03: Deck	
Turn-in directory : $ex03/$	
Files to turn in : `Deck.mli, Deck.ml and main.ml`	
Allowed functions : `Allowed functions and modules from the previous exercices, plus raise and the Random module`	
Remarks : `n/a`	

We have cards, it's time to organize them in a deck represented by the `Deck` module. First write the interface for that module in the file `Deck.mli` according to the following statements:

- The `Deck` module embeds the `Card` module from the previous exercice.
- The `Deck` module exposes an **abstract** type `t` that represents a deck. Its definition is up to you.
- The `Deck` module exposes a function `newDeck` that takes no argument and returns a deck of the 52 cards (i.e. the type `t`) in **random** order. This means that upon two different calls to the function `newDeck`, the order of the deck will be different.
- The `Deck` module exposes a function `toStringList` that takes a deck as a parameter and returns a list of the string representations of each card.
- The `Deck` module exposes a function `toStringListVerbose` that takes a deck as a parameter and returns a list of the verbose string representations of each card.
- The `Deck` module exposes a function `drawCard` that takes a deck as a parameter and returns a couple composed of the first card of the deck and the rest of the deck. If the deck is empty, raise the exception `Failure` with a relevant error message.

Now implement the `Deck` module in the file `Deck.ml` according to its interface.

Provide some tests in the file `main.ml` to prove that your `Deck`, `Deck.Card`, `Deck.Card.Color` and `Deck.Card.Value` modules work as intended.



Piscine OCaml - d05

Imperative features

42 staff pedago@staff.42.fr
nate alafouas@student.42.fr

Abstract: *This is the subject for d05 of the OCaml piscine.*

Contents

Chapter I

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 II

Day-specific rules

- For exercises 00, 01, 02 , 03 and 04, the `rec` keyword is forbidden and considered cheating. No questions asked. Yes, yes, I know. But still.
- For every exercise of today's subject, you are allowed to use every function in the Pervasives module. Each exercise's header specifies what you are allowed to use **aside** from these functions.
- For today et **only** for today, the keywords `for` and `while` are allowed.
- Any warning at compilation or uncaught exception means your exercise is non-functional and you will therefore earn no points for it.
- Some exercises require you to write a full program, others just specify a function. If you are just asked to write a function, you still need to turn in a full program, including a `let ()` definition with examples to show that your function is working properly. You can use any functions you want or see fit to use in your `let ()` definition, as long as you don't have to link an external library.
- If you are required to provide your own examples, remember that your tests must be comprehensive. A non-tested feature is a non-functional feature.

Chapter III

Foreword

Here are some words of wisdom for you to meditate.

```
 Fear is freedom!
Subjugation is liberation!
Contradiction is truth!
Those are the facts of this world!
And you will all surrender to them, you pigs in human clothing!
```

Letting aside being a pig, today you will be doing some pretty nasty stuff. Make sure you wash your hands after you push your exercises.

If you know where this quote is from, that's nice. But you're still a pig in human clothing.

Chapter IV

Acknowledgements

Exercises 05, 06, 07 and 08 on Machine Learning use the Ionosphere Data Set from UC Irvine's Machine Learning Repository, and I would like to thank them for making these data public. Click here to access the repository.

Chapter V

Exercise 00: Micro-nap

	Exercise 00
A tribute to polyph...poly...you know, that polysleep thingie.	
Turn-in directory : *ex*00/	
Files to turn in : `micronap.ml`	
Allowed functions : `The Sys and Array modules`	
Remarks : `n/a`	

You will write a program which takes an integer command line argument. This argument will be the number of seconds your program will wait before exiting. Invalid or missing argument quits the program immediatly, no specific output is expected.

You must use the following function `my_sleep` to do the actual wait:

```
let my_sleep () = Unix.sleep 1
```

You will turn in this function along with your work. Feel free to sleep while your program is running, if you need to.

`You might have to do something "special" to compile this exercise.`

Chapter VI

Exercise 01: ft_ref

	Exercise 01
A tribute to something you should normally never use.	
Turn-in directory : $ex01/$	
Files to turn in : `ft_ref.ml`	
Forbidden functions : `ref`	
Remarks : `n/a`	

Crate a type `ft_ref` to reproduce the `ref` type, and implement the following functions:

- `return: 'a -> 'a ft_ref`: creates a new reference.
- `get: 'a ft_ref -> 'a`: Dereferences a reference.
- `set: 'a ft_ref -> 'a -> unit`: Assign a reference's value.
- `bind: 'a ft_ref -> ('a -> 'b ft_ref) -> 'b ft_ref`: This one is a bit more complicated. It applies a function to a reference to transform it. You can see it as a more complicated `set` function.

The use of the standard type `ref` is obviously forbidden, but playing with it in the interpreter should tell you how it is implemented internally. Your goal is to do the same thing. Oh, and by the way, after this exercise, you will have done your first monad. Monads are some kind of ancient black magic you'll get to play with soon enough. See you on d09...

Chapter VII

Exercise 02: Bad jokes

	Exercise 02
What's red and goes up and down?	
Turn-in directory : *ex*02/	
Files to turn in : `jokes.ml`	
Forbidden functions : `None`	
Remarks : `n/a`	

You will write a program to print a joke on the standard output, followed by an end-line character.

Your jokes can be whatever you want, but you get bonus points if they are bad. The only restriction is, you will store them in an array and there must be at least **five (5)** of them. Your program will randomly pick a joke from this array and print it to the standard output.

`A joke is bad if your grader wants to slap you after reading it.`

Chapter VIII

Exercise 03: Bad jokes, improved.

	Exercise 03
A tomato in an elevator.	
Turn-in directory : $ex03/$	
Files to turn in : `jokes.ml, and any jokes file you want.`	
Forbidden functions : `None`	
Remarks : `n/a`	

In this exercise you will take the `jokes.ml` file from the previous exercise and make some improvements to it.

Your program will now load its jokes from a file. You can organise your file however you like, as long as anyone can modify it and add/remove jokes from it if given short instructions; if I need a 2-hour course on how to edit your file, there's obviously a problem.

The name of the jokes file will be provided to your program through a command line argument. You are not required to handle non-compliant/weird files, as your program will only be tested with your file, or any file that fully complies to your formatting rules. **IMPORTANT**: your jokes **MUST** be different from the previous exercice in order to earn bonus points!

What does not change, is that your jokes still have to be stored in an array, they still have to be bad, and your program still has to pick one randomly from that array.

Chapter IX

Exercise 04: Sum

	Exercise 04
Seriously, just a sum. There's no catch.	
Turn-in directory : *ex*04/	
Files to turn in : `sum.ml`	
Allowed functions : `None`	
Remarks : `n/a`	

You will write a function named `sum` which takes two floating-point numbers and adds one to the other. Yes. That's it.

Your function's type will be `float -> float -> float`.

`Don't forget to turn in a full program with examples to show your work is functional. This is harder than it seems.`

Chapter X

Exercise 05: You are (not) alone.

	Exercise 05
You didn't actually think I was that kind, did you?	
Turn-in directory : $ex05/$	
Files to turn in : `eu_dist.ml`	
Allowed functions : `The Array module`	
Remarks : `n/a`	

I mean come on, you know I'm much meaner than that. Now let's do some funny stuff. In the next series of exercises we'll try to do some machine learning. If you don't know what machine learning is, look it up on Wikipedia or ask your hipster entrepreneur NodeJS friend.

But first, we need to do the basic things. You will write a function named `eu_dist` which takes two points and calculates the Euclidian distance between them. If you don't know what the Euclidian distance is, here it is: if we consider a a point as an array of coordinates $a_1, a_2, a_3...a_n$ and b another point as an array of coordinates $b_1, b_2, b_3...b_n$, the Euclidian distance between a and b is:

$$eu\_dist(a, b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

Our model for a point will be an `array` of floating-point numbers, with each cell containing the coordinate in a given dimension.

Your function's domain will be: $eu\_dist : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}^+, D \in \mathbb{N}^*$.

Your function's type will be: `float array -> float array -> float`. You don't have to handle cases with two vectors having different lengths.

Okay, now you should start to understand that machine learning is not just a buzz word. It's mostly math. And it's just the beginning. Still with me ?

Chapter XI

Exercise 06: You can (not) advance.

	Exercise 06
This is boring. But you have to do it anyway.	
Turn-in directory : $ex06/$	
Files to turn in : `examples_of_file.ml, *.csv`	
Forbidden functions : `None`	
Remarks : `n/a`	

I said we would be doing some machine learning today; it is now time to give you some more details about what we will be doing. What we will implement is an algorithm for *supervised classification*. If you don't know what "supervised" and "classification" means, look it up on the Internet because your hipster entrepreneur NodeJS friend probably doesn't know either.

In this exercise you will write a function named `examples_of_file` which takes a path to a file as argument, and returns a set of examples read from the file input, which is formatted as csv.

Each line in the input describes a radar used to detect free electrons in the ionosphere. A radar is described by a set of `float` fields, which are some complicated stats I don't understand, and a letter at the end of the line to specify if the radar could detect evidence of free electrons in the ionosphere or not. Anyway, you just have to remember that a radar is defined by a bunch of complicated stats which form a vector, and a class under the form of a character. If you don't know what a vector is, just ask your 15-year old sibling. Chances are he or she knows.

In other words, the type of an example will be `float array * string`, and your function's type will be `string -> (float array * string) list`.

For instance, `1.0,0.5,0.3,g` will be converted to `([|1.0; 0.5 ;0.3 |], "g")`.

You can use the files named ionosphere.test.csv and/or ionosphere.train.csv for your tests, or you can use any file you want, as long as it has the same format (but it doesn't have to have the same number of float columns).

Chapter XII

Exercise 07: You can (not) redo.

	Exercise 07
This is very interesting! But you have to do it anyway.	
Turn-in directory : $ex07/$	
Files to turn in : `one_nn.ml`	
Forbidden functions : `None`	
Remarks : `n/a`	

Here we go! Now that we have our examples, we can do some prediction. This exercise is where you're going to implement the `K-nearest` neighbours algorithm — or `K-nn` for short.

Do you remember our radars? Our radars can be either good or bad. Our objective will be to use the stats describing the radars: let's say you're trying to guess the type of a radar `A`. You know that `A` is bad. You know a radar named `B` that looks a lot like `A`. That means `B` is probably bad, right? That's the spirit of the K-nn algorithm. You pick the K nearest radars to the one you're trying to guess, and you can say `good` or `bad` depending on whether there's more good radars or bad radars.

And how do you know two radars are close to each other? DUH! You know how to compute an Euclidian distance, right? ... Right?

But right now, that sounds like a lot to do. It's complicated. And I know you're tired. So we're going to implement that with just **ONE** nearest neighbour. 1-nn. Your function will do just that: guess if the radar you give to it is good or bad, using the type of the nearest radar.

That means your function's type will be `radar list -> radar -> string`, with `type radar = float array * string`. But I bet you already figured that for yourself. You are not required to handle the case when the radars have different vector lengths, or when the train set is empty.

As usual, don't forget your tests. It could be interesting to show that your one-nn can guess correctly, but also make mistakes! Feel free to use your examples_of_file function to provide examples, or write in-memory examples if you couldn't solve the previous exercise.

Your one-nn has to be able to handle any class (not just g or b) and any vector length, as long as it's always the same for all radars.

Chapter XIII

Exercise 08: This is (not) the end.

	Exercise 08
Let's put everything together!	
Turn-in directory : $ex08/$	
Files to turn in : `k_nn.ml`	
Forbidden functions : `None`	
Remarks : `n/a`	

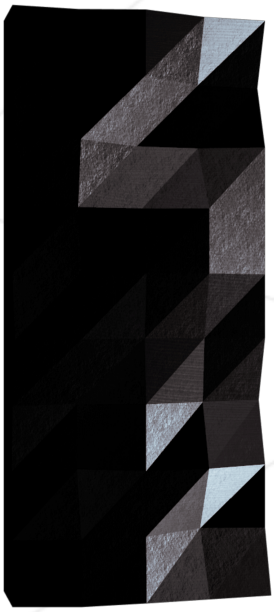
Up for another challenge? Cool! :) So now we're going to really implement a `K-nn` algorithm, using what you already know, especially your `one_nn`.

The K-nn algorithm works like your `one_nn`, except you're going to pick the K nearest radars and return the class that's more represented in those K radars. In other words, you're making a generalization of `one_nn` to implement your K-nn.

By the way, K is called a `hyperparameter`, but that's just vocabulary. It'll be another argument for your function, which means its type will be `radar list -> int -> radar -> string`.

In case your K hyperparameter is even and you have a tie, do something smart. You won't be deducted if you randomly pick either choice, but really that's a shame.

`As usual, don't forget your tests. Bonus points if your tests can run a full test set and measure your K-nn classifier's accuracy.`



Ocaml piscine - D06

Functors

Staff 42 bocal@staff.42.fr

Abstract: This document is the subject for day 06 of 42's OCaml piscine.

Contents

Chapter I

Foreword

Dane Cross, born John David Schardt, was born in San Francisco, California october 3rd 1983. Half Greek and half German, former film student, Dane Cross worked as a news cameraman for a little while.

Dane and his girlfriend at the time answered an on-line ad for adult film industry and started out performing in the alt-porn niche at age 24 in 2007. Soon he began working for such major hardcore companies as Vivid, Wicked, Adam & Eve, Penthouse, and New Sensations.

His X-rated movie pseudonym is a cross between Dane Cook and David Cross, two american stand-up comedian he enjoyed a lot. Dane is perhaps best known as Bud in the popular porno parody series "Not Married With Children XXX".

Dane Cross won several notable adult cinema awards in 2010: The AVN Award for Best New Male Newcomer, the XRCO Award for Best New Stud, and the XBIZ Award for New Male Performer of the Year. He is also famous for his engagement to porn star and frequent co-star Faye Reagan, but as of late 2010 they are no longer together due to Faye's drug addiction. Dane admited he still loved Faye tremendously in a Reddit AMA in 2013.

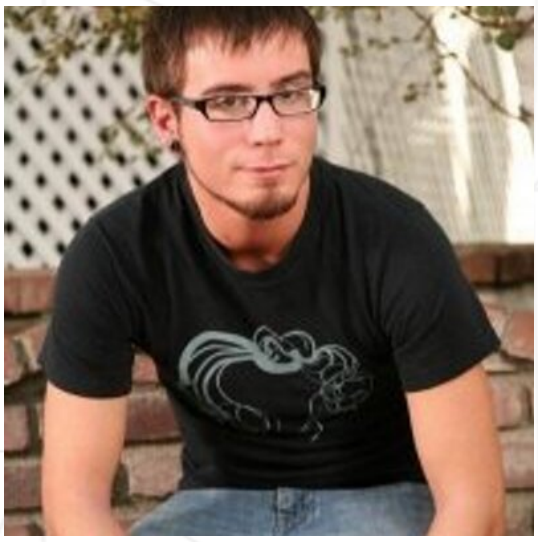


Figure I.1: Dane Cross

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 additionnal 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 exercice right.
- 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. Anyway, the interpreter is a powerfull ally, learn to use it at its best as soon as possible !
- The subject can be modified up to 4h 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 she or he can't grade. As usual, big fonctions is a weak style.

- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are afforded a certain amount of freedom in how you choose to do the exercises. Anyway, 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`	
Remarks : `n/a`	

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`	
Remarks : `n/a`	

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 : `Pervasives.fst and Pervasives.snd`	
Remarks : `n/a`	

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 Pervasives module`	
Remarks : `n/a`	

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 Pervasives module`	
Remarks : `n/a`	

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.

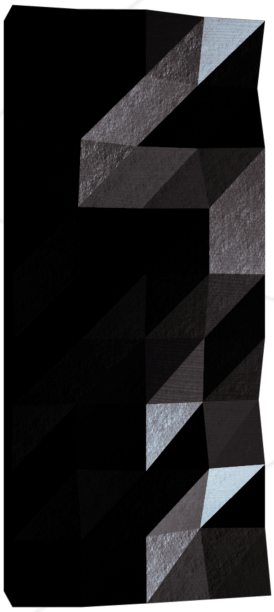
```
module IntVal : VAL =
struct
  type t = int
  let add = ( + )
  let mul = ( * )
end

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 !!!



OCaml Pool - d07

Object Oriented Programming 1

42 pedago pedago@staff.42.fr
kashim vbazenne@student.42.fr

Abstract: This is the subject for d07 of the OCaml piscine. The main theme of this day is to introduce the object oriented programming style with OCaml. This first day will focus on basic interaction with the 'O' from 'O'Caml.

Contents

Chapter I

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 II

Day-specific rules

- This day is pure fun. Bonus points will be allowed if you provide clever references to the Doctor Who universe.
- You are in a functional programming piscine, so your coding style MUST be functionnal (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 functionnality that can't be tested won't be graded!**
- Don't be lazy. If something is not tested in an exercise, the defence will stop immediately.

Chapter III

Foreword : The ultimate mega gangsta chocolate butter cake

III.1 Ingredients :

- 250g of chocolate
- 250g of butter (Yeah... I know)
- 150g of sugar
- 4 eggs
- 1 big spoon of flour

III.2 Recipe :

- Melt the chocolate with the butter.
- Add sugar.
- Add eggs one by one by mixing between each one.
- Add flour progressively while mixing the whole thing
- Pour into a cake plate (or something similar)
- Put the cake plate into a larger plate with a little water in it (bain marie style)
- Put the two plates like this in the oven for 45 minutes at 180 degrees
- Put it 12 hours (Not a joke) into your refrigerator.

Done!

Chapter IV

Exercise 00: Do What I do. Hold tight and pretend it's a plan!

	Exercise 00
Exercise 00: Do What I do. Hold tight and pretend it's a plan!	
Turn-in directory : *ex*00/	
Files to turn in : `people.ml, main.ml, Makefile`	
Allowed functions : `Pervasives modules`	
Remarks : `n/a`	

- Write a class *people* that has the following attributes :
  - A name attribute of type string.
  - An hp attribute of type int initialized to 100.
  - A to_string method that returns the name of the object with attributes values.
  - A talk method that print the following string on the standard output : *I'm [NAME]! Do you know the Doctor?*
  - A method die which prints the following sentence on the standard output : *Aaaarghh!*
  - An initializer which indicate that the object has been created (feel free to use something explicit and wise to describe it!)
- You have to simulate all the methods in the main to provide sufficient testing for the defence.

Chapter V

Exercise 01: The Name Of The Doctor!

	Exercise 01
Exercise 01: The Name Of The Doctor!	
Turn-in directory : *ex*01/	
Files to turn in : `doctor.ml, people.ml, main.ml, Makefile`	
Allowed functions : `Pervasives modules`	
Remarks : `n/a`	

- Write a class *doctor* that has the following attributes :
  - A name attribute of type string.
  - An age attribute of type int.
  - A sidekick attribute of type people
  - An hp attribute of type int initialized to 100.
  - A to_string method that returns the name of the object with attributes values.
  - A talk method that print the following string on the standard output : *Hi! I'm the Doctor!*
  - An initializer which indicate that the object has been created (feel free to use something explicit and wise to describe it!)
  - A method travel_in_time which takes two arguments of type int : *start* and *arrival* and changes the age of the doctor logicaly (Think before coding some weird arithmetics... Please...). This method also draw a TARDIS on the standard output. (If you don't know what a TARDIS is, google it!)
  - A method use_sonic_screwdriver which prints the following sentence on the standard output : *Whiiiiwhiiiwhiii Whiiiiwhiiiwhiii Whiiiiwhiiiwhiii*

  - A private method regenerate that sets the hp of the doctor to 100 (the maximum)

- You have to simulate all the methods in the main to provide sufficient testing for the defence.

Chapter VI

Exercise 02: You are a good Daaaaaalek!

	Exercise 02
Exercise 02: You are a good Daaaaaalek!	
Turn-in directory : *ex*02/	
Files to turn in : `dalek.ml, doctor.ml, people.ml, main.ml, Makefile`	
Allowed functions : `Pervasives, String and Random modules`	
Remarks : `n/a`	

- Write a class *dalek* that has the following attributes :
  - A name attribute of type string randomly generated with the format : *DalekXXX* with XXX is a random set of chars (DalekSec for example).
  - An hp attribute of type int initialized to 100.
  - A shield attribute of type bool mutable, initialized to true and change it's value each time the exterminate method is used.
  - A to_string method that returns the name of the object with attributes values.
  - A talk method that randomly prints one of the following strings on the standard output :
    - *Explain! Explain!*
    - *Exterminate! Exterminate!*
    - *I obey!*
    - *You are the Doctor! You are the enemy of the Daleks!*
  - A method exterminate which takes an argument of type people object and kill it instantly.

  - A method die which prints the following sentence on the standard output : *Emergency Temporal Shift!*

- You have to simulate a battle between the doctor, a dalek and a human in the main to provide sufficient testing for the defence. Also feel free ot add any setter that you want.

Chapter VII

Exercise 03: The Day of The Doctor!

	Exercise 03
Exercise 03: The Day of The Doctor!	
Turn-in directory : *ex*03/	
Files to turn in : `army.ml, dalek.ml, doctor.ml, people.ml, main.ml, Makefile`	
Allowed functions : `Pervasives and List modules`	
Remarks : `n/a`	

- Write a parameterized class *army* that has the following attributes :
  - A member attribute of type 'a list which contains a list of instance of one of the 3 previous classes.
  - An add method that adds an instance ot the list (front or back).
  - A delete method that removes the head of the list member. (front or back)
- You have to simulate a construction and a destruction of an army of each type in the main to provide sufficient testing for the defence.

Chapter VIII

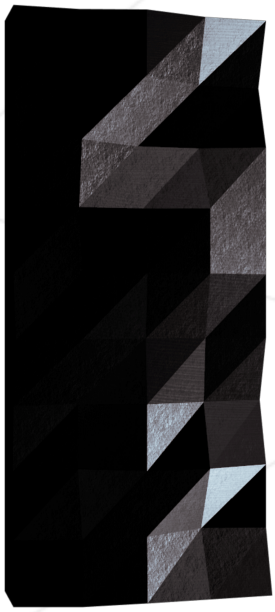
Exercise 04: The Time War!

	Exercise 04
Exercise 04: The Time War!	
Turn-in directory : *ex*04/	
Files to turn in : `galifrey.ml, army.ml, dalek.ml, doctor.ml, people.ml, main.ml, Makefile`	
Allowed functions : `Everything functional (so no while, for, Array etc...)`	
Remarks : `n/a`	

- Write a class *galifrey* that has the following attributes :
  - A member attribute of type dalek list which contains a list of instance of dalek type.
  - A member attribute of type doctor list which contains a list of instance of doctor type.
  - A member attribute of type people list which contains a list of instance of people type.
  - A do_time_war method that launch the greatest battle in time and space. You will need to add more methods to handle correct behaviour. For example

one to select one of the attack of an instance or another to check if there is any object of a list that is still alive. Feel free to add any method that seems necessary.

- You have to simulate a time war in the main to provide sufficient testing for the defence.



d08

Object-Oriented Programming 2

42 staff pedago@staff.42.fr
nate alafouas@student.42.fr

Abstract: This is the subject for d08 of the OCaml pool.

Contents

Chapter I

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 II

Day-specific rules

- For every exercise you are required to turn-in a full program, with examples to prove your classes are working correctly. You may use any function or operator you want or see fit to use in your examples — as long as you don't need to link an external library to use it.

- Each exercise in today's subject is meant to be a sequel to the previous one; as a consequence, you will likely need to use your previous exercises to solve the one you're working on.

- As an obvious consequence, any exercise with 0 points for any reason (not turned in, does not compile, crashes, etc.) means the defence will not continue any further.

- You are generally free to use any type of data structure you want. But every time a method returns an associative list, it must be sorted in ascending order by its index.

- All chemical formulae for molecules must be written using the Hill notation.

- All the classes you implement today must be functional. Any imperative class in your code means no points for the entire exercise.

- Your code will not be modified during defences, which means a non-tested feature is a non-functional feature.

- For each exercise you are required to write a Makefile, which will compile your entire exercise. You can use OCamlMakefile to write your Makefile.

Chapter III

Foreword

It was a pleasure to burn.

It was a special pleasure to see things eaten, to see things blackened and changed. With the brass nozzle in his fists, with this great python spitting its venomous kerosene upon the world, the blood pounded in his head, and his hands were the hands of some amazing conductor playing all the symphonies of blazing and burning to bring down the tatters and charcoal ruins of history. With his symbolic helmet numbered 451 on his stolid head, and his eyes all orange flame with the thought of what came next, he flicked the igniter and the house jumped up in a gorging fire that burned the evening sky red and yellow and black. He strode in a swarm of fireflies. He wanted above all, like the old joke, to shove a marshmallow on a stick in the furnace, while the flapping pigeon-winged books died on the porch and lawn of the house. While the books went up in sparkling whirls and blew away on a wind turned dark with burning.

Montag grinned the fierce grin of all men singed and driven back by flame.

He knew that when he returned to the firehouse, he might wink at himself, a minstrel man, burnt-corked, in the mirror. Later, going to sleep, he would feel the fiery smile still gripped by his face muscles, in the dark. It never went away, that smile, it never ever went away, as long as he remembered.

By `Ray Bradbury`, in `Fahrenheit 451`.

Chapter IV

Exercise 00: Atoms

	Exercise 00
Our whole universe was in a hot, dense state...	
Turn-in directory : $ex00/$	
Files to turn in : `*.ml, Makefile`	
Forbidden functions : `None`	
Remarks : `n/a`	

You will write a virtual class named `atom` for...an atom. Today we're doing some chemistry! An atom is defined by a few things:

- A `name`, of type `string`.
- A `symbol`, of type `string`.
- An `atomic_number`, of type `int`.

All these things have to be methods, and you have to be able to set what they return through the atom's constructor. Of course, feel free to look up what an atom is if you don't understand what these are.

Your class will contain at least a `to_string` method to describe shortly what your atom is, and an `equals` method to compare atoms; aside from that, you can implement anything you think is suitable or you may need in the next exercises.

To go along with your atom virtual class, you will also write some real atoms, which will of course inherit your `atom` class. You have to write at least the following classes:

- `hydrogen`
- `carbon`
- `oxygen`
- And three more atoms of your choice, as long as they really exist.

Bonus points if you write more atoms. Learning important atoms is good for you.

Chapter V

Exercise 01: Molecules

	Exercise 01
Then nearly fourteen billion years ago expansion started. Wait...	
Turn-in directory : $ex01/$	
Files to turn in : `*.ml, Makefile`	
Forbidden functions : `None`	
Remarks : `n/a`	

Now that you have your atoms, you can make some molecules! If you don't know what a molecule is, please look it up before beginning this exercise.

Your molecule class is virtual, and it needs at least the following things:

- A `name`, of type `string`.
- A `formula`, of type `string`.

All these things have to be methods. Of course, feel free to look up what a molecule is if you don't understand what these are.

One very important constraint: you have to store the molecule's atoms internally and this atom storage must be used to compute the molecule's chemical formula. Also, the chemical formula must be formatted using the `Hill notation`. As such, your constructor will only accept a name and a list of atoms.

The chemical formula is simple: it's a small string which describes what atoms and how many of them are contained in the molecule. Let's build an example with `Trinitrotoluene` (or `TNT` for short). We get the list of atoms, and we end up with:

- 3 atoms of `Nitrogen`
- 5 atoms of `Hydrogen`
- 6 atoms of `Oxygen`
- 7 atoms of `Carbon`

We know the symbols of these atoms are `N`, `H`, `O` and `C`. All we have to do is enumerate their symbols and their quantity, right? So you would get something like: $N_3H_5O_6C_7$. But **NO! WAIT!** That's not how it works. The Hill notation says we get carbon, then hydrogen, then everything else in alphabetical order. So the result is: $C_7H_5N_3O_6$. Of course, you can't really write subscripts in a terminal, so writing it behind the symbol as it is is fine, like so: `C7H5N3O6`.

Your class will contain at least a `to_string` method to describe shortly what your molecule is, and an `equals` method to compare molecules; aside from that, you can implement anything you think is suitable or you may need in the next exercises.

To go along with your molecule class, you will write some real molecules, which will of course inherit your `molecule` class. You have to write at least the following classes:

- Water (H_2O))
- Carbon dioxyde (CO_2)
- And three more molecules of your choice, as long as they really exist.

Bonus points if you write complex molecules. You don't want to be lazy, do you?

Some elements like salts, ions and other things are not considered as molecules in their strictest meaning. However, for today we'll keep it simple and assume that a molecule is simply an aggregation of atoms, as long as its overall electrical charge is neutral.

Chapter VI

Exercise 02: Alkanes

	Exercise 02
The Earth began to cool, the autotrophs began to drool...	
Turn-in directory : $ex02/$	
Files to turn in : `*.ml, Makefile`	
Forbidden functions : `None`	
Remarks : `n/a`	

Molecules are cool, but today we're focusing on a particular type of molecules, which are alkanes. Alkanes are a family of simple molecules composed of just carbon and hydrogen, which means we can create alkanes easily! The formula of an acyclic alkane is C_nH_{2n+2}. The name of the alkane simply depends on the value you give to n.

As such, your alkane's constructor only needs one parameter, which is n. It must be able to guess the name and the formula from this only n parameter.

You don't have to handle every possible alkane (because actually, you can't). Your data should be able to handle $1 \leq nleq12$; you won't be tested (and have the right to refuse to be tested) with $n > 12$.

Note that an alkane is still a molecule, which means you still have to provide the following methods:

- `name`
- `formula`
- `to_string`
- `equals`

To go along with your alkane class, you will write...some real alkanes! As usual. Of course, they will inherit your `alkane` class and you will write at least the following ones:

- `methane`
- `ethane`
- `octane`

Chapter VII

Exercise 03: Reactions

	Exercise 03
Neanderthals developed tools, we built a wall (we built the pyramids!)	
Turn-in directory : $ex03/$	
Files to turn in : `*.ml, Makefile`	
Forbidden functions : `None`	
Remarks : `n/a`	

Now we're doing interesting things! Chemical reactions. A chemical reaction is a simple thing, really. You have some molecules at the start of your reaction, and you have...some molecules at the end. There is only one really important rule to follow, which is: you must have the same exact amount of atoms at the start and at the end of your reaction. "Nothing is lost, nothing is created, everything is transformed". If you've never heard that sentence, look up Antoine de Lavoisier and what he has done for chemistry.

Basically, you can instantiate a new chemical reaction with a list of molecules. Then you can call a `get_result` method to get the result.

All that explanation was good, but now let's talk about code! You will write a virtual class named `reaction`, which will be instantiated with two collections of molecules, one for the start and one for the end of your reaction.

Your class will also provide the following virtual methods:

- `get_start: (molecule * int) list`
- `get_result: (molecule * int) list`
- `balance: reaction`
- `is_balanced: bool`

Your class doesn't have to do anything: all you have to provide is the structure. But if you think hard enough, you might think that some behaviour is common to all possible chemical reactions. Feel free to implement those you feel useful to implement. *wink wink*

Chapter VIII

Exercise 04: Alkane combustion

	Exercise 04
Math, science, history unravelling the mystery...	
Turn-in directory : $ex04/$	
Files to turn in : `*.ml, Makefile`	
Forbidden functions : `None`	
Remarks : `n/a`	

Now that we know what a chemical reaction is, let's do a real one! Do you remember the alkanes I had you write a few exercises ago? Because now we're going to burn them. Our class will be named `alkane_combustion`, and it will be a subtype of `reaction` (what did you expect?).

But first, let's clear up some theory. An alkane combustion means you start with an alkane and some molecular oxygen, and you end up with carbon dioxyde and water. Always. That means our reaction will be (for example, with methane):

$$CH_4 + O_2 \rightarrow CO_2 + H_2O$$

But wait! It's not actually that simple. I did say you had to have the same exact amount of atoms at the start and at the end, right? And if you look at what I just wrote, that doesn't really add up: you start with 1 carbon, 4 hydrogen and 2 oxygen, and you end with one carbon, 2 hydrogen and 3 oxygen; so you end up with not enough hydrogen and too much oxygen.

That's where `stoichiometrical coefficients` come in. Of course don't forget to look up the word if it sounded like Hebrew to you. But basically they are coefficients you add to the molecules in your reaction, so that you have the same amount of atoms on both sides. As a result:

$$CH_4 + 2\,O_2 \rightarrow CO_2 + 2\,H_2O$$

Now on both sides you have 1 carbon, 4 hydrogen and 4 oxygen. Good, everything works!

Now let's talk about code. You will construct your class with a list of `alkane` objects, and it will implement the `reaction` class's methods:

- `get_start`: returns the list of molecules at the beginning of the reaction. Throws an exception if the reaction is not balanced.
- `get_result`: returns the list of molecules at the end of the reaction. Throws an exception if the reaction is not balanced.
- `balance`: returns a new alkane combustion, with the right (and smallest possible) stoichiometrical coefficients so that your combustion is balanced.
- `is_balanced`: `true` if your reaction is balanced, false otherwise.

Your balance method will have to process the list of alkanes, possibly removing duplicates, and then compute all the adequate coefficients for your reaction to work.

Chapter IX

Exercise 05: Incomplete combustion

	Exercise 05
It all started with the Big Bang! (BANG!)	
Turn-in directory : *ex*05/	
Files to turn in : `*.ml, Makefile`	
Forbidden functions : `None`	
Remarks : `n/a`	

Balancing alkane combustions was the main goal of your day, and if you succeeded in doing that, congratulations! I'm very proud of you. This final exercise is for those of you who want to go a bit further (but you still have to do it if you want full points on the day. `:)`)

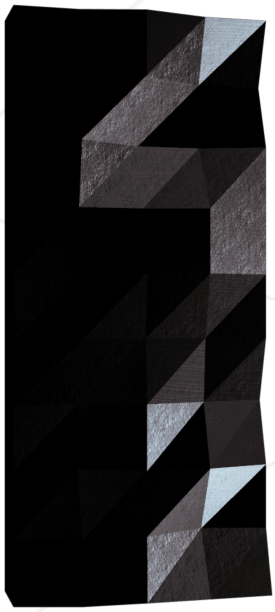
You will take your `alkane_combustion` class and add a new method to it, called `get_incomplete_results`. And by incomplete results, I mean incomplete combustion. There are cases when your alkane isn't provided enough oxygen to completely burn, which leads to carbon monoxyde or soot being created. For example, with ethane and propane:

$$C_2H_6 + 3\,O_2 \rightarrow CO_2 + CO + 3\,H_2O$$
$$C_2H_6 + 2\,O_2 \rightarrow CO + C + 3\,H_2O$$
$$C_3H_8 + 4\,O_2 \rightarrow 2\,CO_2 + C + 4\,H_2O$$
$$C_3H_8 + 3\,O_2 \rightarrow CO_2 + 2\,C + 4\,H_2O$$
$$C_3H_8 + 3\,O_2 \rightarrow CO_2 + 2\,C + 4\,H_2O$$
$$C_3H_8 + 3\,O_2 \rightarrow 2\,CO + C + 4\,H_2O$$

And so on...

Of course, these aren't the only possible solutions. There might be solutions with no carbon dioxyde at all, but obviously if there's no carbon monoxyde then it's a complete combustion and it doesn't count. Your new method will be able to change the amount of oxygen in the reaction to compute the possible outcomes, which will be returned with type `(int * (molecule * int) list) list`, using the amount of oxygen as the list's index.

Obviously, chemical reactions are much more complex than that, and usually involve the molecules' internal structure. Today we'll keep it simple and consider that a reaction is valid if and only if their stoichiometrical coefficients add up correctly. That's it. If you can work out all the possible outcome using CO2, CO, C and H2O, I'm happy with that.



OCaml Pool - d09

Monoids and Monads also known as The Day the student stood still

42 pedago pedago@42.fr
kashim vbazenne@student.42.fr
nate alafouas@student.42.fr

Abstract: This is the subject for d09 of the OCaml pool.
The main theme of this day is to introduce the Monoids and the Monads. This is hard but not as hard as it seems. Don't forget to check the videos made specially for this day since they help a lot in the whole understanding of this concept.

Contents

Chapter I

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 II

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 functionnal (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 functionnality that can't be tested won't be graded!**

- YOU MUST WATCH This video : Bryan Beckman - Don't fear the Monad

Chapter III

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 admited 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 IV

Exercise 00: All Along the Watchtower

	Exercise 00
Exercise 00: All Along the Watchtower	
Turn-in directory : $ex00/$	
Files to turn in : `*.ml, Makefile`	
Forbidden functions : `None`	
Remarks : `n/a`	

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

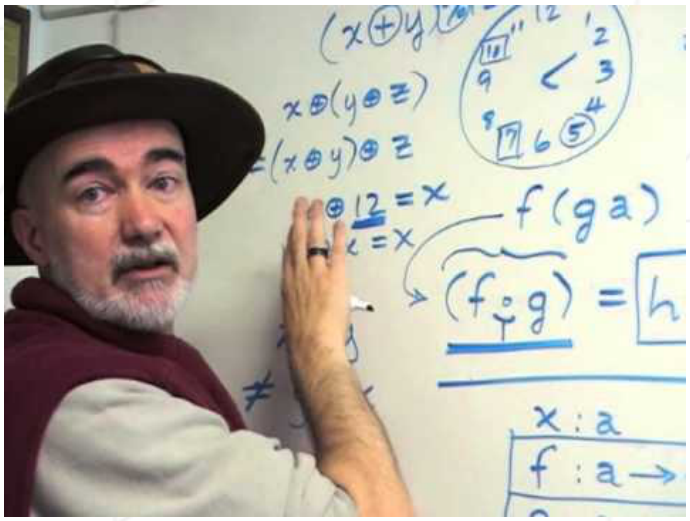
Your monoid will contain :

- `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.


x ⊕ (y ⊕ z)
(x ⊕ y) ⊕ z
⊕ 12 = x
= x
f (g a)
(f ∘ g) = h
x : a
f : a →

Chapter V

Exercise 01: The "Alan Parson's Project"

	Exercise 01
Exercise 01: The "Alan Parson's Project"	
Turn-in directory : $ex01/$	
Files to turn in : `*.ml, Makefile`	
Forbidden functions : `None`	
Remarks : `n/a`	

In this exercise you will implement a basic monoid named `App`. This monoid is an implementation of a project manager.

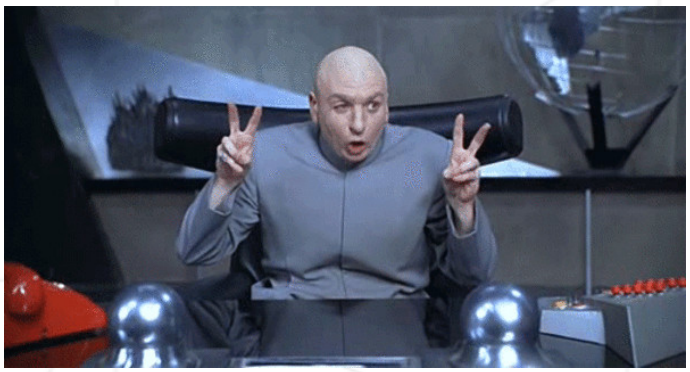
Your monoid will contain :

- `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 VI

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

	Exercise 02
Exercise 02: These aren't the functoids you're looking for	
Turn-in directory : $ex02/$	
Files to turn in : `*.ml, Makefile`	
Forbidden functions : `None`	
Remarks : `n/a`	

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 obious matching type`
- `A zero for the addition and substraction rule named zero1`
- `A zero for the multiplication and division rule named zero2`
- `An add and a sub rules to add and substract 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...


ONE DOES NOT
SIMPLY
PASS THE D09 WITHOUT
CRYING
memegenerator.net

Chapter VII

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 : $ex03/$	
Files to turn in : `*.ml, Makefile`	
Forbidden functions : `None`	
Remarks : `n/a`	

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:

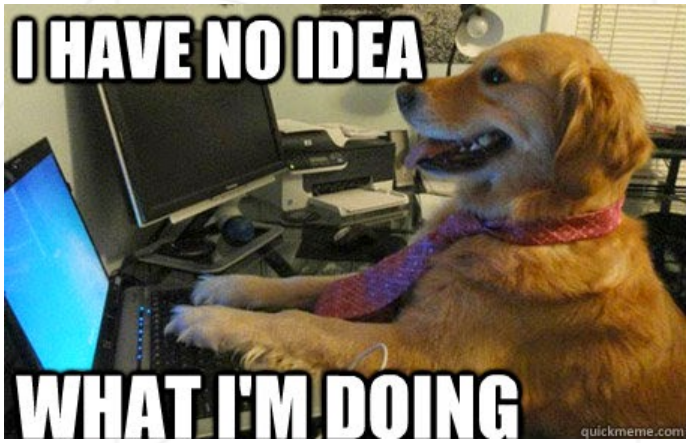
- `Success of 'a`
- `Failure of exn`

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`.

Keep in mind that Monads are a class of hard drugs.



Chapter VIII

Exercise 04: Game, Set and Match.

	Exercise 04
Somebody set up us the bomb.	
Turn-in directory : $ex04/$	
Files to turn in : `*.ml, Makefile`	
Forbidden functions : `None`	
Remarks : `n/a`	

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.

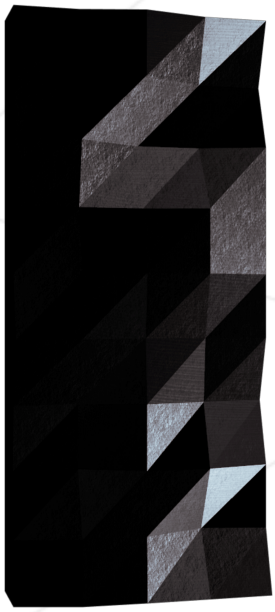
- `filter: 'a Set.t -> ('a -> bool) -> 'a Set.t`
  Returns a new set containing only the elements that satisfy the predicate given in argument.

- `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.

When you do this exercise, remember that you do it `For great Justice`.



OCaml Pool - Rush 00

Tic-tac-tic-tac-toe

42 pedago pedago@42.fr
kashim vbazenne@student.42.fr
nate alafouas@student.42.fr

Abstract: *This is the subject for rush00 of the OCaml pool.*

Contents

Chapter I

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 II

Day-specific rules

- You **MUST** provide an author file named `auteur` at the root folder of your repository, as always.

```
$> cat -e auteur
alafouas$
vbazenne$
$>
```

- This project will be entirely graded by humans. As such, you are free to have some variation between this subject's examples and your program's actual output. However, you do have to obey the spirit of the subject and turn in a program with consistent and relevant output formatting.

- For the same reasons, you are entirely free to choose your file names and general hierarchy. But keep in mind that a consistent and relevant file organization is a good code practice and will earn you bonus points.

- Any unexpected termination, stack overflow, segmentation fault, uncaught exception or unsafe behaviour means a grade of 0 — with the exception of `End_of_file`.

- You can use the modules `Pervasives`, `String` and `List` and every syntaxes and semantics covered in this first week's videos. On the other hand, you cannot use any of the following: `ref`, exceptions, arrays, mutable records, objects. If you do, you will get a grade of -42 for cheating.

- You are free to organize your files and modules as you see fit, but you must turn in a `Makefile` to build your work. Your `Makefile` **MUST** be able to build your work as a bytecode executable and as a native executable. Obviously, both binaries must behave exactly the same way. You can use `OCamlMakeFile` if you want.

Chapter III

Foreword

Food for thought:

> This is the only time! The players fight with all their strength: the fans cheer for their favorite team. They forget pain, suffering... Only the game matters! That's why blitz has been around for so long. Least that's what I think.

By Wakka, from Final Fantasy X.

Chapter IV

Mandatory Part

This week was a lot of fun; or at least, I hope it was a lot of fun for you. But now that you've polished your skills in training, fighting your way through mud, dirt, and horrendously boring exercises only limited by the cruel imagination of yours truly, here's your time to rise and shine, and show the world what you can do with what you've learnt of OCaml so far !

This first rush will be your very first full and "complex" program. But don't worry, let's take a few minutes so everything's clear in your minds, and I'm sure you'll do great.

For the next two days, you'll be working on a game of tic-tac-toe. This game is played by two players on a 3x3 cells board.

```
- - -
- - -
- - -
```

Let's number the cells like this:

```
1 2 3
4 5 6
7 8 9
```

Each player fills a cell, player one with `O` and player two with `X`. By convention, the player with `O`s always starts. The player's goal is to make a straight line with three of his symbols.

```
X - -
O O O
- - X

O wins!

X - -
X O O
X O O

X wins!
```

This is the simple tic-tac-toe; but our tic-tac-tic-tac-toe is a bit more complicated. Each cell is a board itself, meaning that a game of tic-tac-tic-tac-toe is actually a set of nine tic-tac-toe games.

Players pick a cell one after another, using a simple format like `(row) (column)`. No line discipline at all is expected. Use a simple `read_line`, and note that `ctrl-d` won't be tested in defence.

```
- - - | - - - | - - -
- - - | - - - | - - -
- - - | - - - | - - -
-------------------
- - - | - - - | - - -
- - - | - - - | - - -
- - - | - - - | - - -
-------------------
- - - | - - - | - - -
- - - | - - - | - - -
- - - | - - - | - - -

O's turn to play.
1 3

- - O | - - - | - - -
- - - | - - - | - - -
- - - | - - - | - - -
-------------------
- - - | - - - | - - -
- - - | - - - | - - -
- - - | - - - | - - -
-------------------
- - - | - - - | - - -
- - - | - - - | - - -
- - - | - - - | - - -

X's turn to play.
toto
Incorrect format.
33 7
Illegal move.
1 3
Illegal move.
5 5

- - O | - - - | - - -
- - - | - - - | - - -
- - - | - - - | - - -
-------------------
- - - | - - - | - - -
- - - | - X - | - - -
- - - | - - - | - - -
-------------------
- - - | - - - | - - -
- - - | - - - | - - -
- - - | - - - | - - -
```

Both players try to win the game by winning the main board by winning the nested grids. If a nested grid ends up in a draw, the winner is the one who puts the last (and ninth) symbol into the grid.

```
- - O | - - - | - - -
- - - | - - - | - - -
- - - | - X - | - - -
-------------------
- - - | - - - | - - -
- O - | X X - | - - -
- O - | - - - | - - -
-------------------
- - - | - - - | - - -
- - - | - - - | - - -
- - - | - - - | - - -

O's turn to play.
4 2
O wins grid 4!

- - O | - - - | - - -
- - - | - - - | - - -
- - - | - X - | - - -
-------------------
/ - \ | - - - | - - -
|   | | X X - | - - -
\ - / | - - - | - - -
-------------------
- - - | - - - | - - -
- - - | - - - | - - -
- - - | - - - | - - -
```

The game ends when one player has a straight line of his symbols on the main board.

```
\   / | \   / | X - -
  X   |   X   | O X O
/   \ | /   \ | O - -
-------------------
/ - \ | - - - | - - -
|   | | - - - | - - -
\ - / | - - - | - - -
-------------------
/ - \ | - - - | - - -
|   | | - - - | - - -
\ - / | - - - | - - -

X's turn to play.
3 9
X wins grid 3!
X wins the game!

\   / | \   / | \   /
  X   |   X   |   X
/   \ | /   \ | /   \
-------------------
/ - \ | - - - | - - -
|   | | - - - | - - -
\ - / | - - - | - - -
-------------------
/ - \ | - - - | - - -
|   | | - - - | - - -
\ - / | - - - | - - -
```

Chapter V

Bonus Part

V.1 Mandatory bonuses

These bonuses are considered "mandatory" because they are fundamental and common improvements for your work. Other bonuses will not be counted if these are not implemented fully and perfectly.

- A one-player mode against an IA. It **must** try to win and to prevent you from winning.
- Letting the players enter their name, and display it instead of `O`/`X`. Both players can't use the same names, or use empty names.
- Letting the players begin a new game after the previous one has ended.

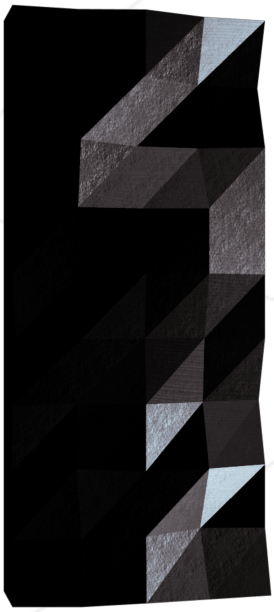
V.2 Optional bonuses

You're free to provide any feature you want as bonus. But keep in mind that everything you present as a bonus must not keep the program from behaving like specified in the subject's mandatory part. If your feature does alter the program's behaviour as such, the end-user must be able to disable and enable it at runtime (i.e. without having to recompile the application).

Here are some suggestions for you:

- A badass graphical interface, with mouse control and colors!
- A badass ncurses interface, with an arrow-controlled cursor and colors!
- A customizable grid size! Why not 4x4, 5x5...?
- A customizable grid nesting! Because we need to go deeper.
- A game timer like in chess, Go or other zero-sum games!

Really the only limit for this section is your imagination, so go ahead and show the world how you can create rainbows and unicorns with OCaml!



OCaml Pool - Rush 01

Instant Tama

42 pedago pedago@42.fr
kashim vbazenne@student.42.fr
nate alafouas@student.42.fr

Abstract: This is the subject for rush01 of the OCaml pool.
The main theme of this day is to use a graphic library with OCaml in order to create a basic Tamagotchi like game.

Contents

Chapter I

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 II

Day-specific rules

- You MUST not push your graphical library in the repository but provide a clever Makefile rule to install it instead.
- You are in a functional programming pool, so your coding style MUST be functionnal (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 functionnality that can't be tested won't be graded!**
- You must provide an author file at the root of your git with your login in it, as usual...

Chapter III

Foreword

Here lies a pretty song based on the famous Instant Karma from John Lennon
Instant Tama's gonna get you
Gonna knock you right on the head
You better get yourself together
Pretty soon you're gonna be dead
What in the world you thinking of
Laughing in the face of love
What on earth you tryin' to do
It's up to you, yeah you

Instant Tama's gonna get you
Gonna look you right in the face
Better get yourself together darlin'
Join the OCaml race
How in the world you gonna see
Laughin' at fools like me
Who in the hell d'you think you are
A super star
Well, right you are

Well we all cried out
Like the staff and the other students
Well we all cried out
Ev'ryone come on

Instant Tama's gonna get you
Gonna knock you off your feet
Better recognize your brothers
Ev'ryone you meet
Why in the world are we here
Surely not to live in pain and fear
Why on earth are you there
When you're ev'rywhere
Come and get your share

Well we all cried out
Like the staff and the other students
Well we all cried out

Well we all cried out
Like the staff and the other students
Well we all cried out

Well we all cried out
Like the staff and the other students
Well we all cried out
Like the staff and the other students
Well we all cried out
Like the staff and the other students
Well we all cried out
Like the staff and the other students

Chapter IV

Mandatory Part

In this part you must provide sufficient functionnality to demonstrate your almighty power in OCaml.

- `Your program must draw a wibbily wobbly timey wimey creature on the main screen`
- `You must also draw basic meters : health, hygiene, energy, happyness (you're free to draw it as you like but it must remain consistent with the concept.)`
- `You must also draw basic button/area to allow the end user to select an action to perform : EAT, THUNDER, BATH, KILL`

Your pet must obey the following rules:

- `Health is initially set to 100 and is depleted by 20 each time an action other than EAT is performed and depleted by 1 each second`
- `Energy is initially set to 100 and is depleted by 10 each time an actionother than THUNDER is performed`
- `Hygiene is initially set to 100 and is depleted by 20 each time an EAT action is performed`
- `Happiness is initially set to 100 and is depleted according to the following list of actions`

Here is the list of the actions and their side effects:

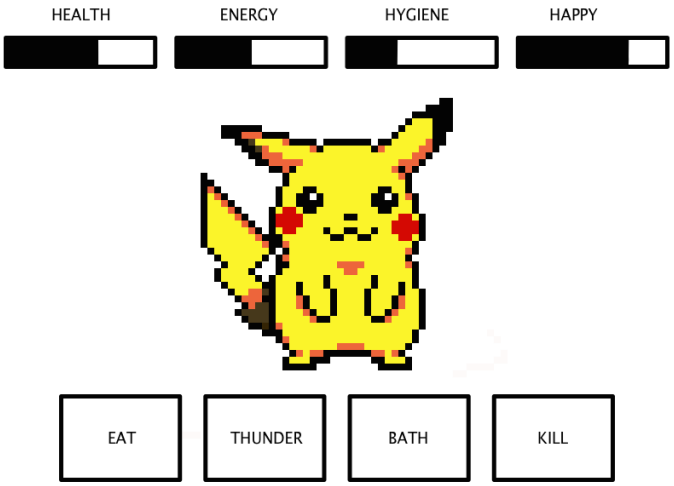
- `EAT : Health + 25, Energy - 10, Hygiene - 20, Happiness + 5`
- `THUNDER : Health - 20, Energy + 25, Happiness - 20`
- `BATH : Health - 20, Energy - 10, Hygiene + 25, Happiness + 5`
- `KILL : Health - 20, Energy - 10, Happiness + 20`

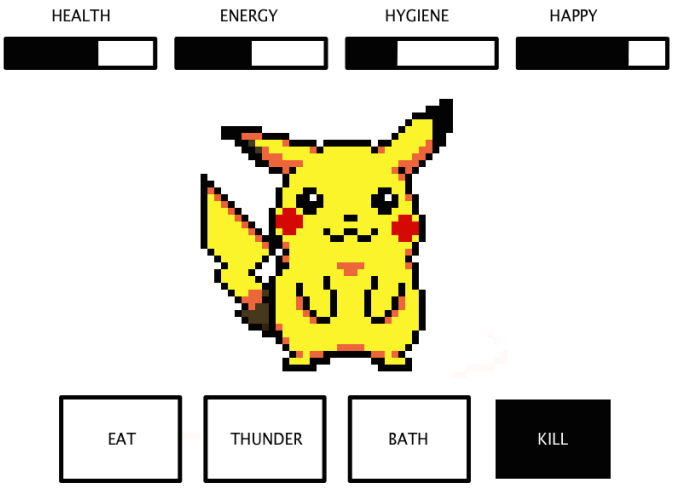
Also if any of the stats of your pet reach the value 0, your pet is instantly killed dramatically with a GAME OVER.

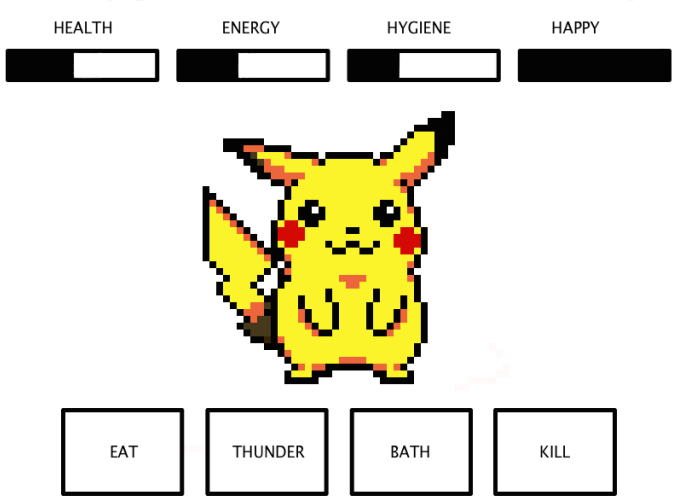
You should also provide an efficient way to save the state of your game into a file named **save.itama** and load it correctly at the next launch. (It would be wise to use the notions from the d09 of your OCaml pool... Isn't it?)

You're free to use any graphic library you like to achieve this rush.

Below are some screenshots of a relatively beautiful majestic as fuck Instant Tama.






HEALTH
ENERGY
HYGIENE
HAPPY
EAT
THUNDER
BATH
KILL

Chapter V

Bonus Part

You're free to provide any functionnality you want as bonus. Here are some of the most common functionnnality :

- `New actions : SLEEP, SING, DANCE... but you must not alter the mandatory actions!`
- `Super Gangsta Graphics with vertex shaders, tiles, and all the mega features of you library`
- `Sonor effects when an action is performed`

WARNING : if you program crash at any time or throw an unhandled exception, your work won't be graded, even if the crash comes from a BONUS.