B4- Functional Programming

B-PAV-360

Bootstrap : Bases

Cities breaking down on a camel’s back

1.3

Bootstrap : Bases

Cities breaking down on a camel's back

binary name:	
repository name:	OCAML_2016_bootstrap_base
repository rights:	ramassage-tek
language:	OCaml
group size:	1
compilation:	via Makefile, including re, clean and fclean rules

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

This Bootstrap consists of two parts. The first "Preliminaries" part should be done **before** coming to the Bootstrap.
The second "Bootstrap" part is to be done during the Bootstrap session, but keep in mind that it is really long because it covers a lot of different language.
Therefore, I recommend that you read it and prepare beforehand.

For this Bootstrap, your sources must be turned in **at the root of the repository** in different folders, whose names are given in each exercise.
Make sure to use the file names in order to be able to take the autograder tests. The tests will be carried out with an OCaml interpreter, so make sure your files function correctly in this environment.

To help you out, I strongly recommend consulting the official information on the language at this address, and especially the information about the Pervasives module here.
By default, this standard library module is open in all of your programs.

The preliminaries exercises are to be done before coming to the Bootstrap.
They are not difficult and will enable you to familiarize yourselves with OCaml.
Each required file must be at the root of the repository.

Preliminaries

What does the camel do?

This exercise consists of trying to predict OCaml's instructions' return.
As you may have seen, when you declare something with the keyword, `let`, OCaml displays the name, the type and the value of the declared element.
Therefore, for each instruction, you can try to predict what the return is that is displayed by the interpreter.

The instructions are contained in the `instructions.txt` file, which is available on the intra.
You must turn in your answers in a file named `retour.txt`.
The first line will correspond to the return from the first intruction, the second line will correspond to the second instruction,...
Play along and try to predict the returns rather than directly looking at what happens in the interpreter.

Don't forget to put an empty line in if you don't know what to put, and display the error if need be.
In each exercise, remember that the instructions come one after another.

First function

In a file named `incr.ml`, write the `incr` function, whose type is `int -> int` and which returns the integer passed as parameter that is incremented by 1. Your function should perform like the following example:

Terminal

```
∼/B-PAV-360> ocaml
OCaml version 4.00.1
# incr 0;;
-: int = 1
# incr 41;;
-: int = 42
```

Factorial

In a file named `fact.ml`, write two functions (`fact_if` and `fact_match`), whose type must be `int -> int` and defined in the following way:

$$fact(n) = \begin{cases} n \times fact(n-1) & \text{when } n \geq 1 \\ 1 & \text{when } n = 0 \end{cases}$$

Both functions must be **recursive**.
The `fact_if` function should use an `if .. then .. else`.
The `fact_match` function should use a filter with the `function` keyword.
Your functions should perform like the following example:

```
Terminal
~/B-PAV-360> ocaml
OCaml version 4.00.1
# fact_if 0;;
-: int = 1
# fact_match 5;;
-: int = 120
# fact_if 7;;
-: int = 5040
```

Variants

In a file named `boolean.ml`, you are going to define a variant type named `boolean` and a function named `my_and`.
The `boolean` variant type can take `True` and `False` values.
The `my_and` function (of type `boolean -> boolean -> boolean`) will mimic the logical "AND" to both booleans in parameter.
You must use a `match .. with` filter in this function.

Your function should perform like the following example:

```
Terminal
~/B-PAV-360> ocaml
OCaml version 4.00.1
# my_and False True;;
-: boolean = False
# my_and True True;;
-: boolean = True
# my_and False False;;
-: boolean = False
# my_and False True;;
-: boolean = False
```

Tuples

As a reminder, a tuple (or "n-uplet") is a non-mutable list. Once it's created, a tuple cannot, in any way, be modified. It's a tuple's order and element type that defines a tuple's type.

For example, let's take the following tuple, `(42, "epitech", true)`.
This tuple is an `(int * string * bool)` type.
Think about a student represented in the form of a `(string * string * int * bool)` type tuple.

The elements that make up this tuple are, respectively:

- the first name (string),
- the last name (string),
- the graduation year (int),
- if the student is a closed-account or not (bool).

In a file named `tuple.ml`, write the following functions that will allow our tuple to be handled:

```
get_first_name   : 'a * 'b * 'c * 'd -> 'a
get_last_name    : 'a * 'b * 'c * 'd -> 'b
get_grad_year    : 'a * 'b * 'c * 'd -> 'c
is_close         : 'a * 'b * 'c * 'd -> 'd
is_in_grad_year  : 'a * 'b * 'c * 'd -> 'c -> bool
```

Your functions should perform like the following example:

Terminal

```
~/B-PAV-360> ocaml
OCaml version 4.00.1
# let student = ("David", "GIRON", 2009, false);;
val student : string * string * int * bool = ("David", "GIRON", 2009, false)
# get_first_name student;;
-: string = "David"
# get_last_name student;;
-: string = "GIRON"
# get_grad_year student;;
-: int = 2009
# is_close student;;
-: bool = false
# is_in_grad_year student 2009;;
-: bool = true
# is_in_grad_year student 2015;;
-: bool = false
```

Bootstrap

Fibonacci

In a file named `fibo.ml`, write the `fibo` function, whose type is `int->int` and is defined in the following way:

$$fibo(n) = \begin{cases} fibo(n-1) + fibo(n-2) & \text{when } n \geq 2 \\ 1 & \text{when } n = 1 \\ 0 & \text{when } n = 0 \end{cases}$$

Your function should be recursive and **should not** use a `if then else` construction. Instead, you should use a filter.
Your function should perform like the following example:

```
Terminal
~/B-PAV-360> ocaml
OCaml version 4.00.1
# fibo 0;;
-: int = 0
# fibo 1;;
-: int = 1
# fibo 2;;
-: int = 1
# fibo 13;;
-: int = 233
# fibo 25;;
-: int = 75025
```

Tuples

By using the `is_in_grad_year` function from the preliminaries, define the following **recursive** function:

```
is_in_grad_year_range : ('a * 'b * int * 'c) -> int -> int -> bool
```

Which should perform as follows:

```
Terminal
~/B-PAV-360> ocaml
OCaml version 4.00.1
# is_in_grad_year_range student 2000 2010;;
-: bool = true
# is_in_grad_year_range student 2000 2003;;
-: bool = false
# is_in_grad_year_range student 2000 2000;;
-: bool = false
# is_in_grad_year_range student 2010 2010;;
-: bool = false
# is_in_grad_year_range student 2009 2009;;
-: bool = true
# is_in_grad_year_range student 2010 2002;;
-: bool = false
```

Variants

Variants are extremely common in OCaml. Therefore, it is crucial that you know how to use them.
I'd like to remind you that a variant allows you to define a type and to express its definition domain (like the enums in C), but also allows you to configure this domain's values. Also, contrary to C, the values (or constructors) of a variant self-report toward themselves instead of toward the integers.
Remember that each of a variant's values is called a "constructor" (can be configured), whose username must start with an uppercase letter.

You're going to create a little calendar and a few function tools to go along with it.
This exercise's code must be written in a file named `variant.ml`.

1- Write a variant type named **day** that lists the days of the week.
2- Write a variant type named **month** that lists the months of the year.

The dates will be represented in the form of a tuple that has the following type:

```
day * int * month * int
```

For both of the above functions, you must make a filter that is introduced by the `function` keyword.

3- Write a **next_day** function from `type day -> day` that returns the day following the one passed as parameter.
4- write a **next_month** function from `type month -> month` that returns the months following the one passed as parameter.

The output must be identical to the following:

Terminal

```
~/B-PAV-360> ocaml
OCaml version 4.00.1
# next_day Sunday;;
-: day = Monday
# next_month December;;
-: month = January
```

Now you are going to write a function that determines if a year is a leap year. I would like to remind you that a year is a leap year if it is divisible by 4 and not divisible by 100, or if it is divisible by 400.

The modulo operator in OCaml is "mod".

5- Write a `int -> bool` type **is_leap_year** function that returns true if the year passed as parameter is a leap year, and returns false otherwise.

Terminal

```
~/B-PAV-360> ocaml
OCaml version 4.00.1
# is_leap_year 2010;;
-: bool = false
# is_leap_year 2000;;
-: bool = true
# is_leap_year 2012;;
-: bool = true
```

You can now write a function that returns the number of days per month. The first parameter is the year and the second is the month. This will enable you to declare a filter on the months by using the function keyword.

6- Write a `int -> month -> int` type **nb_days** function that returns the number of days in the month of the year passed as parameter.

Terminal

```
~/B-PAV-360> ocaml
OCaml version 4.00.1
# nb_days 2011 January;;
-: int = 31
# nb_days 2011 February;;
-: int = 28
# nb_days 2012 February;;
-: int = 29
```

You also need a function that increments the day's number by taking into account the number of days in a month, thanks to the previous function.

7- Write a `int -> month -> int -> (int * bool)` type **next_nday** function.
It takes the day (in numbers), the month and the year as parameters.
It returns a couple (tuple) that is composed of the day, incremented by 1, as well as a boolean that represents the remainder if the incrementation makes it go on to the 1st of the following month.
You must use the `nb_days` function that was previously defined to determine if the day incrementation makes it go on to the following month.
Of course, you must handle the leap years.

Terminal

```
~/B-PAV-360> ocaml
OCaml version 4.00.1
# next_nday 22 January 2011;;
-: int * bool = (23, false)
# next_nday 31 January 2011;;
-: int * bool = (1, true)
# next_nday 28 February 2012;;
-: int * bool = (29, false)
# next_nday 28 February 2011;;
-: int * bool = (1, true)
```

Finally, you can increment a complete date by taking the end of weeks, months, years and leap years into account.

8- Write a `(day * int * month * int) -> (day * int * month * int)` type `next` function.
It takes a date as as parameter and returns the next day.

Terminal

```
~/B-PAV-360> ocaml
OCaml version 4.00.1
# next (Friday, 31, December, 2010);;
-: day * int * month * int = (Saturday, 1, January, 2011)
# next (Sunday, 29, February, 2004);;
-: day * int * month * int = (Monday, 1, March, 2004)
# next (Saturday, 28, February, 2004);;
-: day * int * month * int = (Sunday, 29, February, 2004)
# next (Monday, 28, February, 2005);;
-: day * int * month * int = (Tuesday, 1, March, 2005)
```

Lists

The `list` type is a type that is native to OCaml and is still the most common data structure in this language. The standard OCaml library offers a very comprehensive and practical list-handling module called List. The subject of the module's first project consists of rewriting several of these functions, which makes for an excellent technical exercise and which will certainly inspire you to write your own module in C, if you haven't already done so.

In certain specific cases, the List module's functions can raise an exception.
A reminder about exceptions is given a little further on.
Feel free to watch the video about lists again before starting the following exercises.

In order to introduce what you need to do, and to give you some practice, let's looks at a lists function together: `iter`.

Almost all of the project's functions rely on scanning a list. Therefore, the iter function is an ideal place to start.
As its name indicates, this function consists of scanning each element on a list and applying the following side effect, like displaying it.

Let's decode this type of function together: `('a -> unit) -> 'a list -> unit`.

- `'a -> unit`
  the first parameter is a function that takes an 'a type in parameter and returns unit. This function will be the one we will apply to each of the list's elements.
- `'a list`
  the second parameter is a list of 'a type elements. Of course, it's the same 'a type as above.
- `unit`
  iter returns "nothing". It only consists of a series of one-function applications that self-report toward unit.

You will probably have noticed that this function is very similar to C++'s STL for_each algorithm that you encountered at the end of your C++ pool.
To start this function's execution, I suggest writing iter by only scanning the list, without worrying about the function passed as parameter.

Let's remind ourselves of the pseudo code for scanning a list recursively:

```
Scan(list) :
        If list == empty (* Basic case *)
                Return;
        Otherwise (* Recursive case *)
                head = First_element(list);
                rest = Next_list(list);
                Scan(rest);
        Return;
```

Thanks to the on-the-fly deconstruction filter, this pseudo code can be adapted to so that it is more in-line with this language:

```
Scan(list) :
        List filter:
                .Case list_empty  -> Return
                .Case head::rest  -> Scan(rest)
```

Now it's your turn to write this function in OCaml, starting from a pseudo code. The work is already well under-way!
Write the `'a list -> unit` type **iter** function that scans the list passed as parameter.

In OCaml, a sequence of instructions is expressed in an unlimited block by the "begin" and "end" keywords. The instructions are separated by a ';'.

Let's now insert a random treatment in the iter function.
In a file named `list.ml`, write the `('a -> unit) -> 'a list -> unit` type **iter** function that applies the function passed as parameter to each element of the list that is passed in second parameter.

Terminal

```
~/B-PAV-360> ocaml
OCaml version 4.00.1
# iter print_endline ["Hello"; "you"; "tech2's!"];;
Hello
you
tech2's!
- : unit = ()
# iter (fun n -> print_int n) [0; 1; 2; 3];;
0123- : unit = ()
```

Exceptions

To complete your project, you will need to use OCaml's expressions.
In fact, some of the project's functions are liable to raise an exception and we are going to take advantage of this exercise to learn about their syntax in OCaml.

You have already learned about the exception mechanism during your C++ pool, and you will be happy to hear that the OCaml exceptions function in a rather similar way.
A certain number of standard exceptions exist, but you can naturally declare yours thanks to the following syntax:
`exception <Name> [of <type>].`

The name of the exception must take an uppercase letter and can be followed by a type.
We can easily recognize a variant's syntax! We will see that it is not a coincidence that the exceptions are a special variant type that we can dynamically increase, as opposed to classic variants.

Let's look at a new exception declaration example:

Terminal

```
∼/B-PAV-360> ocaml
OCaml version 4.00.1
# exception My_excep;;
exception My_excep
# My_excep;;
- : exn = My_excep
# exception Other of int;;
exception Other of int
# Other 42;;
- : exn = Other 42
```

We can see that all of the exceptions are an exn type. So, each time we declare a new exception, we're really just increasing the exn variant with a new constructor.
Increasing any random variant is impossible, but exn is a special variant that has this characteristic.

To raise an exception, we use the `raise` keyword, followed by a constructor for the exception to be raised.
The syntax is surprisngly simple.

Terminal

```
∼/B-PAV-360> ocaml
OCaml version 4.00.1
# exception My_excep;;
exception My_excep
# raise My_excep;;
Exception: My_excep.
```

The exception will move up the call stack until it is caught. If it's never caught, the program will terminate...brutally...
In order to avoid this brutal termination, and to handle an exception (whether it's defined in the standard library or manually, as seen above), we will use the `try` keyword, in front of the expression that is likely to raise an exception, followed by a filter on the possible exceptions to handle them.

The following example illustrates how exceptions are handled:

```
Terminal
~/B-PAV-360> ocaml
OCaml version 4.00.1
# exception My_excep_1;;
exception My_excep_1

# exception My_excep_2;;
exception My_excep_2

# try
raise My_excep_1
with
| My_excep_1 -> print_endline "[exception 1 !]"
| _ -> print_endline "exception not handled"
;;
[exception 1 !]
- : unit = ()

# try
raise My_excep_2
with
| My_excep_1 -> print_endline "[exception 1 !]"
| _ -> print_endline "exception not handled"
;;
exception not handled
- : unit = ()
```

For practice, in a file named `exception.ml`, write the `int list -> int` type `list_mul` function, which takes a list of integers as parameters and returns their product.
If one of the elements is equal to 0, raise an exception of your choice that interrupts the list scan and directly returns the value 0.

Conclusion

This bootstrap should have helped you understand several basic concepts of functional programming and OCaml, to start serenely your projects.

Don't hesitate to go back over typos, English errors and even your own feelings about these exercises.
Student feedback is always interesting to improve pedagogical materials.