B4- Functional Programming

B-PAV-360

MyList

Recoding OCaml lists

1.2

MyList

Recoding OCaml lists

binary name:	
repository name:	OCAML_2016_mylist
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).

You must turn in a file named `mylist.ml` at the root of your repository. This file IS NOT a binary file, it is a simple text file. The functions in the file MUST have exactly the same type (except the lists that become my_list) and the same performance as those in the standard library's List module (except for the iter type, which has an equivalent).

- You MUST NOT define your own exceptions. Use the standard exceptions (Failure, Invalid_argument, Not_found,...) with the same values.
- All of the standard library's modules are prohibited, List included, obviously.
  Only the Pervasives module, open by default in every Ocaml program, is authorized.
- Usage of the list's standard type and its `::` constructor are prohibited.
  The Pervasives `@` module is also prohibited.
- The bonus points will only be counted if you have all of the points for the mandatory section.
- All of your functions MUST be located in a file named `mylist.ml` in the root of your repository.
  Only the content of this file will be evaluated by the autograder during the oral presentation.
- The definition of the my_list type is located in your turn-in file and is identical to the one given in the project description.
- All of OCaml's imperative traits are prohibited, except those concerning the exceptions and instruction sequences when necessary. This is obviously limited to the bare minimum.
- There is no norm for OCaml at Epitech. However, any unreadable code could be randomly penalized.
  A clear identation, intelligent naming, space between the functions and complying with the 80 columns are strongly recommended.
- As the code to produce is restricted, the oral presentation will particularly emphasize your own personal knowledge.

Advice

This is a short and particularly easy project. If you take it seriously, you will understand the basics of functional language and paradigm.

- Don't hesitate to create values and local functions in your (`let ... in`) functions. This will make it so you won't get mixed up. I insist.
- Avoid coding everything on one line! Lighten your code and indent it. That's an order.
- If your code seems dirty, that's because it probably is. Each of this project's functions is short and OCaml is a clean language for clean people.
- Getting used to declaring a filter with several cases with the `function` keyword rather than `match with` makes the code much more readble. Do it.
- Don't hesitate to use the _ username when a username isn't useful.
- Forcing yourself to put the `|` in front of the first filter case and aligning the -> for each case also usually makes it easier on the eyes.
- Your code will be loaded and evaluated in the interpreter. So make sure that everything functions correctly in this environment!

mylist

Handling lists is a common action in OCaml.
The language's standard library offers a very comprehensive module to carry out these actions.
We suggest recoding your own list type and its related functions.
It's an excellent exercise for learning about OCaml's paradigm and syntax!

You will use the following concepts from class:

- variants,
- parameterized types,
- parametric polymorphism,
- recursion,
- filters.

The my_list type
For this project, you won't use OCaml's list's standard type that you might have seen during the Bootstrap, but a personal type that emulates its behavior.

For this, let's look at the behavior of the standard `list` type.
The `::` symbol allows you to add an element to a list's header.
We call it the ''lists constructor''.

Therefore, we can create a list by successively adding elements to the header of the empty list.
Since the list constructor is associated with the right, the parentheses aren't necessary.
That's why a list can be created in the two following ways:

Terminal

```
~/B-PAV-360> ocaml
OCaml version 4.02.2
# let ingredients = "flour"::("butter"::("milk"::("sugar"::([]))));;
val ingredients : string list = ["flour"; "butter"; "milk"; "sugar"]
# let ingredients = "flour"::"butter"::"milk"::"sugar"::[];;
val ingredients : string list = ["flour"; "butter"; "milk"; "sugar"]
```

If you have any questions about one of the above lines, take the time to read the class material again, redo the Bootstrap section on lists and ask yourselves some questions.

We can deduce that only one type of list should offer both values:

- a recursive element that represents a value in a list's header,
- a list's terminal element.

A recursive variant is the best solution!
Also, OCaml's standard list type can have any type of values.
Our list type should therefore be **parametric**. Hence the following type:

```
type 'a my_list = Item of ('a * 'a my_list) | Empty
```

Let's use our ingredients list with our list type:

Terminal

```
~/B-PAV-360> ocaml
OCaml version 4.02.2
# let ingredients = Item ("flour", Item ("butter", Item ("milk" , Item ("sugar", Empty))));;
val ingredients :  string my_list =
Item ("flour", Item ("butter", Item ("milk", Item ("sugar", Empty))))
```

Victory! Our `my_list` type emulates perfectly the standard List type!

Mandatory section

Now that you have your own type of list, the project consists of recoding 20 functions from the List module that are adapted to your `my_list` type, that was introduced and defined in the previous chapter.

You will find official information about functions here.

The functions to recode are as follows:

```
cons         : 'a -> 'a my_list -> 'a my_list
length       : 'a my_list -> int
hd           : 'a my_list -> 'a
tl           : 'a my_list -> 'a my_list
nth          : 'a my_list -> int -> 'a
rev          : 'a my_list -> 'a my_list
append       : 'a my_list -> 'a my_list -> 'a my_list
rev_append   : 'a my_list -> 'a my_list -> 'a my_list
flatten      : 'a my_list my_list -> 'a my_list
iter         : ('a -> 'b) -> 'a my_list -> unit
map          : ('a -> 'b) -> 'a my_list -> 'b my_list
fold_left    : ('a -> 'b -> 'a) -> 'a -> 'b my_list -> 'a
for_all      : ('a -> bool) -> 'a my_list -> bool
exists       : ('a -> bool) -> 'a my_list -> bool
mem          : 'a -> 'a my_list -> bool
memq         : 'a -> 'a my_list -> bool
filter       : ('a -> bool) -> 'a my_list -> 'a my_list
split        : ('a * 'b) my_list -> 'a my_list * 'b my_list
combine      : 'a my_list -> 'b my_list -> ('a * 'b) my_list
partition    : ('a -> bool) -> 'a my_list -> 'a my_list * 'a my_list
sort         : ('a -> 'a -> int) -> 'a my_list -> 'a my_list
```

Optional section

If you want more practice, this section was made for you!
Have fun coding the other functions from the List module:

```
mem_assoc     : 'a -> ('a * 'b) my_list -> bool
assoc         : 'a -> ('a * 'b) my_list -> 'b
remove_assoc  : 'a -> ('a * 'b) my_list -> ('a * 'b) my_list
mem_assq      : 'a -> ('a * 'b) my_list -> bool
remove_assq   : 'a -> ('a * 'b) my_list -> ('a * 'b) my_list
fold_right    : ('a -> 'b -> 'b) -> 'a my_list -> 'b -> 'b
iter2         : ('a -> 'b -> 'c) -> 'a my_list -> 'b my_list -> unit
map2          : ('a -> 'b -> 'c) -> 'a my_list -> 'b my_list -> 'c my_list
merge         : ('a -> 'a -> int) -> 'a my_list -> 'a my_list -> 'a my_list
```

Don't hesitate to invent functions that don't exist in the module, but that could be useful. For example:

```
remove_all_assoc : 'a -> ('a * 'b) my_list -> ('a * 'b) my_list
```

Same behavior as `remove_assoc`, but deletes all of the elements that correspond to the given key.