B4- Functional Programming

B-PAV-360

Bootstrap

Walk like a camel

1.6

Bootstrap

Walk like a camel

repository name:	OCAML_2016_bootstrap_module
repository rights:	ramassage-tek
language:	OCaml
group size:	1

- 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).

Now that you are familiar with the syntax and basics of OCaml, it's time to move on to more serious things. You're going to be able to apply the skills that you saw during the third class (programming with OCaml). To begin, let's leave behind our OCaml interpreter and switch into compiled mode...

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 preliminary exercises are to be done before coming to the Bootstrap.

Each required file or directory must be at the root of the repository.

Preliminaries

Makefile

In this first exercise, you are going to write a `Makefile` file that you can use for your projects. You MUST put this Makefile file in a directory named `my_makefile`.

- Read the ocamlc and ocamlopt manuals L.
- Ẃrite a Makefile to compile the OCaml code.
- The default rule will compiles your sources towards the native code.
- A rule of your choice will compile your sources towards the bytecode.
- Your make MUST include the usual clean, clean, and re rules.
- Add all of the rules you feel are relavent.
- Take the time to make your Makefile correctly and once and for all. Remember: a smart developer avoids all useless effort.

Module Calendar

You are going to create your first module that will contain a signature in a `.mli` file. You must write two files, `calendar.mli` and `calendar.ml`, as well as a `main.ml` file that will use the calendar module. These files will be placed in the `calendar` directory.

Attention, don't add the Makefile to the `calendar` directory. We will use our own for the tests.

We suggest reviewing the work that was done on the variants during the first Bootstrap. It dealt with creating a list of functions that allowed dates to be manipulated with the help of two variants for the days and the months. You are going to reuse the developed functions during this Bootstrap in order to incorporate them into a Calendar module. The Calendar module signature will be as follows:

```
module type CALENDAR =
  sig
                type date = day * int * month * int

                val next_day      : day -> day
                val next_month    : month -> month
                val is_leap_year  : int -> bool
                val nb_days       : int -> month -> int
                val next_nday     : int -> month -> int -> int * bool
                val next          : date -> date
  end

module Calendar : CALENDAR
```

You will notice the `date` type declaration inside the signature and its usage in the `next` function. In this version, the type is defined (we use a '=' to define it), but we could also not define it and make it abstract. I urge you to test this option to see how the interpreter behaves in this situation.

I suggest you review the class on module language in order to really understand how to write your module and use it

in a main. In particular, the declarations of these variants for the days and the months should be placed wisely. The tests will take your module and test it on a main (compilation + functionalities).

Bootstrap

A program's arguments

Passing arguments to a program is obviously possible in OCaml. Reading the module information from `Sys` standard library shows us the following:

```
val argv: string array
       The command line arguments given to the process.
       The first element is the command name used to invoke the program.
       The following elements are the command-line arguments given to the program.
```

`Sys.argv` is therefore a `string array` type value. You haven't used `'a array` type yet, but a quick look at the information will enable you to find the follow function that transforms it into a list:

```
       val to_list : 'a array -> 'a list
```

Ẃwrite a `printme` program that will display its arguments on the standard output. You must use the `Array.to_list`, `List.iter` and `print_endline` functions in your program.

Your program should be in the `print_me` directory, which will contain a Makefile that allows your program to compile. The executable should be found in the `print_me` directory. How the inside of the `print_me` directory is organized is up to you.

Input/Output

Read the Input/Output section of the Pervasives module information.

In a directory named `my_cat`, write a program named `cat` that has a similar behavior to the `cat` binary without options. Add the "`-e`" option to your program with the usual behavior for this option.