# B4- Functional Programming

B-PAV-360

# Agenda

Don't forget your appointments

1.5

# Agenda

## Don't forget your appointments

| | |
|---|---|
| **repository name**: | OCAML_2016_agenda |
| **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).

- Your repository should not contain a Makefile.
- Your respository should contain the follow files at its root: `agenda.ml`, `agenda.mli`, `contact.ml`, `contact.mli`, `event.ml`, `event.mli`.
- The `agenda.mli` file is provided for you and you MUST NOT change the interface's function prototypes. You can, however, add others if they don't conflict with the mandatory functions.

Please follow the guidelines found below to the best of your ability:

- All of the iterative functions MUST be recursive.
- All of the recursive functions MUST be terminal.
- If you use filters on the last parameter, your MUST use the keyword function.
- Don't hesitate to use the matching pattern: match...with.
- Don't use conditions after filters; use guards (when).
- If your function uses constant values (or does the same calculations with a result that doesn't change), don't hesitate to create a value that is local to the function.
- If your function requires non-constant local values, and it's recursive, you should pass this value as function parameter instead of creating it.
- Don't force your functions' typing.
- The value names should be clear and explicit: 3 or 4 letters in English.
- For the sake of clarity, align your filters, indent your code and comment on your code OUTSIDE of your functions.

# Instructions

The goal of this project is to explore the unit's concepts and to give you practice with manipulating OCaml's data structures: the lists, tuples, types, and filters that you have seen prior.

Feel free to refer to the official OCaml reference documents : Modules, Strings, Exceptions.

You have to create a functions library that allows you to store contacts and events. Your code will therefore be contained in three modules: `Agenda`, `Contact`, `Event`.

The `Contact` module must implement the Contact.contact abstract type.
Each contact must be made up of the following elements: first name, last name, age, email and telephone.
The `Event` module must implement the Event.event abstract type.
Each event must be made up of the following elements: date, time, duration and the list of invited contacts.
The rest of the content for the `Contact` and `Event` modules is up to you.

The `Agenda` module's interface is given in the `agenda.mli` file.
The predefined functions should not be modified. They will be used for the autograder tests. However, you can add functions if you would like.

Outside of the `bonus` directory, all external libraries or modules are strictly prohibited, except for the `List`, `String`, `Char` and `Buffer` modules.

Don't hesitate to regularly check out the OCaml group on Yammer.

Some functions in this project consist of looking up contacts according to the following list elements. You MUST use the following variant, which defines the different possible types of elements:

```
type field = All | Id | Firstname | Lastname | Age | Email | Phone
```

# Mandatory Functions

The following list of functions is mandatory in the Agenda Module:

- **addContact** : adds a contact
- **getContactId** : sends the contact's position on your list
- **removeContact** : deletes a contact
- **replaceContact** : replaces one contact with another
- **printContacts** : displays the contact list

## Optional Functions

The following list of functions is optional in the Agenda Module:

- **addEvent** : adds an event
- **getEventId** : sends the event's index or position on your list
- **removeEvent** : deletes an event
- **printEvents** : displays the list of events

You can add as many functions as you'd like, as far as they don't conflict with the mandatory functions.

# Specifications

## Examples

In order to show how the Agenda module functions, here is an example:

```
# let empty_data = [];;
val empty_data : 'a list = []
# let one_data = Agenda.addContact empty_data ("Guillaume", "Collet", 36,
    "guillaume.collet@epitech.eu", "01 84 07 42 10");;
val one_data : Contact.contact list = [<abstr>]
# let two_data = Agenda.addContact one_data ("Jo", "La Frite", 42, "jo.la-frite@epitech.eu", "01 84
    07 78 10");;
val two_data : Contact.contact list = [<abstr>; <abstr>]
# let three_data = Agenda.addContact two_data ("Felix", "Elcat", 65, "flechat@epitech.eu", "01 84
    07 92 22");;
val three_data : Contact.contact list = [<abstr>; <abstr>; <abstr>]
```

As you can see, since the `Contact.contact` type is abstract, `<abstr>` is written on the display.
We will never use words with accents for this project.

## addContact

The `addContact` function adds a contact to the contact list and is prototyped the following way:

```
val addContact: Contact.contact list -> string * string * int * string * string -> Contact.contact
    list
```

- The first argument is the contact list.
- The second argument is a tuple that represents a contact composed as follows: (first name, last name, age, e-mail, telephone).
- Adding a contact to the list should not change the alphabetical order of the contacts. This order is based on the last name, then the first name. Your list should always remain sorted.
- The first name MUST imperatively be capitalized. Since the capital letter is always located on the first letter, like in `François`. For hyphenated named, you must also have a capital letter after a space, a hyphen or an apostrophe, like in `Jean-Pierre`.
- The last name MUST imperatively be written in CAPITAL LETTERS.
- Useless spaces in last or first names must be deleted (no double or single spaces at the beginning or end of a line).
- If the age is less than 0 or higher than 120, you MUST raise an exception.
- An e-mail address MUST meet the following criteria:
  - contain the **@** symbol.
  - contain at least one character before and after the **@**.
  - contain a dot after the **@**.
  - contain at least one character before and after the dot.

  The e-mail address, a**@**b.c is therefore valid. If an e-mail is invalid, you MUST raise an exception.
- The telephone number must always include 10 numbers and spaces. No other symbols will be accepted. Whatever the input format, a telephone number will always be given in the following format: "## ## ## ## ##". Also, a telephone number must imperatively begin with a 0.
- Concerning an invalid age, e-mail address or telephone number and an empty first or last name (""), you MUST raise `Add_Contact_With_Invalid_Data` exception. Since this exception is defined in the Agenda module, it must appear in the following form:

  ```
  Fatal error: exception Agenda.Add_Contact_With_Invalid_Data
  ```

## getContactId

The `getContactId` function looks for a contact according to a search field and a character string. It returns the contact's position on the list. The search fields are defined in the following variant:

```
type field = All | Id | Firstname | Lastname | Age | Email | Phone
```

The function is prototyped the following way:

```
val getContactId : Contact.contact list -> field -> string -> int
```

- The first argument is the contact list, the second argument is the search field and the third argument is the character string to look for in the specified field.
- The contacts are numbered starting with 0.
- The search should not be case-sensitive.
- The `All` field searches in all of the other fields.
- If a search fails, the function returns -1.
- If the search field corresponds to one or several contacts, the function always returns the ID of the first found contact.

## removeContact

The `removeContact` function deletes a contact from a contact list according to its ID and is prototyped the following way:

```
val removeContact: Contact.contact list -> int -> Contact.contact list
```

- If the list passed in parameter is empty, you must raise a `Remove_Impossible_On_An_Empty_List` exception.
- If the ID is invalid, you must raise a `Remove_Using_An_Invalid_Id` exception.

## replaceContact

The `replaceContact` function replaces a contact with another one given in parameter according to its ID and is prototyped the following way:

```
val replaceContact: Contact.contact list -> int -> string * string * int * string * string ->
    Contact.contact list
```

- If the list passed as parameter is empty, you must raise a `Replace_Impossible_On_An_Empty_List` exception.
- If the ID is invalid, you must raise a `Replace_Using_An_Invalid_Id` exception.
- Make sure to manage the same error instances as the `add` function for the last name, first name, age, telephone number and e-mail address.
- In the event of an invalid age, e-mail address or telephone number, you must raise a `Replace_Contact_With_Invalid_Data` exception.

## printContacts

The `printContacts` displays the contacts on the standard output according to a search in a given field and is prototyped the following way:

```
val printContacts: Contact.contact list -> field -> string -> unit
```

- The search shouldn't be case-sensitive.
- If the third parameter is empty "", you must display the ENTIRE contact list.
- If the list passed as parameter is empty, or if the search fails, you shouldn't display anything.
- The contacts' data should never contain accents.
- Each display column has a fixed width that is defined for each field: 4, 16, 16, 4, 32 and 14 respectively.
- IF the fields go over the maximum length, only display the first x characters (x = the size of the column).
- Between each field, you should fill in the emptiness with spaces and not tabs.