B4 - Functional Programming

B-FUN-400

Haskell Basics

you should spend more time thinking than typing

1.1

{EPITECH.}

Be careful about the vocabulary used here, it's different from what you are used to. For example, 'variables' and 'instructions' are not used anymore, instead, we talk about 'values' and 'expressions'.

Before we start this Bootstrap, it is paramount that you become familiar with Haskell. You can use this resource:
The official Haskell website is packed with resources, tutorials and various tools' user guides.
You'll definitely need those for the project, so go check them right away.

Here is a non-exhaustive list of relevant websites:

- learnyouahaskell
- schoolofhaskell
- Functors, Applicatives, And Monads In Pictures
- a wiki for Haskell

In Haskell, pay extra attention to the types of expressions, functions, etc.

Stack

Install Stack, the most used platform for developping in Haskell, and find out details about this tool.
After installation, you'll get access to its main tools:

- the compiler: **ghc**,
- the interpreter: **ghci**.

From now on, write your functions in a file and load them into ghci.

If you want your functions to appear in the code, but not to actually do anything, use the keyword 'undefined'.

Step 1 : simple functions

In a file named `step1.hs`, write:

- a **prev3** function, that returns the integer passed as parameter minus 3.
  The type of the function must be : `prev3 :: Int -> Int`
- a **min3** function, that returns the smallest of the 3 integers passed as parameters.
  The type of the function must be : `min3 :: Int -> Int -> Int -> Int`
- a **isneg** function, that returns `True` if the integer passed as parameter is negative, or `False` if not.
  The type of the function must be : `isneg :: Int -> Bool`

:t

Step 2 : recursion

In a file named `step2.hs`, write:

- a **myFact** function, that calculates the factorial of the number passed as parameter (nope, you won't escape this!)
  If the parameter is negative, return 0.
  The type of the function must be : `myFact :: Int -> Int`
- a **mySqrt** function, that calculates the square root of the number passed as parameter.
  If the parameter is negative, return 0.
  The precision of the result shall be verified up to the third decimal.
  The type of the function must be : `mySqrt :: Float -> Float`.

Of course, you are not supposed to use Haskell `sqrt` function, but to code your own one.

STEP 3 : PATTERN MATCHING

In a file named `step3.hs`, write a **myBaro** function, that, depending on the season and the atmospheric pressure (`String` and `Int`), returns a description of the weather for the day.
The type of the function must be `:myBaro :: String -> Int -> String`

The weather descriptions stem from Moreux's Table hereunder:
Spring

1. 1020 hPa or more => Sunny or fair weather.
2. from 1006 to 1020 hPa => Showers or hail showers.
3. less than 1006 hPa => Rain and wind.

Summer

1. 1020 hPa or more => Sunny or fair weather.
2. from 1013 to 1020 hPa => Fair or possible showers.
3. from 1006 to 1013 hPa => Showers or hail showers.
4. less than 1006 hPa => Rain and wind.

Automn

1. 1020 hPa or more => Sunny or fair weather.
2. from 1013 to 1020 hPa => Local showers.
3. from 1006 to 1013 hPa => Showers, fresh weather.
4. less than 1006 hPa => Showers, chilly weather.

Winter

1. 1020 hPa or more => Fair and foggy.
2. from 1013 to 1020 hPa => Fair weather.
3. from 1006 to 1013 hPa => Snow or hail showers.
4. less than 1006 hPa => Snow, cold weather.

You can choose the data structures that contain the descriptions.

It will be interesting to compare your choices to those of other students.

STEP 4 : LISTS AND TUPLES

In a file named `step4.hs`, write:

- a **mySumEven** function, that, given a list of integers, returns the sum of the even numbers contained in the table.
  The type of the function must be : `mySumEven :: [Int] -> Int`
- a **myPal** function, that, given a list of elements, returns `True` if the list forms a palindrome.
  The type of the function must be : `myPal :: [Int] -> Bool`
- a **myAverage** function, that calculates the average of the integers contained in the list given as parameter.
  If the list is empty, return 0.
  The type of the function must be: `myAverage :: [Int] -> Int`
- a **myDicho** function, that carries out a divide-and-conquer sort algorithm on the integers list given as parameter.
  The type of the function must be : `myDicho :: [Int] -> [Int]`
- a **my2Sum** function, that, given a list of integers `l` and an integer `k`, returns a tuple containing 2 integers `m` and `n` such as `m + n = k` (where `m` and `n` belong to `l`).
  If no pair is found, the function returns the tuple `(-1,-1)`.
  The type of this function must be : `my2Sum :: [Int] -> Int -> (Int, Int)`

You can import the library Data.List to use the advanced functionalities of the lists.

STEP 5: MAYBE

In Haskell, some types encapsulate optional values.
This is the case for Maybe, which is defined as follows:

```
Data Maybe a = Just a | Nothing
```

It lets you express the fact that a function can return a value, or not.
For example, if you are searching for a value in a list, you may not be able to find it.
Thus, you may just have a value of type a (`Just a`), or no value (`Nothing`).

Have a look at the functions `lookup` and `find` of the Data.List library. They return a Maybe.

Write a function named myFind in the file step5.hs, that searches an element in a list, while respecting the following prototype (that you are expected to carefully dissect in order to understand it):

```
myFind :: Eq a => a -> [a] -> Maybe a
```

Fully code the function from scratch, it is about 3 lines of code, and do not use the `elem` function.

Maybe can be very useful (though limited) for error handlong.