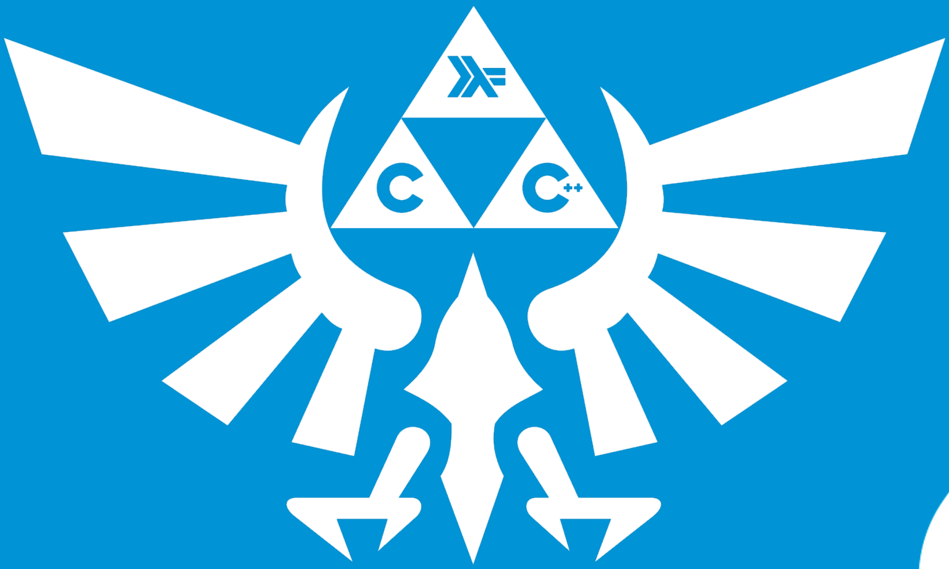
DAY 02

A GENTLE INTRODUCTION TO FUNCTIONAL PROGRAMMING



DAY 02

binary name: DoOp.hs, doop
language: haskell
compilation: via Makefile, including re, clean and fclean rules

- ✓ The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- ✓ All the bonus files (including a potential specific Makefile) should be in a directory named bonus.

Today's goal is to discover how to:

- ✓ handle errors in a more elegant way than with the error function.
- ✓ manage side effects (input and outputs).
- ✓ build an executable with Haskell.

Good news: you have now the right to use most of the functions of the standard library. You are free to use functions from your My.hs if you want, but all of the useful functions are included (bug free) in the standard library in a way or another, just look for it (myMap is map, myTake is take, etc...).

Banned functions for today: fromJust, readMaybe, reads, elem, lookup.

You still have to respect the Haskell Coding Style.

All functions are going in DoOp.hs. Additionaly, a Makefile must be present so that when make is called in the delivery directory, a doop binary is built from DoOp.hs.

Step 0 - Prerequisite

Let's start with something simple.

Task 01

```
myElem :: Eq a => a -> [a] -> Bool
```

A function which takes an argument which implements **equality** and a list of the same type, and returns True if the argument is present in the list, False otherwise.

For example:

```
Terminal
~/B-PDG-300> ghci DoOp.hs
Prelude> myElem 3 [1,2,3,4,5]
True
Prelude> myElem 7 [1,2,3,4,5]
False
```

Step 1 - Maybe

Maybe is a data type included in the Haskell's standard library which is a simple and elegant way to handle errors without using exceptions. It's also a **monad**.

You can inspect it's definition with ghci:

```
Terminal
~/B-PDG-300> ghci
Prelude> :info Maybe
data Maybe a = Nothing | Just a - Defined in 'GHC.Base'
[...]
```

Task 02

```
safeDiv :: Int -> Int -> Maybe Int
```

A function which takes two Ints as arguments, and returns **Nothing** if it's a division by zero, or **Just N** (with N the result of the division) if it's not.

Example:

```
~/B-PDG-300> ghci
Prelude> safeDiv 10 0
Nothing
Prelude> safeDiv 10 2
Just 5
```

Task 03

```
safeNth :: [a] -> Int -> Maybe a
```

A function similar to the function myNth, but which returns **Nothing** in case of an error, or **Just N** (with N the result) in case of success.

Task 04

```
safeSucc :: Maybe Int -> Maybe Int
```

A function which takes a Maybe Int and returns its successor wrapped in a maybe.

```
Terminal
~/B-PDG-300> ghci
Prelude> safeSucc (Just 41)
Just 42
Prelude> safeSucc (safeDiv 10 2)
Just 6
Prelude> safeSucc (safeDiv 10 0)
Nothing
```

Can you write safeSucc using the fmap function ?

Can you write safeSucc using the bind function/operator (>>=) ?

Task 05

```
myLookup :: Eq a => a -> [(a,b)] -> Maybe b
```

A function which takes an argument which implements equality and a list of tuples. Looks successively if the first element of each tuple is equal to the first argument. If a match is found, the second element of the tuple is returned wrapped in a **Maybe**. Otherwise, **Nothing** is returned.

Example:

```
Terminal
~/B-PDG-300> ghci
Prelude> myLookup 2 [(1,“foo”),(2,“bar”),(3,“baz”)]
Just “bar”
Prelude> myLookup 5 [(1,“foo”),(2,“bar”),(3,“baz”)]
Nothing
```

Task 06

```
maybeDo :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
```

A functions which takes a function as argument and two more arguments wrapped in **Maybes**, and returns Nothing if any of the arguments are **Nothing**, or **Just n** (with n the result of the function applied to the arguments) otherwise.

Example:

```
Terminal
~/B-PDG-300> ghci
Prelude> maybeDo (\ x y -> x + y) (Just 1) (Just 2)
Just 3
Prelude> maybeDo (*) (safeDiv 10 0) (Just 5)
Nothing
Prelude> maybeDo (+) (safeDiv 10 5) (safeSucc $ Just 39)
Just 42
```

Can you write maybeDo using the bind function/opertor (>>=) ?

Can you write maybeDo using the do notation?

Can you write maybeDo using fmap/(<$>) and the <*> operators?

Task 07

```
readInt :: [Char] -> Maybe Int
```

A function which takes a string as argument and returns Nothing if its not a parsable number, or an Int wrapped in a maybe otherwise.

Example:

```
Terminal
~/B-PDG-300> ghci
Prelude> readInt ‘‘42’’
Just 42
Prelude> readInt ‘‘foobar123’’
Nothing
```

You’re free to use the read function if you want, but **readMaybe** and **reads** are forbidden, obviously

Step 2 - IO

To handle Inputs and Outputs (IO), Haskell uses the IO monad, which permits to keep our function pure, while exchanging data from the outside world.

Task 08

```
getLineLength :: IO Int
```

A function which reads a line from the standard input and returns its length as a IO Int.

You should have a look on the getLine function

Task 09

```
printAndGetLength :: String -> IO Int
```

A function which takes a string as argument, prints it on the standard output followed by a carriage return, and returns its length in the IO monad.

Terminal

```
~/B-PDG-300> ghci
Prelude> printAndGetLength ‘‘hello world’’
hello world
11
```

Task 10

```
printBox :: Int -> IO ()
```

A function which takes an Int (N) as argument, and draw a box of height N and width N*2 on the standard output (see example bellow). If N is zero or less, it does nothing.

Terminal

```
~/B-PDG-300> ghci
Prelude> printBox 5
+--------+
|        |
|        |
|        |
+--------+
Prelude>
```

Task 11

```
concatLines :: Int -> IO String
```

A function which takes an Int (N) as argument, read N lines from the standard input and returns a concatenation of these lines in the IO monad. If N is 0 or negative, an empty string is returned.

Example:

```
Terminal
~/B-PDG-300> ghci
Prelude> concatLines 3
foo
bar
baz
“foobarbaz”
```

Task 12

```haskell
getInt :: IO (Maybe Int)
```

A function which reads a line from the standard input and returns a Maybe Int in the IO monad.

Example:

```
Terminal
~/B-PDG-300> ghci
Prelude> getInt
42
Just 42
```

Step 4 - doop

Task 13

A program called **doop**, which will be built by the command “make”.

This program takes three arguments on the command line. The first and third arguments are numbers (signed integers). The second argument is an arithmetic operator (either + - * / or %). The program must print the result of the operation to the standard output.

In case of error, your program must return 84.

```
Terminal
~/B-PDG-300> ./doop 1 + 2
3
~/B-PDG-300> ./doop 10 / 0
~/B-PDG-300> echo $?
84
```

v 1.1

{EPITECH}