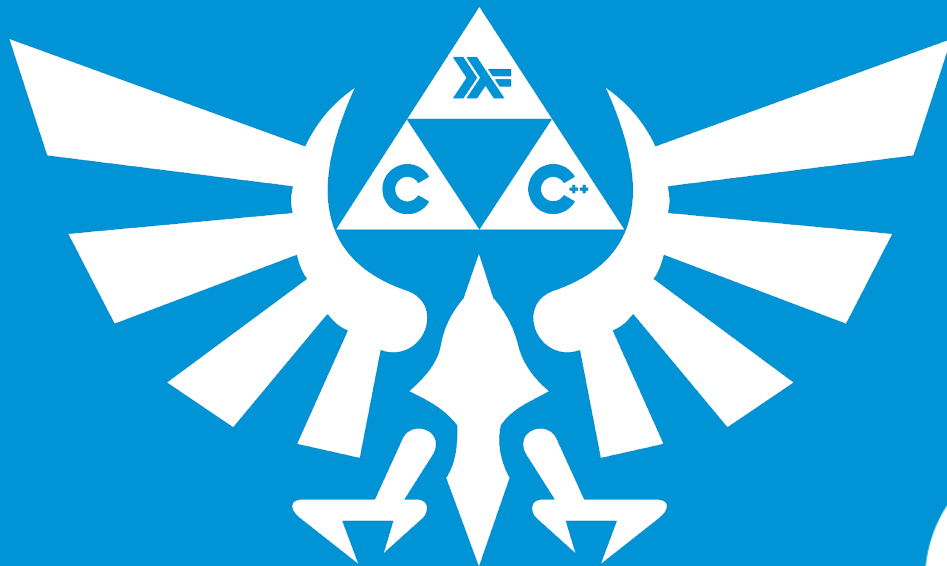




# DAY 01

A GENTLE INTRODUCTION TO FUNCTIONAL PROGRAMMING



# DAY 01



**binary name:** My.hs  
**language:** haskell



- ✓ 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 learn the basics of Haskell and functional programming in a bottom up manner.

You will have to implement from scratch a bunch of functions which are present in a way or another in the Haskell's standard library.

# Step 0 - Prerequisites

## GHC and GHCi

For this functional pool, you will need GHC, The Glorious Glasgow Haskell Compilation System, in version 8 or superior (check the version installed in the official docker image if in doubt)

You should have the commands **ghc** and **ghci** installed.

If you don't, now is a good time to install them.



Now is also a good time to configure your favorite editor (which is Emacs) to handle syntax highlighting and default indentation.



You have to respect the **Epitech Haskell coding style**, which you will find on the intranet here: <https://intra.epitech.eu/file/Public/technical-documentations/Haskell/>

## What to turn in

You have to turn in a file called `My.hs` which will contain several haskell functions.



Your source file must not contain a `main` and you can't import any module today.

We will test your functions using **GHCI**, so you should use it to be sure everything works fine.

**GHCI** is the interactive version of **GHC**, which is very useful to test your functions:

```
Terminal
~/B-PDG-300> ghci My.hs
Prelude> mySucc 41
42
```

## Authorized and forbidden functions and syntax

For today tasks, most of the functions of the standard library will be forbidden.

You are only allowed to use data types constructors and the **error** function from the standard library. You are allowed to use arithmetic operators (+, -, \*, ..., ==, <, ...). All other functions are forbidden (especially “++” and “!!”). You can't use list comprehensions.

You can use any function you have coded yourself to implement another function. Even if a task requires to implement a single function, this function may need auxiliary functions to be implemented, you are free to use as many functions as you want.

## Step 1 - Basic functions

Let's start with a bunch of very easy functions. Each task starts with the function's signature (prototype). All your functions must have an explicit signature and must abide to the signature specified in the subject.

### Task 01

```
mySucc :: Int -> Int
```

A function which takes one Int as argument and returns its successor.  
For example: **mySucc 41** returns **42**.

### Task 02

```
myIsNeg :: Int -> Bool
```

A function which takes one Int as argument and returns True if it's negative or False otherwise.

## Task 03

```
myAbs :: Int -> Int
```

A function which takes one Int as argument and returns it's absolute value.

## Task 04

```
myMin :: Int -> Int -> Int
```

A function which takes two Ints as arguments and returns the minimum of the two.

## Task 05

```
myMax :: Int -> Int -> Int
```

I'm sure you can guess what this function returns :) .

# Step 2 - Tuples

Tuples are data structures which allow to “pack” values in groups.



you can also “unpack” a tuple and bind its content to values thanks to pattern matching...

## Task 06

```
myTuple :: a -> b -> (a, b)
```

A function which takes two arguments and return a tuple of those.

## Task 07

```
myTriple :: a -> b -> c -> (a, b, c)
```

A function which takes three arguments and return a tuple of those.

## Task 08

```
myFst :: (a, b) -> a
```

A function which takes a tuple as argument and returns its first value.



Of course, the function **fst** is forbidden...

## Task 09

```
mySnd :: (a, b) -> b
```

A function which takes a tuple as argument and returns its second value.



Guess what? **snd** is forbidden too!

## Task 10

```
mySwap :: (a, b) -> (b, a)
```

A function which takes a tuple as argument and returns a new tuple, with it's two values swaped.

## Step 3 - Simple lists

Let's see a more advanced data structure: **lists**.

Like tuples, you can manipulate a list with its constructors and using pattern matching.

### Task 11

```
myHead :: [a] -> a
```

A function which takes a list as argument and returns its first value. If the list is empty, an exception is raised with the function **error**.

### Task 12

```
myTail :: [a] -> [a]
```

A function which takes a list as argument and returns a new list without its first element. If the list is empty, an exception is raised with the function **error**.

### Task 13

```
myLength :: [a] -> Int
```

A function which takes a list as argument and returns the number of elements in the list.

### Task 14

```
myNth :: [a] -> Int -> a
```

A function which takes a list and an Int (N) as argument and returns the element at index N in the list, or an error if the index is too large or negative.



This function can be written as : **myNth = (!!)** , but that's cheating, you have to implement it yourself.)

## Task 15

```
myTake :: Int -> [a] -> [a]
```

A function which takes an Int (N) and a list and returns a list with the Nth first elements of the list. If the list is too short the whole list is returned.

## Task 16

```
myDrop :: Int -> [a] -> [a]
```

A function which takes an Int (n) and a list and returns a list without the N first elements. If the list is too short an empty list is returned.

## Task 17

```
myAppend :: [a] -> [a] -> [a]
```

A function which takes two lists and returns a new list with the second list appened to the first one.



Of course, the function **++** is forbidden...

## Task 18

```
myReverse :: [a] -> [a]
```

A function which takes a list and returns a list with all its elements in reverse order.

## Task 19

```
myInit :: [a] -> [a]
```

A function which takes a list and returns a list with all its elements except the last one, or an error if the list is empty.



## Task 20

```
myLast :: [a] -> a
```

A function which takes a list and returns its last element, or an error if the list is empty.

## Task 21

```
myZip :: [a] -> [b] -> [(a, b)]
```

A function which takes two lists as arguments, and returns a list of tuples. The list produced is as long as the shortest list.

## Task 22

```
myUnzip :: [(a,b)] -> ([a], [b])
```

A function which takes a list of tuples, and return a tuple of lists.

# Step 4 - Advanced lists

Hold on tight, we're almost there for today. The functions in this section are very useful and are present in a form or another in all functional languages or libraries. Most of them have the characteristic that they are taking another function in argument to do their thing: they are higher-order functions.



They all exist in Haskell's standard library, playing with them is a good way to understand how they work and how to re-implement them...

## Task 23

```
myMap :: (a -> b) -> [a] -> [b]
```

A function which takes a function and a list, and apply this function to every element of the list.

## Task 24

```
myFilter :: (a -> Bool) -> [a] -> [a]
```

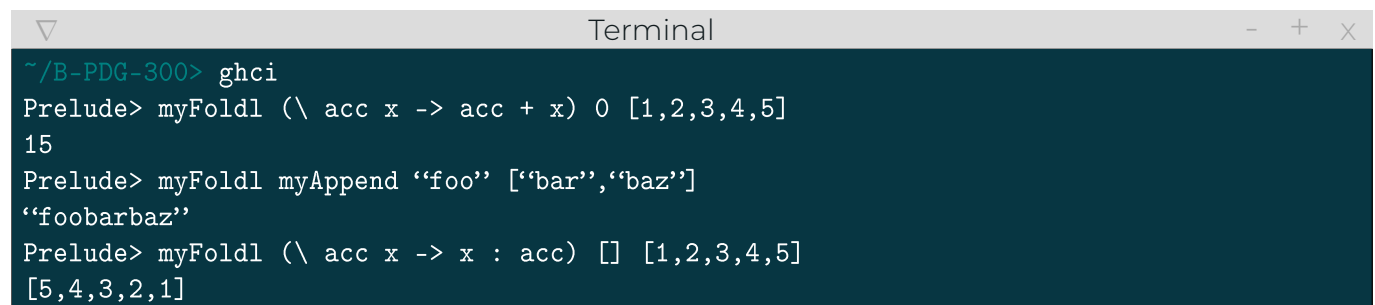
A function which takes a predicate (a function returning a boolean value) and a list, and returns a list of all the elements for which the predicate has returned True.

## Task 25

```
myFoldl :: (b -> a -> b) -> b -> [a] -> b
```

A function which takes a function, a starting value and a list as argument. It takes the second argument and the first item of the list and applies the function to them, then feeds the function with this result and the second argument and so on.

### Examples:

A terminal window titled "Terminal" with a dark background. It shows the execution of Haskell code in the ghci interpreter. The first command calculates the sum of a list using myFoldl, resulting in 15. The second command appends a list of strings using myFoldl and myAppend, resulting in "foobarbaz". The third command reverses a list using myFoldl, resulting in [5,4,3,2,1].

```
~/B-PDG-300> ghci
Prelude> myFoldl (\ acc x -> acc + x) 0 [1,2,3,4,5]
15
Prelude> myFoldl myAppend "foo" ["bar","baz"]
"foobarbaz"
Prelude> myFoldl (\ acc x -> x : acc) [] [1,2,3,4,5]
[5,4,3,2,1]
```

## Task 26

```
myFoldr :: (a -> b -> b) -> b -> [a] -> b
```

Like myFoldl, but from right to left.

## Task 27

```
myPartition :: (a -> Bool) -> [a] -> ([a], [a])
```

A function which takes a predicate and a list as argument, and returns a tuple of lists, with in the first list the elements for which the predicate returns true, and in the second list the other elements.



**partition** is not imported by default in ghci, you have to import it explicitly with **import Data.List**

## Task 28

```
myQuickSort :: (a -> a -> Bool) -> [a] -> [a]
```

A function which takes a predicate and a list as arguments, and returns the list sorted according to the predicate.

### Example

```
~/B-PDG-300> ghci
Prelude> myQuickSort (\ x y -> x < y) [9,1,8,2,7,3,6,4,5]
[1,2,3,4,5,6,7,8,9]
Prelude> myQuickSort (>) ["alice", "eve", "carl", "bob"]
["eve","carl","bob","alice"]
```



The functions you have already coded may be very helpful to implement this one...

## Step 5 - Conclusion

Now that you have implemented quite advanced functions, you may want to go back to the simpler ones and think if you can implement them in a better, shorter way (maybe using other functions).

Are you sure all your functions are behaving correctly? There's only one way to be sure: write unit tests for all of them!



There is the HUnit module for Haskell which has everything needed to write unit tests, go check it out.

{EPITECH}