

TP 1 : Differences

©2022 Ghiles Ziat
ghiles.ziat@epita.fr

EXERCISE I : First Program

Q1 – Create two programs *hello.hs* and *hello.lisp* which display on the standard output the string “Hello world”

Q2 – Compile and run both programs.

Note :

- To compile one (or more) Haskell source file(s), you have to run the command `ghc file.hs`. You can then run the executable produced by doing `./file`
- To compile and evaluate one (or more) lisp file(s), you must run the command `sbcl --script file.lisp`.
- In Haskell, a **'main'** needs to be defined as the entry point of the program.
- You can use the `putStrLn` function in Haskell and the `write-line` function in lisp for printing.

EXERCISE II : Recursion

We want to have a function `sumDigits` which calculates the sum of the digits of a number. For example, `sumDigits 1234` should return 10 (4+3+2+1). An *imperative* way of **doing** this calculation is given by :

```
sumDigits(n){  
  sum = 0;  
  while (n != 0) do  
    sum = sum + (n % 10);  
    n=n/10;  
  done;  
  return sum  
}
```

A *functional* way of **defining** this calculation is to say that the sum of the digits of an integer n is equal to n if $n < 10$, otherwise, it is equal to the units digit of n to which we add the sum of the remaining digits of n (Note that the definition is naturally recursive).

Q1 – Write a recursive function with this behavior. (you can use the functions `div` and `rem` in Haskell, as well as the `floor` and `rem` functions in lisp to compute the Euclidean division and the remainder)

Q2 – A number is divisible by 3 if

- it's less than 10 and it's 0, 3, 6 or 9
- it's greater than or equal to 10, and the sum of its digits is itself divisible by 3

Define a recursive function with this behaviour.

EXERCISE III : Logic

We recall here the truth tables of certain Boolean operators classics (and, or and not). True (resp. false) is identified by \top (resp. \perp) :

\wedge	\top	\perp
\top	\top	\perp
\perp	\perp	\perp

\vee	\top	\perp
\top	\top	\top
\perp	\top	\perp

\neg	\top	\perp
\top	\perp	\top

We also recall that the implication (\Rightarrow), the equivalence (\Leftrightarrow) and the xor (\oplus) can be defined by :

- $A \Rightarrow B \equiv (\neg A) \vee B$
- $A \Leftrightarrow B \equiv A \Rightarrow B \wedge B \Rightarrow A$
- $A \oplus B \equiv (A \vee B) \wedge \neg(A \wedge B)$

Q1 – Define the operators `andOp` and `orOp`, from type `bool -> bool -> bool` using a conditional.

Q2 – Define the operators `imply`, `equiv` and `xor` respectively performing the operations (\Rightarrow , \Leftrightarrow , \oplus).

Q3 – Compare the result of the calls in Haskell and in Lisp (the calls are written in Haskell, you will use the Lispian value `t` instead of `True`) :

- `orOp True (error "failure")`
- `orOp (error "failure") True`

What do you notice ?

EXERCISE IV : Arithmetic

The Collatz conjecture¹ is an unsolved problem in mathematics that asks if the repetition of two simple arithmetic operations will eventually transform every positive integer into 1.

Q1 – Consider the following operation on an integer strictly positive : if it is even, we divide it by 2 ; if it is odd, we multiply it by 3 and add 1. Write a function with this behavior.

1. https://en.wikipedia.org/wiki/Collatz_conjecture

Q2 – By repeating the operation, we obtain a sequence of integers strictly positive, each of which depends only on its predecessor. Write a function that builds this list until it hits the value 1.

Q3 – Test your function and display its result on input 10.

EXERCISE V : Lists

Merge sort² is a stable comparison sorting algorithm that consists in dividing a list into sublists, sorting these sublists separately and merge their result.

Q1 – Define a function that splits a list into two sublists of equal size (up to one). You will probably need to store the result of a recursive call in a variable³. For this, you can use a `let` binding that associates values with names using either the Lispian syntax `(let ((a (+ 20 1)) (b 2)) (* a b))` or the Haskellian `let a = 20 + 1 in let b = 2 in a*b`, which declares two variables “a” and “b” and uses them in the expression “a*b”

Q2 – Define a function that given two sorted lists, constructs a sorted list containing the elements resulting from these two lists.

Q3 – Implement the following sort function :

- if the list is empty or contains only one item, it is already sorted.
- otherwise,
 - divide the list into two sublists of the same size (up to 1),
 - sort the two sublists separately,
 - merge the sorted sublists while maintaining the order.

2. https://en.wikipedia.org/wiki/Merge_sort

3. Note that in functional programming, the term variable is used in the mathematical sense of the term. Variable being immutable, their contents do not ... vary