Declaratieve Talen

Haskell 1

Introduction

Starting & using GHCi

The interactive Haskell environment GHCi can be started by executing the `ghci` command in a terminal window. Haskell programs are stored in files with the `.hs` extension. Familiarise yourself with the following GHCi *directives*.

- To load a program type `:l filename.hs` in GHCi.
- To reload the last file, type `:r` in GHCi.
- To find out the type of an expression, type `:t expr` with `expr` being an expression, e.g. `not True`.
- To find out more information about a type, use `:i type` with `type` being a type, e.g. `Bool` or `[]`.
- To exit GHCi, type `:q`.
- Haskell is indentation sensitive. Use spaces (not tabs!) to indent your program.

HLint

HLint is a tool that helps to improve your code style. It suggests changes for making your code better by indicating redundant brackets, better usage of built-in functions, eta reductions, ... It is a good idea to run HLint on your file after you are done with an exercise. Remember that we pay attention to style when correcting your exam!

```
$ hlint example.hs
example.hs:10:15: Error: Redundant bracket
Found:
  (x)
Why not:
  x

1 suggestion
```

1 Hors d'Oeuvres

Implement the functions below. At the end of this exercise you'll find some examples to verify the correctness of your solutions. Note that many of these function are available in the standard library, you are not allowed to use them (yet)! When writing a recursive function involving lists, put some thought into choosing the right base case.

- Write a function `myLast :: [Int] -> Int` which returns the last element of a list.
- Write a function `myRepeat :: Int -> Int -> [Int]` such that `myRepeat n x` returns a list with `n` times the number `x`.
- Write a function `flatten :: [[Int]] -> [Int]` which converts a list of lists to a single list.
- Write a function `range :: Int -> Int -> [Int]` which returns a list of the consecutive numbers between the two given numbers, both numbers included.
- Write a function `removeMultiples :: Int -> [Int] -> [Int]` which removes all multiples of a number from the list. Use `n `mod` d` or `mod n d`.

Examples

Hint: You can use the `it` function (without any arguments) to recall the result of the last execution in the GHCi environment.

```
Main> myLast [1,2,3,4,5]
5

Main> myRepeat it it
[5,5,5,5,5]

Main> flatten [[1,2],[3,4],[5,6]]
[1,2,3,4,5,6]

Main> flatten []
[]

Main> range 1 10
[1,2,3,4,5,6,7,8,9,10]

Main> range (-10) (-5)
[-10,-9,-8,-7,-6,-5]

Main> removeMultiples 2 (range 1 10)
```

```
[1,3,5,7,9]

Main> removeMultiples 5 []
[]
```

Now implement these functions again by reusing as many of the standard library (Prelude) functions as possible. An overview of the predefined library modules can be found on `http://downloads.haskell.org/~ghc/7.6.3/docs/html/libraries/base/`. For this exercise, we recommend you to have a look in the `Data.List` module.

2 Folds

2.1 Your own implementation

Implement the functions below. At the end of this exercise you'll find some examples to verify the correctness of your solutions. Just as in the previous exercise, you are not allowed to use the versions of these function present in the standard library.

- Write a function `mySum :: [Int] -> Int` which calculates the sum of the numbers in a list.
- Write a function `myProduct :: [Int] -> Int`, which, similarly to the previous function, calculates the product of the numbers in a list.
- After writing these two functions, you should have noticed they look very similar, and only differ in a few places. Write a function `fold` which has the common characteristics of `mySum` and `myProduct`, and accepts the different characteristics as parameters. Using this `fold` function, `mySum` and `myProduct` could be implemented as follows:

  ```
  mySum     = fold (+) 0
  myProduct = fold (*) 1
  ```

  Don't forget the type signature of `fold`.
- Write the most general (polymorphic) type signature of `fold` as possible.

Hint: Use `:t` to find out the type of a function in GHCi.

Examples

```
Main> mySum (removeMultiples 3 (range 1 10))
37

Main> mySum []
0

Main> myProduct [1,2,3]
6

Main> myProduct []
1

Main> fold (+) 0  [1,2,3,4]
10

Main> fold (*) 1  [1,2,3,4]
24
```

2.2 Associativity

Your `fold` function implicitly puts the operator between the elements of the list. So:

```
fold (+) 0  [1,2,3,4] = 0+1+2+3+4
```

The `(+)`-operator is commutative, so the order of execution is not relevant. However, what should happen when we execute `fold (-) 0 [1,2,3,4]`? Here, there are two options, either we associate to the left: $((((0-1)-)2-)3-4) = -10$ or we associate to the right: $(1-(2-(3-(4-0)))) = -2$.

- Test whether your implementation associates to the right or to the left.
- `f = fold (:)  []` is a valid function, what does this function do? If we associate differently, does the behaviour of the function change?

In Haskell, 4 fold functions are available by default:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldr :: (a -> b -> b) -> b -> [a] -> b

foldl1 :: (a -> a -> a) -> [a] -> a
foldr1 :: (a -> a -> a) -> [a] -> a
```

The last 2 functions are versions of the functions above where the first element of the list is taken as starting element. For example `foldl1` can be implemented as:

```
foldl1 f (x:xs) = foldl f x xs
```

- Write a function `readInBase :: Int -> [Int] -> Int` using `fold` that takes a list of digits in base B and outputs the number in base 10.

```
Main> readInBase 2 [1,0]
2

Main> readInBase 6 [1,3,0]
54
```

2.3 Map

In Prolog exercise session 3, you implemented the `map/3` predicate. Now implement this function in Haskell, once with and once without using `fold`.

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

This function is available in the Haskell Prelude under the name `map`.

3 Unpair

Implement the function `unpair` which converts a list of pairs (or tuples) into a pair of lists. Make sure the function has the most general (polymorphic) type.

Examples

```
Main> unpair [(1,2),(3,4),(5,6)]
([1,3,5],[2,4,6])

Main> unpair [(1,'a'),(2,'b'),(3,'c')]
([1,2,3],"abc")
```

If you have written the function using list comprehensions, try writing it without, or if you haven't, try writing it with list comprehensions.

4 Transpose

Write a polymorphic function `transpose` which calculates the transpose of a matrix. Put some thought into choosing the right base case.

Examples

```
Main> transpose [[1,2,3],[4,5,6]]
[[1,4],[2,5],[3,6]]

Main> transpose [[1,1],[2,2],[3,3],[4,4]]
[[1,2,3,4],[1,2,3,4]]
```

5 Prime numbers

Write a function `sieve :: Int -> [Int]` which returns all prime numbers smaller than the given number. Implement your function following Eratosthenes' algorithm.[1] You can actually stop filtering the moment you've reached the square root of the input number. Ignore this optimisation in your first implementation.

Examples

```
Main> sieve 20
[2,3,5,7,11,13,17,19]

Main> sieve 49
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

[1] See `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`.

Extension

Haskell provides the `sqrt` function to find the square root of a number. However, this function requires an argument of type `Double`, whereas your argument is an `Int`. To convert this `Int` to a `Double`, use the `fromIntegral` function. The result of `sqrt` will also be a `Double`. To convert this `Double` back to an `Int`, use the `floor` function.[2] Because these functions work with type classes, we give you versions of these functions with the right types: `sqrtMono`, `i2d`, and `floorMono`. Don't forget to add them to your file!

```
sqrtMono :: Double -> Double
sqrtMono = sqrt

i2d :: Int -> Double
i2d = fromIntegral

floorMono :: Double -> Int
floorMono = floor
```

Using these functions, try to write the function `floorSquare` which *floors* the square root of the given `Int` argument. Use this `floorSquare` function to make `sieve` more efficient.

[2] See `http://en.wikipedia.org/wiki/Floor_and_ceiling_functions#Examples` for more information about the `floor` function.