Declaratieve Talen

Haskell 1

1 List Operations

Implement the functions below. Note that many of these functions are available in the standard library,[1] but the goal of this exercise is to practice by implementing them from scratch. When writing a recursive function involving lists, put some thought into choosing the right base case.

Recall that the syntax of pattern-matching on a list is as follows (where `x` is the head of the list and `xs` is the tail):

```
function :: [...] -> ...
function []     = ...
function (x:xs) = ...
```

Note that HLint (on E-Systant) may generate warnings, you can ignore these.

- Write a function `myProduct :: [Integer] -> Integer`, which takes a list of integers and computes their product.
  Note that we use `Integer` here instead of `Int`. The former can represent numbers of arbitrary size, whereas the latter will overflow.

  ```
  Main> myProduct [1,2,3]
  6

  Main> myProduct []
  1

  Main> myProduct [-2,3,-4,5,-6]
  -720
  ```

- Write a function `insert :: Int -> [Int] -> [Int]`, which takes an integer and a list of integers and inserts the integer into the list at the first position where it is less than or equal to the next element.

[1] For example, see module `Data.List`, which can be found at `http://downloads.haskell.org/~ghc/7.6.3/docs/html/libraries/base/`.

```
Main> insert 0 [1,2,3]
[0,1,2,3]

Main> insert 2 [1,0,3]
[1,0,2,3]

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

- Write a function `myLast :: [Int] -> Int` which returns the last element of a list. Assume that the input lists are non-empty[2]

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

2 Rock - Paper - Scissors

In the Rock, Paper and Scissors game, two players choose one of the following gestures after counting to three:

- **A clenched fist** which represents a rock.
- **A flat hand** representing a piece of paper.
- **Index and middle finger extended** which represents a pair of scissors.

The result of a round is decided this way:

- **Rock defeats scissors**, because a rock will blunt a pair of scissors.
- **Paper defeats rock**, because a paper can wrap up a rock.
- **Scissors defeat paper**, because scissors cut paper.
- **Otherwise**, the players chose the same gesture and it's **a draw**.

2.1 Moves

- Define a datatype `Move` with three choices `Rock`, `Paper` and `Scissors` that represent the valid moves. Note that the "`deriving (Eq, Show)`" should not be removed from the declaration otherwise the testing framework won't work.
- Write a function `beat :: Move -> Move` such that `beat m` is the move that beats move `m`.
- Write a function `lose :: Move -> Move` such that `lose m` is the move that will lose against move `m`.

[2]You may return an error using the `error` function in case the list is empty.

2.2 Playing the Game

- Define a datatype `Result` that represents the outcome of a round of Rock - Paper - Scissors. As we explained above, a player can either `Win`, `Lose` or the game may end up in a `Draw`. Like before, the "`deriving (Eq, Show)`" should not be removed from the declaration otherwise the testing framework won't work.

- Write a function `outcome :: Move -> Move -> Result` that takes as arguments two moves (the first argument is the move of the first player and the second is the move of the second player) and calculates the outcome for the **first player**.

3 Lists, Ranges and List Comprehensions

In contrast to the previous exercise on lists, try to implement these functions using list comprehensions (e.g., `[ a+1 | a <- as ]`) as well as list ranges: `[1..n]`, `[1,5,..,n]`. **Do not use explicit recursion!**

Note that many of these functions are available in the standard library,[3] but the goal of this exercise is to practice by implementing them from scratch.

- In mathematics, the factorial of a non-negative integer number `n` is defined recursively as follows:

$$factorial\ (n) = \begin{cases} 1 & \text{, if } n = 0 \\ n * factorial\ (n-1) & \text{, if } n > 0 \end{cases}$$

  Alternatively, we can also express it as: $factorial\ (n) = 1*2*\cdots*(n-1)*n$

  Write a function `factorial :: Integer -> Integer`, such that `factorial n` is the factorial of `n`. If `n` is a negative number, `factorial n` should result in 1. **Hint:** You can use the standard library function `product`, to compute the product of every element in a list.

```
Main> factorial 5
120

Main> factorial 0
1

Main> factorial (-10)
1
```

- Write a function `myRepeat :: Int -> Int -> [Int]` such that `myRepeat n x` returns a list with `n` times the number `x`. If `n` is less than zero return the empty list.

[3] For example, see module `Data.List`, which can be found on `http://downloads.haskell.org/~ghc/7.6.3/docs/html/libraries/base/`.

```
Main> myRepeat 4 5
[5,5,5,5]

Main> myRepeat (-1) 5
[]

Main> myRepeat 0 5
[]
```

- Write a function `flatten :: [[Int]] -> [Int]` which converts a list of lists to a single list.

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

Main> flatten []
[]
```

- Write a function `range :: Int -> Int -> [Int]` which returns a list of the consecutive numbers between the two given numbers, both numbers included. If the first number is greater than the second you should return the empty list.

```
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> range 10 1
[]
```

- Write a function `sumInts :: Int -> Int -> Int`, which takes an integer `low` and an integer `high` and computes the sum:

  $$\texttt{sumInts low high} = \texttt{low} + (\texttt{low} + 1) + (\texttt{low} + 2) + \cdots + (\texttt{high} - 1) + \texttt{high}$$

  If `low` > `high` then the sum should be zero. **Hint:** You can use the standard library function **sum**, to compute the sum of every element in a list.

```
Main> sumInts 3 5
12

Main> sumInts 5 3
```

```
0

Main> sumInts 5 5
5
```

- 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`. Assume that the first argument will never be zero.

```
Main> removeMultiples 2 (range 1 10)
[1,3,5,7,9]

Main> removeMultiples 5 []
[]
```

4 List Comprehensions

Rewrite the following functions using *list comprehensions*:

- `mapLC :: (a -> b) -> [a] -> [b]`, which takes a function and a list and applies the function to each element in the list (corresponds to the predefined function `map`).
- `filterLC :: (a -> Bool) -> [a] -> [a]`, which takes a predicate and a list and retains only the elements in the given list for which applying the predicate function returns `True`. The function should preserve the original order of the elements (corresponds to the predefined function `filter`).

5 Folds

I Did It My Way

Implement the functions below. Just as in the previous exercise, avoid using the versions of these functions in the standard library and do it your way.

- Write a function `mySum :: [Integer] -> Integer` which calculates the sum of the numbers in a list.
- Write a function `myProduct :: [Integer] -> Integer`, which, similarly to the previous function, calculates the product of the numbers in a list.[4]
- After writing these two functions, you should have noticed they look very similar, and only differ in a few places. Write a function `foldInts` which has the common characteristics of `mySum` and `myProduct`, and accepts the different characteristics as parameters.

[4] We use `Integer` (arbitrary-precision-integers instead of plain `Int`, since the size of the product can rise reapidly.

```
foldInts :: (Integer -> Integer -> Integer)
         -> Integer -> [Integer] -> Integer
```

Using this `foldInts` function, `mySum` and `myProduct` could be implemented as follows:

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

Examples

```
Main> mySum [1,4,7,10]
22

Main> mySum []
0

Main> myProduct [1,2,3]
6

Main> myProduct []
1

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

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

Associativity and Folds

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

```
foldInts (+) 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 `foldInts (-) 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: $(1-(2-(3-(4-0))))=-2$. Since this is a very common operation in functional programming, the `Prelude` predefines the following 2 functions:

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

- Implement the two functions yourself:

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

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

```
Main> myFoldl (+) 0 [1,2,3]
6

Main> myFoldl (-) 0 [1,2,3]
-6

Main> myFoldl (++) "" ["Hello", " ", "World"]
"Hello World"

Main> myFoldr (+) 0 [1,2,3]
6

Main> myFoldr (-) 0 [1,2,3]
2

Main> myFoldr (:) [] [1,2,3]
[1,2,3]

Main> myFoldr (++) "" ["Hello", " ", "World"]
"Hello World"

Main> readInBase 2 [1,0]
2

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

Hint: You can write a number in base b, given as a list of digits $d_n d_{n-1} ... d_1 d_0$ (with d_n the most significant digit), as a polynomial like this

$$b^n d_n + b^{n-1} d_{n-1} + ... + bd_1 + d_0.$$

For example, the number 130 in base 6 can be written as:

$$1 * 6^2 + 3 * 6^1 + 0 * 6^0$$

Use Horner's method in combination with a fold function. Horner's method calculates polynomials using the following scheme:

$$b^n d_n + b^{n-1} d_{n-1} + ... + bd_1 + d_0 = ((...(d_n b + d_{n-1})b + ...)b + d_1)b + d_0$$

For the number 130 in base 6, this becomes

$$(1 * 6 + 3) * 6 + 0$$

Map

Function `map` is also a common function in Haskell (it is available in the Haskell Prelude under the name `map`). It takes a function and a list of elements and applies the function to all elements:

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

- Implement function `myMap :: (a -> b) -> [a] -> [b]`, that is your own implementation of `map`. Do not use folds for this implementation.
- Implement function `myMapF :: (a -> b) -> [a] -> [b]`, this time using a fold function.

Examples

```
Main> myMap (+1) [1,2,3,4]
[2,3,4,5]

Main> myMap not [True, False]
[False, True]

Main> myMap not []
[]

Main> myMapF (+1) [1,2,3,4]
[2,3,4,5]

Main> myMapF not [True, False]
[False, True]

Main> myMapF not []
[]
```

6 Function Chaining

It is very common in a program to apply multiple functions one after another. For example, to apply `f`, `g` and `h` to a value `x`, one could write:

```
myFunc x = h (g (f x))
```

These parentheses become cumbersome very quickly. The solution to this, is to introduce a higher-order function '`.`' that "chains" two functions after each other:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Using this operator, we can now write `myFunc` much more elegantly:

```
myFunc x = (h . g . f) x
```

The '`.`' function is read as "after", which means that the right-hand side is read as "`h` after `g` after `f` applied to `x`". This means that we apply `f` to `x`, apply `g` to the result, and finally apply `h` to this result. Note that the parentheses here are necessary, as `h . g . f x` is actually parsed as `h . g . (f x)`, which means something completely different.

Intermediate Haskell programmers don't like to write parentheses, so they have come up with a way to omit these parentheses. The solution is the `$`-function:

```
($) :: (a -> b) -> a -> b
```

This function seems completely pointless as it just represents function application. However, due to how it is parsed, this operator can separate the function and argument without the need for parentheses. We can now rewrite `myFunc` to:

```
myFunc x = h . g . f $ x
```

Notice that the function can be defined by just chaining `f`, `g` and `h` together. The argument `x` is now redundant. Thus, the function `myFunc` can be defined as:

```
myFunc = h . g . f
```

That is, by evaluating `h` after `g` after `f`.

- Write a function `applyAll :: [a -> a] -> a -> a` which applies a list of functions to a value, one after the other.

  ```
  Main> applyAll [(+ 2),(* 2)] 5
  12

  Main> applyAll [(: []). sum, filter odd] [1..8]
  [16]
  ```

- Write a function `applyTimes :: Int -> (a -> a) -> a -> a`, which applies a function a given number of times to a value. When this number is ≤ 0, the function should be applied zero times, i.e. leave its argument unchanged. You should use only two explicit arguments in your code. Hint: you can use the `applyAll` function in your definition.

  ```
  Main> applyTimes 5 (+ 1) 0
  5

  Main> applyTimes 4 (++ "i") "W"
  "Wiiii"

  Main> applyTimes 0 (error "Error!") 3.14
  3.14
  ```

- As a variation on this theme, write a function `applyMultipleFuncs :: a -> [a -> b] -> [b]`, which takes an argument and a list of functions, and applies these functions to the given argument.

```
Main > applyMultipleFuncs 2 [(*2), (*3), (+6)]
[4,6,8]
```

7 EXTRA: Caesar Cipher

Note: For this assignment you may find some functions from the `Data.Char` library useful.

Julius Caesar before sending his messages often encoded them, by replacing each letter by the letter three places further down in the alphabet (wrapping around at the end of the alphabet). For example, the string

```
"haskell is fun"
```

would be encoded as

```
"kdvnhoo lv ixq"
```

In general, we can do even more than Caesar did, and encode our strings using any integer between 1 and 25 (since the alphabet has 26 letters), having 25 different ways of encoding a string. For example, with a shift factor of 10, the original string would be encoded as:

```
"rkcuovv sc pex"
```

Encoding and Decoding

For simplicity, in this exercise we will only encode lowercase letters, leaving all other letters unchanged. Function `let2int` converts a lowercase letter between `'a'` and `'z'` to an integer from `0` to `25`, and function `int2let` does the inverse:

```
let2int :: Char -> Int
let2int c = ord c - ord 'a'

int2let :: Int -> Char
int2let n = chr (ord 'a' + n)
```

(The library functions `ord :: Char -> Int` and `chr :: Int -> Char` convert a character to its unicode representation and vice-versa) For example:

```
Main> let2int 'a'
0

Main> int2let 0
'a'
```

- Define function `shift :: Int -> Char -> Char`, which applies a shift factor to a lowercase letter and leaves any other character unchanged (**Hint:** Use the above functions, as well as function `mod` to ensure that the resulting integer representation does not exceed 26).

- Define function `encode :: Int -> String -> String` by means of function `shift`, which, given a shift factor, encodes a whole string.

Examples

```
Main> shift 3 'a'
'd'

Main> shift 3 'z'
'c'

Main> shift (-3) 'c'
'z'

Main> shift 3 ' '
' '

Main> encode 3 "haskell is fun"
"kdvnhoo lv ixq"

Main> encode (-3) "kdvnhoo lv ixq"
"haskell is fun"
```

Note that there is no need for a "decode" function, since if a string is encoded using a shift factor `n`, we can always take it back be re-encoding it using `(-n)` as a shift factor.

Frequency Tables

The key to crack the Caesar cipher is the observation that some letters of the English alphabet appear more often than others. In fact, by analyzing a large volume of text, one can derive the following table of approximate percentage frequencies of the 26 letters of the alphabet:

```
table :: [Float]
table = [ 8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0, 6.1, 7.0, 0.2, 0.8, 4.0, 2.4
        , 6.7, 7.5, 1.9, 0.1, 6.0, 6.3, 9.1, 2.8, 1.0, 2.4, 0.2, 2.0, 0.1 ]
```

For example, letter `'e'` occurs most often, with a frequency of 12.7%, while `'q'` and `'z'` appear least often, with a frequency of 0.1% each.

- Define function `percent :: Int -> Int -> Float` which computes the percentage of an integer with respect to another (**Hint:** Use library function `fromIntegral :: (Integral a, Num b) => a -> b` to convert the arguments to `Float` before dividing them). For example:

```
Main> percent 6 12
50.0

Main> percent 3 15
20.0
```

- Define function `freqs :: String -> [Float]` which computes the frequencies of the 26 letters of the alphabet for a given string. Assume that the given string will contain at least one lowercase letter. For example:

```
Main> freqs "abbcccdddeedeee"
[ 6.7, 13.3, 20.0, 26.7, 33.3, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ]
```

  That is, letter `'a'` appears with frequency 6.7, letter `'b'` with frequency 13.3 and so on.

Cracking The Cipher

Now that we have laid the foundations, it is time to crack Caesar's Cipher.

A standard method for comparing a list of observed frequencies `o` with a list of expected frequencies `e` is the *chi-square* statistic, defined as follows:

$$\texttt{chisqr o e} = \sum_{i=0}^{n-1} \frac{(\texttt{o}_i - \texttt{e}_i)^2}{\texttt{e}_i}$$

The details of the chi-square method are not important to us, only the fact that the smaller the result of `chisqr o e`, the better the match between frequency tables `o` and `e`.

- Implement function `chisqr :: [Float] -> [Float] -> Float`. **Hint:** this exercise can be easily solved using the `zip` function and list comprehensions.

- Implement function `rotate :: Int -> [a] -> [a]`, which rotates the elements of a list a given number of times to the left. For example:

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

  You can assume that the integer argument is always between 0 and the length of the list. **Hint:** Use functions `take` and `drop` to implement this exercise.

Now, if we are given an encoded string but not the shift factor used for the encoding, we can find the shift factor as follows:

1. We produce the frequency table of the encoded string
2. We calculate the chi-square statistic for each possible rotation of this table with respect to the expected frequencies (value `table`)
3. The position of the minimum chi-square value is the most probable shift-factor used to encode the string.

For example, if `table' = freqs "kdvnhoo lv ixq"`, then

```
[ chisqr (rotate n table') table | n <- [0..25] ]
```

gives the result

$$[1408.8, 640.3, 612.4, 202.6, 1439.8, 4247.2, 651.3, \cdots, 626.7]$$

, in which the minimum value is 202.6, appearing in position 3 in this list (counting from 0). Hence, we can conclude that the shift factor used to encode the string was 3, and to retrieve the original string, we just need to encode it again using -3.

- Define function `crack :: String -> String` which takes an encoded string and, using the above method, computes the original string. **Hint:** In addition to all the functions you have already defined, you will also find functions `minimum :: Ord a => [a] -> a` and `elemIndex :: Eq a => a -> [a] -> Maybe Int` useful for solving this exercise (function `elemIndex` is defined in the `Data.List` library).

Examples

```
Main> crack "kdvnhoo lv ixq"
"haskell is fun"

Main> crack "vscd mywzboroxcsyxc kbo ecopev"
"list comprehensions are useful"
```

Note that cracking is not always accurate, especially in cases where the encoded word is too short, or has an unusual distribution of letters.

```
Main> crack (encode 3 "haskell")
"piasmtt"

Main> crack (encode 3 "boxing wizards jump quickly")
"wjsdib rduvmyn ephk lpdxfgt"
```

8 Extra: Approximating π

The goal of this exercise is to compute π using known mathematical formulas. Use both types of list ranges (`[1..n]`, `[1,5,..,n]`) and list comprehensions `[ a+1 | a <- as ]` to solve this exercise. **Do not use explicit recursion!**

Some Basic Functions

The following functions will come in handy when implementing the π-approximations, so it would be better to implement them first:

- Function `sumf :: [Float] -> Float`, which computes the sum of the elements of a list.[5]
- Function `productf :: [Float] -> Float`, which computes the product of the elements of a list.

Examples

```
Main> sumf []
0.0

Main> sumf [1, 5.0, 6.32]
12.32

Main> productf []
1.0

Main> productf [1, 5.0, 6.32]
31.6
```

Approximation 1

One way to compute π analytically is by using the following formula:

$$\pi(n) \approx 8 * \left(\frac{1}{1*3} + \frac{1}{5*7} + \frac{1}{9*11} + \ldots + \frac{1}{(4n+1)*(4n+3)} \right)$$

The higher the value of n, the closer the value of $\pi(n)$ to the actual value of π:

$$\begin{array}{lcll}
\pi(0) & = & 8 * \left(\frac{1}{1*3}\right) & = \quad 2.6666667 \\
\pi(1) & = & 8 * \left(\frac{1}{1*3} + \frac{1}{5*7}\right) & = \quad 2.8952382 \\
\pi(2) & = & 8 * \left(\frac{1}{1*3} + \frac{1}{5*7} + \frac{1}{9*11}\right) & = \quad 2.9760463 \\
 & & \ldots & \\
\pi(100) & = & 8 * \left(\frac{1}{1*3} + \frac{1}{5*7} + \ldots + \frac{1}{401*403}\right) & = \quad 3.1366422 \\
 & & \ldots &
\end{array}$$

[5] `Float` is a single-precision floating-point number, `Double` is a double-precision floating-point number.

- Implement function `piSum :: Float -> Float` that, given an n, approximates π using the above formula. You can assume that n is a natural number, e.g., 0.0, 1.0, 2.0, ...

Approximation 2

Similarly, we can also implement π using the following formula:

$$\pi(n) \approx 4 * \frac{2*4}{3*3} * \frac{4*6}{5*5} * \frac{6*8}{7*7} * \frac{8*10}{9*9} * \ldots * \frac{(2n+2)*(2n+4)}{(2n+3)^2}$$

- Implement function `piProd :: Float -> Float` that, given an n ,approximates π using the above formula. You can assume that n is a natural number, e.g., 0.0, 1.0, 2.0, ...

9 Extra: Prime numbers

Write a function `sieve :: Int -> [Int]` which returns all prime numbers smaller than the given number. Implement your function following Eratosthenes' algorithm.[6] You can actually stop filtering the moment you've reached the square root of the input number. Ignore this optimization 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]
```

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.[7] Because these functions work with type classes, we give you versions of these functions with the right types: `sqrtMono`, `i2d`, and `floorMono`.

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

i2d :: Int -> Double
```

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

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

```
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.