1. Introduction

The purpose of this assignment is to get started writing Haskell functions.

- For the purposes of this assignment, don’t use any of the higher-order built-in functions such as map, foldr, etc. — I want you to write all functions “from scratch”! You may use the ++-function for list concatenation and the reverse function for list reversal.

- Unless otherwise specified, you should use the guard syntax rather than the if-then-else syntax when you define recursive functions.

HINT: It’s never wrong to introduce an auxiliary function when it makes the program easier to write or prettier to read! In fact, breaking up a larger function in two or more smaller ones is encouraged.

2. Non-Recursive Functions

1. Define a function doublestring s which takes a string argument s and returns a new string consisting of two copies of s:

doublestring :: String -> String
doublestring s = ...

> doublestring ""
"
> doublestring "hello"
"hellohello"

2. Write a function charToString a which returns the one-character string consisting only of the character a:

charToString :: Char -> String
charToString c = ...

> charToString 'A'
"A"
3. Using the formula

\[ V = 2\pi rh + 2\pi r^2 \]

define a function \( \text{cylinderSurfaceArea} \ (r,h) \) which computes the surface area of a cylinder of height \( h \) and radius \( r \), rounded down to the nearest integer:

\[
cylinderSurfaceArea :: (Float,Float) -> Int
cylinderSurfaceArea (r,h) = ... \\
\]

\[ \text{cylinderSurfaceArea} \ (2.0,5.0) = 87 \]

Use the \text{floor} function to round down and the \text{pi} constant function to approximate \( \pi \):

\[ \text{floor} \ (5.5) = 5 \]
\[ \text{pi} = 3.14159 \]

4. Use \text{head} and \text{tail} to write a non-recursive function \( \text{third} \ \text{xs} \) which returns the third element of a list of \text{Ints}:

\[
third :: \text{[Int]} -> \text{Int} \\
third \ \text{xs} = ...
\\
> \text{third} \ [1,2,3,4,5] = 3 \\

You don’t have to check that the list contains at least three elements.

5. Use \text{head} and \text{tail} to write a non-recursive function \( \text{yahtzee} \ \text{xs} \) which takes a list of five \text{Ints} (between 1 and 6) as argument (the result of rolling five dice) and returns \text{True} if all numbers are the same, and \text{False} otherwise.

\[
yahhtzee :: \text{[Int]} -> \text{Bool} \\
yahhtzee \ \text{xs} = ...
\\
> \text{yahtzee} \ [1,1,1,1,1] = \text{True} \\
> \text{yahtzee} \ [1,1,1,1,2] = \text{False} \\
> \text{yahtzee} \ [6,6,6,6,6] = \text{True} \\

You don’t have to check the input for correctness — we assume that there are exactly five elements in the list and that the numbers are between 1 and 6.
3. Recursive Functions

1. Write a recursive function \texttt{msum n} that returns the sum of the integers $0 + 1 + 2 + 3 + \ldots + n$, where $n \geq 0$:

\texttt{msum} :: \texttt{Int} $\rightarrow$ \texttt{Int}
\texttt{msum n} = ...

\texttt{> msum 0} \\
\texttt{0}
\texttt{> msum 1} \\
\texttt{1}
\texttt{> msum 5} \\
\texttt{15}

\texttt{msum} should make use of a conditional expression (if-then-else syntax).

2. Write a recursive function \texttt{gsum n} that returns the sum of the integers $0 + 1 + 2 + 3 + \ldots + n$, where $n \geq 0$:

\texttt{gsum} :: \texttt{Int} $\rightarrow$ \texttt{Int}
\texttt{gsum} ...

\texttt{gsum} should make use of guards.

3. Define a recursive function \texttt{copystring (s,n)} which returns a string consisting of $n$ copies of the string $s$:

\texttt{copystring} :: (\texttt{String},\texttt{Int}) $\rightarrow$ \texttt{String}
\texttt{copystring (s,n)} ...

Your function should have the following behavior:

\texttt{> copystring ("hello",-1)} \\
\texttt{""}
\texttt{> copystring ("hello",0)} \\
\texttt{""}
\texttt{> copystring ("hello", 1)} \\
\texttt{"hello"}
\texttt{> copystring ("hello",2)} \\
\texttt{"hellohello"}
\texttt{> copystring ("hello",10)} \\
\texttt{"hellohellohellohellohellohellohellohellohellohellohellohello"}

I.e., for any values of $n \leq 0$ \texttt{copystring} should return the empty string, for positive values it should return $n$ copies of the string, concatenated together.

4. Write a recursive function \texttt{numlist n} which generates a list of the integers $[0,1,2,3,\ldots,n]$, where $n \geq 0$. Generate the error "illegal argument" for $n < 0$.

\texttt{numlist} :: \texttt{Int} $\rightarrow$ [\texttt{Int}]
\texttt{numlist n} ...

> numlist 0
  [0]
> numlist 4
  [0,1,2,3,4]
> numlist (-5)
program error: illegal argument

5. Write a recursive function `allsame xs` which takes a list of Ints and returns `True` if all numbers are the same, and `False` otherwise.

   ```haskell
   allsame :: [Int] -> Bool
   allsame xs ...
   ```

   ```haskell
   > allsame []
   True
   > allsame [1]
   True
   > allsame [1,2]
   False
   > allsame [1,1,1,1,1]
   True
   > allsame [1,1,1,1,2]
   False
   ```

6. Write a recursive function `swap xs` which takes a list of Ints and returns a new list where pairs of adjacent elements have been swapped.

   ```haskell
   swap :: [Int] -> [Int]
   swap xs ...
   ```

   ```haskell
   > swap []
   []
   > swap [1]
   [1]
   > swap [1,2]
   [2,1]
   > swap [1,2,3]
   [2,1,3]
   > swap [1,2,3,4]
   [2,1,4,3]
   > swap [1,2,3,4,5,6,7,8]
   [2,1,4,3,6,5,8,7]
   ```

7. Write a recursive function `split xs` which takes a list of Ints and returns a tuple of two new lists such that every other element (starting with the first) goes in the first list, and the remaining elements in the second list.

   ```haskell
   split :: [Int] -> ([Int],[Int])
   split xs ...
   ```

   ```haskell
   > split []
   ([],[])
   > split [1]
   ([1],[])
   ```
> split [1,2]
([1],[2])
> split [1,2,3]
([1,3],[2])
> split [1,2,3,4]
([1,3],[2,4])
> split [1,2,3,4,5,6,7,8,9,10]
([1,3,5,7,9],[2,4,6,8,10])
> split [10,1000,111,42,666,99,999,9999]
([10,111,666,999],[1000,42,99,9999])

Note: The function \texttt{fst} returns the first element of a tuple, \texttt{snd} returns the second one.
4. Applications

1. Assume that we have a database of people and their corresponding phone numbers, implemented as a list of \((\text{name},\ \text{number})\) pairs:

\[
\text{phoneDB} = [(\"Jenny\",\"867-5309\"), (\"Alice\",\"555-1212\"), (\"Bob\",\"621-6613\")]
\]

Given someone’s name you’d like to be able to see if they’re in the database:

\>
\begin{align*}
\text{> member nameEQ ("Alice","") phoneDB} \\
\text{True} \\
\text{> member nameEQ ("Jenny","") phoneDB} \\
\text{True} \\
\text{> member nameEQ ("Erica","") phoneDB} \\
\text{False}
\end{align*}

And, given someone’s phone number you’d also like to be able to see if they’re in the database:

\>
\begin{align*}
\text{> member numberEQ ("","867-5309") phoneDB} \\
\text{True} \\
\text{> member numberEQ ("","111-2222") phoneDB} \\
\text{False}
\end{align*}

Write a function \text{member eq x xs} which looks up \text{x} in the list \text{xs}:

\[
\text{nameEQ (a,_) (b,_) = a == b} \\
\text{numberEQ (_,a) (_,b) = a == b} \\
\text{intEQ a b = a == b}
\]

\[
\text{member :: Eq a => (a -> a -> Bool) -> a -> [a] -> Bool} \\
\text{member eq x ys = …}
\]

Note a few things:

(a) \text{member} is \text{polymorphic}, i.e. \text{a} is a \text{type variable}, a place-holder for any type.
(b) \text{member} takes an equality function \text{eq} as argument, which returns true if its arguments are equal.
(c) By giving \text{member} different equality functions we can make it behave differently.

Here are some more examples:

\>
\begin{align*}
\text{> member intEQ 4 [1,2,3,4]} \\
\text{True} \\
\text{> member intEQ 4 [1,2,3,5]} \\
\text{False}
\end{align*}

We will study polymorphic functions in more detail later!

2. Credit card numbers are validated by the so-called \textit{Luhn algorithm}. You can read more about it here: \text{http://en.wikipedia.org/wiki/Luhn_algorithm}.

\[
\text{NOTE: Although in the examples below, credit card numbers are shown to be 11 digits, you cannot make such assumptions in your code! American Express, for example, uses 15 digits and Visa 16 digits.}
\]
Here’s a sketch of the validation algorithm:

(a) Let
\[ x_{10}, x_9, x_8, x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0 ] \]
be the credit card number. \( x_0 \) is the check digit.

(b) Double every other digit, starting from \( x_1 \):
\[ x_{10}, 2 \times x_9, x_8, 2 \times x_7, x_6, 2 \times x_5, x_4, 2 \times x_3, x_2, 2 \times x_1, x_0 ] \]

(c) Let \( S \) be the function which sums the digits of a number. For example, \( S(12) = 1 + 2 = 3 \). Now apply \( S \) to every other number, again starting with \( x_1 \):
\[ x_{10}, S(2 \times x_9), x_8, S(2 \times x_7), x_6, S(2 \times x_5), x_4, S(2 \times x_3), x_2, S(2 \times x_1), x_0 ] \]

(d) Sum up all the numbers in the list. If the sum is 0 mod 10, the account number is probably valid.

Let’s write a function `isValidCC x` which takes a string of digits as input and returns True or False depending on whether it’s a correct credit card number or not.

### NOTE: None of the functions below need to do any error checking.

(a) The `isValidCC` function itself has been given to you:
```haskell
isValidCC :: String -> Bool
isValidCC x = luhnSum (string2IntList x) 'mod' 10 == 0
```

(b) Now write a function `string2IntList x` which returns a list of the digits (all integers) of \( x \), where \( x \) is a string consisting only of digits:
```haskell
> string2IntList ""
[]
> string2IntList "12345"
[1,2,3,4,5]
```
You can assume that \( x \) only contains digits.

(c) Now write the function `sumDigits x` which sums the digits of \( x \), where you can assume that \( x \) is an integer between 0 and 99:
```haskell
> sumDigits 0
0
> sumDigits 7
7
> sumDigits 10
1
> sumDigits 11
2
> sumDigits 23
5
> sumDigits 99
18
```

(d) Now write a function `sumDigitsList` that sums all the digits of a list of integers:
```haskell
> sumDigitsList []
0
> sumDigitsList [1,2,3]
6
> sumDigitsList [11,12,13]
9
> sumDigitsList [0,7,10,11,23,99]
33
```
(e) Now write a function that doubles all the elements of a list of integers:

```haskell
> double []
[]
> double [1]
[2]
> double [1,2,3,4,5]
[2,4,6,8,10]
```

(f) Now write the `everyOther xs` function which returns every other element of a list `xs` of integers, starting with the first one:

```haskell
> everyOther []
[]
> everyOther [1]
[1]
> everyOther [1,2]
[1]
> everyOther [1,2,3]
[1,3]
> everyOther [1,2,3,4]
[1,3]
> everyOther [1,2,3,4,5]
[1,3,5]
```

(g) Finally, write the `luhnSum xs` function which, using the functions you've just defined as help, computes the Luhn sum according to the algorithm above:

```haskell
> luhnSum [4,9,9,2,7,3,9,8,7,1,6]
70
```

HINT: reverse and ++ are good friends to have.

(h) Now your `isValidCC` function should be working:

```haskell
> isValidCC "49927398716"
True
> isValidCC "49927398717"
False
> isValidCC "49927398726"
False
```
5. Submission and Assessment

The deadline for this assignment is noon, Mon Feb 4. It is worth 5% of your final grade.
You should submit the assignment to d2l.arizona.edu.

NOTE: The TAs will grade the assignment using the ghci interpreter on lectura. Make sure your code works on lectura before turning it in!

NOTE: Put all your functions into one file, named ass1.hs.

Don’t show your code to anyone, don’t read anyone else’s code, don’t discuss the details of your code with anyone. If you need help with the assignment see the instructor or the TA.