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 string concatenation.

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.

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

2 Simple 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`: [5 points]

```haskell
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`: [5 points]

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

> charToString 'A'
"A"
```

3. Using the formula

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

define a function `cylinderSurfaceArea (r, h)` which computes the surface area of a cylinder of height `r` and radius `h`, rounded down to the nearest integer: [5 points]
cylinderSurfaceArea :: (Float,Float) -> Int
cylinderSurfaceArea (r,h) = ...

> cylinderSurfaceArea (2.0,5.0)
87

Use the \texttt{floor} function to round down and the \texttt{pi} constant function to approximate $\pi$:

> floor 5.5
5
> pi
3.14159

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

\begin{verbatim}
third :: [Int] -> Int
third xs = ...
\end{verbatim}

> third [1,2,3,4,5]
3

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

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

\begin{verbatim}
yahtzee :: [Int] -> Bool
yahtzee xs = ...
\end{verbatim}

> yahtzee [1,1,1,1,1]
True
> yahtzee [1,1,1,1,2]
False
> yahtzee [6,6,6,6,6]
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 Simple Recursive Functions

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

\begin{verbatim}
msum :: Int -> Int
msum n = ...
\end{verbatim}

> msum 0
msum should make use of a conditional expression (if-then-else syntax).

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

\[
gsum :: \text{Int} -> \text{Int} \\
gsum ... \\
\]

\( gsum \) should make use of guards.

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

\[
\text{copystring} :: (\text{String},\text{Int}) -> \text{String} \\
\text{copystring} (s,n) ...
\]

Your function should have the following behavior:

\[
> \text{copystring} \ ("hello",-1) \\
""
> \text{copystring} \ ("hello",0) \\
""
> \text{copystring} \ ("hello",1) \\
"hello"
> \text{copystring} \ ("hello",2) \\
"hellohello"
> \text{copystring} \ ("hello",10) \\
"hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohello hellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohellohello helicopter
allsame :: [Int] -> Bool
allsame xs ...

> 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 \texttt{swap} \( xs \) which takes a list of \texttt{Int}s and returns and new list where pairs of adjacent elements have been swapped. [8 points]

\texttt{swap} :: [Int] -> [Int]
\texttt{swap} \( xs \) ...

> 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 \texttt{split} \( xs \) which takes a list of \texttt{Int}s and returns and tuple of two new lists such that every other element (starting with the first) goes in the first list, every other element in the second list. [9 points]

\texttt{split} :: [Int] -> ([Int],[Int])
\texttt{split} \( xs \) ...

> 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])
Note: The function `fst` returns the first element of a tuple, and `snd` returns the second one.

8. Write a function `hash xs` which computes a hash-value from a string `xs = [x_1, x_2, x_3, ...]`. The hash function should be defined as

\[
\text{hash}(x_1, \ldots, x_n) = |3^n + \left( \sum_{i=1}^{n} x_i 3^{i-1} \right)|
\]

or, expressed using Horners rule:

\[
\begin{align*}
\text{hash}[\ ] & = |1| \\
\text{hash}[x_1] & = |x_1 + 3 \times 1| \\
\text{hash}[x_1, x_2] & = |x_1 + 3 \times (x_2 + 3 \times 1)| \\
\text{hash}[x_1, x_2, x_3] & = |x_1 + 3 \times (x_2 + 3 \times (x_3 + 3 \times 1))| \\
\text{hash}[x_1, x_2, x_3, x_4] & = |x_1 + 3 \times (x_2 + 3 \times (x_3 + 3 \times (x_4 + 3 \times 1)))|
\end{align*}
\]

(HINT: Using Horner’s rule is easier since it translates directly into a recursive definition.) All arithmetic should be performed \( \text{mod} 2^{32} \), i.e. using 32-bit integers. \( |x| \) means the absolute value of \( x \). The `hash` function should be declared like this:

\[
\textbf{hash :: String -> Int} \\
\textbf{hash x = ..}
\]

Here are some examples:

\[
\begin{align*}
> & \text{hash } [\ ] \\
& 1 \\
> & \text{hash } ['\1'] \\
& 4 \\
> & \text{hash } ['\1', '\2'] \\
& 16 \\
> & \text{hash } ['\1', '\2', '\3'] \\
& 61 \\
> & \text{hash } ['\1', '\2', '\3', '\4'] \\
& 223 \\
> & \text{hash } "hello!" \\
& 22034 \\
> & \text{hash } "hello world! I'm a really long string!" \\
& 1224194303
\end{align*}
\]

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

Given someone’s name you’d like to be able to see if they’re in the database:
> member nameEQ ("Alice","") phoneDB
  True
> member nameEQ ("Jenny","") phoneDB
  True
> member nameEQ ("Erica","") phoneDB
  False

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

> member numberEQ ("","867-5309") phoneDB
  True
> member numberEQ ("","111-2222") phoneDB
  False

Write a function `member eq x xs` which looks up `x` in the list `xs`:

```haskell
nameEQ (a,_) (b,_) = a == b
numberEQ (_,a) (_,b) = a == b
intEQ a b = a == b

member :: (a -> a -> Bool) -> a -> [a] -> Bool
member eq x ys = ...
```

Note a few things:

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

Here are some more examples:

> member intEQ 4 [1,2,3,4]
  True
> member intEQ 4 [1,2,3,5]
  False

We will study polymorphic functions in more detail later!

4 Submission and Assessment

The deadline for this assignment is noon, Tue Sep 14. It is worth 5% of your final grade.

You should submit the assignment to `d2l.arizona.edu`.

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.