University of Arizona, Department of Computer Science

CSc 372 — Assignment 1 — Due noon, Tue Sep 13 — 5%

Christian Collberg
September 5, 2011

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`: [5 points]

```
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]

```
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 `floor` function to round down and the `pi` constant function to approximate π:

```
> floor 5.5
5
> pi
3.14159
```

4. Use `head` and `tail` to write a non-recursive function `third xs` which returns the third element of a list of `Int`s: [5 points]

```
third :: [Int] -> Int
third xs = ...

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

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

5. Use `head` and `tail` to write a non-recursive function `yahtzee xs` which takes a list of five `Int`s (between 1 and 6) as argument (the result of rolling five dice) and returns `True` if all numbers are the same, and `False` otherwise. [5 points]

```
yahtzee :: [Int] -> Bool
yahtzee xs = ...

> 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 Recursive Functions

1. Write a recursive function `msum n` that returns the sum of the integers $0 + 1 + 2 + 3 + \ldots + n$, where $n \geq 0$: [6 points]

   ```
   msum :: Int -> Int
   msum n = ...

   > msum 0
   0
   > msum 1
   1
   > msum 5
   15
   ```

   `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 $0 + 1 + 2 + 3 + \ldots + n$, where $n \geq 0$: [7 points]

   ```
   gsum :: Int -> Int
   gsum ...
   ```

   `gsum` should make use of *guards*.

3. Define a recursive function `copystring (s,n)` which returns a string consisting of `n` copies of the string `s`: [7 points]

   ```
   copystring :: (String,Int) -> String
   copystring (s,n) ...
   ```

   Your function should have the following behavior:

   ```
   > copystring ("hello",-1)
   ""
   > copystring ("hello",0)
   ""
   > copystring ("hello", 1)
   "hello"
   > copystring ("hello",2)
   "hellohello"
   > copystring ("hello",10)
   "hellohellohellohellohellohellohellohellohellohello"
   ```

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

4. Write a recursive function `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 $n < 0$. [7 points]

   ```
   numlist :: Int -> [Int]
   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 `Int`s and returns `True` if all numbers are the same, and `False` otherwise. [7 points]

```
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 `swap xs` which takes a list of `Int`s and returns and new list where pairs of adjacent elements have been swapped. [7 points]

```
swap :: [Int] -> [Int]
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 `split xs` which takes a list of `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. [8 points]

```
split :: [Int] -> ([Int],[Int])
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])
>  split [10,1000,111,42,666,99,999,9999]
([10,111,666,999],[1000,42,99,9999])
```

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

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

$$hash(x_1, \ldots, x_n) \quad = \quad |3^n + (\sum_{i=1}^{n} x_i 3^{(i-1)})|$$

or, expressed using *Horners rule*:

$$\begin{aligned}
hash[] &= |1| \\
hash[x_1] &= |x_1 + 3 * 1| \\
hash[x_1, x_2] &= |x_1 + 3 * (x_2 + 3 * 1)| \\
hash[x_1, x_2, x_3] &= |x_1 + 3 * (x_2 + 3 * (x_3 + 3 * 1))| \\
hash[x_1, x_2, x_3, x_4] &= |x_1 + 3 * (x_2 + 3 * (x_3 + 3 * (x_4 + 3 * 1)))| \\
\ldots & \quad \ldots
\end{aligned}$$

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

```
hash :: String -> Int
hash x = ..
```

Here are some examples:

```
> hash []
1
> hash ['\1']
4
> hash ['\1','\2']
16
> hash ['\1','\2','\3']
61
> hash ['\1','\2','\3','\4']
223
> hash "hello!"
22034
> hash "hello world! I'm a really long string!"
1224194303
```

4 Applications

1. Assume that we have a database of people and their corresponding phone numbers, implemented as a list of (*name*, *number*) pairs: [8 points]

```
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:

```
> 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`:

```
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!

2. Credit card numbers are validated by the so-called *Luhn algorithm*. You can read more about it here: `http://en.wikipedia.org/wiki/Luhn_algorithm`. [10 points]

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:

$$[x_{10}, 2 * x_9, x_8, 2 * x_7, x_6, 2 * x_5, x_4, 2 * x_3, x_2, 2 * x_1, x_0]$$

(c) Let $\mathcal{S}$ be the function which sums the digits of a number. For example, $\mathcal{S}(12) = 1 + 2 = 3$. Now apply $\mathcal{S}$ to every other number, again starting with x_1:

$$[x_{10}, \mathcal{S}(2 * x_9), x_8, \mathcal{S}(2 * x_7), x_6, \mathcal{S}(2 * x_5), x_4, \mathcal{S}(2 * x_3), x_2, \mathcal{S}(2 * x_1), x_0]$$

(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:

```
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:

```
> 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:

```
> 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:

```
> 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:

```
> 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:

```
> 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:

```
> 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:

```
> isValidCC "49927398716"
True
> isValidCC "49927398717"
False
> isValidCC "49927398726"
False
```

5 Submission and Assessment

The deadline for this assignment is noon, Tue Sep 13. 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.