CIS 552: Advanced Programming

Haskell List Processing and recursion

Due Saturday, September 17 at noon

This is the first homework assignment for CIS 552. It provides practice with the basic built-in data structures of Haskell, including lists, tuples and maybes, as well as recursion and pattern matching.

Before you get started, you should make sure that you have installed the Haskell Platform and emacs.

This file is a "literate" Haskell program, meaning that explanation is interspersed with actual Haskell code. To complete your assignment, edit this file, which contains just the code, and submit it through the course web page. Make sure the beginning of the file contains your name in the comments.

```
-- Advanced Programming, HW 1
-- by <YOUR NAME HERE>
```

Before we go any further, you should make sure that you can compile and run this code. Under emacs: C-c C-l will load the it interactively into ghci. You can then type `main` to run the main routine. Alternatively, to compile the code from the command line type

```
ghc --make Main
```

which will produce an executable (named `Main`, `a.out` or `Main.exe` depending on your system).

The first line of actual code for this homework assignment is a few pragmas to GHC (these are specific instructions for the compiler.)

```
{-# OPTIONS -Wall -fwarn-tabs -fno-warn-type-defaults  #-}
```

The first option controls what warnings we would like GHC to generate. `-Wall` is a large set of warnings that will help you satisfy many of the the code style requirements. However, -Wall doesn't warn about using tabs instead of spaces (as the style guide requires), so we enable that with `-fwarn-tabs`. Finally, -Wall includes one warning we don't want (about type defaulting, a feature of Haskell that changed in 1998), so we disable that warning with `-fno-warn-type-defaults`.

When you complete your assignment, it would be a good idea to add the option `-Werror`, which turns all warnings into errors. **We will not grade your assignment if there are any compiler warnings.**

The next directive, below, suppresses the import of the `Prelude`, the set of builtin functions that are automatically available for all Haskell programs. We're going to be writing some of these functions ourselves, so they shouldn't be imported. We will only do this supression for this particular homework assignment, in future assignments, you are encouraged to use any function in the prelude or standard library.

```
{-# LANGUAGE NoImplicitPrelude #-}
```

Next, we declare that we are creating a module called `Main` and using functions defined in the modules `Prelude` and `Test.HUnit`.

```
module Main where
import Prelude hiding (all, reverse, takeWhile, zip)
import Test.HUnit
```

The main program for this assignment merely runs the tests for each homework problem. You should not edit this code. Instead, your goal is to modify the problems below so that all of the tests pass. Note that the definitions in Haskell modules do not need to come in any particular order. The main function may use `test0`, `test1`, etc, even though their definitions are later in the file.

```
main :: IO ()
main = do
   _ <- runTestTT $ TestList [ test0, test1, test2, test3 ]
   return ()
```

Now that we have the preliminaries out of the way, we can start the actual problems for the assignment.

Problem 0

All of the following Haskell code does what it is supposed to do (i.e. the tests pass), but it is difficult to read. Rewrite the following expressions so that they follow the style guide. (You don't need to rewrite the Test following each problem.)

```
-- Part 0 (a)

abc x y z =
  if x then if y then True else
   if (x && z) then True else False
  else False


t0a :: Test
t0a = "0a1" ~: TestList [abc True False True ~?= True,
                         abc True False False ~?= False,
                         abc False True True ~?= False]

-- 0 (b)

arithmetic :: ((Int, Int), Int) -> ((Int,Int), Int) -> (Int, Int, Int)
arithmetic x1 x2 =
     let a = fst (fst x1) in
     let b = snd (fst x1) in
     let c = snd x1 in
     let d = fst (fst x2) in
     let e = snd (fst x2) in
     let f = snd x2
       in
       ((((((b*f) - (c*e)), ((c*
       d) - (a*f)
       ), ((a*e)-(b*d))))))

t0b :: Test
t0b = "0b" ~: TestList[ arithmetic ((1,2),3) ((4,5),6) ~?= (-3,6,-3),
                        arithmetic ((3,2),1) ((4,5),6) ~?= (7,-14,7) ]

-- 0 (c)

cmax :: [Int] -> Int -> Int
cmax l t
 = g (length l - 1) t
     where g n t = if n < 0 then t else
                   if (l !! n) > t then g (n-1) (l !! n) else g (n-1) t



t0c :: Test
t0c ="0c" ~: TestList[ cmax [1,4,2] 0 ~?= 4,
                       cmax []      0 ~?= 0,
                       cmax [5,1,5] 0 ~?= 5 ]

-- 0 (d)

reverse l  = reverse_aux l [] where
  reverse_aux l acc =
    if null l then acc
       else reverse_aux (tail l) (head l : acc)

t0d :: Test
t0d = "0d" ~: TestList [reverse [3,2,1] ~?= [1,2,3],
                        reverse [1]     ~?= [1] ]

test0 :: Test
test0 = "test0" ~: TestList [ t0a , t0b, t0c, t0d ]
```

Problem 1 - Validating Credit Card Numbers

Have you ever wondered how websites validate your credit card number when you shop online? They don't check a massive database of numbers, and they don't use magic. In fact, most credit providers rely on a checksum formula for distinguishing valid numbers from random collection of digits (or typing mistakes).

In this section, you will implement the validation algorithm for credit cards. It follows these steps:

- Double the value of every second digit beginning with the rightmost.
- Add the digits of the doubled values and the undoubled digits from the original number.
- Calculate the modulus of the sum divided by 10.

If the result equals 0, then the number is valid.

```
-- Part 1 (a)
```

We need to first find the digits of a number. Define the functions

```
toDigits :: Integer -> [Integer]
toDigits = error "unimplemented"

toDigitsRev :: Integer -> [Integer]
toDigitsRev = error "unimplemented"
```

`toDigits` should convert an Integer to a list of digits. `toDigitsRev` should do the same, but with the digits reversed. (Note that `Integer` is Haskell's type for arbitrarily large numbers. The standard 32-bit `Int` type overflows is not large enough to store credit-card numbers.)

```
t1a :: Test
t1a = "1a" ~: toDigitsRev 1234 ~?= [4,3,2,1]

-- 1 (b)
```

Once we have the digits in the proper order, we need to double every other one. Define a function

```
doubleEveryOther :: [Integer] -> [Integer]
doubleEveryOther = error "unimplemented"

t1b :: Test
t1b = "1b" ~: doubleEveryOther [8,7,6,5] ~?= [8,14,6,10]

-- 1 (c)
```

The output of doubleEveryOther has a mix of one-digit and two-digit numbers. Define the function

```
sumDigits :: [Integer] -> Integer
sumDigits = error "unimplemented"
```

to calculate the sum of all digits. Note that sumDigits needs to handle two-digit numbers correctly, as in the test case.

```
t1c :: Test
t1c = "1c" ~: sumDigits[8,14,6,10] ~?= 20

-- 1 (d)
```

Define the function

```
validate :: Integer -> Bool
validate = error "unimplemented"
```

that indicates whether an Integer could be a valid credit card number. This will use all functions defined in the previous exercises.

```
t1d :: Test
t1d = "1d" ~: validate 4012888888881881 ~?= True

test1 :: Test
test1 = TestList [ t1a, t1b, t1c, t1d ]
```

[This problem is adapted from the first practicum assigned in the University of Utrecht functional programming course taught by Doaitse Swierstra, 2008-2009.]

Problem 2 - Vedic Multiplication

Vedic Mathematics is the early stage of the Indian Mathematical heritage. In this tradition, which is based on mental arithmetic, common arithmetic functions are reformulated so that little or no intermediate results need to be written down.

One example is the algorithm for multiplication, called Ūrdhva Tiryagbhyām, which is Sanskrit for "vertically and crosswise". This algorithm computes products of numbers digit-by-digit, one digit at a time. Suppose we would like to multiply two n-digit numbers,

```
x_n ... x_1 x_0   and   y_n ... y_1 y_0
```

to produce the (at most) 2n+1 digit result

```
z_{2n+1} z_{2n} ... z_1 z_0
```

We can compute each digit of the output z_k by working right-to-left, using two results: c_k, the "carry" from the previous step (initially zero), and s_k, the discrete convolution of the first k input digits (x_k ... x_0) and (y_k ... y_0).

```
z_k = (s_k + c_k) mod 10
```

The carry for the next step is

```
c_k = (s_k + c_k) div 10
```

The *discrete convolution* of two lists of numbers is the dot product of the first list and the reverse of the second list. For example, the convolution of the lists `[2,4,6] [1,2,3]` is `20` because `20 = 2*3 + 4*2 + 6*1`.

For example, suppose we wanted to multiply

```
     2 4 6

  x  1 2 3
 ----------
```

We compute the result digits (called `z6 z5 z4 z3 z2 z1 z0`) in stages. We start with the first carry `c0` (which is `0`), and the first convolution s0, which is:

```
s0 = conv [6] [3] = 18
```

From these, we can compute the rightmost digit of the output (`z0`) and the next carry (`c1`):

```
z0 = (s0 + c0) `mod` 10 = 8
c1 = (s0 + c0) `div` 10 = 1
```

For the next step, we compute the next convolution (of `x1 x0` and `y1 y0`) and use that result for the next digit of the output (`z1`) and its carry:

```
s1 = conv [4,6] [2,3] = 24

z1 = (s1 + c1) `mod` 10 = 5
c2 = (s1 + c1) div 10 = 2
```

We continue as before, computing the convolution at each step and remembering the carry.

```
s2 = conv [2,4,6][1,2,3] = 20

z2 = (s2 + c2) `mod` 10 = 2
c3 = (s2 + c2) `div` 10 = 2

s3 = conv [0,2,4,6][0,1,2,3] = 8 (= conv [2,4][1,2])

z3 = (s2 + c2) `mod` 10 = 0
c4 = (s2 + c2) `div` 10 = 1

s4 = conv [0,0,2,4,6][0,0,1,2,3] = 2 (= conv [2][1])

z4 = (s3 + c3) `mod` 10 = 3
c5 = (s3 + c3) `div` 10 = 0

s5 = conv [0,0,0,2,4,6][0,0,0,1,2,3] = 0

z5 = (s4 + c4) `mod` 10 = 0
```

```
c6 = (s4 + c4) `div` 10 = 0

s6 = conv [0,0,0,0,2,4,6][0,0,0,0,1,2,3] = 0

z6 = (s5 + c5) `mod` 10 = 0
c7 = (s5 + c5) `div` 10 = 0
```

So the result of the multiplication is:

```
[0,0,3,0,2,5,8]

-- 2a)
```

Implement the convolution function.

```
-- | The conv function takes two lists of numbers, reverses the
-- second list, multiplies their elements together pointwise, and sums
-- the result.  This function assumes that the two input
-- lists are the same length.

conv :: [Int] -> [Int] -> Int
conv xs ys = error "conv: unimplemented"

t2a :: Test
t2a = "2a" ~: conv [2,4,6] [1,2,3] ~?= 20

-- 2b)
```

Given two numbers, written as lists of digits, the first step of vedic multiplication is to add extra zeros to the beginning of both lists.

```
-- | The normalize function adds extra zeros to the beginning of each
-- list so that they each have length 2n + 1, where n is
-- the length of the longer number.

normalize :: [Int] -> [Int] -> ([Int], [Int])
normalize = error "normalize: unimplemented"

t2b :: Test
t2b = "2b" ~: normalize [1] [2,3] ~?= ([0,0,0,0,1], [0,0,0,2,3])

-- 2c)
```

Now use conv and normalize to implement the Ūrdhva Tiryagbhyām algorithm.

```
-- | multiply two numbers, expressed as lists of digits using
-- the Ūrdhva Tiryagbhyām algorithm.

multiply :: [Int] -> [Int] -> [Int]
multiply = error "unimplemented"

t2c :: Test
t2c = "2c" ~: multiply [2,4,6][1,2,3] ~?= [0,0,3,0,2,5,8]

-- 2d) OPTIONAL CHALLENGE PROBLEM
```

Your definition of convolution probably traverses each list multiple times. Rewrite this function so that each lists is traversed only once. (You may assume that both lists are of the same length.)

```
convAlt :: [Int] -> [Int] -> Int
convAlt = error "unimplemented"

t2d :: Test
t2d = "2d" ~: convAlt [2,4,6][1,2,3] ~=? 20

test2 :: Test
test2 = TestList [t2a,t2b,t2c,t2d]
```

[The inspiration for this problem comes from the paper: Olivier Danvy and Meyer Goldberg, "There and Back Again", BRICS RS-01-39]

Problem 3 - List library chops

Define, debug and test the following functions. (Some of these functions are part of the Haskell standard prelude or standard libraries like Data.List.) Their solutions are readily available online. You should not use these resources, and instead, implement them yourself.

There are eight parts of this assignment, and for each part, you need to declare the type of the function, define it, and replace the testcase for that part based on the problem description. The declared type of each function should be the most general one. Make sure to test the functions with multiple inputs using "TestList".

```
test3 :: Test
test3 = "test3" ~: TestList [t3a, t3b, t3c, t3d, t3e, t3f, t3g, t3h]

-- 3 (a)

-- The intersperse function takes an element and a list
-- and `intersperses' that element between the elements of the list.
-- For example,
--    intersperse ',' "abcde" == "a,b,c,d,e"

t3a :: Test
t3a = "3a" ~: assertFailure "testcase for 4a"

-- 3 (b)

-- invert lst returns a list with each pair reversed.
-- for example:
--   invert [("a",1),("a",2)] returns [(1,"a"),(2,"a")]

t3b :: Test
t3b = "3b" ~: assertFailure "testcase for 4b"

-- 3 (c)

-- takeWhile, applied to a predicate p and a list xs,
```

```
-- returns the longest prefix (possibly empty) of xs of elements
-- that satisfy p:
-- For example,
--     takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
--     takeWhile (< 9) [1,2,3] == [1,2,3]
--     takeWhile (< 0) [1,2,3] == []

t3c :: Test
t3c = "3c" ~: assertFailure "testcase for 4c"

-- 3 (d)

-- find pred lst returns the first element of the list that
-- satisfies the predicate. Because no element may do so, the
-- answer is returned in a "Maybe".
-- for example:
--     find odd [0,2,3,4] returns Just 3

t3d :: Test
t3d = "3d" ~: assertFailure "testcase for 4d"

-- 3 (e)

-- all pred lst returns False if any element of lst
-- fails to satisfy pred and True otherwise.
-- for example:
--    all odd [1,2,3] returns False

t3e :: Test
t3e = "3e" ~: assertFailure "testcase for 4e"

-- 3 (f)

-- map2 f xs ys returns the list obtained by applying f to
-- to each pair of corresponding elements of xs and ys. If
-- one list is longer than the other, then the extra elements
-- are ignored.
-- i.e.
--   map2 f [x1, x2, ..., xn] [y1, y2, ..., yn, yn+1]
--        returns [f x1 y1, f x2 y2, ..., f xn yn]

t3f :: Test
t3f = "3f" ~: assertFailure "testcase for 4f"

-- 3 (g)

-- zip takes two lists and returns a list of corresponding pairs. If
-- one input list is shorter, excess elements of the longer list are
-- discarded.
-- for example:
--    zip [1,2] [True] returns [(1,True)]

t3g :: Test
t3g = "3g" ~: assertFailure "testcase(s) for zip"

-- 3 (h)  WARNING this one is tricky!

-- The transpose function transposes the rows and columns of its argument.
-- If the inner lists are not all the same length, then the extra elements
-- are ignored.
-- for example:
```

```
--    transpose [[1,2,3],[4,5,6]] returns [[1,4],[2,5],[3,6]]

t3h :: Test
t3h = "3h" ~: assertFailure "testcase for 4h"
```

News :

Welcome to CIS 552!
See the home page for basic information about the course, the schedule for the lecture notes and assignments, the resources for links to the required software and online references, and the syllabus for detailed information about the course policies.

Links :

- Piazza
- Haskell.org
- GHC manual
- Library documentation
- Hackage

Design by Minimalistic Design
Powered by Pandoc