Homework 4 — assigned Monday 28 April — due Monday 5 May

Your programs should be prepared as literate Haskell scripts `hw4n.lhs` (where `n` is a digit) containing, in addition to your program code, some descriptive text outlining your design decisions, and a detailed testing log, showing a comprehensive test suite. Your code should be compiled using `ghc`, and the executable `hw4n` and all other output files from the compiler should be submitted as well. If you are using the Latex literate Haskell script style, submit the final typeset form as a PDF file.

Wherever this aids clarity, use Haskell's predefined higher-order functions. Wherever this aids clarity, use the point-free style of definition.

4.1 Using lists for arithmetic: writing recursive functions over lists (50pts)

Numerals can be represented as lists of integers. For instance, decimal numerals can be expressed as lists of integers from 0 to 9. The integer 12345678901234567890 might be represented as the Haskell list `[1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0] :: [Int]`. However, the representation should allow a radix (base) other than 10 as well.

We use the following type abbreviation:

```
type Numeral = (Int, [Int])
```

where the first component of the pair is the radix and the second the list of digits.

The above example number is then represented as:

```
example = (10, [1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0])
```

Write the following functions:

1. (10pts) `makeLongInt: Integer -> Int -> Numeral`, such that `makeLongInt` *n r* computes the list representation of the integer *n* in radix *r*. You can assume that $n \geq 0$, and that $r > 1$. For example, `makeLongInt 123 10` should evaluate to `(10, [1,2,3])`.

2. (10pts) `evaluateLongInt: Numeral -> Integer`, such that `evaluateLongInt (`*r*`,` *l*`)` converts a numeral back to a Haskell integer. You can assume that *l* is a valid list for radix *r*. For example, `evaluateLongInt (10, [1,2,3])` should evaluate to `123`.

3. (10pts) `changeRadixLongInt: Numeral -> Int -> Numeral`, such that `changeRadixLongInt` *n r* computes the representation of the same number as *n* in a new radix *r*. For example, `changeRadixLongInt (10, [1,2,3]) 8` should evaluate to `(8, [1,7,3])`; on the other hand, `changeRadixLongInt (10, [1,2,3]) 16` should evaluate to `(16, [7,11])`. The computation should be carried out without the use of Haskell's built-in `Integer` arithmetic.

4. (10pts) `addLongInts: Numeral -> Numeral -> Numeral`, such that `addLongInts` *a b* computes the sum of the numbers given by the numerals *a* and *b*. If *a* and *b* use the same radix, that radix should be used for the result. If *a* and *b* use different radices, the result should use the larger one. For example, `addLongInts (10, [1,2,3]) (3, [1])` should evaluate to `(10, [1,2,4])`. The computation should be carried out without the use of Haskell's built-in `Integer` arithmetic.

5. (10pts) `mulLongInts: Numeral -> Numeral -> Numeral`, such that `mulLongInts` *a b* computes the product of the numbers given by the numerals *a* and *b*. If *a* and *b* use the same radix, that radix should be used for the result. If *a* and *b* use different radices, the result should use the larger one. For example, `mulLongInts (10, [1,2,3]) (3, [1])` should evaluate to `(10, [1,2,3])`. The computation should be carried out without the use of Haskell's built-in `Integer` arithmetic. It is not permissible to implement the multiplication of *a* and *b* as $\sum_1^a b$.

4.2 Higher-order functions over lists (40pts)

Answers to these two questions should be submitted as a text file `hw42.txt` or as a PDF file `hw42.pdf`.

4.2.1 Filter (20pts)

Prove that the following equality holds for all finite lists `xs` and for all predicates `p` and `q` of suitable type:

```
filter p (filter q xs) = filter (p &&& q) xs
```

where the infix operator `&&&` is defined as follows:

```
p &&& q = \x -> p x && q x
```

4.2.2 Map and concat (20pts)

Prove that the following equality holds for all finite lists `xs` and for all functions `f` of suitable type:

```
concat (map (map f) xs) = map f (concat xs)
```

4.3 Trees (30pts)

We can define binary trees without any interesting content as follows:

```
data T = Leaf | Node T T
```

A path from the root to any subtree consists of a series of instructions to go left or right, which can be represented using another datatype:

```
data P = GoLeft P | GoRight P | This
```

where the path `This` denotes the whole tree. Given some tree, we would like to find all paths, i.e., the list of all paths from the root of the given tree to each of its subtrees. Write a function `allpaths :: T -> [P]` to do so.

For instance, `allpaths (Node Leaf (Node Leaf Leaf))` should evaluate to
`[This,GoLeft This,GoRight This,GoRight (GoLeft This),GoRight (GoRight This)]`
(but the ordering of the paths is immaterial).

How to turn in

Submission instructions: see course mailing list.

Include the following statement with your submission, signed and dated:
I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual.