Homework 2 — assigned Monday 26 February — due Friday 9 March

General instructions

A skeleton Haskell source file homework2.hs will be provided with the type signatures of all the functions you are to write. Please edit that file and submit.

The skeleton file will start with `import` declarations for all Haskell libraries that you are allowed to use in your solution. You must not add any other `import` declarations.

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

Hints

It may be convenient to develop the solution to Exercise 2.6 in stages, and in the following order of increasing difficulty (in the instructor's opinion at least): `evaluateLongInt`, `makeLongInt`, `addLongInts`, `mulLongInts`, `changeRadixLongInt`. We've provided a few specifications that work via `Integer`; these may be convenient to use in place of the final definition while you are developing other functions.

2.1 Simple functions on numbers (10pts)

(See Project Euler, `https://projecteuler.net/problem=14`.)

The following iterative sequence is defined for the set of positive integers:

$$n \to \frac{n}{2}, \quad n \text{ even}$$
$$n \to 3n+1, \quad n \text{ odd}$$

Using the rule above and starting with 13, we generate the following sequence:

$$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

It can be seen that this sequence (starting at 13 and finishing at 1) contains 10 terms. Although it has not been proven yet (Collatz Problem), it is thought that all starting numbers finish at 1.

Define a function, `collatz :: [Int] -> Int`, that takes in a list of starting numbers and returns the one which gives rise to the longest Collatz sequence. For example, `collatz [1..20]` should evaluate to `19`.

If multiple starting numbers have the same sequence length, your function should return the largest of them. For example, 18 and 19 both produce a sequence length of 21, and so 19 is reported.

NB. Once the sequence starts, the terms can become quite large. Your code must be prepared to handle this possibility.

2.2 Simple functions on lists and strings (10pts)

Write a function `haskellFileNames :: [String] -> [String]`, which, given a list of strings, will extract those strings which (ignoring any leading or trailing space characters) look like Haskell source file names (they end exactly in .hs or .lhs). (We'll consider this.is.a.file.hs to be a good file name.) For example,

```
haskellFileNames
  [
    "pure.hs",
    "best.lhs",
    "good.better.hs",
    "      pure.hs      ",
    "impure.c",
    "bad.java",
    "awesome.Hs",
    "cool.hs.txt"
  ]
```

evaluates to `["pure.hs","best.lhs","good.better.hs","      pure.hs      "]`.

2.3 Simple functions on lists (10pts)

Write a function `select :: (t -> Bool) -> [t] -> [a] -> [a]`, which takes a predicate and two lists as arguments and returns a list composed of elements from the second list in those positions where the predicate holds when applied to the element in the corresponding position of the first list. For example, `select even [1..26]  "abcdefghijklmnopqrstuvwxyz"` evaluates to `"bdfhjlnprtvxz"`.

2.4 Simple functions on lists and numbers (10pts)

Write a function `prefixSum :: [Int] -> [Int]`, which takes a list of numbers as its argument and returns a list of sums of all prefixes of the list. For example, `prefixSum [1..10]` evaluates to `[1,3,6,10,15,21,28,36,45,55]`.

2.5 Simple functions on lists and numbers (10pts)

Write a function `numbers :: [Int] -> Int`, which takes a list of integers (each of them between zero and nine) as its argument and returns the integer which has those numbers as digits in the usual decimal notation. For example, `numbers [1..4]` evaluates to `1234`.

2.6 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 is the list of digits.

The above example number is then represented as:

```
example :: Numeral
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`. (This is a generalization of Exercise 2.5 above.)

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. In particular, the following implementation must be understood only as a specification (because it uses `Integer` arithmetic within the functions `makeLongInt` and `evaluateLongInt`):

   ```
   changeRadixLongInt (r1, ds1) r2 = makeLongInt (evaluateLongInt (r1, ds1)) r2
   ```

   Additional examples: `changeRadixLongInt (16, [13,14,10,13,11,14,14,15]) 17` evaluates to `(17, [9,1,13,3,6,16,7,8])`.

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. In particular, the following implementation must be understood only as a specification (because it uses `Integer` arithmetic in `(+)` as well as within the functions `makeLongInt` and `evaluateLongInt`):

```
addLongInts (r1, ds1) (r2, ds2)
  | r1 == r2 = makeLongInt (evaluateLongInt (r1, ds1) + evaluateLongInt (r2, ds2)) r1
  | r1 < r2 = addLongInts (changeRadixLongInt (r1, ds1) r2) (r2, ds2)
  | r1 > r2 = addLongInts (r1, ds1) (changeRadixLongInt (r2, ds2) r1)
```

It is not permissible to implement the addition of a and b as $b + \sum_1^a 1$ (repeated Succ).

Additional examples: `addLongInts (16, [13,14,10,13,11,14,14,15]) (8, [7, 7, 7])` evaluates to `(16, [13,14,10,13,12,0,14,14])`.

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. In particular, the following implementation must be understood only as a specification (because it uses `Integer` arithmetic in `(*)` as well as within the functions `makeLongInt` and `evaluateLongInt`):

```
mulLongInts (r1, ds1) (r2, ds2)
  | r1 == r2 = makeLongInt (evaluateLongInt (r1, ds1) * evaluateLongInt (r2, ds2)) r1
  | r1 < r2 = mulLongInts (changeRadixLongInt (r1, ds1) r2) (r2, ds2)
  | r1 > r2 = mulLongInts (r1, ds1) (changeRadixLongInt (r2, ds2) r1)
```

   It is not permissible to implement the multiplication of a and b as $\sum_1^a b$ (repeated addition).

   Additional examples: `mulLongInts (16, [13,14,10,13,11,14,14,15]) (8, [7, 7, 7])` evaluates to `(16, [1,11,12,7,12,13,0,1,15,1,1])`.

How to turn in

Use the UNM Learn facility as follows: Navigate to `https://learn.unm.edu/` and log in. Then click on $\boxed{\text{CS-357L-000 (Spring 2018)}}$. Now click on $\boxed{\text{Assignments}}$ in the left side navigation menu. After that, click on the appropriate homework assignment link. Now attach your .hs file(s) and click submit. You are allowed to submit as many times as you like but only the latest submission will be graded.