CSCI 365 Problem Set 4: Lists

due Friday, 16 February 2024

Specification

To receive credit for this problem set:

- You must complete at least 8 out of 11 exercises.
- Any code you write must adhere to the Haskell style guide linked from the course web page.

Haskell code should be submitted in one or more `.hs` files. Written exercises may be submitted either as a PDF, or as comments in one of the `.hs` files.

List comprehensions

Exercise 1 Using a list comprehension, find all the primitive Pythagorean triples with three positive numbers less than 100. A *primitive Pythagorean triple* is a triple (a, b, c) where $a^2 + b^2 = c^2$, the three numbers do not all share a common divisor other than 1, and $a < b < c$.

Hint: to find the greatest common divisor of three integers you can use `gcd a (gcd b c)`.

Exercise 2 Using a list comprehension, write a function

```
unique :: Eq a => [a] -> [a]
```

which returns the list of those elements from the input list which occur exactly once. For example,

It is OK if your function takes $O(n^2)$ time.

```
unique "hello" = "heo"
unique [1,3,5,3,2,7,2,0,1] = [5,7,0]
unique [1,2,3] = [1,2,3]
unique [5,5,5] = []
```

Hint: you may find it helpful to write a helper function

```
count : Eq a => a -> [a] -> Int
```

which counts how many times the given element occurs in the given list.

Exercise 3 Turn the following English descriptions into Haskell code defining the infinite list of all prime numbers:

This is not the most efficient way to define the infinite list of primes, but it is probably the simplest.

- `primes :: [Integer]` is defined as the result of sieving the list `[2..]`.
- The result of sieving a list whose first element is `p` is defined as the list which begins with `p`, and continues with the result of sieving all the numbers from the remainder of the list which are not divisible by `p`.
- Sieving the empty list yields the empty list (this case is not strictly needed since we will only sieve infinite lists, but we include it so GHC doesn't yell at us about missing cases).

List processing

Exercise 4 Write a function `isSorted :: Ord a => [a] -> Bool` which decides whether or not a list is in sorted order.

Exercise 5 Write functions

```
rle :: Eq a => [a] -> [(Int,a)]
unrle :: [(Int,a)] -> [a]
```

which implement *run-length encoding*. The first function, `rle`, takes a list and turns each run of consecutive equal elements into a pair containing the element and the length of the run. For example,

```
rle [1,1,1,1,2,2,2,1,1,1] = [(4,1), (3,2), (3,1)]
rle "hello" = [(1,'h'), (1,'e'), (2,'l'), (1,'o')]
```

The second function, `unrle`, should invert `rle`, that is, it expands a run-length-encoded list back into the original list. In other words, it should be the case that `unrle . rle = id`.

Optional challenge question: why is it *not* the case that `rle . unrle = id`?

Exercise 6 Consider the algebraic data type `PTree`, defined below.

```
data PTree a where
  Leaf :: a -> PTree a
  Fork :: PTree a -> PTree a -> PTree a
```

`PTree a` values represent binary trees with data stored at the leaves. Write a function

```
flatten :: PTree a -> [a]
```

which returns the list of all the elements stored in the leaves of a `PTree`, in order from left to right.

Exercise 7 The most straightforward solution to the previous exercise takes quadratic time on trees which are deeply nested to the left. Write a version of `flatten` which runs in linear time. *Hint*: write a helper function

```
flattenTrees :: [PTree a] -> [a]
```

which takes a *stack* of `PTree` values (represented as a list) and returns the list of all their elements in order.

String Processing

Exercise 8 Without looking at its implementation in the standard library, write a function

```
lines :: String -> [String]
```

which splits a string into lines, by breaking at each newline (`'\n'`) character. For example,

```
lines "Hello" = ["Hello"]
lines "Hello\nthere\nworld" = ["Hello", "there", "world"]
lines "Hello\n\n\nthere" = ["Hello", "", "", "there"]
```

Exercise 9 Without looking at its implementation in the standard library, write a function

```
words :: String -> [String]
```

which splits a string into nonempty words delimited by whitespace. "Whitespace" includes not only space characters, but also tabs (`'\t'`) and newlines (`'\n'`). To test whether a given character is whitespace, you are welcome to import the `isSpace` function from `Data.Char`. Unlike with the `lines` function, multiple consecutive whitespace characters should not result in any empty words being generated. Overall it should behave much like `.split()` in Python. For example:

```
words "Hello there world" = ["Hello", "there", "world"]
words "Hello     there\t \n   world  " = ["Hello", "there", "world"]
words "Hello" = ["Hello"]
words "" = []
```

Exercise 10 Write a function

```
wordList :: String -> [String]
```

which returns a sorted list of unique words which occur in the input. All words should be converted to lowercase and stripped of any punctuation (defined for the purposes of this exercise as any character other than letters or whitespace). For example,

```
wordList "I say! I said, \"Hello hello! What did you say?\""
```

should return

```
["did","hello","i","said","say","what","you"].
```

Of course you are welcome to use any functions written previously on this problem set, functions written in class, or functions you find in the Haskell standard libraries. Rather than writing one monolithic function to do everything, try to use simple functions that incrementally transform the input into something one step closer to the desired output, then chain all these small functions together via function composition.

Further Exploration

Exercise 11 Read about the `TransformListComp` language extension at `https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/generalised_list_comprehensions.html`. Play with it and come up with an interesting example that is different than the examples shown on the page linked above. Could you have used it to help on any of the other exercises on this problem set?

Remember that you can enable language extensions by writing `{-# LANGUAGE ExtensionName #-}` at the very top of your `.hs` file.