Homework Assignment 2

Functional and Logic Programming, 2023

Due date: Thursday, May 18, 2023 (18/05/2023)

Bureaucracy

- Submission is in pairs, but solo submission is also allowed. We very much suggest you pair-up, as solving this exercises with another person greatly enhances your learning (as well as being more fun!).
- To submit, create a zip file named `HW2_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters IDs.
  - Or `HW2_<id>.zip` if submitting alone.
- Make sure your submission compiles successfully. Submissions which do not compile will receive a 0 grade!
  - We will be using the following command to compile the file: `ghc -Wall -Werror HW2.hs`.

General notes

- The instructions for this exercise are split between this file and `HW2.hs`. This file offers a more high-level overview of the exercise, as well as offering a few hints. The Haskell file details all the required functionality for this assignment.
- You may not modify the **`import`** statement at the top of the file, nor add new **`import`**s.
  - If you are unsure what some function does, you can either ask HLS or Hoogle.
  - Hoogle also support module lookups, e.g., `Prelude.notElem`.
  - Do be aware however, that some functions from the standard library are more general than what we have learned so far in class.
    - And some cases their definition may not be entire clear just yet!
- The exercises and sections are defined linearly. That is, it is a good idea to use previously defined functions (either from the same section or ones before it). It is also a good idea to use functions you saw in class; some of them are already imported, and some of them you would have to define yourself.
  - Do not be alarmed by the large amount of functions! Many of them are simple one-liners, and were designed to aid you in solving the more complex functions.

  - In general, you may define as many helper functions as you wish.

- Try to write elegant code, as taught in class. Use point-free style, η-reductions, and function composition to make your code shorter and more declarative. Prefer `foldr` over manual recursion where possible, and use functions like `map` and `filter` when appropriate. Although it is not required in the homework assignments, non-elegant code *will* be penalized in the test, so this is a good exercise. HLS and hlint can be very helpful in this.

  - Do note that in some cases, hlint may suggest functions which are not imported, or which you are trying to implement right now!

- If possible, please ask your questions first in Piazza, as this will allow all students to take part in the discussion.

Section 1: String intersections

In this part we will implement a few basic `String` intersection functions.

```
isPrefixOf :: String -> String -> Bool
isInfixOf :: String -> String -> Bool
isSuffixOf :: String -> String -> Bool
isSubseuenceOf :: String -> String -> Bool
```

Here are the example usages:

```
"foo" `isPrefixOf` "foobar"
True
"foo" `isPrefixOf` "xfoobar"
False
"bar" `isSuffixOf` "foobar"
True
"bard" `isPrefixOf` "foobar"
False
"oob" `isInfixOf` "foobar"
True
"ooo" `isInfixOf` "foobar"
False
"oa" `isSubseuenceOf` "foobar"
True
"ao" `isSubseuenceOf` "foobar"
False
```

Note that the empty `String` `""` is a prefix/suffix/infix/subsequence of any other `String`.

Section 2: Document search

In this section we will implement an expression tree for searching **phrases** inside **documents**. For our purposes, both phrases and documents are plain Haskell `String`s. Given a list of documents, we want to partition to ones which match our search phrase and ones which don't. To support complicated search options, we will be using an expression algebraic data type, called `Query`.

```
type Phrase = String
data Query = All [Query] | Any [Query] | None [Query] | Literal Phrase
type Document = String
findDocuments :: Query -> [Document] -> ([Document], [Document])
```

The simplest option is to simply search for a single phrase, using `Literal`.

```
documents = ["Hello!", "Goodbye!"]
findDocuments (Literal "ood") documents
(["Goodbye!"],["Hello!"])
```

The other `Query` nodes allow us to aggregate multiple phrases together.

- `All` requires **all** of its queries to match the document.
- `Any` requires **any** of its queries to match the document.
- `None` requires **none** of its queries to match the document.

Note that searching is **case-sensitive**. In all cases, the returned order of the documents should maintain the original relative order in the list. Let's take a look at a few examples:

```
documents = ["abcd", "abef", "xyz", "abcdefg"]
findDocuments (All [Literal "a", Literal "c"]) documents
(["abcd","abcdefg"],["abef","xyz"])
findDocuments (Any [None [Literal "x"], All [Literal "ab", Literal "yz"]]) documents
(["abcd","abef","abcdefg"],["xyz"])
findDocuments (None [None [Literal "abc"]]) documents
(["abcd","abcdefg"],["abef","xyz"])
```

Hint: It might be helpful to define a function which checks if a single `Document` matches a `Query`.

Section 3: Infinite lists

In this section we will deal with infinite lists. Unlike regular lists `[a]`, which may or may not be infinite, we will define a list which is **necessarily** infinite.

```
-- (:>) Is the constructor in this case
data InfiniteList a = a :> InfiniteList a
infixr 3 :>
```

Sampling infinite lists

In order to be able to actually verify our infinite lists, it is useful to be able to evaluate a few starting elements. For that, we will implement `itoList :: InfiniteList a -> [a]`. Combined with the regular list function `take :: Int -> [a] -> [a]` from `Prelude`, we can now evaluate finite prefixes of infinite lists:

```
sample :: InfiniteList a -> [a]
sample = take 10 . itoList
smallSample :: InfiniteList a -> [a]
smallSample = take 5 . itoList
```

Creating infinite lists

Below are usage examples of the basic ways of creating infinite lists:

```
sample $ irepeat 1
[1,1,1,1,1,1,1,1,1,1]
sample $ icycle [1, 2, 3]
[1,2,3,1,2,3,1,2,3,1]
smallSample $ iiterate (\ x -> x * x + x) 1
[2,6,42,1806,3263442]
```

- `icycle []` should never terminate, i.e., it should enter an infinite loop.

The `integers` and `naturals` are infinite lists of all the integers (0, positives, and negatives) and natural numbers (0 and positives). Note that the order of both lists is important!

```
sample integers
[0,1,-1,2,-2,3,-3,4,-4,5]
sample naturals
[0,1,2,3,4,5,6,7,8,9]
```

`imap` and `iscan` are the two ways of converting one infinite list to another:

```
sample $ imap (* 3) integers
[0,3,-3,6,-6,9,-9,12,-12,15]
sample $ iscan (+) 0 naturals
[0,1,3,6,10,15,21,28,36,45]
```

Hint: You can pattern match on an infinite list the same way you would a regular list!

```
imap f (x :> xs) = undefined
```

`zip` and `interleave` combine two infinite lists to one infinite list:

```
l1 = imap (*3) integers
l2 = imap (*5) naturals
sample $ izip l1 l2
[(0,0),(3,5),(-3,10),(6,15),(-6,20),(9,25),(-9,30),(12,35),(-12,40),(15,45)]
sample $ interleave l1 l2
[0,0,3,5,-3,10,6,15,-6,20]
```

`inits` and `tails` expand a single infinite list into multiple infinite lists:

```
smallSample $ iinits naturals
[[],[0],[0,1],[0,1,2],[0,1,2,3]]
smallSample $ imap smallSample $ itails naturals
[[1,2,3,4,5],[2,3,4,5,6],[3,4,5,6,7],[4,5,6,7,8],[5,6,7,8,9]]
```

Bonus

Last but not least, you may implement the following function for a bonus of 15 points:

```
ifind :: (a -> Bool) -> InfiniteList (InfiniteList a) -> Bool
```

This function accepts an infinite list of infinite lists and returns `True` if any list contains an element matching the predicate. It doesn't have to return `False` if there is no such element—how would it even know?—it can simply iterate forever instead.

```
powers = imap (iiterate (\x -> x * x)) naturals
ifind (\x -> x > 1000 && x < 1100) powers
True

-- Will loop forever!
ifind (\x -> x > 1000 && x < 1010) powers
```

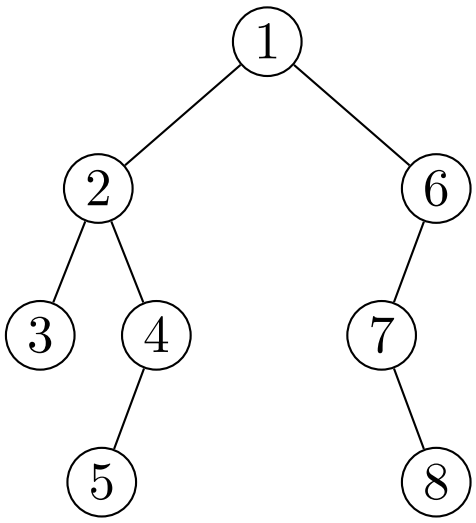
Hint of dubious efficacy: The power of an `InfiniteList` is $\aleph_0$. Of course, we can iterate over all elements of a set with power $\aleph_0$. Luckily, the power of an `InfiniteList (InfiniteList a)` is $\aleph_0 \times \aleph_0 = \aleph_0$! How does one iterate over such a set though?

Section 4: Binary trees

In this section we will deal with (non-search) binary trees. By non-search, we mean that they are no necessarily ordered.

```
data Tree a = EmptyTree | Tree (Tree a) a (Tree a)
```

As you know, there are multiple ways of traversing a tree. Since we have no side effects, one way of defining tree traversal is by converting a tree to a list. For the below code, the tree looks like this:




```
single t = Tree EmptyTree t EmptyTree
tree =
  Tree
    (Tree (single 3) 2 (Tree (single 5) 4 EmptyTree))
    1
    (Tree (Tree EmptyTree 7 (single 8)) 6 EmptyTree)
preOrder tree
[1,2,3,4,5,6,7,8]

inOrder tree
[3,2,5,4,1,7,8,6]

postOrder tree
[3,5,4,2,8,7,6,1]

levelOrder tree
[1,2,6,3,4,7,5,8]
```

We can also build trees from lists. The function `fromListLevelOrder` builds a **complete** binary tree from a list such that applying `levelOrder . fromListLevelOrder` should return the list itself. For example, invoking `fromListLevelOrder [1..10]` will return the following tree: