TEXAS A&M UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSCE 314: Programming Languages

Homework Assignment 5

Objective

In this assignment you'll work with more advanced data types. Also there is another problem for which higher-order functions can be fruitfully employed.

Submission

- Like before, this homework is will be graded as a 0, 1, or 2 for each question. (The questions are: "Tree traversal", "Sets and Set products", "Set partitions, and Bell Numbers"). With 0 = inadequate or no attempt; 1 = solid effort showing a good attempt at all questions; 2 = faultless/exemplary submission. You may find it helpful to discuss your homework with the peer teachers, TAs, or instructors (say, at their office hours).
- Prepare a **single** Haskell file with a `.hs` extension.
- *Recommended*: Include as a comment at the top of your code: your name and UIN.
- The file should contain functions named precisely as described below. Those functions must have proper type signatures and should follow exactly the ones that have been explicitly given. In addition, you may include any number of additional helper functions you like; you may not include any extra modules, packages, or external libraries.
- Additional documentation in the form of comments throughout your code is strongly encouraged. If you have used outside resources, document these directly in your source file.
- You are encouraged to use the class slack to communicate about/solicit help/ask questions clarifying aspects of the assignment.
- Submission will be via Canvas, due: 10 November 2021.

Instructions

- Some of the questions ask you to produce attractive looking output, this simply involves producing the appropriate string. GHCI will output it; if you're trying to use `putStr` then you're barking up the wrong tree.

Tree traversal

Consider the following data type.

```
data Tree a b = Branch b (Tree a b) (Tree a b)
              | Leaf a
```

Displaying trees

Make `Tree` an instance of `Show`. Do not use `deriving`; define the instance yourself. Make the output look somewhat nice (e.g., indent nested branches). For example, if you define `mytree` as

```
mytree = Branch "A" (Branch "B" (Leaf (1::Int)) (Leaf (2::Int))) (Leaf (3::Int))
```

when you input `mytree` in the command line, the screen outputs something like

```
"A"
      "B"
            1
            2
      3
```

Traversing trees

Traverse the tree in the given order with a corresponding Haskell function, which collects the values from the tree nodes into a list:

```
preorder  :: (a -> c) -> (b -> c) -> Tree a b -> [c]

postorder :: (a -> c) -> (b -> c) -> Tree a b -> [c]

inorder   :: (a -> c) -> (b -> c) -> Tree a b -> [c]
```

Note: The values in the tree cannot be collected to a list as such because the values on the leaves are of a different type than the values on the branching nodes. Thus each of these functions takes two functions as arguments: The first function maps the values stored in the leaves to some common type `c`, and the second function maps the values stored in the branching nodes to type `c`, thus, resulting in a list of type `[c]`.

Sets and Set products

The following problems are to implement mathematical sets and some of their operations using Haskell lists. A set is an *unordered* collection of elements (objects) without duplicates, whereas a list is an *ordered* collection of elements in which multiplicity of the same element is allowed.

We do not define a new type for sets, but instead define `Set` as a *type synonym* for lists as follows:

```
type Set a = [a]
```

Note: if your type definitions have the extra class constraint: (Ord a) => ... for any of the solutions given below, that is acceptable too.

Even though the types `Set a` and `[a]` are the same to the Haskell compiler, to the programmer they communicate that values of the former are sets while the values of the latter are arbitrary lists.

Set constructor

Write a recursive function that constructs a set.

```
mkSet :: Eq a => [a] -> Set a
```

Constructing a set from a list simply means removing all duplicate values.

Subset

Write a recursive function `subset`, such that `subset set1 set2` returns `True` if `set1` is a subset of `set2` and `False` otherwise.

```
subset :: Eq a => Set a -> Set a -> Bool
```

Set equality

Using `subset` you have already defined, write a function `setEqual` that returns `True` if the two sets contain exactly the same elements, and `False` otherwise.

```
setEqual :: Eq a => Set a -> Set a -> Bool
```

Set product

The product of two sets **A** and **B** is the set consisting of all pairs draw from either set, where the pairs are ordered having elements **(a_i, b_j)**. The first element is from **A** and the second from **B**.

```
setProd :: (Eq t, Eq t1) => Set t -> Set t1 -> Set (t, t1)
```

Here's an example:

```
> setProd [1,2,3] ['a', 'b']
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
> setProd [1..4] ['a', 'b', 'c']
[(4,'c'),(4,'b'),(4,'a'),(3,'a'),(3,'b'),(3,'c'),(2,'a'),(2,'b'),(2,'c'),(1,'a'),(1,'b'),(1,'c')]
```

Set partitions, and Bell Numbers

This question follows directly on from your previous one; you should use your `Set` type for this. (It is separated because it is sufficiently challenging to be worth points on its own.)

Set partition

> **Note:**
> This is the hardest part of the assignment. Like most recursive functions, it is helpful to work out several of the cases by hand. Start from the smallest problems and build up, until you see that pattern. Also, you may find Section 6.6 of the Haskell textbook has some quite helpful general advice here.

The partition of a set **S** is defined as a set of nonempty, pairwise disjoint subsets of **S** whose union is **S**. For example, the set `{red, green, blue}` can be partitioned in 5 ways:

```
{ {red}, {green}, {blue} }
{ {red}, {green, blue} }
{ {green}, {red, blue} }
{ {blue}, {red, green} }
{ {red, green, blue} }.
```

Write a Haskell function to compute the partition of any set provided as input. (The colors are just to help you see the pattern; your code isn't expected to produce the colored output.)

```
partitionSet :: Eq t => Set t -> Set( Set (Set t))
```

Computing Bell numbers

The Bell number $\mathbf{B_n}$ is the number of partitions of a set of size **n**. Use your previous answer to write a function that computes the Bell number for any non-negative **n**. (Note that the standard defintion of the numbers declares that both $\mathbf{B_0 = B_1 = 1}$.)

```
bellNum :: Int -> Int
```

Here's an example:

```
> bellNum 5
52
> bellNum 1
1
```

Acknowledgement

Tree traversal: Dr. Dylan Shell and Dr. Hyunyoung Lee's previous CSCE 314 assignments