Exercise (Week 5)

DUE: Tue 2 July 14:00:00

Getting Started

CSE Stack

Download the exercise tarball and extract it to a directory on your local machine. This tarball contains a file, called `Ex03.hs`, wherein you will do all of your programming.

To test your code, run the following shell commands to open a GHCi session:

```
$ stack repl
Configuring GHCi with the following packages: Ex03
Using main module: 1. Package 'Ex03' component exe:Ex03 ...
GHCi, version 8.2.2: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Ex03             ( Ex03.hs, interpreted )
Ok, one module loaded.
*Ex03> quickCheck prop_mysteryPred_1
...
```

Calling `quickCheck` in the above way will run the given QuickCheck property with 100 random test cases.

Note that you will only need to submit `Ex03.hs`, so only make changes to that file.

QuickCheck and Search Trees

The file `Ex03.hs` includes the support code described in the following as well as stubs for the three functions that you must implement. The QuickCheck properties discussed below are also included.

We include a binary tree implementation, plus a predicate, `isBST`, which returns `True` if a tree is a binary *search* tree (that is, an infix traversal of the tree is sorted). This is a *data invariant* for our BST operations.

```
data BinaryTree = Branch Integer BinaryTree BinaryTree
                | Leaf
                deriving (Show, Ord, Eq)

isBST :: BinaryTree -> Bool
isBST Leaf = True
isBST (Branch v l r) = and [ allTree (< v) l, allTree (>= v) r
                           , isBST l, isBST r ]
  where allTree :: (Integer -> Bool) -> BinaryTree -> Bool
        allTree f (Branch v l r) = f v && allTree f l && allTree f r
        allTree f (Leaf) = True
```

We also include `insert` and `deleteAll` functions for binary search trees:

```
-- Add an integer to a BinaryTree, preserving BST property.
insert :: Integer -> BinaryTree -> BinaryTree
```

```
-- Remove all instances of an integer in a binary tree,
-- preserving BST property
deleteAll :: Integer -> BinaryTree -> BinaryTree
```

An arbitrary generator that generates binary search trees:

```
searchTrees :: Gen BinaryTree
```

If we wanted to check that our generator always generates wellformed inputs, we can check it by running the additional property:

```
prop_searchTrees = forAll searchTrees isBST
```

We use the arbitrary search tree generator rather than prefix our properties with a guard like `isBST tree ==>` to prevent QuickCheck generating lots of spurious test cases.

Implementing A Mystery Function (Part 1a, 2 Mark)

Write a predicate function `mysteryPred`, which has the following type signature:

```
mysteryPred :: Integer -> BinaryTree -> Bool
```

It must satisfy the following QuickCheck properties:

```
prop_mysteryPred_1 integer
  = forAll searchTrees $ \tree ->
      mysteryPred integer (insert integer tree)

prop_mysteryPred_2 integer
  = forAll searchTrees $ \tree ->
      not (mysteryPred integer (deleteAll integer tree))
```

You can test your `mysteryPred` implementation by simply running `quickCheck prop_mysteryPred_[1-2]` in the playground or `GHCi`.

Even more mysterious (Part 1b, 3 Marks)

Write a function mysterious, with the following type signature:

```
mysterious :: BinaryTree -> [Integer]
```

It must satisfy the following QuickCheck properties:

```
prop_mysterious_1 integer
  = forAll searchTrees $ \tree ->
      mysteryPred integer tree == (integer `elem` mysterious tree)

prop_mysterious_2 = forAll searchTrees $ isSorted . mysterious

isSorted :: [Integer] -> Bool
isSorted (x:y:rest) = x <= y && isSorted (y:rest)
isSorted _ = True
```

You can test your mysterious implementation by simply running `quickCheck prop_mysterious_[1-2]` in a playground or GHCi.

Depending on your exact definition, this could be an abstraction function for binary search trees. Consider how our binary tree operations like `insert` and `deleteAll` could be specified using *data refinement*.

Balancing Act (Part 2, 4 Marks)

First, we have the generator:

```
sortedListsWithoutDuplicates :: Gen [Integer]
```

We use a generator to produce `sortedListsWithoutDuplicates` because guarding our properties with `isSorted list ==>` etc. means that QuickCheck will waste a lot of time randomly generating lists and checking if they satisfy the guard. We can check if our generator works with the following properties:

```
prop_sortedListsWithoutDuplicates_1
  = forAll sortedListsWithoutDuplicates isSorted
prop_sortedListsWithoutDuplicates_2
  = forAll sortedListsWithoutDuplicates $ \x -> x == nub x
```

Write a function `astonishing`, of type `[Integer] -> BinaryTree` that satisfies these properties:

```
prop_astonishing_1
  = forAll sortedListsWithoutDuplicates $ isBST . astonishing

prop_astonishing_2
  = forAll sortedListsWithoutDuplicates $ isBalanced . astonishing

prop_astonishing_3
  = forAll sortedListsWithoutDuplicates $ \ integers ->
      mysterious (astonishing integers) == integers
```

Where `isBalanced` is a predicate that returns true if a tree is *height-balanced*:

```
isBalanced :: BinaryTree -> Bool
isBalanced Leaf = True
isBalanced (Branch v l r) = and [ abs (height l - height r) <= 1
                                , isBalanced l
                                , isBalanced r
                                ]
  where height Leaf = 0
        height (Branch v l r) = 1 + max (height l) (height r)
```

Note: This definition of `isBalanced` is a rather generous definition of balanced trees. Can you find an example of a tree that is not very balanced for which this function would still return `True`? What would a stricter predicate look like?

Submission instructions

You can submit your exercise by typing:

```
$ give cs3141 Ex03 Ex03.hs
```

on a CSE terminal, or by using the `give` web interface. Your file *must* be named `Ex03.hs` (case-sensitive!). A dry-run test will *partially* autotest your solution at submission time. To get full marks, you will need to perform further testing yourself.