CIS 194: Homework 9

Due Friday, 7 November

- Files you should submit: `HW09.hs`.

Testing Ring properties

QuickCheck gives us the amazing power to do randomized testing. This means that all you, the programmer, have to do is to write down general *properties* you wish to test for, and you can have QuickCheck do the actual test case generation. In our case, you will be testing to make sure that `Ring` instances (provided, in `Ring.hs`, available from the website) obey the ring properties.

Exercise 1 If you want to test the rings, you will need `Arbitrary` instances for `Mod5` and `Mat2x2` so that QuickCheck can create arbitrary values of these types. Check out the documentation for the `Arbitrary` class. You will see the method `arbitrary :: Gen a` — this is the one you must implement. See the part of the documentation page labeled "Generator combinators". These will be useful. Most importantly, note that `Gen` is a monad! So, you can start with `arbitrary = do ...` and go from there.

http://hackage.haskell.org/package/QuickCheck-2.7.6/docs/Test-QuickCheck.html

Write `Arbitrary` instances for `Mod5` and `Mat2x2`.

Hint: `Integer` has both `Arbitrary` and `Random` instances already.

Exercise 2 (Optional: $\frac{1}{2}$pt. extra credit) Implement the `shrink` method for `Mat2x2` by reading the `shrink` documentation.

Exercise 3 Visit the Wikipedia page for rings, at http://en.wikipedia.org/wiki/Ring_(mathematics). A little bit down the page, you'll see 8 properties that all rings should have. Encode these properties into Haskell, in a form suitable for `quickChecking`. For compatibility with our automated testing, name these `prop_1` through `prop_9`.[1] (Normally, you'd use better names than that.)

[1] Nine properties? Yes, nine. Although Wikipedia shows eight bullets, there are actually nine properties embedded within that list.

Make sure that your properties work with `quickCheck` by running, say, `quickCheck prop_1` in GHCi. If you want to test something other than `Integer`s (the default), you'll need to add a type annotation, like `quickCheck (prop_1 :: Mat2x2 -> Mat2x2 -> Bool)`, though your `prop_1` may need more or fewer arrows.

Exercise 4 Write a property to rule them all, named `prop_ring`, that checks to see if all the ring properties hold. There are many ways to do this, some cleaner than others. Take a look at the documentation for `Test.QuickCheck` (available at `http://hackage.haskell.org`) to find useful combinators, like `conjoin` and `.&&.`. Note that the types for these combinators pose a bit of a challenge.

It is also possible to write `prop_ring` without using these combinators, but it is neither as beautiful nor as fun. (You will get full correctness points for such an implementation, but not full style points. But remember, you have a chance to revise style!)

Exercise 5 One of the rings as defined in `Ring.hs` is broken. Use your tests to find which one and write up your discovery in comments in your code.

Generating binary search trees

Take a look at `BST.hs`, ripped straight from HW04.

Exercise 6 The `isBSTBetween` and `isBST` functions there were written without the benefit of type classes. Copy their definitions (along with the datatype definiton for `BST`) to your code. Remove the first parameter to both functions (the one of type `a -> a -> Ordering`), and instead, add an `Ord a` constraint to both, and get them working again.

Exercise 7 Write an `Arbitrary` instance for `BST a` that creates proper binary search trees. Recall that, in a BST, a left sub-tree has values less than the value stored in a node, and a right sub-tree has values greater than the value stored in a node. A basic implementation of `arbitrary` could proceed as follows:

- Write a helper function `genBST :: (...) => a -> a -> Gen (BST a)` (where you'll have to fill in the `...`). The two parameters are the lower and upper bounds of the tree. The function does the following:

  1. First, it decides whether it will make a leaf or an interior node. This can be done, say, by generating an arbitrary `Bool`, though there are other `Gen` combinators that make this more elegant.
  2. If you wish to make a leaf, do so.
  3. Otherwise, generate a value of type `a` in the range given by the parameters. Call this value `x`.
  4. Generate a tree bounded above by `x` via a recursive call; this will be your left sub-tree.

5. Generate a tree bounded below by `x` via a recrusive call; this will be your right sub-tree.
6. Put `x` together with the left and right sub-trees in a `Node`, and you're done.

- Back in the implementation for `arbitrary`, generate arbitrary lower and upper bounds (making sure that the lower bound is indeed less than the upper one!) and call `genBST`.

You can test your `arbitrary` by using the `generate` and `sample` functions in the QuickCheck library. You may need a type annotation for this to work out. For example, type `sample (arbitrary :: Gen (BST Integer))`.

Exercise 8 (Optional: $\frac{1}{2}$pt. extra credit) If you follow the implementation plan above, you may notice that roughly half of the generated trees are `Leafs`. This is because the first step (depending on your implementation) chooses to make a `Leaf` half of the time.

Modify your implementation to make a greater variety of trees.

Good starting points for this are the `Arbitrary` instance in the `BST.hs` file (which does *not* make trees in order) and the example in the lecture notes for `MyList`.

Testing parsers

There are `Parsable` instances in `Ring.hs`. These are harder to test using property-based tests, so we'll use old-fashioned unit tests using HUnit.

Exercise 9 Write a test

```
parserTests :: Test
parserTests = TestList [ ... ]
```

that includes at least two tests for each `Parsable` instance. Make sure to give descriptive names to each test using `(~:)`. You may find it convenient to use the `parseAll` function, which parses assuming that the entire string is consumed — that is, it checks to see that there are no leftovers after parsing.

Note: To test the `Integer Parsable` instance, you will probably need a type annotation, to make clear that you mean `Integer` and not, say, `Int`. For example, you may have code something like `parseAll "3" ~?= Just (3 :: Integer)`.