Homework 6: Generics

This assignment will exercise your knowledge of generic interfaces,covariance and bounded quantification.

Preliminaries

You should create a directory-tree that looks like this:

```
./generics
├── project
│   └── plugins.sbt
└── src
    ├── main
    │   └── scala
    │       └── Solution.scala ........................ Your solution goes here
    └── test
        └── scala ..................................... Yours tests go here
```

The `project/plugins.sbt` file must have exactly this line:

```
addSbtPlugin("edu.umass.cs" % "compsci220" % "1.0.2")
```

The support code for this assignment is in the package `hw.generics`.

Programming with Bounded Quantification

The trait `hw.generics.ListLike` is a trait for "list-like" collections. (i.e., collections that are either *empty* or have a *head* and *tail*.) The class `hw.generics.MyList` is a typical list that implements the `ListLike` trait.

1. In `Solution.scala`, create the following type for binary trees:

   ```
   sealed trait BinTree[A]
   case class Node[A](lhs: BinTree[A], value: A, rhs: BinTree[A]) extends BinTree[A]
   case class Leaf[A]() extends BinTree[A]
   ```

   Furthermore, have `BinTree` extend `ListLike`. The *head* of a binary-tree is the value on the left-most node and the *tail* of a binary-tree is the tree with the left-most value removed.

2. In `Solution.scala`, create an object called `ListFunctions` with the following functions:

   (a) Create a function `filter(f, alist)` where `alist` is a list-like collection and `f` is a predicate that can be applied to elements in the list. The function should produce a new list-like collection with the same type as `alist` that only contains the elements on which `f` produces `true`.

   i.e., this filtering function should behave exactly the same as Scala's filtering function.

   (b) Create a function `append(alist1, alist2)`, where `alist1` and `alist2` are two list-like collections of the same type. The result should be a new list-like collection (with the same type as `alist1` and `alist2`) that has the elements of `alist1` followed by the elements of `alist2` in order.

   i.e., on linked lists, `append(alist1, alist2)` should behave in a manner simlar to `alist1 ++ alist2`.

   (c) Define a function that sorts in ascending order with the following name and type:

   ```
   def sort[A <: hw.generics.Ordered[A], C <: hw.generics.ListLike[A, C]](alist: C): C = {
     ...
   }
   ```

You should test applying these functions to the provided `MyList` type and to the `BinTree` type that you defined. You should also be able to apply it any other new data structure that extends the `ListLike` trait.

Extending Traits

The package `hw.generics` defines three traits `T1`, `T2`, and `T3` that define exactly the same set of methods, but have a different set of type-parameters. In `Solution.scala`, create the following classes:

```
class C1 {
  def f(a: Int, b: Int): Int = 0
  def g(c: String): String = ""
  def h(d: String): Int = 0
}


class C2 {
  def f(a: Int, b: Int): Int = 0
  def g(c: Int):  Int = 0
  def h(d: Int): Int = 0
}

class C3[A](x: A) {
  def f(a: Int, b: A): Int = 0
  def g(c: A): String = ""
  def h(d: String): A = x
}

class C4[A](x: Int, y: C4[A]) {
  def f(a: Int, b: C4[A]): C4[A] = b
  def g(c: Int): C4[A] = y
  def h(d: C4[A]): Int = x
}
```

Moreover, have these classes extend as many of the traits traits `T1`, `T2`, and `T3` as possible. Some classes may be able to extend several traits. **You should not modify the class bodies.**

Hand In

From the `sbt` console, run the command `submit`. The command will create a file called `submission.tar.gz` in your assignment directory. Submit this file using Moodle.

For example, if the command runs successfully, you will see output similar to this:

```
Created submission.tar.gz. Upload this file to Moodle.
[success] Total time: 0 s, completed Jan 17, 2016 12:55:55 PM
```

Note: The command will not allow you to submit code that does not compile. If your code doesn't compile, you will receive no credit for the assignment.

Template

You can use the following template for `Solution.scala`:

```
import hw.generics._

sealed trait BinTree[A]
case class Node[A](lhs: BinTree[A], value: A, rhs: BinTree[A]) extends BinTree[A]
case class Leaf[A]() extends BinTree[A]

object ListFunctions {
  // def filter(f, alist)
  // def append(alist1, alist2)
  def sort[A <: Ordered[A], C <: ListLike[A, C]](alist: C): C = ???
}

class C1 {
  // Do not change the class body. Simply extend T1, T2, and/or T3.
  def f(a: Int, b: Int): Int = 0
  def g(c: String): String = ""
  def h(d: String): Int = 0
}
```

```scala
class C2 {
  // Do not change the class body. Simply extend T1, T2, and/or T3.
  def f(a: Int, b: Int): Int = 0
  def g(c: Int):  Int = 0
  def h(d: Int): Int = 0
}


class C3[A](x: A) {
  // Do not change the class body. Simply extend T1, T2, and/or T3.
  def f(a: Int, b: A): Int = 0
  def g(c: A): String = ""
  def h(d: String): A = x
}

class C4[A](x: Int, y: C4[A]) {
  // Do not change the class body. Simply extend T1, T2, and/or T3.
  def f(a: Int, b: C4[A]): C4[A] = b
  def g(c: Int): C4[A] = y
  def h(d: C4[A]): Int = x
}
```