CSci 555-01: Functional Programming
Assignment #2, Spring 2019

H. Conrad Cunningham

25 February 2019

Contents

Assignment #2

Revised Deadline Monday, 4 March, 11:59 p.m.] (Original deadline Thursday, 28 February)

General Instructions

All homework and programming exercises must be prepared in accordance with the instructions given in the Syllabus. Each assignment must be submitted to your instructor by its stated deadline.

Citations: In accordance with expected scholarly and academic standards, if you reference outside textbooks, reference books, articles, websites, etc., or discuss an assignment with individuals inside or outside the class, you must document these by including appropriate citations or comments at prominent places in your submission such as in the header of the primary source file.

Identification: Put your name, course name, and assignment number as comments in each file you submit.

Assignment Description

1. This is an individual assignment.
2. When complete, submit your Scala source code files to the course Blackboard site for Assignment #2.
3. Be sure to document your source code appropriately using program comments. Give attention to the general instructions given above and in the Syllabus.

   Consider documenting the source code with preconditions, postconditions, termination arguments, and time and space complexity.
4. Do Exercise set A below. A student taking the course for undergraduate credit may omit one exercise.
5. Do Exercise set B below. A student taking the course for undergraduate credit may omit one exercise.
6. Test your program thoroughly.

Note: Exercise Set A below consists of the first six items in Exercise Set A in the notes on "Functional Data Structures". Exercise Set B below is the same as Exercise Set B in that set of notes.

Exercise Set A

In the following exercises, extend the `List2.scala` algebraic data type implementation developed in the notes on "Functional Data Structures" to add the following functions. In the descriptions below, type `List` refers to the trait defined in that package, not the standard Scala list.

1. Write a Scala function `orList` that takes a `List` of `Boolean` values and returns the logical `or` of the values (i.e. true if any are true, otherwise false).
2. Write a Scala function `andList` that takes a `List` of `Boolean` values and returns the logical `and` of the values (i.e. true if all are true, otherwise false).
3. Write a Scala function `maxList` that takes a nonempty `List` of values and returns its maximum value.

   Hint: First solve this with `Int`, then generalize to a generic type. Study the subsection on insertion sort in this set of notes.
4. Write a Scala `remdups1` that is like `remdups` except that it is implemented using either `foldRight` or `foldLeft`.

5. Write a Scala function `total` that takes a nonnegative integer `n` and a function `f` of an appropriate type and returns `f(0) + f(1) + ... f(n)`.

6. Write a Scala function `flip` that takes a function of polymorphic type `(A,B) => C` and returns a function of type `(B,A) => C` such that, for all `x` and `y`:

```
f(x,y) == flip(f)(y,x)
```

General Tree Algebraic Data Type

A *general tree* is a hierarchical data structure in which each node of the tree has zero or more subtrees. We can define this as a Scala algebraic data type as follows:

```
sealed trait GTree[+A]
case class Leaf[+A](value: A) extends GTree[A]
case class Gnode[+A](gnode: List[GTree[A]]) extends GTree[A]
```

An object of class `Leaf(x)` represents a *leaf* of the tree holding some value `x` of generic type `A`. A leaf does not have any subtrees. It has height 1.

An object of type `Gnode` represents an *internal* (i.e. non-leaf) node of the tree. It consists of a nonempty list of subtrees, ordered left-to-right. A `Gnode` has a height that is one more than the maximum height of its subtrees.

Exercise Set B

In the following exercises, write the Scala functions to operate on the `GTree`s. You may use functions from the extended `List` module as needed.

1. Write Scala function `numLeavesGT` that takes a `GTree` and returns the count of its leaves.
2. Write Scala function `heightGT` that takes a `GTree` and returns its height (i.e. the number of levels).
3. Write Scala function `sumGT` that takes a `GTree` of integers and returns the sum of the values.
4. Write Scala function `findGT` that takes a `GTree` and a value and returns `true` if and only if the element appears in some leaf in the tree.
5. Write Scala function `mapGT` that takes a `GTree` and a function and returns a `GTree` with the structure but with the function applied to all the values in the tree.
6. Write Scala function `flattenGT` that takes a `GTree` and returns a `List` with the values from the tree leaves ordered according to a left-to-right traversal of the tree.