Homework: Self-referential data

COMP 50

Fall 2013

This homework is due at 11:59PM on **Monday, October 7**.
Submit your solutions in a single file using the COMP 50 Handin button on DrRacket; the homework is the `trees` homework.

All the exercises should be done using the Beginning Student Language *with list abbreviations*.

In this homework, all lists are made with `empty` and `cons` as described in the book. Selector functions for the `cons` case are `first` and `rest`. For testing, you may use the `list` function to make lists.

Special data description for this homework

In the first problem of this homework you will be locating stations on the Northeast Corridor railway line. Boston is the northern terminus, and we will be locating stations by measuring their distance (south) from Boston. You can get real-world information about stations on the Wikipedia page for the Northeast Corridor.

A *station* is a structure:

```
(make-station name distance)
```

where `name` is a string and `distance` is a number representing the distance in miles from Boston.

A *1D-tree of stations* is either:

- A station
- A structure (representing a boundary point on the line)

  ```
  (make-boundary distance south north)
  ```

  where

  - `distance` is the distance of the boundary point from Boston
  - `south` is a 1D-tree of stations, all of which are south of the boundary (that is, they are further away from Boston)
  - `north` is a 1D-tree of stations, all of which are north of the boundary (that is, they are closer to Boston than the given `distance`)

Finger Exercises

For this homework I am recommending the following finger exercises:

- In the first edition, Exercises 9.5.2, 9.5.6, 10.1.2, 10.2.2 (but using strings instead of symbols), 10.2.4, 10.2.5, 11.3.5, 12.2.1, and 12.2.2.
- Define a function that computes how many stations there are in a 1D-tree of stations.
- Define a function that computes the depth of a 1D-tree of stations.

Problems to submit

1. *Define a function* that takes as arguments a distance in miles and a 1D-tree of stations, and which returns information saying the name of the nearest station, how far away it is, and in what direction. You must *write a data definition* that says how the information you return will be represented.

   Use the design recipe for self-referential data (natural recursion). In addition, plan on writing some helper functions that operate on distances and stations. Use your wish list!

   Hint: For purposes of testing, you will do well to define a function that takes a tree and two numbers `lo` and `hi`, and checks the given tree to be sure that

   - All stations have distances between `lo` and `hi`
   - For every boundary in the tree, the stations south and north of that boundary are located at distances that are consistent with the location of the boundary

   Use this function to make sure that your test trees are well formed.

2. Two fisherman go out in a boat and use nets to pull a whole bunch of bluegills out of the Mystic Lakes. They agree to divide the catch evenly. You are given two buckets and two scales and are charged with placing the fish into the buckets, dividing evenly *by weight*. A friend of Mr Turing's whispers to you that dividing as evenly as possible might take more time than you have. The fishermen agree that your time is worth something, and they will be content if you place one fish at a time in the fairest way possible.

   To prove to the fishermen that your division of fish is reasonably fair, you will use "computational buckets". A computational bucket not only holds a list of fish but also contains a readout that instantly gives the total weight of the fish in the bucket.

   Write a data definition for a pair of computational buckets. The only important thing about a fish is its weight.

   Define a function that takes as arguments a list of fish and returns a pair of computational buckets such that each bucket contains about the same weight of fish as the other bucket.

   Use the design recipe for self-referential data (natural recursion).

   It turns out that the fairness of your division is affected by the order in which you consider the fish. Please create a random list of 30 fish and *run three experiments*:

   - Divide the fish
   - Sort the list of fish in *decreasing* order of weight, then divide them
   - Sort the list of fish in *increasing* order of weight, then divide them

   How do the experiments affect the fairness of the division?

   A convenient way to report on the experiments is to produce a list of three strings, as in the following example:

```
> (three-experiments (N-random-fish 30))
'("When divided, randomly ordered fish give buckets that differ by [REDACTED]"
  "When divided, increasing fish give buckets that differ by [REDACTED]"
  "When divided, decreasing fish give buckets that differ by [REDACTED]")
```

Hints:

- To sort the fish, you will want to use the insertion sort defined in section 12.2 of the first edition textbook. You can develop a similar sort function that sorts in increasing order.
- If you are not sure how to create a list of N random fish, have a look at the treatment of natural numbers in Section 11 of the first edition.

Karma Problem

A. If a 1D-tree is *balanced*, there is a very efficient way to locate stations quickly. I will give you the idea with an example:

- Suppose you are looking for a point 150 miles from Boston, and you find a boundary located at 180 miles from Boston. Your first step should be to look north of the boundary for the closest station. If that station is closer than 30 miles, *you don't have to look south of the boundary*. Why? Because every station south of the boundary is at least 30 miles away.

Using this insight, define a function that finds the closest station *without* always looking at all stations.

(To test such a function, you can create a 1D-tree in which certain stations are replaced by the value `'hole`, and you can check to make sure that you can find the nearest station without ever looking at a hole. You can also use `check-error` to confirm that your original code *does* look at holes.)

Later in the term we will see how to build such balanced trees, and I hope we will look at 2D-trees, which can be used to locate nearby cities quickly.