Homework: Multiple lists

COMP 50

Fall 2013

This homework is due at 11:59PM on **Tuesday, October 15** (because Monday is a holiday).
Submit your solutions in a single file using the COMP 50 Handin button on DrRacket; the homework is the `multiple-lists` homework.

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

As in the previous 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, and you may also use quotation and quasiquotation.

Overview

There are only three problems on the homework. Do not be deceived; it is a big homework.

1. The first problem asks you to combine a demonstration from lecture (all stations on a railway) with ideas from Section 17 (Processing Two Complex Pieces of Data). It is relatively easy.

2. The second problem asks you for a *safe* function that finds the nearest station on a railway, even in the presence of zombies. The *efficient*, logarithmic-time function for finding the nearest station will do for this problem. If you understood the sketch in class, the problem is relatively easy. If not, get yourself to office hours right away. Either way, you get to use any code you like from my *inefficient* solution.

   The second problem demands *stringent* testing.

   If you find it hard to make progress on the second problem, *move on to the third problem*, which is more important.

3. The third problem is a `big-bang` problem for building a graphical editor. This problem also asks you to work with two lists. The problem can be solved in a number of ways; the key to finding a solution is to be very clear about the meaning of your data description. I have given you a "starter" description to begin with, but you'll need to refine it.

   This problem is challenging, but when you complete it, you'll have a very good idea how editor widgets work, including the widgets you use on the search box or location box of your web browser.

The apocalyptic railway

A zombie apocalypse is destroying the Northeast Corridor. To handle a zombie apocalypse, we need multiple data definitions:[1]

- A *station* is a structure:

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

[1] Given the right data definitions, we can handle anything. Zombies? No problem.

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

- A *horde-of-zombies* is a structure:

  ```
  (make-zombies distance)
  ```

  where `distance` is a number representing the station's distance in miles from Boston. (Because it is not safe to get close enough to count them, nobody knows how many zombies there are in a horde.)

- A *stop* is either:
  - A station
  - A horde-of-zombies

- An *apocalyptic-railway* is either:
  - A stop
  - 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 an apocalyptic-railway, all of whose stops are south of the boundary (that is, they are further away from Boston)
    * `north` is an apocalyptic-railway, all of whose stops are north of the boundary (that is, they are closer to Boston than the given `distance`)

  Please note that the data definition for *apocalyptic-railway* uses definition by choices, and one of those choices (*stop*) is itself *another* definition by choices. The appropriate template for railway functions will be based on a `cond` with *two* cases, one of which may either call a helper function or may be a nested `cond`.

- A *point-on-the-line* is a number representing the point's distance in miles from Boston.

From the previous homework you may have helper functions that involve stations. You may use those helper functions, together with their signatures, purpose statements, examples, and tests. *You may also copy helper functions from my solution,* provided you copy the examples and tests as well.

Finger Exercises

In the textbook, I am recommending these finger exercises:

- Exercises 12.2.2, 14.1.1, 14.1.3, 14.1.5, 14.2.3, 17.1.1, 17.2.2, 17.3.1, 17.4.1, and 17.4.2

And in addition, I recommend these finger exercises:

i) *Define a function* `stop?` that tells whether a value represents a *stop* according to the data definition.

ii) *Define a function* that takes as argument an apocalyptic-railway and returns the list of all *stations* on the railway, in order of distance from Boston with the southernmost station first.

iii) *Define a test function* that takes as argument a list of stations and returns a Boolean saying whether the distances in the list are decreasing.

iv) *Define a function* that takes as argument an apocalyptic-railway and returns the list-of-numbers containing all *distances* mentioned on the railway, in decreasing order with the greatest distance first.

The distances must include

- Distances of stations
- Distances of zombie hordes
- Distances of boundaries

v) *Define a test function* that takes as argument a list of numbers and returns a Boolean saying whether the numbers are decreasing.

Reminder: Solutions to the book exercises are on the web. You and your classmates may share solutions *to finger exercises only*, including my exercises above.

Problems to submit

1. Define a function `same-stations?` that takes as arguments *two* apocalyptic-railways and returns a Boolean saying whether those railways have exactly the same stations. Ignore boundary points and zombie hordes.

   Do *not* use a sorting function; in particular, do not use insertion sort. If you do, you will earn No Credit for this problem.

   Hints: Revisit Section 17 (Processing Two Complex Pieces of Data) and see what case applies. Use the results of the finger exercises.

2. Georgia Mason has stolen a train and wants to drop her brother Shaun on the apocalyptic railway. Stations are occupied by strong people with guns, and zombies cannot cross them—so any point between two stations is safe. But a government conspiracy has shut down the GPS system, and Georgia's only guides are the mileage markers on the railway line. Dropping Shaun near a horde of zombies is a *fatal* error.

   Define a function `safe-nearest-station` that takes as arguments a point-on-the-line and an apocalyptic-railway, such that calling the function has one of two outcomes:

   - If the point on the line is between two stops, and both of the stops are stations (so zombies can't reach Shaun), the function returns the nearest stop.
   - If the nearest stop on the line is a horde of zombies, or possibly if the nearest stop is a station but there is a horde of zombies on the other side, the function calls

     ```
     (error "Shaun was killed and eaten.")
     ```

     and does not return a value.

   Idea of the problem: If Georgia drops Shaun between two stations, you can find the nearest station without ever looking upon a zombie.

   Details of the problem: The function you define must obey a complicated set of rules. Depending on the location of the point and the structure of the apocalyptic-railway, there are three kinds of situations:

   - In some situations the function must return a station.
   - In some situations the function must call `error`.
   - In some situations the function may do one or the other.

   In all situations, looking at a horde of zombies causes instant death.

   Here are the rules in detail:

   - If the point falls on a station (that is, the point's distance from Boston is the same as the station's distance from Boston), the function must return that station.

- If the point falls between two stations, the function must return the nearest station.
- If the point falls on a horde of zombies, the function must call `error` with the message given above.
- If the point falls between two hordes of zombies, the function must call `error` with the message given above.
- If the point falls between a station and a horde of zombies, and the horde is closer, the function must call `error` with the message given above.
- If the point falls between a station and a horde of zombies, and the station is closer, the function may return the station, or it may call `error` with the message given above.
- If the function is given an apocalyptic-railway that is only a horde of zombies, it must call `error` with the message given above.
- If the function is given an apocalyptic-railway that is only a station, it must return that station.

The net effect of all these rules is that *it is death even to **look** at a horde of zombies.* You must find a way to write the function so that if the point falls between adjacent stops that are stations, it never looks at a horde of zombies.

To earn full credit for this problem, you must check every required behavior. You will use both `check-expect` and `check-error`.

Hint: This problem is basically the same as the 1D-tree problem from the previous homework, except you mustn't look at all the stops (because if you look at zombies, Shaun will be eaten). You will need new data examples, but you can probably reuse some helper functions and their tests. You are welcome to use any of my code from the solution to the previous homework.

Hint: Review Section 14 of the first edition.

How to avoid zombies: If you find yourself exactly on a boundary point, you have to look at both sides. Otherwise, you're on one side of the boundary, which I'll call the *near side*; the other side of the boundary is the *far side*. Find the closest stop on the near side. If that stop is closer than any stop on the far side can possibly be, you don't even have to look on the far side. Not looking on the far side (except when necessary) is the secret to avoiding zombies.

3. The final problem is to build a one-line graphical editor widget such as you would find in a web browser or other search tool. We're following the model explained in the section A Graphical Editor in the second edition of *How to Design Programs*. That section describes a `big-bang` program that acts as an editor for a single line of text:

   - You will build an editor that shows black text and a red cursor on a background empty scene of 300 by 40 pixels. For the cursor, use a tall, thin red rectangle. For the text, use black text of size 24.
   - Use the following data description:
     - A list-of-1strings is either:
       * `empty`
       * `(cons c cs)`, where `c` is a one-character string and `cs` is a list-of-1strings
     - An *editor* is a structure:

       ```
       (make-editor left right)
       ```

       where `left` and `right` are both lists-of-1strings. The `left` list contains the text that appears to the left of the cursor; the `right` list contains the text that appears to the right of the cursor.
   - The editor responds to keystrokes as follows:
     - The keystroke `"escape"` causes the editor to stop running, returning its current contents as a string. (When the editor stops, the position of the cursor is thrown away.)
     - The keystroke `"left"` moves the cursor one character to the left (if possible).
     - The keystroke `"right"` moves the cursor one character to the left (if possible).
     - Any other keystroke that is represented by a string of length 2 or more is ignored.
     - The keystroke `"\b"` (backspace) deletes the character to the left of the cursor, if any.

- The keystrokes `"\t"` (tab) and `"\u007F"` (rubout) are ignored.
- Any other keystroke that is represented by a string of length 1 causes that string to be inserted to the left of the cursor.

Complete the following parts of this problem:

A. *Finish the data definition for "editor"* so that it says
   - *Exactly* how the strings in the `left` and `right` lists are *ordered*
   - How the two lists of strings relate to the text as it appears on the screen

B. *Write a data definition* for the world-state of a graphical editor.

C. Using the design guidelines for `big-bang` programs, write a function that takes as argument a string representing the initial contents to the editor, that calls `big-bang` to implement the editor, and that when the editor is finished, returns the final (edited) string.

Hint: The hard parts of this problem are the *data description* of the editor and the *keyboard-event handler*. I strongly recommend that you **build your program one kind of keyboard event at a time**. For example,

- Begin with the `"escape"` key, so that all your program does is stop when it is asked to stop.
- Continue with the code to insert keyboard events of string length 1
- Continue by ignoring tab and rubout characters
- Continue by implementing the `"left"` keyboard event
- Continue by implementing the `"right"` keyboard event
- Finish by making sure that events like `"up"` and `"down"` are ignored

If you use helper functions and `check-expect`, and if you stop and run your `big-bang` function at each stage, you will make steady progress. If you try to build the whole thing at once, before testing any of it, you may as well just find a brick and hit yourself in the head with it—the net effect will be the same.

Racket library knowledge: the BSL function `explode` converts a string to an equivalent list of characters. Function `implode` does the opposite conversion. Function `string-length` tells the number of characters in a string, which represents a keyboard event.

Karma problem

A. *Define a function* whose input is a *nonempty* list of *stops*, sorted by distance from Boston with the southernmost stop first, and returns an apocalyptic-railway containing the same stops. Be sure that the *depth* of the apocalyptic-railway is at most one more than the base-2 logarithm of N, where N is the number of stops.

Warning: This function goes beyond natural recursion. Try it only if you want to stretch your brain.

Math: You can define the base-2 logarithm as follows:

```
(define (log2 x)
   (inexact->exact (/ (log x) (log 2))))
```

Hint: A stop is also an apocalyptic-railway, and therefore a list of stops is also a list of apocalyptic-railways.

Hint: It may help to adapt the `southmost` and `northmost` functions designed in class.

How your work will be evaluated

For *every* function we expect to see results that are consistent with a systematic approach to program design:

- A signature that is consistent with the header
- A purpose statement that goes beyond what's in the signature and ideally is sufficient to predict results with any valid input
- Test cases that exercise every possibility from the data description of the input(s)
- Test cases that **pass**, i.e., a complete absence of failing tests
- Code we understand that is derived from a template we recognize

The most common mistakes I have seen on previous homeworks are

- Very complex templates
- Complex code used to fill in templates

Avoid these mistakes!

Unless you can fill in the `...` parts of the template with code that is very simple indeed, **define helper functions.** I know it is a lot of work to write all those signatures, purpose statements, and tests, but it is the only way for anyone to *know* that your code is right. I have seen too much code that sort of almost worked—and in every case, the problem was too much complexity in the function bodies.

Additional things we will be looking for:

1. In problem 1 we will be looking for code that follows one of the patterns identified in Section 17 (Processing Two Complex Pieces of Data).
2. In problem 2 we will be looking for *comprehensive* test cases. There is one case in which `safe-nearest-station` may either return a station or may call `error`. That outcome can't be tested. But in every other case, `safe-nearest-station` *must* do one or the other, and the rules always say which. **Every one of these testable cases must be tested.**

   We will look for you to use the long form of `check-error`, which says exactly what error message is expected:

   ```
   (check-error (...)  "Shaun was killed and eaten.")
   ```

3. In problem 3, here are the additional things we will be looking for:
   - Please write clear and comprehensive data descriptions, with examples, for *editor* and *world-state*, including descriptions that explain the significance of the *order* in which strings appear.
   - Please lay out of the elements of the solution in this order:
     a) Constants that describe unchanging elements of the program which need not be included in the world-state
     b) Data descriptions, including the world-state
     c) A main function that calls `big-bang`
     d) Handlers and other auxiliary functions
   - Please make sure that when we click `Run`, *your code does not call `big-bang`.* All calls to `big-bang` must occur inside functions.
   - Please provide tests for *every* function except the one that calls `big-bang`.
   - For each event handler, please provide a specific and precise purpose statement supported by examples and tests. Your purpose statement and tests should be sufficient to give us complete confidence in the handler's correctness.