CS61A Summer 2010

George Wang, Jonathan Kotker, Seshadri Mahalingam, Steven Tang, Eric Tzeng

Homework 2

Due: Tuesday, July 6 2010, at 7AM

Note: The number of stars (★) besides a question represents the estimated difficulty of the problem: the more the number of stars, the harder the question.

1 How to Start and Submit

First, download `http://inst.eecs.berkeley.edu/~cs61a/su10/hw/hw2.scm` and use it as a template while filling out your procedures. See `http://www-inst.eecs.berkeley.edu/~cs61a/su10/hw-faq.pdf` for submission instructions.

2 Big-Theta: What is the Runtime?

We strongly recommend you do this section individually. This is because analyzing the runtime of a procedure yourself may be challenging, but if somebody tells you the solutions, you will have a very easy time convincing yourself that it is correct. We want you to make sure you are able to derive the answers on your own, and the easiest way you can be assured of that is to do this individually.

For the following pieces of code, please give a big-Theta runtime in terms of n, where n is the input to each function. When we talk about runtimes, we are primarily concerned with the *number of recursive calls.* This is because the number of operations is almost always going to be some constant multiple of the number of recursive calls. Remember that if a procedure calls a helper procedure, take the time that it takes to call that helper function into account.

Note, the return value is *not* what we are concerned with here. In other words, it does not matter what the procedure does; for this exercise, determine how long the procedure takes.

1. (★)

```
(define (one n) n)
```

2. (★)

```
(define (foo n)
  (if (= n 0)
      0
      (foo (- n 1))))
```

3. (★★)

```
(define (bar n)
  (if (or (> n 10) (< n 0))
      n
      (bar (+ n 1))))
```

4. (★★)

```
(define (mystery n)
  (cond ((< n 0) 0)
        ((odd? n) (+ 2 (mystery (- n 2))))
        (else (+ 1 (mystery (- n 1))))))
```

5. (★★)

```
(define (foofoo n)
  (if (= n 0)
      0
      (+ (foo n) ; defined as above
         (foofoo (- n 1)))))
```

6. (★)

```
(define (baz n)
  (define (bar a b)
     (if (>= b a)
         0
         (bar (/ a 2) b)))
  (bar n 1))
```

7. (★★)

```
(define (foobaz n)
  (if (= n 0)
      0
      (+ (baz n) ; defined as above
         (foobaz (- n 1)))))
```

3 Lists

1. (★) Write a procedure `insert-front`, which adds an element to the front of a list. It should take two arguments: the element to be added, and the list to add the element to. This should be easy.

   For example:

```
> (insert-front '4 '(2 4 2 1))
(4 2 4 2 1)
```

2. (★★) Now, code `insert-back`, which inserts an element to the back of a list. It should take the same arguments that `insert-front` does. Notice how much harder this is? You should keep this list asymmetry in mind as we move from sentences to lists.

   For example:

   ```
   > (insert-back '4 '(2 3 2 1))
   (2 3 2 1 4)
   ```

4 Deep Lists

1. (★★) Write a procedure `substitute` that takes three arguments: a list, an old word, and a new word. It should return a copy of the list, but with every occurrence of the old word replaced by the new word, even in sublists. For example:

   ```
   > (substitute '((lead guitar) (bass guitar) (rhythm guitar) drums) 'guitar 'axe)
   ((lead axe) (bass axe) (rhythm axe) drums)

   > (substitute '(((((hello))))) 'hello 'goodbye)
   (((((goodbye)))))
   ```

2. (★★) Now write `substitute2` that takes a list, a list of old words, and a list of new words; the last two lists should be the same length. It should return a copy of the first argument, but with each word that occurs in the second argument replaced by the corresponding word of the third argument.

   ```
   > (substitute2 '((4 calling birds) (3 french hens) (2 turtle doves))
                  '(1 2 3 4) '(one two three four))
   ((four calling birds) (three french hens) (two turtle doves))
   ```

   Hint: You may find the `position` and `list-ref` procedures helpful. `position` takes in an element and a list, and returns the index at which the element appears in the list. `list-ref` takes a list and an index, and returns the item at that index of the list. Note that list indices start at `0`, so the first element of a list has index `0`. However, you do not have to use these procedures.

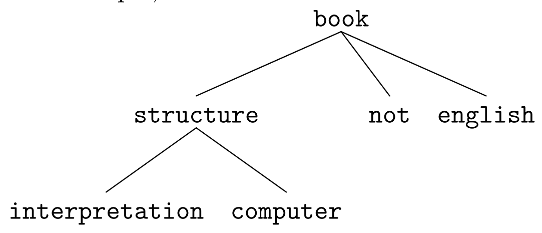
5 Trees

For this section, remember to use the Tree Abstract Data Type that we are studying. As a reminder, here are the relevant functions:

```
(make-tree DATUM CHILDREN) ; Constructor
(datum TREE)               ; Accessor
(children TREE)            ; Accessor
```

1. (★★) Write a function `dfs-find` that accepts two arguments: a Tree and a predicate. `dfs-find` should return the first element, according to a depth-first traversal, that satisfies the predicate.

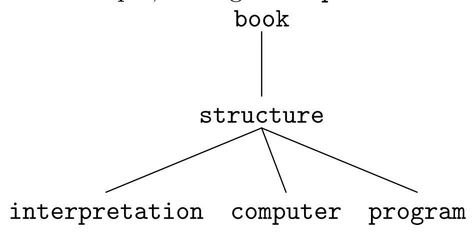
   For example, if `Tree` were defined as



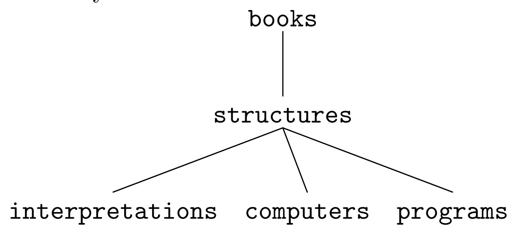
   then `(dfs-find begins-with-vowel? Tree)` should yield `interpretation`. (Note that `begins-with-vowel?` is not a predicate already available in our version of Scheme.) If the tree does not contain any element that satisfies the predicate, the call to `dfs-find` should return `#f`. You may assume that the tree does not already contain `#f`.

2. (★) Write `tree-plural`, a procedure that takes in a Tree of English nouns, and outputs a Tree where every noun has been pluralized. You should make use of the `plural` procedure that you wrote in lab 1a.

   For example, calling `tree-plural` on

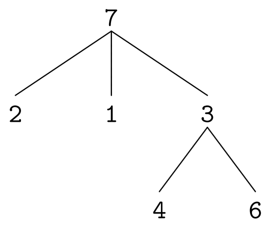


   should yield

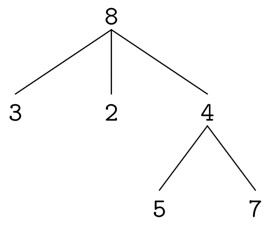


3. (★★) Generalize `tree-plural` to `tree-map`, which takes a Tree and a procedure and returns a copy of the Tree, where every item has been replaced with the result of calling the procedure on the original item.

   For example, `(tree-map 1+ Tree)`, where `Tree` is defined below:



should yield



4. (★★★) Write `has-path-sum?`, a procedure that takes a Tree and a number and returns `#t` if there is a path from the root to a leaf, such that the sum of the nodes along the path (including the root and leaf) total up to the number. [1]

6 Binary Trees

For this section, remember to use the Binary Tree Abstract Data Type that we are studying. Note that to signify that a node has no child, we use an empty list.

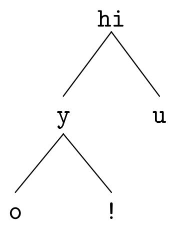
As a reminder, here are the relevant functions:

```
(make-tree DATUM LEFT-CHILD RIGHT-CHILD) ; Constructor
(datum TREE)                             ; Accessor
(left-branch TREE)                       ; Accessor
(right-branch TREE)                      ; Accessor
```

1. (★★) Write `tree-max` and `tree-min`, functions that, when given a Binary Tree, return the largest element or the smallest element, respectively. You may use the `max` and `min` functions and assume that the Tree is composed of only numbers.

2. (★★★) Generalize these functions into a tree-accumulate function, which takes in a function and a Binary Tree. You should write two versions of this procedure: `tree-accumulate-inorder`, which performs an in-order accumulation of the tree and `tree-accumulate-preorder`, which performs a pre-order accumulation of the tree. Note that for non-commutative functions, you get different results!

   If `Tree` is the following tree:



   then we should have

[1] Pulled from a Stanford exercise. This is article #110 in the Stanford CS Education Library. This and other free CS materials are available at the library (`http://cslibrary.stanford.edu/`). That people seeking education should have the opportunity to find it. This article may be used, reproduced, excerpted, or sold so long as this paragraph is clearly reproduced. Copyright 2000-2001, Nick Parlante, `nick.parlante@cs.stanford.edu`.

```
> (tree-accumulate-preorder word Tree)
hiyo!u
> (tree-accumulate-inorder word Tree)
oy!hiu
```

7 Case Study: Geopositioning via Wi-Fi Signals

7.1 Acknowledgements

This was adapted from a Stanford Programming Assignment: `http://see.stanford.edu/materials/icsppcs107/34-Assignment-7-Where-Am-I.pdf`, with modifications made by course staff members Eric Tzeng and Hoa-Long Tam.

7.2 Background

Many of you probably have a device that can access Wi-Fi signals, be it a laptop or your phone. What you may not know is that these signals can also be used to locate you geographically. Apple products use the Wi-Fi signals of nearby Wi-Fi hotspots to locate where they are in addition to GPS. This technology is developed on a large scale by Skyhook Wireless. I have done my best to guess at how their system works, and for this programming assignment, your job will be to develop a working prototype of such a system.

For simplicity, we will make a few assumptions about the signal. First, we assume that the signal degrades linearly and radially at a rate of ten percent per meter. In other words, if we are ten meters away from a signal source, we will be unable to receive the signal that it is producing, and if we are receiving a signal that is 50% strong, we are about 5 meters away from the hotspot. We will further assume that all signal sources transmit equally strong signals.

Each signal from a hotspot is uniquely associated with a *MAC Address*. We may think of these MAC addresses as "names" for each signal. What the computer receives is a list of these names and distances. We will use a database of where these hotspots are geographically to determine the location of the computer.

7.3 Plan of Attack

Since we know that the signal degrades radially, we can assume that when our computer receive a signal's MAC address and signal strength, we deduce that the computer is approximately on a circle whose center is determined by the location of the signal source and whose radius is determined by the signal strength. All the points on this circle are receiving that signal with that particular signal strength. Once we do this with multiple signals, all the signals should intersect at a point, if our calculations are perfect. However, due to small measurement errors, we should assume instead that the circles will intersect in a small clump near our actual location. All we have to do is find out where this clump is and thus conclude our location.

Take a moment to familiarize yourself with the three abstract data types defined: points, rated points, and circles. These are defined in the `hw2.scm` template file provided. In all of the questions below, remember to **respect the data abstraction**.

1. (★) An *association list* is a list of pairs, each of whose `car` is a key and whose `cdr` is the associated value. An example of this is your database of MAC addresses and locations, where a MAC address is a key and the location is the corresponding value.

   To obtain a circle from a name and a value for signal strength, query the database using `assoc` with the name to find the center of the circle. Then, calculate how far you are from the signal using the signal strength.

Write a function `find-circle` that returns a circle, given a pair whose `car` is a MAC Address and whose `cdr` is the strength of the signal, represented as a decimal number from 0 to 1.

```
> (assoc '48-2C-6A-1E-59-3D database)
(48-2c-6a-1e-59-3d 0 0)

> (find-circle (cons '48-2C-6A-1E-59-3D 0.2))
(8.0 (0 0))
```

2. (★) Write a function `intersection-points` that takes a list of circles and returns a list of all the points where the circles intersect. Use the `intersect` function provided, which takes two circles and returns a list of the points where the circles intersect. Each circle in the list may intersect with each of the other circles in the list. You can be sure to obtain all the possibilities by intersecting the first circle in the list with all of the ones to the right. Then intersect the second circle with all the ones to its right. Repeat this process until there is only one circle left in the list. N circles should yield at most $(N^2 - N)$ points of intersection. The list of points may contain duplicates; in fact, duplicates are good—they're affirmation that your measurements aren't totally hokey.

```
> (intersection-points (list (make-circle 1 (make-point 0 0))
                             (make-circle 1 (make-point 1 0))))
((0.5 0.8660254037844386) (0.5 -0.8660254037844386))
> (intersection-points (list (make-circle 1 (make-point 0 0))
                             (make-circle 1 (make-point 1 0))
                             (make-circle 1 (make-point 1 1))))
((0.5 0.8660254037844386) (0.5 -0.866025437844386))
 (2.7755575615628914E-16 1.0) (1.0 2.7755575615628914E-16)
 (0.1339745962155614 0.5) (1.8660254037844386 0.5))
```

3. (★) Write a function `distance-product` that takes a point and a list of points and returns the product of the distances between that point and each of the points in the list. The point itself may be in the list. In that case, it should be removed so that it doesn't force the product to be zero. You may use the `dist` function provided, and you may assume the list contains at least two different points.

```
> (distance-product (make-point 2 0) (list (make-point 0 0)
                                           (make-point 2 0)
                                           (make-point 6 0)))
8
> (distance-product (make-point 3 3) (list (make-point 2 5)
                                           (make-point 7 8)
                                           (make-point 10 1)
                                           (make-point 3 2)))
104.23531071570709
```

Rather than using `car-cdr` recursion, use `map` and a `lambda` expression wrapping around the `dist` function you have been given to compute a list of distances, and then take the product. You will need to cope with the problem where the base circle may be in the circle list, leaving a `0` where you would rather have a `1`. You can overcome this problem in a number of ways, and any one of them is more than acceptable.

4. (★★) The next step is to rate how far each intersection point is from all the other intersection points. To do this, write a function `rate-points` which takes a list of points and returns a list of rated points, where each point is rated with its distance-product from the other points. Use `map` with an appropriate `lambda` expression to rate the points.

```
> (rate-points (list (make-point 0 0)
                     (make-point 2 0)
                     (make-point 6 0)))
((12 (0 0)) (8 (2 0)) (24 (6 0)))
```

```
> (rate-points (list (make-point 2 5)
                     (make-point 7 8)
                     (make-point 10 1)
                     (make-point 3 2)))
((164.92422502470643 (2 5)) (320.22492095400696 (7 8))
 (481.66378315169186 (10 1)) (161.24515496597098 (3 2)))
```

5. (★★) We will need a function called `sort-points` that will take a list of rated points, and sort them in ascending order of rating.

```
> (sort-points (rate-points (list (make-point 2 5)
                                  (make-point 7 8)
                                  (make-point 10 1)
                                  (make-point 3 2))))
((161.24515496597098 (3 2)) (164.92422502470643 (2 5))
 (320.22492095400696 (7 8)) (481.66378315169186 (10 1)))
> (sort-points (rate-points (list (make-point 0 0)
                                  (make-point 2 0)
                                  (make-point 6 0))))
((8.0 (2 0)) (12.0 (0 0)) (24.0 (6 0)))
```

You will do so using insertion sort, which you were introduced to in the last homework assignment. The `hw2.scm` template contains the `insertion-sort-general` procedure you were asked to write in the last homework assignment. The included version has been altered to work with lists rather than sentences. Use this updated version of `insertion-sort-general`, in combination with a comparator that you will write, to write the `sort-points` procedure.

6. (★★★) The points with small distance ratings tend to be in a clump, while the points with large distance ratings tend to be out by themselves. Half of the points will be clumped and the other half will be spread out. To isolate the points in the clump, use the above functions to rate and sort the points, and then just extract the first half of the list. If the list is of odd length, the extra element should be discarded. Using the provided `first-n-points` procedure, write a function `clumped-points` which takes a list of points, rates them, sorts them, and then returns the half of the points with the smallest ratings. `clumped-points` should return the points without the ratings.

```
> (clumped-points (list (make-point 0 0)
                        (make-point 2 0)
                        (make-point 6 0)))
((2 0))
> (clumped-points (list (make-point 0 0)
                        (make-point 2 0)
                        (make-point 6 0)
                        (make-point 1 0)))
((1 0) (2 0))
```

7. (★★) The next step is to take the clumped points and average them together. The function `average-point` should take a list of points and average them all down to a single point. The average point is obtained by averaging all the x-values to get an x-value and all the y-values to get a y-value. `average-point` should also include the distance rating indicating how far the average point was from all the points. A small distance rating is small implies that the points were in a clump. You should use the `let` construct to avoid computing the average point twice.

```
> (average-point (list (make-point 0 0) (make-point 2 0) (make-point 6 0)))
(5.925925925925926 (2.6667 0))
> (average-point (list (make-point 0 0) (make-point 2 0)
                       (make-point 6 0) (make-point 1 0)
                       (make-point 5 4) (make-point 4 5)))
```

```
(590.8865213418864 (3 1.5))
```

8. (★) Building on all of the functions so far, define the function `find-location` that takes a list of signal strength measurements, turns them into circles, computes all the points of intersection, winnows those points down to those which are most clumped, and returns their average point. Note, average point returns a rated point, which you will then have to extract the single point from.

```
>(find-location (list (cons '48-2C-6A-1E-59-3D 0.2) (cons '2C-1E-54-A2-84-12 0.4)
                      (cons '82-1E-52-3B-DC-24 0.3) (cons 'A3-C1-5D-73-12-9A 0.292)))
(5.01174019404359 4.94809451778074)
```

In this case, the implication is the circles essentially intersect around the point $(5, 5)$. Note that the numbers can vary somewhat between implementations due to rounding error, but they should all be approximately the same.

8 Feedback

Now that you are done, please leave me some feedback at the following link regarding how the course is going. This is not worth points, but will give me valuable feedback that in turn improves your course experience. Thanks! `https://spreadsheets.google.com/viewform?formkey=dHY4eDNaUjR5VjBIRHlEWDlsa0tqMmc6MA`