# Homework: More Abstraction, Trees, and Lists

COMP 50

Fall 2013

This homework is due at 11:59PM on **Monday, November 18**. Submit your solutions in a single file using the COMP 50 Handin button on DrRacket; the homework is the `loops` homework.

All the exercises should be done using the Intermediate Student Language with lambda.

### How to tackle this homework

This homework has a lot of parts. I want to call your attention to where I think the challenges are.

- Problem 1 asks you to write parametric data definitions and use them to describe examples. Once you understand parametric data definitions, all these parts should be easy. If you need to review the concepts or notation, look at Section 19.2 in the first edition textbook.
- Problems 2 and 3 should present only a mild challenge: you need to be able to apply the design recipe for self-referential data to binary-search trees, and in Problem 3A you need to find a suitable abstract list function. The key is to remember that *there is one template for binary trees*. If you understand and follow that template, all three problems will go smoothly.

  Take advantage of data examples from previous assignments.
- Problem 4 is just a collection of list problems like the ones you did in the hospitals lab. At this stage you should find them very easy, except that you may have to think a little bit about how to write the signatures.
- Problem 5 may present a bit of a challenge. The key idea is one that we haven't emphasized in lecture: a `local` function has *direct* access to the parameters of its enclosing function. The goal is to simplify signatures, headers, and call sites by eliminating parameters to which the `local` function has direct access.

  Pay attention during the process, because I ask you for your opinion about the results.

  Take advantage of data examples and tests from previous assignments.
- There are three karma problems aimed at further abstraction of the 2D-tree search. The most interesting aspect is being able to use a single function for both Euclidean distance (on the plane) and GPS distance (on the surface of a sphere).

## Data definitions

A `(bst X)` is either of

- `false`
- A structure `(make-node left key value right)`
  where `key` is a string, `value` is an X, `left` is a (bst X) in which all `key`s are *less than* this key, and `right` is a (bst X) in which all `key`s are *greater than* this key.

A `(setof X)` is either of

- `empty`
- `(cons x xs)` where `x` is an `X` and `xs` is a `(setof X)` not containing `x`.

# Finger exercises

The first group of finger exercises from the book are basic exercises in the topics needed for this homework. The last three finger exercises are there in case you are not sure of your ground; in that case, tackle them with a study partner or guidance of a TA. And definitely post your results to Piazza so you can be sure you are on the right track.

- Exercises 19.2.4, 20.2.2 (by "contract" the problem means "signature"), 20.2.4, 21.1.1, 22.2.1, 22.2.2
- *Write a parametric data definition* of a "simplified binary-search tree" in which each internal node contains one value (of unknown type) and two subtrees. Such a tree can be used to represent a set of values efficiently.
- *Define a function* to search a simplified binary-search tree. Pay close attention to its signature and purpose statement. Test it using values of two different types.
- Define a function `bst?` which is given a BSL value and tells whether it is a binary search tree that respects all the relevant less-then and greater-than properties. Do not worry about efficiency.

# Problems to submit

## Parametric data definitions and polymorphic signatures

I remind you of the rules for type variables:

- If two different values with unknown data definitions must belong to the *same* unknown data definition, then these values must be referred to using a *single* type variable.

  Conversely, every appearance of a single type variable always refers to a value of from the *same* unknown data definition.

- If two different values with unknown data definitions *may* belong to *different* unknown data definitions, then these values must be referred to using two *different* type variables.

  Conversely, if a data definition or signature uses two different type variables, then in actual use the two type variables *may* refer to different data definitions, but they need not—they could also refer to the same data definition.

Solve all parts of the following problem:

1. *Writing and applying parametric data definitions*

   A. After a visit to a corn maze, Professor Hescott writes a maze-searching function that will run on a robot. This function is given a predicate and returns one of two kinds of results:

      - A value satisfying the predicate
      - An explanation indicating why no such value could be found

An explanation always includes string, such as "I looked everywhere but the maze does not contain the object you wanted" or "I could not find my way through the entire maze" or "Part of the maze was closed for construction".

*Write a parametric data definition* that describes the results returned by this function. Be sure that if Professor Hescott sends the robot to look for a string containing both consonants and vowels, that there is some way to tell whether such a string has been found.

B. Write a *parametric* data definition for a list containing an even number of elements, all of which are the same kind of data.

Examples:

```
'(10 20 30 40 50 60)
'("Hi" "there" "I'm" "sorry" "about" "Greasy")
```

Non-examples:

```
'(1 "fish" 2 "fish" red "fish" blue "fish")
'("Hi" "there" "I'm" "sorry" "about" "the" "frog")
```

C. Using your parametric definition from the previous problem, describe the data of each of the examples.

D. Write a *parametric* data definition for a list containing an even number of elements in which their are two possibly different kinds of data and elements alternate.

Examples:

```
'(one "fish" two "fish" red "fish" blue "fish")
'("Hescott" 7 "Ramsey" 8 "Guyer" 4)
'(lecture ("M" "W") lab ("W" "Th") main-office ("M" "T" "W" "Th" "F"))
'(10 20 30 40 50 60)
```

Non-examples:

```
'(1 "fish" 2 "fish" red "fish" blue "fish")
'(ol (li "Homework") (li "Lab") (li "Portfolio))
```

E. Using your parametric definition from the previous problem, describe the data of each of the examples.

F. The quote notation can easily be used to define a form of "association lists", which is a list of lists. Each inner list contains two elements: a key and a value.

Examples:

```
'((one "fish") (two "fish") (red "fish") (blue "fish"))
'(("Hescott" 7) ("Ramsey" 8) ("Guyer" 4))
'((lecture ("M" "W")) (lab ("W" "Th")) (main-office ("M" "T" "W" "Th" "F")))
'((10 20) (30 40) (50 60))
```

Write a *parametric* data definition to describe this form of association list.

G. Using your parametric definition from the previous problem, describe the data of each of the examples.

## Binary search trees

Binary search trees will play a significant role in your final homework (identify what language a web page is written in). In all the problems below, We will pay special attention to **signatures** and **purpose statements**.

2. *Adding information to a binary search tree*
Solve all three parts:

A. *Key properties*
*Design a function* `every-key?` which tests to see if every *key* in a binary search tree satisfies a given predicate.

B. *Adding information to a binary search tree*
*Design a function* `insert` which takes a key, a value, and a binary search tree. It returns a new binary search tree that is exactly like the given tree, except that in the new tree, the given key is associated with the given value. If the original tree contains the given key, then the new tree should have the same number of nodes as the original.

C. *Test* the results of `insert` to be sure that insert always returns a true binary search tree that respects ordering properties.

3. *Converting between trees and lists*.

A. **Without using recursion**, write a function that takes an association list with string-valued keys and returns the corresponding binary search tree. Assume that the list contains no duplicate keys.
We will pay special attention to your *signature* and *purpose statement*.

B. *Design a function* that converts a binary search tree to an association list. Your function should make sure that for any node in the tree, the key-value pair of that node always *precedes* the key-value pairs of the nodes in its two subtrees.
It may help you name the function to know that the resulting list is called a *preorder listing* of the tree

C. *Design another function* that converts a binary search tree to an association list. Your function should make sure that for any node in the tree, the key-value pair of that node always *follows* the key-value pairs of the nodes in its two subtrees.
It may help you name the function to know that the resulting list is called a *postorder listing* of the tree

D. *Design a third function* that converts a binary search tree to an association list. Your function should make sure that in the association list, keys appear in increasing order.
It may help you name the function to know that the resulting list is called an *inorder listing* of the tree

E. For long-term-storage, trees are written out to disk as a sequence of characters. It should be possible to recover the tree exactly.
*Demonstrate by testing* that if you run *one* of the three tree-to-list functions, then convert the list back to a tree using your function from part (a), then you will recover the original tree *exactly*.

## More standard abstract functions on lists

4. Using the built-in abstract functions for list processing, solve the following parts. In this problem of the assignment, **do not use any recursion**.

A. *Define a function* `is-in?` that tests to see if a number is in a set of numbers.

B. *Define a function* `set-insert` that inserts a number into a set of numbers. That is, if number `z` is inserted into set `zs`, the resulting value is a set that contains all the elements of `zs` and also contains `z`.

C. *Define a function* `set-difference` that computes the difference between two sets of numbers. (The difference of two sets contains all elements that are in the first set but *not* in the second.)

D. *Define a function* `set-intersection` that computes the intersection of two sets of numbers. (The intersection of two sets contains all element that are in *both* sets.)

E. *Define a function* `set-union` that computes the union of two sets of numbers. (The union of two sets contains every element that is in either set.)

### Improving code through abstraction and local definitions

In this section you use local definition and abstraction to improve my solution to the `nearest-point` problem (Problem 5 on the abstractions homework).

5. My solution to `nearest-point` uses a number of helper functions. *Without* trying to combine `maybe-above-or-below` and `maybe-left-or-right`, improve my solution as follows:

   - If any helper function receives an argument that is identical to any argument of `nearest-point`, *make the helper function `local`*.
   - In each `local` helper function, *remove arguments that are also arguments to `nearest-point`*. (A `local` helper function can already "see" those arguments; for examples, look at Section 22.2 in the first-edition textbook.)

     You will also need to

     – Remove the superfluous arguments from all the call sites
     – Repair the signatures and purpose statements of the local helper functions

   Does this *refactoring* improve the solution overall? Why or why not? In a comment, *write your opinion*.

   **How to solve this problem without making yourself crazy**:

   - Start by finding a helper function that is called *only* from `nearest-point`, and move *that* function into a `local` definition. Remove that function's test cases. Press Run and make sure all the other test cases still work.
   - Examine the function you just moved and see what arguments are identical to arguments of `nearest-point`. Take them out of the function definition, and take them away from any place the function is called.

     Again press Run and make sure all tests pass.

     Now update the signature and purpose statement.
   - Now find *another* helper function that is called only from `nearest-point` or from one of `nearest-point`'s *local* helper functions. Make *that* function local. Run all the tests.
   - Take the function you just moved and eliminate its superfluous arguments. Eliminate superfluous arguments from call sites.

     Now press Run again.

     Now update the signature and purpose statement.
   - Continue with each helper function: first move it, then remove its superfluous parameters, the fix its signature and purpose statement. Press Run every time. Your new motto is "Run insanely often."
   - When you finish, go back and check all the signatures and purpose statements, in case you forgot one. (Tragically, Run does not help with signatures and purpose statements.)

   Slow is fast: if you are methodical and you press Run at each step of the process, the only things you'll really have to think about are which parameters are superfluous. Everything else will just work.

   If you're greedy and you try to do big chunks at once, you could get lucky… or you could create a mess that you cannot debug.

   When you finish, *don't forget to write your opinion.*

## Karma Problems

Some abstractions can go too far:

X. In my solution to `nearest-point`, the functions `maybe-above-or-below` and `maybe-left-or-right` are nearly identical. Please *write a signature, purpose statement, and header* that answer these two questions:

   i) If the nearly identical functions were replaced by a single abstraction, what additional parameter(s) would have to be passed?

   ii) If the nearly identical functions were replaced by a single abstraction, what would be its purpose statement?

   Is this abstraction worth it? Why or why not? In a comment, *write your opinion*.

It would be pleasant to use 2D-tree search directly on the USGS points-of-interest dataset. But the USGS uses GPS coordinates, not Euclidean coordinates. Address this deficiency by solving the following two problems:

Y. Copy your improved solution from problem 6 and use it to implement a function `nearest-point-gps`, in which the `x` coordinate is longitude *in degrees* and the `y` coordinate is latitude *in degrees*. Your computation of distance will have to change:

   - If two points have the same latitude $y$ but different longitudes $x_1$ and $x_2$, then the distance between them is *approximately* $(x_2 - x_1) \cos\ y$.[1] You will want to define a helper function that takes the cosine of an angle in degrees.
   - If two points have the same longitude $x$ but different latitudes $y_1$ and $y_2$, the distance between them is $y_2 - y_1$. This distance is exact.
   - If two points have different latitudes and longitudes, you can approximate the distance between them as the square root of $(y_2 - y_1)^2 + (x_2 - x_1)^2(\cos\ y)^2$, where $y$ is the average of $y_1$ and $y_2$.

Z. Abstract over your two 2D-search functions to *define a single higher-order function* that takes as arguments functions that compute distances and returns as result an efficient 2D-search function.

   Take care with the signature and purpose statement.

   Demonstrate your solution by

   i) Instantiating the function for Euclidean distance and showing it computes the same results as my original

   ii) Instantiating the function for GPS distance and showing it computes the same results as your modified code

   Is the abstraction worth it? Why or why not? In a comment, *write your opinion*.

*Hint for testing*: A point 1.2 degrees west of Boston is actually closer to Boston than a point 1.0 degrees north of Boston.

[1] The formula above gives the distance traveling along a line of latitude. As you get further from the equator, the shortest route loops away from the equator, and the actual distance is shorter.