

CS245 Lab 1: Data Structures in C++ and Scheme

Due Wednesday of the 4th week of classes (23 Sept 2009); handed out 3rd Tuesday.

This lab is designed to introduce you to data structures in C++ and Scheme, via the projects `Tree-C++` and `Tree-Scheme`. In each, you will create a data structure to represent n-ary trees: trees with any number of child nodes and a value (of type `String`) in each node. Your representation should be based on a list of nodes, since an n-ary tree just has a value and a (possibly empty) list of children.

For both projects, you should have:

- constructor(s) to create trees (you are free to make your simplest case either an empty tree or a tree with one element, but your choice should be clear in your code or comments).
- predicate(s) to test for simple (empty or one-element) vs. larger trees
- two accessor functions to process trees (similar to the “process” function for queues):
 - a function `map_tree` that takes (a) a tree and (b) a function that maps a string to a string, and then computes a new tree with the same structure as the tree argument, but in which each string is replaced by the result of applying the function to the node’s string; and
 - a function `reduce_tree` that takes (a) a tree and (b) a function that maps a string and a list of strings to one result string, and then computes a single string value for the tree by calling the function with the tree root’s value and a list of the function’s results on the subtrees.
- a test program that builds a non-trivial example tree in which some nodes have strings of letters and others strings of digits, and then produces a single string giving the concatenation of all the strings of digits — your program could compute this string in the following two ways, and check to make sure it gets the same string in either case
 - combine a simple map and a simple reduce: call `map_tree` with a function that takes a string and returns it if it is made of digits or returns an empty string if it isn’t; then use `reduce_tree` on the resulting tree with a function that just concatenates strings;
 - use only reduce but with a more problem-specific function — call `reduce_tree` with a function that takes a list of strings and returns the concatenation of those that are strings of digits.

(note that the function `is_numeric` in `hc_string.h` returns true if a string is made up of digits; for the purposes of this lab, you may pretend that the standard Scheme function `string->number` also does this, though that is not exactly true).

You do not have to create any “mutator” functions (functions that modify a parameter), though you are allowed to do so if you know how and would like to. (In other words, use the “pure functional” style of the queue data types, in which we had separate functions to return the head of a queue, or the remainder of the queue without the head, rather than a single operation to return the head after modifying the queue to remove it.

You should obtain the two projects (through the CVS repository exploring perspective in Eclipse, or command-like use of CVS if you prefer to use that), and do the following:

Tree-C++:

- Create a class `n_ary_tree` to represent n-ary trees as an abstract data type. Create the appropriate files for this class, and add a demonstration/test to `main.cc`.
- Run your test to make sure your program works.
- Draw a memory evolution diagram showing the first seven function calls of your call to `map_tree`. You may draw on the computer with inkscape, gimp, or whatever you like, but please create a PDF file to be sure I can read it, or just draw this by hand and give me the paper.

Tree-Scheme:

- Practice working with Scheme's built-in list data type by doing the following:
 1. In the file `length.scm`, write a scheme function `list-length` that gives the number of elements in the "top level" of a list — in other words, your function should return 3 for both `(list-length '(1 2 3))` and `(list-length '(1 (2.1 2.2 2.3) 3))`.
 2. In the file `length.scm`, write a scheme function `list-width` that gives the number of non-list elements anywhere in a list or sublists of the list — in other words, your function should return 3 and 5 if applied to the examples above. Remember that `null?` returns true for the empty list (i.e., for `()` but not for `(1 3)`), and `list?` returns true for any list (i.e., for `()` and `(1 3)` but not for 12 or "text").
 3. Include examples (such as those above) to show your functions work.
- Create operations to allow manipulation of n-ary trees, and the demonstration/test.
- Run your test to make sure your functions work.

You do **not** have to do a version of this lab in HERA. Phew.

REMEMBER to submit **both** projects using Team->Commit when you're done.