Computer Science & Engineering 171
Assignment #3: Functional Programming

Due: June 6th at 6:00 pm

In this assignment, you will write programs in ML and Scheme. Use the Rocky 9 machines in the Engineering Computing Center for this project. The ML interpreter is `sml` (use "`setup smlnj`" first) and the Scheme interpreter is `guile3.0`. See the "Assignment 3" post on Camino for more details on how you can execute your SML and Scheme code files.

The practicum will be done in class in teams of two or three. Your score will be based on your participation. The homework is to be done individually and submitted using Camino. You score will be based on correctness and coding style. The practicum and the homework are each worth 5 points.

1 Practicum

1.1 Quicksort

On Camino you will find an implementation in ML of ***quicksort***, the sorting algorithm used previously. Translate the program to Scheme, keeping the same value bindings, function names, and parameters.

Goal: To introduce functional programming in strongly-typed and dynamically-typed languages.

Hints: You will need to use `let*` in Scheme to achieve the same effect as `let` in ML.

1.2 Binary Search Trees

As in the first assignment, a ***binary search tree*** is either empty, or it consists of a node with two binary search trees as subtrees. Each node holds an integer. The elements in a binary search tree are arranged so that smaller elements appear in the left subtree of a node and larger elements appear in the right subtree. In ***both*** ML and Scheme, write a program to implement binary search trees. Each program should have at least the following two functions:

- `insert`: given a binary search tree and an integer, inserts the integer into the tree, and returns the new tree; if the integer has already been inserted, then the tree is unchanged and the tree itself is returned
- `member`: given a binary search tree and an integer, returns true if the integer is found in the tree, and returns false otherwise (in Scheme, call this function `member?` to avoid clashing with the built-in function of the same name)

In this assignment, you ***may not*** use assignment or iteration (i.e., any Scheme function whose name ends in an exclamation point such as `set!` or the ML assignment operator). Consequently, the `insert` function must return a tree. For the Scheme program, you will have to use lists to represent the tree, as Scheme does not support user-defined datatypes. However, since ML does support user-defined datatypes, you should define your own datatype to represent the tree.

Goal: To represent structures in languages with structured types and without structured types.

Hints: For the Scheme program, I found it easiest to represent a node as a list with three elements: the data, the left subtree, and the right subtree. Both the left and right subtrees would themselves be lists with three elements. The empty list is used to represent an empty subtree.

1.3 Coding Guidelines

- Use cases in the ML solution for both the `insert` and `member` functions.
- Use currying in your ML solution: `insert` should have type `tree` → `int` → `tree`, and `member` should have type `tree` → `int` → `bool`. If your tree type is polymorphic, which is encouraged but not required, then your functions would use `int tree` rather than just `tree`.
- You do not need to write extra functions such as `inorder` or `preorder` but it is okay to include them in your solution.
- For the Scheme solution to `quicksort`, you are explicitly asked to keep the same binding, function, and parameter names. Include bindings for the head and tail of the list parameter. For the Scheme solution to the binary search tree, you are free to use `let` or `let*` to introduce additional bindings if you like, and to write additional functions such as `left-child`, `right-child`, etc. if you wish.
- Many programmers consider it bad practice to use a `define` inside a function definition (i.e., nesting a `define` within another). A `lambda` inside a `let` or `let*` should be used instead and is more portable.

2 Homework

Build upon the provided solutions to the practicum (and not your own solutions) when solving the homework.

2.1 Quicksort

Modify the quicksort function to accept a comparison function as the first parameter and use the comparison function when comparing two values in the list, which is now the second parameter. For the Scheme solution, the comparison function requires two arguments and returns an integer greater than, equal to, or less than zero depending if the first argument is greater than, equal to, or less than the second argument. You can simply use the subtraction function as the comparison function to test your new function:

```
> (quicksort - '(8 6 7 5 3 0 9 2))
```

For the ML solution, the comparison function behaves similarly but returns a value of `LESS`, `EQUAL`, or `GREATER`. These constants are predefined in ML as values of type `order`:

```
datatype order = LESS | EQUAL | GREATER;
```

Your ML function should be curried, and if done correctly should now be polymorphic. You can use the built-in functions `Int.compare` and `String.compare` to test your new function. Call these programs `sort.scm` and `sort.sml` and submit them using Camino.

2.2 Binary Search Trees

Extend the binary search tree programs to provide a function called `maximum` that computes and returns the maximum value in the given tree. Your ML solution should raise the exception `Empty` if given an empty tree. Call your programs `tree.sml` and `tree.scm` and submit them using Camino.