Homework #2 Solution Set

All problems for this homework were programming problems. The source code for the solution is available at
http://www.ugrad.cs.ubc.ca/~cs418/2016-2/hw/2/sol/hw2.erl Here are some explanatory remarks about the solution for each question.

1. `mean(WTree, DataKey)` **(20 points)**

   Solution: Since this function is just returning a single value, it can easily be implemented with a `reduce`. All the `mean` function needs to do is call `wtree:reduce`, passing in the worker tree, `WTree`, and the appropriately defined functions `Leaf`, `Combine`, and `Root`. It basically works just like a function for sum, except we also need to know the length of the list. So the `Leaf` function gets its data from the `ProcState` variable (in the form of a list of numbers), and puts it into a tuple of the form `{Sum, Length}`. The `Combine` function then just needs to add the corresponding values in the two tuples it gets, and `Root` returns the average by dividing `Sum` by `Length` for the tuple it gets.

2. `vec_mean(WTree, DataKey)` **(20 points)**

   Solution: This is almost identical to question 1 if you use the `vec_sum` function that is given, so again, it can easily be implemented with a `reduce`. The `Leaf` function puts the (vector) sum of its data into a tuple, along with the number of vectors it received. The `Combine` function adds the vectors and lengths that it receives and returns them in a new tuple. And `Root` just needs to divide each element in the vector it receives by the total number of vectors. Note that, given `Sum` (the sum of all vectors) and `N` (the number of vectors), the expression `[ X / N || X <- Sum ]` returns a list of elements `X / N` for all `X` in the list `Sum`.

3. `set_nth(N, Fun, List)` **(10 points)**

   Solution: There are many ways to solve this problem, but it can be done simply and concisely using pattern matching and recursion. The solution provided simply applies `Fun` to the head of `List` if `N` is 1. Otherwise it appends the head element to the result of the recursive call.

   One question that came up in discussing this problem is "Do we need to use tail-recursion?". The problem did not specify tail-recursion; so we won't punish head-recursive solutions. In particular, the provided solution is head-recursive. There is a design trade-off. The head-recursive implementation is simpler, and "obviously" correct. The tail recursive version is needed if we are worried about the case where `N` is huge (i.e. greater than one million). In this case, the solution set chose: "Do the simple way first.", or, as Don Knuth noted "The road to hell is paved with premature optimization.".

   There are some cases where tail-recursion is a must. In Erlang, server processes are implemented with recursive functions. Unless these functions are tail recursive, the server will gradually use more memory until it crashes. Tail-recursion is a must for cases like this. There are other cases where head-recursion is clearly harmless, for example, traversing a balanced binary tree. The height of the tree, and thus the depth of the recursive calls, is bounded by the log of the number of leaves of the tree. There will never be a tree so large that we can't afford the stack frames for a traversal.

4. `bank_statement(WTree, SrcKey, DstKey, InitialBalance)` **(20 points)**

   Solution: Unlike in questions 1 and 2, here we want to return a list of values (i.e., the balance at each point in time), and processes need information about values to their left in order to calculate their own final values. So we should use a `scan`. We need to choose the functions `Leaf1`, and `Leaf2`, `Combine`, and the initial value `Acc0`. The hardest part of this question is probably figuring out how to deal with `interest` transactions. The idea is for each node to calculate a balance along with a tally of accumulated compound interest. The key observation is that any sequence of transactions can be summarized as a linear function:

$$FinalBalance \quad = \quad A * InitialBalance + B$$

where A and B depend on the particular sequence of transactions.

`Leaf1` uses the `lists:foldl` function, calling the helper `bs_leaf1` on each element of `Source`, its data. This returns a tuple of the form `{Balance, CompoundInterest}`, where `Balance` is the result of applying all the transactions in the node's data, and `CompoundInterest` is the (multiplicative) total of any `interest` transactions. The helper `bs_leaf1` applies a single transaction to the running total, returning a tuple of the form `{Balance, CompoundInterest}`.

`Leaf2` gets a list of transactions (from `ProcState`) and the result (i.e., balance and accumulated interest) of all transactions to its left. It then needs to update the `DstList` in `ProcState` with the appropriate values. We want to return a list of balances, given a list of transactions and an initial balance. This is a good place to use a `map` (actually, a `mapfoldl` since we are accumulating the values). The function we are passing to `mapfoldl` is essentially the same as `bs_leaf1`, but the return value needs to be of the form `{Balance, {Balance, CompoundInterest}}` to fit what `mapfoldl` is expecting. The wrapper function `bs_leaf2` is just converting the result of `bs_leaf1` to this form. `Leaf2` then stores this result in the `ProcState` so we can access it after computation finishes.

`Combine` gets two tuples of the form {Balance, CompoundInterest}. Since the transactions of the left tuple occurred before those of the right tuple, the interest of the right tuple needs to be applied to the balance of the left tuple. The value of `Balance` returned is the sum of this and the right balance. The new value of `CompoundInterest` is just the product of the interest from the two input tuples.

`Acc0` needs to be a tuple of the form `{Balance, CompoundInterest}`, so we provide `{InitialBalance, 1.0}`. We chose 1.0 for the initial value for `CompoundInterest` because 1.0 has the effect of applying no interest. However, if you check the calculations, you'll note that the initial value of `CompboundInterest` only affects the final compound interest and has no impact on the per-transaction balances. So, we could put any (numeric) value we want for the `CompoundInterest` field.

5. `sliding_average(WTree, SrcKey, DstKey, Kernel, InitialPrefix)` **(25 points)**

   Solution: Again, a `scan` is appropriate for this question because processes need information about values to their left in order to calculate their own final values. In this case they will need the `length(Kernel)-1` values that come immediately before their own data. This way they can calculate the sliding average of their first `length(Kernel)-1` values.

   `Leaf1` just needs to pass up its rightmost `length(Kernel)-1` values. These values will be used in the `Leaf2` function to calculate the sliding average for the leftmost values coming from nodes to the right.

   `Combine` also just needs to return the rightmost `length(Kernel)-1` values from the two sublists it gets so that nodes to the right can calculate the sliding average of their leftmost values. It is possible that `length(Right)` < `length(Kernel)-1`, in which case we also need to pass up some of the values from `Left`.

   `Leaf2` gets the `length(Kernel)-1` values that come immediately before its own data in `AccIn`. So it can just calculate the sliding average of this using the helper for the sequential algorithm.

6. **Test cases (5 points)**

   Solution: Make sure to test edge cases such as empty lists and badly formed inputs. In some cases your code can just error out on these inputs, but others it should handle properly. You can use `assertError` and `assertException` to make sure you are handling error scenarios properly.