CSC 254 - Assignment 1 - Prikshet Sharma

Languages Used

Python

Python was the easiest to implement. The iterator for T in tree(n - 1): was used.
This works because before iterating, Python determines the value of tree(n - 1), which returns a list, and the iterator henceforth iterates through that list. Since the tree (n - 1) call is recursive, we keep going down the recursive tree until we hit the base case, i.e., tree(1), which directly returns the list [“(.)”]

Haskell

Haskell is a functional programming language, so naturally the implementation was more in tune with what comes naturally and is also idiomatic to Haskell, which is the map function. The map function performs a function to each element of a list. So in our case, we perform the string concatenation of each element in the list that we have already computer using (tree (n - 1)). Since there are two ways in which we concatenate the strings, we call map twice and then merge the two resulting lists.

C#

The logic behind C# was exactly the same as Python, and other than the syntax differences, the pseudocode for both the implementations is exactly the same.

Prolog

Prolog is a logic programming language, and the interesting and novel thing was how the variables are defined. In Prolog, the variable that we return seems like a parameter to a function to the naive person, but in fact it stores the value that the function returns. Similar to the map function in Haskell is the function maplist.

Ada

Ada was challenging to debug since the Syntax was unfamiliar. In Ada, a vector of strings had to be defined so that newly added trees could be appended. The logic is otherwise the same as the Python and the C# implementation.

Limitations

In all the implementations, the time and space complexity is exponential. While the time complexity cannot be decreased because for n nodes there must be 2^n trees, the space complexity perhaps can be decreased by not saving all the n-1 node trees before computing n node trees. In the Ada implementation, there is no prompt for the input, because for some unknown reason, using Put_Line was printing the prompt “Enter the number of nodes: ” *after* the user entered the number of nodes.

Extra credit

Scheme

Scheme was the most intuitive to program in. We put a cond, or conditional, such that tree returns (.) if the n = 1, otherwise it performs the map function twice. Since there are two forms that we must map the (tree (- n 1)) we perform map twice and pass in form1 and form2 corresponding to the two forms, and we cons, i.e., merge, the two resulting lists.

Closed form formula

Every step of recursion returns twice the number of elements than the previous one, since we take the result of the previous recursion and perform two kinds of concatenations for each element in the result of the previous recursion, and add both of these to the list that we return. I.e., if the base case is 1, then the next recursion would give 2, and the next one 2 * 2 and so on. The first few number of trees, therefore, would be len(tree(1)) = 1, len(tree(2)) == 2, len(tree(3)) == 4, len(tree(4)) == 8, len(tree(5)) == 16 and so on. This gives T(n) = 2^n.

Asymptotic Requirements

Python, C# and Ada

All three implementations would have the same time and space complexity since the pseudocode of all three is exactly the same.
Since the number of trees that we are storing in a recursive step increases with the order of two, the space complexity is O(2^n). The time complexity is also O(2^n) because for each n, we must perform two basic computations (concatenations) on each element of the previous recursive (n - 1) step, corresponding to the two different forms.

Haskell and Scheme

All three implementations would have the same time and space complexity since both involve calling map twice associated with each form and then combining the resulting lists, with cons in Scheme and ++ in Haskell.
With each recursive step, we increase the number of computations by a factor of two, and since we are storing the previous lists, the space complexity is O(2 * 2^n) = O(2^n). Since we are storing two trees, because we are calling tree (n - 1) twice. This is one limitation to our implementation, since we should be calling tree(n - 1) only once and storing that in a variable rather than calling the function twice. This can be done but for some reason the let statement didn't work. In any case, the complexity doesn't change although the run time is expected to decrease by half.

Prolog

Prolog's logic programming interface might be different in how we think about the program, but the fundamental concept is the same as that of Haskell and Scheme, i.e., we call the maplist function, which is similar to the map function, and with each recursive step increases by a factor of 2. So both Space and Time Complexity is O(2^n) like in Scheme and Haskell. In this case, however, we were able to call the tree(n - 1) function just once, which is evidently a better implementation than Haskell and Scheme.