Timothy Jelinek

Part 1

A possible structure for expressing the types of C or a similar language as trees would be starting with the subject of the types of C at the start and branching down to the type of variable like int, double, array, or pointer and then branching down to child nodes involved. Next, I will list out the steps in pseudocode that would be used to write a function to check the structural equivalence of the trees. First, you start with the first line of code, where you define the function through code function TypeEqual(node1, node2). You write the following line of code to see if the nodes' types are equal.   After checking if the nodes are equal, then check to see if they represent primitive types. Next, I would check if the nodes represented structs or arrays, followed by checking if the number of child nodes is the same. After checking if the child nodes are the same, I would recursively compare child nodes and move on to checking if the child nodes are equal which would result in returning a response of true. The next step is to check if the nodes represent pointers, which then would be followed by recursively comparing the pointed-to types. If none of the conditions checked for were true, then a false response would be sent. Through these steps and writing the program to check all of these conditions, it is possible to check the types of language to check the structural equivalence.

Part 2

The trees in part one could represent recursive types. I would introduce a reference mechanism where a given node can refer to another node somewhere else in the tree. I would add a branch to a tree node reference. In the function that was used to compare the nodes, I would introduce a new “visited” set to keep track of the nodes that were visited during the process of comparison. Using the “visited” set, you can prevent infinite recursion when processing recursive types. The “visited” set is a great way to check a node to see if it was already visited, and if it wasn’t, then it would be compared and added to the “visited” set. Recursion is a newer concept to me, and I was curious to learn some of the benefits of it. The first advantage of using recursion is that it can reduce time complexity by remembering the result of a function so that there will be less repetition in the running of the code. Another advantage of using recursion is that it adds clarity and reduces the time you use to write and debug code. It adds clarity and reduces the time of writing and debugging code. Finally, recursion is good at tree traversal by recursively looking for specific leaves through traveling up the branch. With these significant advantages of recursion, there are also some cons. The first disadvantage is that recursion uses more memory by saving the results of previous comparisons done through the function. Another disadvantage is that recursion can be slow and take a while for the memory to look and see which results were already found.

Sources:

CodeProject. (n.d.). *tree data structure in C - CodeProject*. CodeProject. https://www.codeproject.com/Questions/988771/tree-data-structure-in-C