Comparing functional programming & logic programming paradigms

For this assignment, I had to implement a binary search tree in a functional programming paradigm and a logic programming paradigm. I chose Haskell for the functional implementation and Prolog for the logic implementation.

I started this assignment by implementing the binary search tree in Prolog. A Prolog program consists of horn clauses, facts and rules. The first thing I wanted to define was a tree. This would be a fact in the program e.g. tree(X, Left, Right). A tree has a value X, a left child and a right child. The left and right children can also be trees or can be empty. I then began to implement the "search" rules. We want to check if X is contained in the binary search tree. The first rule is the base case. It checks if X is the value in the tree and returns true or false e.g "search(30, tree(30, empty, empty))." .This will return True as 30 is contained in the tree. However, if X is not the value, we want to then check if X is less than the Root. If it is then search the left child. The left child is also a tree and so we can recursively call search again but this time searching X in the left child. If X is greater than the Root, we then want to check the right child. Similar to before, we recursively call search but with X and the right child.

Next, I moved on to the "insert" rules. To insert a value, it takes a value X and a tree and checks if X is the root of that tree. The first rule deals with inserting into an empty tree e.g. insert(X, empty, tree(X, empty, empty)). This will only return True or False depending if X is the root so instead we can use Prolog's relations and write it as e.g.insert(30, empty, R). This will return R = tree(30, empty, empty). This allows us to see the binary search tree more clearly. The next two rules defines inserting into a non-empty tree. It takes a value X, a tree and the returned tree. It checks if X is less than the Root value, if it is, we recursively call insert with the value X, the left child and the output "NewLeft". If the value X is greater than the Root, we recursively call insert with the value X, the right child and the output "NewRight". By using recursion and going through all three rules, the value X will eventually be placed in the correct position in the tree.

Next, I had to implement the three different ways to traverse a tree. These are "inorder", "preorder" and "postorder" traversal. I first defined the "inorder" rules. This takes a tree and a list. The first rule is the base case. This takes an empty tree and an empty list or if given an empty tree and list L, L would be empty e.g. inorder(empty, []). The next rule is if the tree is not empty. It takes a tree and "returns" a list of the nodes of the tree in order. It recursively calls the Left child which has its own List called "LeftList", it puts the value X(every child will have a value X) into a list and appends this list to LeftList. Once it traverses through the left child, it will append the original X(the root). Similarly, It then recusively calls the right child which also has its own list called RightList, then appends the result of the left child and root to RightList to get the final List.

"Preorder" and "postorder" are done in the exact same way, the only difference is the placement of where the root value gets placed in the rules of each traversal. In preorder traversal, the root is the first value we see listed, followed by the left children and then the right children. This means the root is appended first in the preorder rules. In postorder traversal, the root is the last value seen in the list. It is therefore appended at the end after the left and right children have already been traversed.

I am aware that there are singleton variables used throughout my program. I purposely left these in to have a better understanding of what I was doing. Best practice would involve using anonymous variable ("_") in rules where a single variable is used.

A remove element operation would have to consider three cases. We would have to account for if the node has no children, if the node has one child and finally if the node has both children. In order to remove the node with no children, I would use the search rules to find the element to be removed. I think the cut operator could be used to prevent backtracking from occurring and somehow help with removing the element but unsure how I would use this. To remove the node with one child, I would again search for that element and somehow make the child the parent which would remove the element we want to remove.

When I finished the logic implementation of a binary search tree, I then moved on to the functional implementation using Haskell. I realized that there was no predefined type in Haskell that would give a good representation of a binary search tree so I knew I had to define a type myself. User-defined types can be defined through the use of datatype declarations e.g. data BST a. From here, I had to create two constructors that would represent the binary search tree. These were “Null”(which is an empty tree) and “Node”. A nullary constructor contains no data while a Node represents a node in a tree. A Node contains data and the type of this data are its parameters. A Node has left child which is of type “BST a” as a child is also a binary search tree. A Node has another type “a” which is the value the Node contains and finally a right child of type “BST a” again. In order to show these values as strings, I had to derive an instance of class “Show”. Throughout the program I realized I also needed to check if one value equals another e.g. X == V. Therefore, I had to derive an instance of class “Eq”. Finally, I wanted to make sure that functions works for any type a, so long as a is comparable. I therefore derived an instance of class “Ord”.

Once I had my binary search tree datatype defined, I moved on to creating functions. The first function that I created was “search”. This function has an “Ord a “constraint which as previously described makes sure that whatever is passed to it is a comparable type. This function takes a binary search tree of type a, a value of type a and returns a Boolean value(true or false). I realized that the functions would work similarly to the rules in prolog. Prolog heavily relies on recursion and this was the same for Haskell. I created a base case in my search function. If given an empty tree(Null) and a value “v”, it would return false. Otherwise if you search a non-empty bst for a value "v", it would do the following: if the current node's value "x" is equal to the value "v", return True. Or if v is less than x, recursively search the left child. Or finally, if v is greater than x, recursively search the right child. The difference between both paradigms here is that the functional programming uses functions which returns a value, while logic programming uses predicates which give true or false as a result rather than values.

I then went on to define the “insert” function. Similar to the search function before it, it takes a BST, a value and this time returns a BST with that value inserted. The base case comes first. If the tree is empty(Null), create a binary tree with the value "v" that has two empty trees as its children e.g. insert Null v = Node Null v Null. On the other hand, If there's a left and right child then do the following: If the value "x" == value "v", then replace v with x. Or if v is less than x, recursively insert the value to the left child. Finally, if v is greater than x, recursively insert the value to the right child.

Similar to the prolog implementation, tree traversal can be done in three ways- inorder, postorder and preorder. These also follow similar logic as the Prolog implementation and the only difference between the three traversals is the placement of the root node. Inorder traversal takes a bst and returns a list of the values in the bst. Our base case describes if the bst is empty, then return an empty list. Otherwise if the bst is not empty, recursively go through the left child using the "++" operator to concatenate lists, followed by the root v, followed by the right child. Every child will have a value “v” and so this value is added to the list every time we find a new node. Preorder and postorder are done in the exact same way but the root node in preorder is appended first and in postorder is appended last.

In order to remove an element from a bst, there are three cases to consider. Just like in Prolog, We would have to account for if the node you want to remove has no children, if the node has one child and finally if the node has both children. To remove the node with no children, you would recursively traverse the tree until you find the node, once you find it, remove it (by letting it equal Null). If the node has one child, you would let the child take over the parent's position in the tree and set the child to Null. If the node has two children, we would have to move up a value that is larger than the nodes in the left child and less than those in the right child. This could be done by getting the maximum value in the left child, or the minimum value in the right child.

After finishing both implementations, I could see the difference between both paradigms more clearly. The major difference is that logic programming uses predicates and functional programming uses functions. A predicate in logic programming is not a function, it doesn't have a return value, instead the predicate can evaluate to true or false. In my Prolog implementation I defined facts and rules (e.g. insert(X, empty, tree(X, empty, empty)).) which can evaluate to true or false. However, using relations, if a value is undefined, it will try to find the correct values so that it would make the predicate true (e.g. insert(30, empty,R). This allowed me to get some sort of "return" value which was similar to Haskell which actually returned values from its functions. A function in Haskell has a name and a set of parameters (e.g. insert :: Ord a => BST a -> a -> BST a). It is evaluated by what comes after the "=" symbol (e.g. | x == v = Node treeLeft x treeRight). Every parameter has a type and the function will always evaluate to a value which also has a type. In my Haskell implantation I was able to use pattern matching. Pattern matching in Haskell is one of its most powerful constructs. Both functions and case expressions can be defined using pattern matching. A pattern is like an expression that can contain binding occurrences of identifiers. Prolog on the other hand uses backtracking. This occurs when the program gives a false outcome, rather than declaring the query as false, it will backtrack to an alternative solution if one exists. Both paradigms were similar when using recursion. They both needed a base case in order to terminate the recursion and until it hits that case, it will keep going. Both implementations described previously used recursion heavily. Although both programming paradigms are declarative, they are different. Haskell being a functional language used mathematical expressions to solve this assignment, while Prolog being a logic language used logic expressions to complete this assignment.

References:

In order to understand prolog's facts, rules and queries better, I researched this site:

http://www.cs.trincoll.edu/~ram/cpsc352/notes/prolog/factsrules.html#:~:text=A%20fact%20is%20a%20predicate,must%20end%20with%20a%20period.

Also went through the following notes:

https://www.computing.dcu.ie/~davids/courses/CA341/CA341_Logic_Programming_Paradigm_2p.pdf

https://www.computing.dcu.ie/~davids/courses/CA341/CA341_Functional_Programming_Paradigm_2p.pdf

https://www.computing.dcu.ie/~davids/CA208_Prolog_2p.pdf

CA320-haskell notes-Alireza Dehghani:
https://ca320.computing.dcu.ie/CA320__Functional_Programming.pdf

In order to understand constructors in Haskell better, used the following link:

https://wiki.haskell.org/Constructor#:~:text=Data%20or%20value%20constructors%20are%20used%20to%20pattern%20match%2C%20e.g.%2C&text=If%20you%20want%20to%20dig,constructor%20itself%20is%20called%20kind.