Declaration on Plagiarism
Assignment Submission Form

This form must be filled in and completed by the student(s) submitting an assignment

Name(s): **Andrew Finn**
Student Number: **18402034**
Programme: **CASE3**
Module Code: **CA341**
Assignment Title: **Report: Comparing Functional Programming and Logic Programming**
Submission Date: **4th December 2020**
Module Coordinator: **Dr. David Sinclair**

Name: **Andrew Finn** Date: **Dec 5th 2020**

Comparing Functional Programming and Logic Programming

Andrew Finn - 18402034

For the completion of this assignment I used Haskell as my functional language and ProLog for my Logic language. Source code is attached with my submission, src - Functional containing my Functional Haskell solution and src - Logic containing my ProLog Logic Solution. For the remainder of this report I will use the following abbreviations:

- Haskell -> HS,
- ProLog -> PL,

Functional Programming Approach:

I began by first implementing my functional approach using Haskell (HS). I choose to implement my solution in HS for a variety of reasons. Firstly, due to CA320 HS is more familiar to me as a result of only recently learning it. Decondly, for a CA320 lab task we recently had to implement a binary tree. As such I used this code as a starting point.

My solution begins by creating the 'Tree()' data type, containing a value (node) and two children, left and right. This data type is the backbone of the program, the rest of the functions are merely manipulation data stored in this data type.

A Tree is useless unless we can add to data as such the insert function was created next. This recursive function takes an integer (node) and adds it to the correct position in the given Tree. This works as the function recursively works its way down the Tree navigating left and right down the tree until it finds the correct position to insert the node.

The makeTree isn't necessary. I simply carried it over from my CA320 solution, this is another recursive function that takes an array as a parameter and calls the insert function for each value in the array. This function allows for easier testing.

For simplicity i will explain inOrder, postOrder and preOrder together as they are all very similar just simply add data to the output array as different indexes. All 3 are recursive functions. These functions all take a Tree type as parameter and return an array of the data in the specified form. These functions recursively work down the tree and add the value at each node to the middle, right or left side of the array respectively.

Search is again a recursive function (sensing a pattern? … this will be explained in the comparison). Search takes in a Tree and integer to search for as a parameters and recursively works its way down the Tree choosing it path left on right based on the value of the provided integer until the value is found or it reaches the end of the tree return a boolean True or false respectively.

Removing nodes from Binary Trees isn't straightforward if the node being removed has children due to the need to reformat the tree afterwards. The simplest way to remove a node in my solution would be to simply recreate the tree. This could be done easily by using the inOrder function to create an array of the tree, then remove the node from the array and then passing this array to the makeTree function to create the new tree with the node removed.

Logic Programming Approach:

Choosing a language for my Logic solution was straightforward due to the fact I am only familiar with one, ProLog (PL). As i haven't really touched this language since CA2 Logic i had to do a crash course before touching any code, luckily i had my HS code so implementation was a matter of transferring this code to PL rather than creating it.

insert is a recursive predicate which takes an integer(X) and a Tree and creates a new Tree containing the addition. It does this by working down through the list to find the correct place to insert the new node

Search is again a recursive predicate that takes the Tree and a value to be found, the predicate then works down the tree based on the value to be found until said value is found or there are no more nodes to be searched.

Again inOrder, preOrder and postOrder are very similar; all predicates take in the tree and create an array of the corresponding ordered output. These predicates are again all recursive and work down the tree adding to the array at different indexed left, right and middle depending on the operation (preOrder, postOrder etc.) requested until no nodes remain.

Again the easiest way to remove a node from this implementation is to simply recreate the tree without the node to be removed. As mentioned earlier this means no reformatting the tree.

Comparison

In one line the implementation and code style of both approaches is very similar, it even looks very similar however under the hood the actions being performed are very different.

In functional programming languages such as Haskell programs are made up of functions. These functions can return values, these return values are made up of mathematical expressions with manipulated data passed to the function as parameters. You build these functions up by specifying edge and base cases and then create a ‘general’ function that will provide the mathematical steps to create the solution. The code will then work down these definitions to find one that works for the given data however once it enters a block the function you cannot fail and move to the next definition, there is no backtracking. The code is essentially a set steps of how to solve the problem

In logic programming languages such as ProLog programs are made up of predicates. Predicates are defined by combining clauses, A clause in this case can be defined using mathematical expressions, propositional calculus and other predefined predicates. ProLog will try predicates if it fails to compute it will just move on to the next predicate in the sequence (if any). The code is essentially a set of relationships that combined will solve the problem. ProLog can also perform backtracking reevaluating previous bindings to reach its goal.

Although a predicate and a function look visually very similar a predicate does not have a return value. The way the languages come to their solution is also fundamentally different despite the code being very similar when reading it.

References

https://stackoverflow.com/questions/39945651/difference-in-implementation-of-gcd-between-logic-and-functional-programming
https://ca320.computing.dcu.ie/labsheet-05.html
http://users.utcluj.ro/~cameliav/lp/lab8.pdf