Student name: Vincent Achukwu
Student number: 17393546
Project title: Comparing Functional Programming and Logic Programming

Introduction

For this assignment, I have chosen Prolog for the Logical programming approach, and Haskell for my Functional approach. I began to implement the binary tree in Prolog first as I was more familiar with the implementation in Prolog, after which I implemented it in Haskell.

For the functional implementation, I added a few test cases to make it easier for the user to test the functions based on the sample trees, if they wish to do so. As is described in the comments of both programs, for the logical implementation, I first begin by defining the node bst predicate for the binary tree, which has an item, left, and right child. The functional implementation also has this, and it defines the binary tree as an empty tree or a node that contains an element with left and right child. Also, the functional implementation can have trees with elements of types that can be compared with each other (hence the "Eq" in "deriving (Show Eq)" in the definition). In the test cases, I created 2 trees, one tree consisting of character elements, and another with integer elements. Both implementations have satisfied the implementation of insertion, searching, and traversals via preorder, inorder, and postorder. Below shows how you can run both programs in the terminal.

```
>>> consult(logical).

>>> ghci
>>> :load functional.hs
```

Logical Programming Approach

The language I have chosen for the logical language implementation for the binary tree is Prolog. Prolog is a declarative programming language, which means it describes the program in terms of the solution's properties. It is considered to be a high-level programming language, meaning that it is easy to read and write for humans rather than machine language (low-level programming). Programming languages in the logical programming paradigm are based on the concept of relations. Prolog uses a special form of logic clauses known as Horn clauses and has logical expressions to evaluate programs. Recursion is used in Prolog as a technique for looping until a base case is reached. Backtracking is how logical programming languages compute the results with the aim of checking every line if it's true or false, then continue to the next step. Logical programming like Prolog uses facts and queries on those facts. These facts are defined by relations

between symbols and then submitting one or more queries to the Prolog run-time system to check for which values of the query variables are the defined relations true. This also introduces the notion of Closed World Assumption, which means that whatever facts are defined in the program remains true, and everything else is false. For example, the following facts are defined:

- human(david).
- human(bob).
- human(mary).

If the user were to query for human(tim). , the program would say it's false. Although we know Tim is a human, Prolog doesn't say so in it's set of facts.

Some advantages of using logical programming languages are how it's easier to use than low-level programming languages, they are reliable, and how programs can quickly develop due to its use of true/false statements rather than, say, objects. Some disadvantages include how predicates are not easily readable. As the program gets more complex, it can become more difficult to track the execution. Also, logical programming languages are used in few applications such as Artificial Intelligence and RBDS. The use of true/false statements limits the number of problems that can be solved.

Functional Programming Approach

For the functional implementation, I used Haskell to implement the binary tree structure. Haskell is also a declarative programming language, as well as a high-level programming language. It is based on mathematical functions and the operations are solely based on the inputs to the program. It doesn't rely on temporary variables to store results in between inputs. Haskell uses recursion and conditional expressions for computations. It is also known as a "lazy" language. It doesn't support the use of control flow like loops and conditional statements. Rather, they use the functions and function calls via recursion. The variables and data can only be set once, meaning that functional programming languages provide immutability. Also, functional programming languages do not have any side effects, other than to return the value of the function.

Some advantages of using functional programming languages are how they are quite simple to understand since pure functions don't change any states and depend on the input. The function signature provides the arguments and return-type of pure functions. It also supports the concept of lazy evaluation (i.e. the value is evaluated and stored only when required). Some disadvantages of using functional programming languages are how inefficient the programs can be compared to other languages. Combining recursion and immutable types may reduce performance. Also, the readability of pure functions can be reduced as the program becomes more complex. Functional programming also requires a large amount of memory since it does not have a state and you need to create new objects every time to perform actions.

Comparison of solutions

For the logical implementation, as well as the bst (Binary Search Tree) predicate being defined (as described in the introduction), the logical implementation defines the search, insert, and the traversal predicates. Below, you can see the predicates being tested. In the binary tree, 10 is present, 180 is not, hence the results are true and false respectively. Inserting 100 to the tree displays a new tree. The inorder, preorder, and postorder traversal of the tree can also be seen below.

```
?- search(10, bst(10, bst(8, bst(2, nil, nil), bst(9, nil, nil)),bst(14,bst(11, nil, nil),bst(16, nil, nil)))).
true .

?- search(180, bst(10, bst(8, bst(2, nil, nil), bst(9, nil, nil)),bst(14,bst(11, nil, nil),bst(16, nil, nil)))).
false.

?- insert(100, bst(10, bst(8, bst(2, nil, nil), bst(9, nil, nil)),bst(14,bst(11, nil, nil),bst(16, nil, nil))), X).
X = bst(10, bst(8, bst(2, nil, nil), bst(9, nil, nil)), bst(14, bst(11, nil, nil), bst(16, nil, bst(100, nil, nil)))) .

?- inorder(bst(10, bst(8, bst(2, nil, nil), bst(9, nil, nil)),bst(14,bst(11, nil, nil),bst(16, nil, nil))), X).
X = [2, 8, 9, 10, 11, 14, 16].

?- preorder(bst(10, bst(8, bst(2, nil, nil), bst(9, nil, nil)),bst(14,bst(11, nil, nil),bst(16, nil, nil))), X).
X = [10, 8, 2, 9, 14, 11, 16].

?- postorder(bst(10, bst(8, bst(2, nil, nil), bst(9, nil, nil)),bst(14,bst(11, nil, nil),bst(16, nil, nil))), X).
X = [2, 9, 8, 11, 16, 14, 10].
```

The functional implementation is done similarly, with the difference being that they're not predicates, but functions. I wrote test functions which are optional for the user to use. Tree1 is a tree of characters, tree2 is a tree of integers. The inorder, preorder and postorder traversals can also be seen in the output below. Searching 8 in tree2 outputs True, whereas a number not in tree2 like 1923 outputs False. You can also see the output of the inorder traversal of the tree after inserting 4 to the tree.

```
*Main> tree1
Node (Node (Node Empty 'a' Empty) 'b' (Node Empty 'c' Empty)) 'd' (Node (Node Empty
'e' Empty) 'f' (Node Empty 'g' Empty))
*Main> inorder tree1
"abcdefg"
*Main> tree2
Node (Node (Node Empty 2 Empty) 8 (Node Empty 9 Empty)) 10 (Node (Node Empty 11 Empt
y) 14 (Node Empty 16 Empty))
*Main> inorder tree2
[2,8,9,10,11,14,16]
*Main> inorder (insert 4 tree2)
[2,4,8,9,10,11,14,16]
*Main> search 8 tree2
True
*Main> search 1923 tree2
False
*Main> preorder tree2
[10,8,2,9,14,11,16]
*Main> postorder tree2
[2,9,8,11,16,14,10]
*Main>
```

For the logical implementation, the search predicate begins with a base case which is true if the item to be searched for is the same as the current node we're on (i.e. leaf node). If that case isn't satisfied, it traverses the left side of the tree if the item we're searching for is less than the current node we're on in the tree. If that isn't satisfied, then we traverse the right side of the tree (i.e. the item we're searching for is greater than the current node of the tree). The use of the anonymous variables means we don't care what's there and it can match anything. I also included test code for the search predicate which can be copied to the terminal for use with other predicates for testing.

```
search(Item, bst(Item, _, _)).
search(X, bst(Item, Left, _)) :-
    X < Item, search(X, Left).
search(X, bst(_, _, Right)) :-
    search(X, Right).
```

The insert predicate works similarly to the search predicate, except it passes in an item to search for in the tree. The base case checks if the tree is empty and doesn't contain the item we're looking for, it adds the item to the tree as a node. Else it traverses the left subtree if the item is less than the current value of the node we're on and searches, else, it traverses the right subtree.

```
insert(X, nil, bst(X, nil, nil)).
insert(X, bst(Item, Left, Right), bst(Item, Left2, Right)) :-
    X < Item, insert(X, Left, Left2).
insert(X, bst(Item, Left, Right), bst(Item, Left, Right2)) :-
    insert(X, Right, Right2).
```

In the logical implementation, the traversal predicates are all similar, with the only difference being how the list of nodes appear in the output according to the traversal. All traversal predicates have the same base case, passing of parameters, and calling of the predicates recursively. For the inorder predicate, we keep the current node as the head and append the right list to the left list. Preorder predicate simply just appends the right list to the left list, ensuring that the node is always the head. As for the postorder predicate, the right list is first appended to the left list after which the current node is appended to the resulting list each time.

```
%Inorder in the form (Left, Node, Right): 1 2 3 4 5
%traversing the tree then lists the nodes of the tree via inorder
inorder(nil, []).
inorder(bst(X, L, R), InorderLst) :-
    inorder(L, InorderL),
    inorder(R, InorderR),
    append(InorderL, [X|InorderR], InorderLst).

%Preorder in the form (Node, Left, Right): 2 1 4 3 5
%traversing the tree then lists the nodes of the tree via preorder
preorder(nil, []).
preorder(bst(X, L, R), PreorderLst) :-
    preorder(L, PreorderL),
    preorder(R, PreorderR),
    append([X|PreorderL], PreorderR, PreorderLst).

%Postorder in the form (Left, Right, Node): 1 3 5 4 2
%traversing the tree then lists the nodes of the tree via postorder
postorder(nil, []).
postorder(bst(X, L, R), PostorderLst) :-
    postorder(L, PostorderL),
    postorder(R, PostorderR),
    append(PostorderL, PostorderR, Postorder1),
    append(Postorder1, [X], PostorderLst).
```

As for the functional implementation, the search function is done in a similar manner where the base case checks if the tree is currently empty, it returns False since the item isn't present. Otherwise, it recursively compares each node to the item we're looking for. If the item is equal to the current node, we return True, else if it is less than it, it traverses left, else, it traverses right.

```
search :: (Ord a) => a -> BinaryTree a -> Bool
search item Empty = False
search item (Node left value right)
    | item == value = True
    | item < value = (search item left)
    | otherwise = (search item right)
```

The insert function is implemented in the same manner, with the difference being the return value of the function. Instead, it returns a tree rather than a boolean value.

```haskell
insert :: (Ord a) => a -> BinaryTree a -> BinaryTree a
insert item Empty = leaf item
insert item (Node left value right)
    | item == value = Node left item right
    | item < value = Node (insert item left) value right
    | otherwise = Node left value (insert item right)
```

Finally, the traversal functions are all similar too, with the difference being how the lists are arranged depending on the traversal type. The base case for each traversal function returns an empty list if the tree is empty. Again, the preorder function returns the list with the nodes in the form “item, left, right”, inorder with “left, item, right”, and postorder with “left, right, item”.

```haskell
-- traverses tree via inorder then returns list
inorder :: BinaryTree a -> [a]
-- if tree empty, return empty list
inorder Empty = []
-- else traverse left and right subtrees, adding item to middle of list each time
inorder (Node left value right) = (inorder left) ++ [value] ++ (inorder right)

-- traverses tree via preorder then returns list
preorder :: BinaryTree a -> [a]
-- if tree empty, return empty list
preorder Empty = []
-- else traverse left and right subtrees, adding item to left of list each time
preorder (Node left value right) = [value] ++ (preorder left) ++ (preorder right)

-- traverses tree via postorder then returns list
postorder :: BinaryTree a -> [a]
-- if tree empty, return empty list
postorder Empty = []
-- else traverse left and right subtrees, adding item to right of list each time
postorder (Node left value right) = (postorder left) ++ (postorder right) ++ [value]
```

Remove Method

If I were to implement the removal operation of an element in a tree for both implementations, I would ensure to have it cover the 3 cases if we found the element to delete from the tree: if the element in the tree has no children, set that node as null or empty. If the element has one child, check if the left or right child is set as null (an empty node), if the right is set as null we go left, else we go right. Else, if the current node has 2 children, get the smallest value of the right subtree to replace the current node after which we would delete the current node.

Conclusions

Both implementations have their pros and cons. I find Haskell to be an easier programming language to understand compared to Prolog due to Prolog having predicates which can become complicated to understand as the program expands. Though, this is also the case with Haskell with larger and more complex programs. In my opinion, processing lists in Haskell is easier than in Prolog since, in Haskell, you can specify the types in the list and can also have lists of functions. Both Prolog and Haskell have very useful built-in functionality for processing lists such as getting the head or tail of a list. In conclusion, Prolog and Haskell represent different approaches to the declarative paradigm. Prolog uses predicates to prove a "theorem" and it does not compute a value, rather answers "yes" or "no". Whereas Haskell is based on the concept of a function which takes a number of arguments and then computes a value.

References

1. **CA208 David Sinclair Prolog:**
    a. https://www.computing.dcu.ie/~davids/CA208_Prolog_2p.pdf
2. **CA341 David Sinclair Comparative Programming languages:**
    a. https://www.computing.dcu.ie/~davids/courses/CA341/CA341.html
3. **Soufiane Bouiti - The difference and the similarity of Functional and Logic Programming Languages**
    a. https://www.academia.edu/11202753/The_difference_and_the_similarity_of_Functional_and_Logic_Programming_Languages
4. **Project 4 – Advantages and Disadvantages of Programming Languages**
    a. http://lildiaz00.blogspot.com/2010/11/project-4-advantages-and-disadvantages.html