COMP1100 Assignment 2

**Huffman Compression**

Report

*ANU CECS*

|  |  |
| --- | --- |
| **Name:** | Aditya Sharma |
| **Student Number:** | u6051965 |
| **Course:** | COMP1100 |
| **Date:** | 05/05/17 |
| **Collaborators:** | Arham Qureshi - u6378881 Osbert Chor u6379604 |

Given the assessment task at hand for completing the modules required for the Huffman Compression implementation in Haskell, there were several different varieties of functions which were to be completed. The solutions for these were presumably required to use trees and recursion given the latest COMP1100 lab however any prelude function could also be used. This report will specify the functions implemented, split into four different sections respective to the file names of the modules needed to be completed and the functions themselves. The thought processes, errors, mistakes and setbacks which occurred during the process of completing said functions will also be documented.

Note that most of the functions required to be completed were done using recursion on the given trees/tree lists, particularly using the “case” function - ie. ‘case list of’.

The only exception to these were the functions which only needed to return a singular value given a singular type; notably the ‘value’ function, and arguably the ‘sort’ function which had an initial recursive implementation and a later implementation using a prelude function, of which the former was used.

**Frequency.hs**

create :: Ord a => Histogram a -> [a] -> Histogram a

The first function to be implemented, and surprisingly one of the most confusing. After many visual representations of the problem, the given inc function was begging to be used. Implementing the histogram creation after this was a breeze as the thought process was to simply apply this inc function to the histogram and the head of the element recursively.

**MakeTrees.hs**

toTreeList :: Ord a => [(a, Int)] -> [Tree a]

A simple function which set the wireframe for the solutions for the others. Seeing that the types taken and returned were both lists (and a list of trees, for that matter), it was obvious that this would need some sort of recursion, and the stage was set for the type of recursive thinking which would have to be applied to complete the rest of the functions. Using said ‘case’ function the immediate best and simplest thought process would be to take the value and key from the head of the list and turn it into a leaf, recursively going over the tail doing the same and adding these leaves together to form a final list as was required by the specifications - which was the first and final implementation. Simple and easy.

value :: Tree a -> Int

Having understood the nature of the first function and the fact that trees would be used for all the functions, the very first thought behind this function implementation may be to use recursion. However, since only a singular value needed to be returned given a definite type, it was simple to think about the two different types that would be possible in ‘Tree a’ because of its recursive implementation (Node and Leaf) and pull the values out of these - ‘case’ was again used for consistency.

merge :: Tree a -> Tree a -> Tree a

Looking at the specification of how this function should work as shown in the Assignment 2 outline, there are two examples. A simple example using two leaves and another using a Leaf and a Node. The direct implementation of this is to take values of the left and right and sum them into a Node – which was also the first thought process and final implementation.

sort :: [Tree a] -> [Tree a]

The first function which required some time and trial & error to complete, sort gave the impression of recursion once again, and rightfully so. Trees were to be used which would already require recursive methods to navigate, but in addition to that, the problem was that this function was to be applied to a *list* of trees, meaning that some sort of merge sort or well-known sorting algorithm would have to be implemented to complete it.

The code which has been commented out underneath the now implemented sort function was the initial thought process implementation. Once again, ‘case’ was used to check against the given list and inside of this, the elements would be sorted in ascending order by simple comparison and not any given sorting algorithm. The problem with this however was that the leaves of the sorted Nodes failed to sort and the function was therefore moot. The thought thereafter this problem was to create a separate helper function to sort the leaves (sortTree), which could be applied to the Nodes as the recursion allows, however this failed to take into consideration when Nodes’ leaves were Nodes and was therefore also inapplicable.

The line of code commented out and marked with a hashtag attempted to fix this issue but did not compile due to an Eq error, which would require adding the Eq definition to every function and would therefore ruin the ordered nature of the binary prefix tree required by the Huffman algorithm.

Given these difficulties, the next option was to do collaboration with peers to find a suitable implementation of sort which worked. This ended up being the prelude defined sortBy function which is implemented now, and simplifies the whole function as it is simply defined in one line and recursion does not need to be implemented.

mergeFirstPair :: [Tree a] -> [Tree a]

Edging closer to the completion of the MakeTrees module, mergeFirstPair proved to be an ambiguously defined function to begin with. The assignment specifications required it to be a given a sorted list, however it was noted as if the list it is given is already sorted. An easy fix, this required the sort function to be used on the list as it was thrown into the argument of the function using ‘case’ once again. There were no setbacks or complications here except for some syntax errors in failed concatenating of x and x1 as ‘++’ was used at first, instead.

makeCodes :: [Tree a] -> Tree a

This final function was also very simple in terms of thought process -> implementation. A repetitive mergeFirstPair applied to each tree in the list recursively was simple to conceive and simple to implement, however, how would a singular tree be given back from the list? This was easily solved by simply returning the amalgamated list (which would eventually have a single tree element) as the its first value.

**CodeTable.hs**

convert :: Ord a => HCode -> Tree a -> [(a,HCode)]

The only function in CodeTable, convert was an easy implementation as it required simply taking the given HCode and the element from the tree that it corresponds to and putting them together in a return value. Simple as it sounds, this was only realised after going through the PAL SOS sheet many times. After that, it was a straightforward recursive implementation like the many other functions.

**Coding.hs**

encodeMessage :: Ord a => Table a -> [a] -> HCode

Strafing away from the ‘case’ implementations of the MakeTrees module, the two functions in the Coding module were far apart in terms of difficulty. encodeMessage seemed difficult at first given the differing types as they were no longer Trees, but was solved easily given the lookup function underneath, applied to each element in [a]. This function was daunting at first due to its nature and how different it seemed from the functions in the MakeTrees module but did not prove to be a great challenge overall.

decodeMessage :: Ord a => Tree a -> HCode -> [a]

The dark horse of the entire implementation, the decompression function required thinking in the form of everything from abstract thought to visual representation. Drawing multiple trees and attempting to understand the problem given the SOS sheet from the PAL sessions took a while to complete but was rewarding as a helper function needed to be used – the actual function almost compiled but failed due to non-exhaustive pattern matching in the helper function. Using collaboration from peers once again, this issue in the recursive implementation which used ‘case’ of both the given tree and the given HCode was realised. The pattern did not match properly with an empty list of bits (HCode) and therefore the collaborated helper function implementation using the ‘where’ clause was better in terms of simplicity and straightforward implementation.

**Learning Outcomes**

**Learned:**

* Proper implementation of trees.
* Effective implementation of recursion.
* Effective use of prelude functions.
* Effective use of collaborative study.

The main result of working on this implementation proved to be the importance of collaborative thinking in addition to the importance of visual representation to break down a problem. Where the problem proved difficult, breaking it down into a visual representation and thinking about it in terms of logical and lossless terms instead of abstract thought processes proved its strength over blind trial and error. Collaborative thinking in terms of the PAL sessions also proved to be a strong way to solve problems which previously seemed conceptually difficult. An overall rewarding implementation assessment.

By Aditya Sharma – u6051965 – 05/05/2017