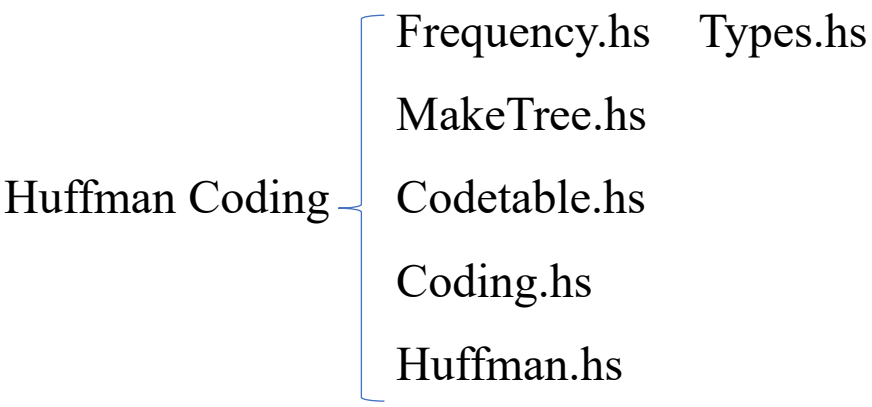
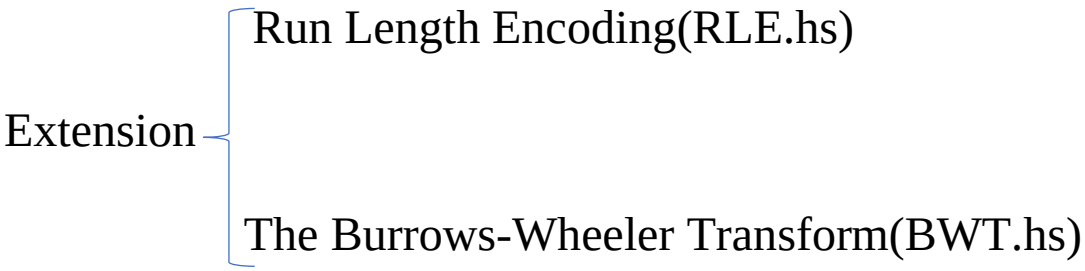
Report

Assignment 2: Huffman Coding

Name:	Zichen Zhang
Student Number:	U6161816
Course:	COMP1130
Date:	5/5/2017
Collaborators:	None

Structure:





Source:

I visited this page to learn the use of exclamation sign:

http://stackoverflow.com/questions/993112/what-does-the-exclamation-mark-mean-in-a-haskell-declaration

I researched about the Burrows-Wheeler Transform from Wikipedia:

https://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform

I learned how to use Test.QuickCheck from

https://hackage.haskell.org/package/QuickCheck-2.9.2/docs/Test-QuickCheck.html

I learned how to use sortBy function from

https://llaisdy.wordpress.com/2009/07/03/haskell-sort-and-sortby/

Implementation:

- Frequency.hs:

  Defining Functions:

  I defined the function “create” using recursion. Recursion helps me to break down the list of characters into single characters so that I can pass the characters one by one to our “inc” function to build up our new histogram.

  Understanding:

  One interesting understanding is that: to some extent, the function “inc” is quite similar to the list constructor “:”. Both of them attach a single element to a collection of elements so I just treated histogram as a list but not a new type.

  Testing:

  I tested the function “frequency” with “test1” defined at the bottom. Test1 returns “True” if our function is correct.

- MakeTree.hs

  Defining Functions:

  I defined “toTreeList” with a helper function to make the definition looks clean and short. An alternative way of defining this function is through recursion:

```
toTreeList [] = []
toTreeList ((x,n):xs) = Leaf n x : toTreeList xs
```

Function “value” is defined with pattern matching to extract the value we want.

The definition of “sort” is quite tricky. I tried hard to define it without using sortBy, resulting in a block of complicated code. But by using sortBy we can complete it with one line of code.

Then I made two mistakes one after another which taught me a lot about recursion:

1. My first mistake was that I forgot to sort the list. That was not a big mistake and I found and “corrected” it when I tested the function.
2. However, I added “sort” in a wrong position and I didn’t realise it until I tested the whole Huffman encoding. Below was the mistake I made:

   ```
   …
   mergeFirstPair (x:y:xs) = sort ((merge x y):xs)
   …
   ```

   I thought I could add my “sort” function to either of the functions “mergeFirstPair” or “makeCodes”. But I found that if I add the “sort” like what I did above, my argument (x:y:xs) is not sorted in the first loop (when the argument is freshly passed to this function). So this mistake reminds me that I should pay more attention to the first loop when

writing a recursion.

Testing:

I used the given “test” to test my code.

- CodeTable.hs

Testing:

I wrote a “test” function to test “codeTable”.

- Coding.hs

Implement:

I want to talk about “decodeMessage” here.

The reason why I chose to define this function with a helper function:

1.If I do recursions on out "decodeMessage" directly, the original tree will not be saved. So after I found one character at one Leaf and there is still HCode left, I cannot go back and start again.

2.I need another parameter "string" to collect the characters got from each recursion.

Testing:

I wrote two tests “test1” and “test2”, testing “decodeMessage” and “encodeMessage” respectively.

- Huffman.hs

Testing:

“encode” and “decode” tested together with “test”

- RLE.hs

  Implement:

  We can separate **“**rle_encode” into two parts: encoding and limiting the number of elements in one run. To encode the message, I just used recursion like what I did when dealing with Huffman encoding. However, I have to add guards to limit the number of elements per run. During the lab, I asked my tutor, Steven, about nesting one guard into another one but that is not allowed by Haskell. Instead, we can put our guard inside a helper function which is included in another guard to achieve the same goal.

  Testing:

  We can test “rle_encode” and “rle_decode” by the “test” function whose argument is about 400 in length (more than 255).

- BWT.hs

  Implement:

  The relatively difficult function to define is “make_bwt_table”. It is tough because we need to split our bwt code into single elements and distribute these elements one per row from top to bottom to our table. I handled this by defining 3 functions: helper, addbwt, splitlist.
  The job of “helper” is to carry on iteration and to sort the table after we combine our bwt code with the previous table.
  The job of “addbwt” is to add our bwt code to the table. (the first

addition should be paid more attention as we cannot add them by match their first element)

"splitlist" helps us to split the list into lists containing a single element.

Testing:

I wrote a function "testbwt" to test the BWT encoding.