**Assignment Submission Declaration**

**Name:** Seán Smith

**Programme:** B.Sc. in Computer Applications (Bachelor Honours Degree)

**Module Code:** CA320 – Computability & Complexity Assignments

**Assignment Title:** String Searching in Haskell

**Submission Date:** 17/12/2012

I declare that this material, which I now submit for assessment, is entirely my own

work and has not been taken from the work of others, save and to the extent that such

work has been cited and acknowledged within the text of my work. I understand that

plagiarism, collusion, and copying are grave and serious offences in the university and

accept the penalties that would be imposed should I engage in plagiarism, collusion or

copying. I have read and understood the Assignment Regulations set out in the

module documentation. I have identified and included the source of all facts, ideas,

opinions, and viewpoints of others in the assignment references. Direct quotations

from books, journal articles, internet sources, module text, or any other source

whatsoever are acknowledged and the source cited are identified in the assignment

references. This assignment, or any part of it, has not been previously submitted by

me or any other person for assessment on this or any other course of study.

Name: Sean Smith Date: 17/12/2012

**CA320 – Computability & Complexity - Assignment 2**

**Rabin-Karp string search algorithm**

***Research***

The Rabin-Karp algorithm is used for multiple pattern matching, rather than simplistic single pattern matching. A practical use of Rabin-Karp is detecting plagiarism. By providing source material, Rabin-Karp can rapidly search for instances of sentences from the source material. The technique in which Rabin-Karp employs is hashing and it is used to find a substring in a text.

***Personalised Functions***

An approach I pursued when attempting to implement the Rabin-Karp algorithm was to research all reliable Haskell functions and to use them to my advantage. As previously experienced in Lab Sessions, I researched thoroughly on multiple functions and generated my own personalised functions based on the information provided on ‘zvon.org’ (refer to references).

The following is a list of personalised list functions I have included upon my source code in order to implement the desired algorithm.

1. **myFirst**

Returns the first element in a pair.

1. **myFoldLeft**

Applies a function, to the second argument and the first item of the list.

1. **myLength**

Returns the number of elements in a list.

1. **myHead**

Returns the first element of a non-empty list.

1. **myLast**

Returns the last element of a non-empty list.

1. **myTake**

Returns the specified number of elements from the front of the list.

1. **myDrop**

Returns a list with the specified number of elements removed from the front of the list.

1. **myZip**

Returns a list of pairs which are formed by tupling together corresponding elements from both lists.

1. **myZipWith**

Returns a list containing elements formed by a specific function to corresponding elements in the list.

1. **myPair**

Returns two tuples formed from pairs of elements.

1. **myMap**

Returns a corrected binding of two values.

The following is used within this function:

* *Maybe* – provides a method of dealing with illegal values without

terminating the program.

* *Just* – correct result of maybe.
* *Nothing* – incorrect result of maybe.

1. **mySizeList**

Reutrns a list (within a list) with a size of length supplied in first argument.

1. **myID**

Returns identity of argument supplied.

1. **myFromMaybe**

Returns error-free value of argument supplied. Due to errors occurring with conflicting elements such as ‘char’ and ‘int’.

1. **myOrd**

Converts character to an integer using the pre-defined ‘fromEnum’ function.

Characters to integer begins at integer 97.

Note

‘fromEnum’ function was not personally implemented due to the requirements of predefined methods.

‘fromEnum’ was imported using *import.Data.List*

1. **mySubstring**

Returns a list with the specified number of elements removed from the front of the list.

***Hashing Functions***

The following are the functions in which I constructed in order to implement hashing within my algorithm.

1. **hash**

Returns an integer representing the amount of characters extracted from a String.

The fourth argument indicates how many characters to extract given the range provided by the first two arguments.

1. **remove\_Hash**

Removes the hash of the first character and pushes each character up by one position.

***Rabin-Karp Algorithm***

The main algorithm requires two inputs from the user.

* **Sentence** – the text which is to be searched upon.
* **Match** – the specific letters/word which is to be searched for in sentence.

The program functions by computing the hash value for the match and for the first characters of the sentence, where the first characters depends on the number of characters in the match.

The application then proceeds through the sentence comparing the match hash with the current version of the sentence hash each time. If the sentence and match hash to the same value, then a match is confirmed and the function is completed.

The program returns the index (using myZip function) within the sentence where the match was first found.

Note

Within my implementation I required the use of the following functions.

* **find**

This returns the element of a list that satisfies a predicate, which in this case is a match. I chose not to implement this function personally as it requires multiple pre-defined methods in order to implement.

Thefunction is structured to deal with illegal values and therefore uses the Maybe type.

* **Scanl**

The scan left function was used as a reduction mechanism. This assisted in viewing the value of the hash on each iteration.

Both function were imported using *import.Data.List*

***Rabin-Karp Execution***

*rabin\_Karp :: String -> String -> Int*

Example

*rabin\_Karp "Hello World" "Wor*"

= 6

**Boyer-Moore string search algorithm**

***Research***

The Boyer-Moore algorithm is the most optimal choice for exact matching. It is suitable for an input with large alphabet use or if the pattern is long. The speed of the Boyer-Moore algorithm derives from shifting the pattern to the right. The shifting patterns are based on the following two techniques:

1. Bad Character Shift Rule

* Avoids repeating unsuccessful comparisons against a specific character.
* Can shift pattern to left, which will not guarantee progress.
* Works well for large alphabets (e.g. the English alphabet).
* A worst case time of O(nm).

1. Good Suffix Shift Rule

* Shift patterns only to the right, which guarantees progress.
* Work well with small alphabets (e.g. “DNA”).
* A worst case time of O(n).

I decided to implement the Boyer-Moore algorithm using the ‘good suffix shift rule’. The reason for this is because it provides sufficient complexity time and guarantees progress.

***Personalised Functions***

Another approach I pursued when attempting to implement to Boyer-Moore algorithm was derived from the Rabin-Karp research documented earlier. I adopted the technique of implementing my own personalised functions, in order to gain an understanding of how I could use functions to my primary advantage in relation to my algorithm.

The following is a list of functions that was implemented within my program.

1. **myHead**

Returns the first element of a non-empty list.

1. **myLast**

Returns the last element of a non-empty list.

1. **myTail**

Returns a list excluding the first element.

1. **myInit**

Returns all elements of a list, excluding the last element.

1. **myTake**

Returns the specified number of elements from the front of the list.

1. **myDrop**

Returns a list with the specified number of elements removed from the front of the list.

1. **myLength**

Returns the number of elements in a list.

1. **myZip**

Returns a list of pairs which are formed by tupling together corresponding elements from both lists.

1. **myZipWith**

Returns a list containing elements formed by a specific function to corresponding elements in the list.

1. **myPair**

Returns two tuples formed from pairs of elements.

1. **myMap**

Returns a list where each element is the result of applying a specific function to the corresponding element in the list.

1. **myLookup**

Returns the corresponding value, depending on if the value is in the map.

***Boyer-Moore Functions***

The following is the functions in which I constructed in order to use the ‘Good Suffix Rule’ approach, within my algorithm.

1. **unequal\_Number**

Returns the third parameter if the last element of both lists are not equal

(‘/=’) .

Otherwise invoke recursion by returning both lists with the next element to the left as the last elements.

1. **good\_Index**

Returns the index of a specific element within the list that corresponds to the suffix of two lists.

By performing the ‘myDrop’ function and the ‘myTake’ function separately within the ‘isSuffixOf’ function, it will confirm that the first list is a suffix of the second list.

The ‘myLength’ function, is used to satisfy that no invalid input was provided.

Note

isSuffixOf function was not personally implemented due to the requirements of outside features, such as ‘isPrefixOf’ function.

isSuffixOf was imported using *import.Data.List*

1. **good\_Suffix\_Rule**

Returns a list of pairs.

The first tupple is the index in which the pair is ordered in and the second tupple. The second tupple is the length of the list supplied in first argument.

However the second tupple of the first pair remains ‘1’. This was implemented within the ‘myZip’ function in order to avoid a non-exhaustive pattern error in the ‘myDrop’ function. Regardless, this has no immediate effect on the final outcome of the algorithm.

***Boyer-Moore Algorithm***

The main algorithm requires two inputs from the user.

* **Sentence** – the text which is to be searched upon.
* **Match** – the specific letters/word which is to be searched for in sentence.

The algorithm extracts the matching letters by taking the length of match and using it as a shifting pattern to check against the sentence. In case of a mismatch, the algorithm shifts right (using the length of match as the size of shift) until the first occurrence of match has occurred in sentence.

The function ‘unequal\_number’ indicates if a pattern was found. If the letters of match corresponded with the letters of sentence then it would return ‘1’ (indicating that elements in match and sentence were equal), otherwise it would return the length of match.

The algorithm is recursively invoked until the base case is called, which signifies that all checking has been completed within sentence.

***Boyer-Moore Execution***

*boyer\_Moore :: [Char] -> [Char] -> [Int]*

Example

*boyer\_Moore "Hello World" "Hello"*

= [1,5,5]

***References***

Zvon.org – provides type and description relating to haskell functions

<http://zvon.org/other/haskell/Outputglobal/index.html>

Researched information on Rabin-Karp String Search Algorithm

(Code included within this website is not related to the Haskell programming language)

<http://www.stoimen.com/blog/2012/04/02/computer-algorithms-rabin-karp-string-searching/>

Researched information on Boyer Moore String Search Algorithm

(Code included within this website is not related to the Haskell programming language)

<http://www.stoimen.com/blog/2012/04/17/computer-algorithms-boyer-moore-string-search-and-matching/>