Report of Assignment 7

Haodong Liao & Liyuan Zhang

July 12, 2019

1 Preface

This was a LaTeXreport written by Haodong Liao and Liyuan Zhang, students of UESTC who participated in the summer school of UCPH. More codes of this summer school were at this repository.

2 Introduction

A string in F# was a sequence of characters[1], and some of the string functions were similar to list functions, after the practicing of the list function, it was time to move on.

In this assignment, we worked with simple text processing and met the requirements of analyzing and generating some text according to statistics. In detail, we used pattern-matching and explicit type definition to create our own recursive functions, besides, we used powerful List-library functions to read, convert, analyze and generate target text.

3 Analysis and Design

3.1 Text Processing of character

3.1.1 ReadText

```
readText: filename:string -> string
// readFile.fsx
let filename = "readFile.fsx"
let line =
  try
    let reader = System.IO.File.OpenText filename
    reader.ReadToEnd ()
  with
    _ -> "" // The file cannot be read, so we return an empty string
printfn "%s" line
```

In this sub-assignment, we needed to convert *readFile.fsx* into a more flexible function which reads the content of any text file.

Referred to the input processing code in assignment 6, there were two branches needed to be handled, one was to set the entry point of our code, the other was to change the fixed argument to a flexible one so that the function could read the content of any text file. The key part of the pseudocode was shown as following:

```
[<EntryPoint>]
let main (input argument) =
  if (argument correct)
    call readText function with responding name of file
  else
    display error message
```

3.1.2 ConvertText

```
convertText: src:string -> string
```

[1] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/lists

The goal of this sub-assignment was to convert letters of a string to lower case and removes all characters except a..z. According to [1], there was a *ToLower()* function *returns a copy of the string where each letter has been converted to lower case*, so we could use it to finish the 'convert' part and moved on to 'remove' part. The key part of the pseudocode is shown as following:

```
let convertText inputString =
  let lowerStr = inputString.ToLower()
    if (elment of lowerStr was in the range of a..z)
      keep the current element
    else remove the current element
```

3.1.3 Get Histogram

```
histogram: src:string -> int list
```

We needed to count occurrences of each lower-case letter of the English alphabet in a string and returned a list in this sub-assignment. According to [1], there was a *String.filter* function similar to *List.filter* function, we used it to filter the specific letter of the string and count the length of it to get the occurrences. The key part of the pseudocode was shown as following:

```
let countchar (src:string)(letter:char) =
      if letter was in the range of a..z then
        filter the current letter
        handle the rest string
      else
        returned an empty list
let histogram (src:string): int list = (countchar src 'a')
```

3.1.4 Generate Ramdom String

```
randomString: hist: int list -> len:int -> string
```

The requirement of this sub-assignment was to change the given program and generate a string of a given length and contained random characters distributed according to a given histogram which used our own functions. The flow of the program was shown as following:

```
step1 - Read the target text file and saved it
step2 - Converted the text to lower case & removed characters except a..z
step3 - Got the histogram of the text
step4 - Generated a new random string according to the histogram
```

3.2 Text Processing of pair of characters

3.2.1 Get Cooccurrence

In this sub-assignment, we needed to count occurrences of each pair of lower-case letter of the English alphabet in a string and return a list of lists. Our solution was to use double recursion and process the letter in each pair separately. The key part of the pseudocode was shown as following:

```
// count the occurrence of each pair
let countnum =
  if the length of src > 2 then
    if the first two char = charpair then
      1 + sum recursively
    else
      ignore current and sum recursively
  else
    return 0
// handle the second letter
let countcharpair =
  if the second letter <= 'z' then
    sum the occurrence and process the left string recursively
```

```
    else return an empty list
// process the first letter
let countchar =
    if the first letter <= 'z' then
        process the second letter and process the left string recursively
    else return an empty list
// call function from here
let cooccurrence (src:string) = countchar src 'a'
```

3.2.2 Markov Model

```
fstOrderMarkovModel: cooc: int list list -> len:int -> string
```

We needed to generate a random string of length *len*, whose character pairs were distributed according to a specified cooccurrence histogram *cooc* of the story "Little Claus and Big Claus" in this sub-assignment. The flow of the program was shown as following:

```
step1 - Count the occurrence of each pairs (like 'aa' to 'az')
step2 - Count the coocurrence of all pairs ('aa' to 'zz')
step3 - Calculate the ratio of each pairs
step4 - Generate a random string of length len
    according to a given cooccurrence and the ratio of each pairs
```

4 Program description

Our implementation of **readText** function was as follows:

```
let printErrorMessage () =
    printfn "Program Input should be 'readfile filename'"

let readText (filename:string) : string =
    let line =
        try
            let reader = System.IO.File.OpenText filename
            reader.ReadToEnd ()
        with
            _ -> "" // The file cannot be read, so we return an empty string
    printfn "%s" line
    line
[<EntryPoint>]
let main (paramList : string[]) : int =
    if paramList.Length <> 2 then
        printErrorMessage ()
        0
    else
        match paramList.[0] with
            | "readfile" ->
                let str = readText paramList.[1]
                printfn "str : %A" str
            | _ ->
                printErrorMessage ()
        1
```

As it showed, the key to this function was to match the argument list, the first argument should be the command 'readfile', and the second argument was the name of a file.

Implementation of **convertText** function was as follows:

```
let rec convertText (src:string) : string =
    let lowerSRC = src.ToLower()
    match lowerSRC with
    | "" -> ""
    | elm ->
```

```
        if ('a' <= elm.[0] && elm.[0] <= 'z')
        then
            elm.[0].ToString() + (convertText elm.[1..])
        else
            convertText elm.[1..]
```

It made us wondering whether there was an expression of string could work as 'elm::rest' to represent a list at the beginning, things did not go well so we changed our mind to use elm[1..] to represent the rest of a string.

Implementation of **histogram** function was as follows:

```
let rec countchar (src:string)(letter:char): int list =
  if letter >= 'a' && letter <= 'z' then
    ((String.filter (fun char -> char=letter) src).Length)::
     (countchar src (char (int letter + 1)))
  else []
let histogram (src:string): int list = (countchar src 'a')
```

This function was the key to solving the following problems.

Assignment 7.4 was a summary of the previous assignments, and the key part to it was as follows:

```
/// step1 readfile
let str = readText "littleClausAndBigClaus.txt"
/// step2 convertfile
let convertRes = convertText str
/// step3 get histogram
let hist = histogram convertRes
let alphabet = List.init hist.Length (fun i -> 'a' + char i)
printfn "A histogram:\n %A" (List.zip alphabet hist)
/// step4 generages a new random string
let ranStr = randomString hist convertRes.Length
printfn "A random string: %s" ranStr
let newHist = histogram ranStr
printfn "Resulting histogram:\n %A" (List.zip alphabet newHist)
```

It was important to have a clear thought of processing flow.

Implementation of **cooccurrence** function was as follows:

```
let rec countnum (src:string) (charpair:string):int =
  if src.Length >=2 then
    if src.[0..1] = charpair then
      1+(countnum src.[1..] charpair)
    else
      (countnum src.[1..] charpair)
  else
    0
let rec countchapair (src:string) (letter1:char) (letter2:char):int list =
  if letter2 <= 'z' then
    let charpair = string letter1 + string letter2
    (countnum src charpair)::(countcharpair src letter1 (letter2+char 1))
  else []
let rec countchar (src:string) (letter1:char):int list list =
  if letter1 <= 'z' then
    (countcharpair src letter1 'a')::(countchar src (letter1 + char 1))
  else []
let cooccurrence (src:string):int list list =
  countchar src 'a'
```

This was a difficult problem for us, and the key to the solution was the idea of 'divide and conquer'. If we could not process the pair of letters at a time, we handle the letter of it separately.

Assignment 7.6 was a summary of all the previous assignments, and the key part of the implementation of it was as follows:

```
let fstOrderMarkovModel (cooc:int list list) (len:int):string =
    let strLengthlist =
      List.map (fun x -> (float x)) (List.map (List.sum) cooc)
```

```
4       let strLengthSum = List.sum strLengthlist
5       let ratiolist = List.map (fun x -> x/strLengthSum) strLengthlist
6       let randomStringAlt (hist: int list):string = randomString hist
7         (int((float (List.sum (hist))/strLengthSum)*(float len)))
8       let strlist = List.map (randomStringAlt) cooc
9       let composestr (acc:string) (elm:string):string = acc + elm
10      List.fold composestr "" strlist
11  printfn "test: \n %A" (fstOrderMarkovModel (cooccurrence a) (a.Length))
12  printfn "present cooc : \n %A"
13    (cooccurrence (fstOrderMarkovModel (cooccurrence a) (a.Length)))
```

This was another difficult assignment for us, the main challenge was to arrange the text properly so the occurrence was similar. We tried to generate a new character according to the letter before the current letter, but things just did not work well for reason we did not figure out, then we tried with the weight of the pair based on the proportion of the sum of the single letters to the sum of the occurrences of all the letters . In this way, when there was a given length, the length of each character list (like 'aa'..'az') would be its weight timed the given length, such as: $Length_{'aa'to'az'} = \frac{Occurrence_{'aa'to'az'}}{Occurrence_{'aa'to'zz'}} * (GivenLength)$, so that we could generate a random string which had a similar distribution but different content.

5 Evaluation

The testing environment was macOS Mojave 10.14.5 system with iTerm and Microsoft (R) F# Compiler version 4.1.

We combined all the assignment into a single program, compiled and tested it with parameter *readfile littleClausAndBigClaus.txt*. Output of it included *content of target file*, *converted string of original string*, *histogram of original string*, *random string of single character*, *histogram of random single character*, *cooc of original string*, *cooc of random single character*, *random string of character pairs* and *cooc of random character pairs*. Part of results (because it was too long) were shown as following (the complete results could be seen at result):

```
→ fsharpc --nologo Assig2.fsx && mono Assig2.exe readfile littleClausAndBigClaus.txt
str :
In a village there lived two men who had the self-same name. Both were named Claus. But
```

Figure 1: Testing of readText function

```
cooc of original string:
[[0; 35; 64; 77; 0; 12; 31; 7; 84; 0; 26; 64; 28; 228; 0; 16; 0; 100; 147; 141;
  122; 36; 26; 3; 43; 3];
 [19; 0; 0; 0; 64; 0; 0; 0; 44; 0; 0; 12; 0; 0; 29; 0; 0; 8; 0; 0; 61; 0; 0; 0;
  16; 0];
```

Figure 2: Testing of fstOrderMarkovModel function

```
cooc of random character pairs:
[[148; 23; 21; 42; 214; 22; 13; 137; 95; 1; 24; 74; 25; 31; 124; 14; 0; 33; 48;
  90; 30; 3; 35; 3; 27; 0];
 [19; 8; 4; 14; 19; 4; 6; 21; 12; 0; 2; 9; 2; 11; 20; 5; 1; 6; 28; 26; 9; 1; 13;
  0; 5; 0];
```

Figure 3: Coocurrence of fstOrderMarkovModel function

6 Conclusion

In this assignment, we worked with simple text processing and met the requirements of analyzing and generating target text according to statistics. It was time-consuming of thinking about the recursive function and avoiding to use things like 'for loop' when we were stuck, it was challenging but this assignment made us get more familiar with F# and functional programming indeed. It was our pleasure to have teachers like Prof.Sporring and we believed that we had a good time. Many thanks!

References

[1] Jon Sporring. *Learning to Program with F#.* 2019.