Learning to Program with F#
Exercises
Department of Computer Science
University of Copenhagen

Jon Sporring, Martin Elsman, Torben Mogensen, Christina Lioma

October 21, 2022

0.1 Random Text

0.1.1 Exercise(s)

0.1.1.1: Write a function,

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

which counts occurrences of each of the characters `['a'..'z']@[' ']` in a string and returns a list. The first element of the list should be the count of 'a's, second the count of 'b's etc. Use this function to replace the mockup function in your library.

0.1.1.2: The function `markovChain` is a random function, and you are to test its output by the cooccurrences of the characters in the string it produces.

Extend your library with the function,

```
diff2 : c1:int list list -> c2:int list list -> double
```

which compares two histograms as the average sum of squared differences,

$$\text{diff2}(c_1, c_2) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (c_1(i,j) - c_2(i,j))^2 \tag{1}$$

where c_1 and c_2 are two cooccurrence histograms of N elements such that $c_1(i,j)$ is the number of times character number i is found following character number j.

Extend your test file with a test of `markovChain` as follows:

(a) Convert The Story using `convertText` and calculate its cooccurrence histogram.

(b) Use this to generate a random text using `markovChain` with length N, where N is the length of the converted The Story.

(c) Calculate the distance between the cooccurrence histograms of The Story and the random texts using `diff2`.

(d) Set the test as passed if the difference is below some threshold.

You must choose a threshold in the above, and you must argue for your choice.

0.1.1.3: The function `randomString` is not easily tested using white- or blackbox testing since it is a random function. Instead you are to test its output by the histogram of the characters in the string it produces.

Extend your library with the function,

```
diff : h1:int list -> h2:int list -> double
```

which compares two histograms as the average sum of squared differences,

$$\text{diff}(h_1, h_2) = \frac{1}{N} \sum_{i=0}^{N-1} (h_1(i) - h_2(i))^2 \tag{2}$$

where h_1 and h_2 are two histograms of N elements.

Extend your test file with a test of `randomString` as follows:

(a) Convert The Story using `convertText` and calculate its histogram.

(b) Use this to generate a random text using `randomString` with length N, where N is the length of the converted The Story.

(c) Calculate the distance between the histograms of The Story and the random texts using `diff`.

(d) Set the test as passed if the difference is below some threshold.

You must choose a threshold in the above, and you must argue for your choice.

0.1.1.4: Add a function to your library which converts a string, such that all letters are converted to lower case, and removes all characters except a...z and space. It should have the following type:

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

0.1.1.5: The script `mockup.fsx` contains a number of functions including

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

The function `randomString` generates an identically and independently distributed string of a given length, where the characters are distributed according to a given histogram. The script is complete in the sense that it compiles and runs without errors, but its `histogram` function is a mockup function and does not produce the correct histograms.

Create the library file `textAnalysis.fs` and add the functions from `mockup.fsx`

0.1.1.6: Extend your library with a function

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

which generates a random string of length `len`, whose character pairs are distributed according to the cooccurrence histogram `cooc`.

0.1.1.7: Write a program that generates a random string of length `len`, whose character triples are distributed according to a user specified trioccurrence histogram `trioc`. The function must have the type:

```
sndOrderMarkovModel : trioc:int list list list -> len:int -> string
```

Use the function developed in Exercise **??**, and test your function by generating a random string, whose character triples are distributed as the converted characters in H.C. Andersen's fairy tale, "Little Claus and Big Claus". Calculate the trioccurrence histogram for the random string, and compare this with the original trioccurrence histogram.

0.1.1.8: The function `randomString` may be considered a zero-order Markov Chain model, since each generated character is independent on any previously generated characters. The function `fstOrderMarkovModel` generates new characters dependent on the previous character and is therefore a first-order Markov Chain model. Consider a similar extension to an n'th-order Markov Chain where the occurrences of n-tupples of characters are stored in `n : int list list ... list`. What possible pit-falls are there with this representation?

0.1.1.9: (a) The script `readFile.fsx` reads the content of the text file `readFile.fsx`. Convert this script into a function which reads the content of any text file and has the following type:

```
readText : filename:string -> string
```

(b) Write a program that converts a string, such that all letters are converted to lower case, and removes all characters except a...z. It should have the following type:

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

(c) Write a program that counts occurrences of each lower-case letter of the English alphabet in a string and returns a list. The first element of the list should be the count of 'a's, second the count of 'b's etc. The function must have the type:

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

(d) The script `sampleAssignment.fsx` contains the function

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

which generates a string of a given length, and contains random characters distributed according to a given histogram. Modify the code to use your histogram function. Further, write a program, which reads the text `littleClausAndBigClaus.txt` using `readText`, converts it using `convertText`, and calculates its histogram and generates a new random string using `histogram` and `randomString`. Test the quality of your code by comparing the histograms of the two texts.

(e) Write a program that counts occurrences of each pairs of lower-case letter of the English alphabet in a string and returns a list of lists (a table). The first list should be the count of 'a' followed by 'a's, 'b's, etc., second list should be the count of 'b' followed by 'a's, 'b's, etc. etc.

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

(f) Write a program that generates a random string of length `len`, whose character pairs are distributed according to a user specified cooccurrence histogram `cooc`. The function must have the type:

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

Test your function by generating a random string, whose character pairs are distributed as the converted characters in H.C. Andersen's fairy tale, "Little Claus and Big Claus", calculate the cooccurrence histogram for the random string, and compare this with the original cooccurrence histogram.

0.1.1.10: Extend your library with a function

```
randomWords : wHist:Tree list -> nWords:int -> string
```

which generates a string with `nWords` number of words randomly selected to match the word-histogram in `wHist`.

0.1.1.11: The function `randomWords` is a random function, and you are to test its output by the histogram of the words in the string it produces.

Extend your library with the function,

```
diffw : t1:Tree list -> t2:Tree list -> double
```

which compares two word-histograms as the average sum of squared differences,

$$\text{diffw}(t_1,t_2) = \frac{1}{M}\sum_{i=0}^{M-1}\left(t_1(i)-t_2(i)\right)^2 \tag{3}$$

where t_1 and t_2 are two word histograms, and M is the total number of different words observed in the two texts.

Extend your test file with a test of `randomWords` as follows:

(a) Convert The Story using `convertText` and calculate its word histogram.

(b) Use this to generate a random text using `randomWords` with M words, where M is the number of words in the converted The Story.

(c) Calculate the distance between the word histograms of The Story and the random texts using `diffw`.

(d) Set the test as passed if the difference is below some threshold.

You must choose a threshold in the above, and you must argue for your choice.

0.1.1.12: The function `randomWords` is a random function, and you are to test its output by the histogram of the words in the string it produces.

Extend your library with the function,

```
diffw : w1: wordHistogram -> w2:wordHistogram -> double
```

which compares two word-histograms as the average sum of squared differences,

$$\text{diffw}(w_1, w_2) = \frac{1}{M} \sum_{i=0}^{M-1} (w_1(i) - w_2(i))^2 \tag{4}$$

where w_1 and w_2 are two word histograms, and M is the total number of different words observed in the two texts.

Extend your test file with a test of `randomWords` as follows:

(a) Convert The Story using `convertText` and calculate its word histogram.

(b) Use this to generate a random text using `randomWords` with M words, where M is the number of words in the converted The Story.

(c) Calculate the distance between the word histograms of The Story and the random texts using `diffw`.

(d) Set the test as passed if the difference is below some threshold.

You must choose a threshold in the above, and you must argue for your choice.

0.1.1.13: Extend your library with a function

```
randomWords : wHist:wordHistogram -> nWords:int -> string
```

which generates a string with `nWords` number of words randomly selected to match the word-histogram in `wHist`.

0.1.1.14: In terms of a Markov Chain, what order is `randomWords`? Suggest an extension of `type Tree` to include first order Markov Chains. Speculate on whether this principle can be used to extend to n'th order mordels, and, in case, speculate on how the storage requirements will grow as n grows.

0.1.1.15: The script `readFile.fsx` reads the content of the text file `readFile.fsx`. Convert this script into a function which can read the content of any text file and has the following type:

```
readText : filename:string -> string
```

Add this function to your library.

0.1.1.16: Write a short report, which

- is no larger than 5 pages;
- includes answers to questions posed;
- contains a brief discussion on how your implementation works, and if there are any possible alternative implementations, and in that case, why you chose the one, you did;
- includes output that demonstrates that your solutions work as intented.

0.1.1.17: Extend your test file with a white-box test of `cooccurrence`.

0.1.1.18: Write a white-box test of the library functions `readText`, `convertText`, and `histogram`, and add these to your test file.

0.1.1.19: Write a program which reads The Story discard all characters that are not in `['a'..'z', 'A'..'Z',' ']`, convert all the remaining characters to lower case and calculate the occurrence of the remaining words as a `Tree list` type.

0.1.1.20: Write a white-box test of `wordHistogram`, and add it to your test file.

0.1.1.21: Extend your test file with a white-box test of `cooccurrenceOfWords`.

0.1.1.22: Write a program that counts occurrences of each triple of lower-case letter of the English alphabet in a string and returns a list of lists of lists. The program must include the function

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

to calculate the number of occurrences of triples.

0.1.1.23: Extend your library with the function

```
type wordCooccurrences = (string * wordHistogram) list
cooccurrenceOfWords : src:string -> wordCooccurrences
```

which counts occurrences of each pair of words in a string and returns a list of pairs. Each returned pair must be a word and the wordHistogram of counts of cooccurrences. E.g., for the string “a hat and a cat” the function must return,

```
[("a", [("hat", 1); ("cat", 1)]);
 ("and", [("a", 1)]);
 ("cat", []);
 ("hat", [("and", 1)])]
```

0.1.1.24: Extend your library with a function that counts occurrences of each word in a string and returns a list. The counts must be organized as a list of trees using the following `Tree` type:

```
type Tree = Node of char * int * Tree list
```

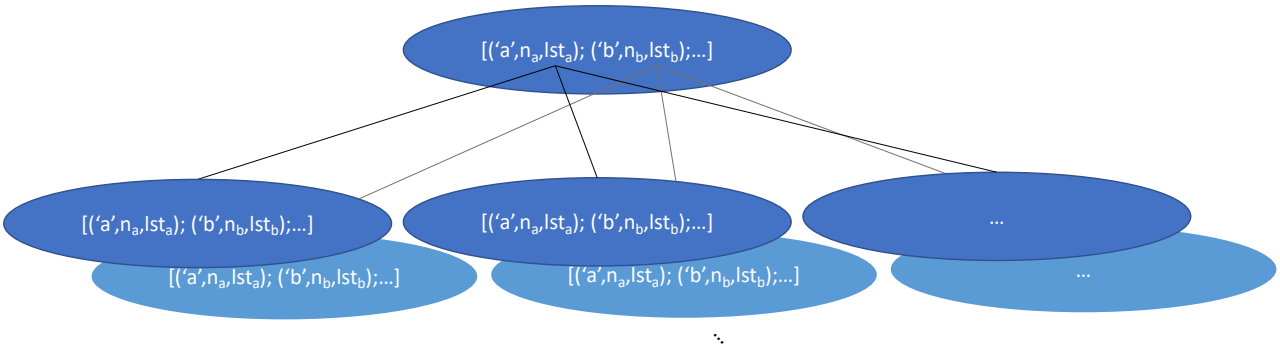

Figure 1: An illustration of a list of values of the type `Tree`.

An illustration of a value of this type is shown in Figure 3.1 Words are to be represented as the sequence of characters from the root til a node. The associated integer to each node counts the occurrence of a word ending in that node. Thus, if the count is 0, then no word with that endpoint has occurred. For example, a string with the words "a abc ba" should result in the following tree,

```
[Node ('a', 1, [Node ('b', 0, [Node ('c', 1, [])])]);
 Node ('b', 0, [Node ('a', 1, [])])]
```

Notice, the counts are zero for the combinations "ab" and "b", which are words not observed in the string. The function must have the type:

```
wordHistogram : src:string -> Tree list
```

0.1.1.25: The function `wordMarkovChain` is a random function, and you are to test its output by the cooccurrences of the words in the string it produces.

Extend your library with the function,

```
diffw2 : c1:wordCooccurrences -> c2:wordCooccurrences -> double
```

which compares two cooccurrence histograms as the average sum of squared differences,

$$\text{diffw2}(c_1, c_2) = \frac{1}{M^2} \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} \left(c_1(i,j) - c_2(i,j)\right)^2 \tag{5}$$

where c_1 and c_2 are two cooccurrence histograms of M elements such that $c_1(i,j)$ is the number of times word number i is found following word number j.

Extend your test file with a test of `wordMarkovChain` as follows:

(a) Convert The Story using `convertText` and calculate its cooccurrence histogram.

(b) Use this to generate a random text using `wordMarkovChain` with length M, where M is the length of the converted The Story.

(c) Calculate the distance between the cooccurrence histograms of The Story and the random texts using `diffw2`.

(d) Set the test as passed if the difference is below some threshold.

You must choose a threshold in the above, and you must argue for your choice.

0.1.1.26: Extend your library with a function that counts occurrences of each word in a string and returns a list. The list must be organized using the following type:

```
type wordHistogram = (string * int) list
```

For example, a string with the words "a abc ba ba" should result in the following list,

```
[("a", 1); ("abc", 1); ("ba",2)]
```

The function must have the type:

```
wordHistogram : src:string -> wordHistogram
```

0.1.1.27: Extend your library with a function

```
wordMarkovChain : wCooc: wordCooccurrences -> nWords:int -> string
```

which generates a random string with `nWords` words, whose word-pairs are distributed according to the cooccurrence histogram `wCooc`.