Declarative Programming Coursework

Pandelis Zembashis, S15101590

July 31, 2017

Contents

1 Rot13

1.1 Brief

Implement a rot-13 utility program (like tr, which acts like a common UNIX utility, performing a line-by-line rot-13 encoding of every line of input contained in each file listed on its command line.

1.2 Planning and Design

ROT13 is a type of Caesar cypher in which each character is replaced with the corresponding letter 13 spaces down the alphabet. When the cypher gets to the end of the usable alphabet it looks back around to A.

1.2.1 Defining behaviour

As we aim to build a command line utility program we will be creating a simple argument based console utility. We will be defining a single entry point to the ROT13 utility in which we will accept a single argument. This argument will be the relative or absolute file path to a file on the users disc.

We may also want to implement a method for accepting a bigger input to the utility in order to be able to pipe a file into it. Such that you could use the cat utility to pipe the contents of a file into ROT13.

Listing 1: Examples of how a user will interact with the utility from the console

```
$ rot13 file.txt
$ rot13 /user/home/u/file.txt
$ cat file.txt | rot13
```

1.2.2 Approach

As we are essentially dealing with a type of encryption we will also face the difficulty of verifying that our implementation works and is reliable. Due to the expected and highly reproducible behaviour of encryption and cyphers the same input will always have the same output.

In order to make sure our code is reliable and is doing exactly what we expect it to do during all phases of development we will be using a test driven development process. This means at every stage we will be defining a test for whatever function or feature we are writing at the time. That feature or function will not be deemed as working until it passes its test. When it does it will be deemed reliable enough to use repeatedly throughout the rest of the application.

This will allow us to build up and reuse our functions piece by piece and quickly fix any errors with the functions we work on later on because we will have already tested the functions feeding into the later functions earlier.

In order to keep the code style as declarative as possible, we will only write unary functions that can be easily pipelined. As we build up our application we will add to the pipeline eventually creating the full feature set of operations.

This will also aid in our testing as we can test minimal code and write many more tests.

1.2.3 Technical Requirements

In order to effectively define and execute tests we will be needed a testing framework. I have chosen to use NUnit and all test will be defined within the NUnit framework.

In order to keep our code clean and reusable the project will be split into 4 files and modules. The files will be (in execution order):

1. File.fs - File IO operations
2. Rot.fs - ROT encryption and transformation
3. Tests.fs - Unit tests
4. Main.fs - Entry point where File meets Rot function

1.3 Implementation

The full source code is available in the appendix. Please note that unit tests will be shown inline with their accompanying function where applicable. In reality all tests exist in Tests.fs

Please also note that double single quotes (”) are used in test function names as there were problems displaying them as double back ticks (`)

1.3.1 Rot.fs

Rot.fs is the main logic of our utility. This is where the shifting and cyphering will take place. This file contains the following modules and functions.

```
module Rot
module Alphabet
    explode: string -> char list
    implode: char list -> string
module Encrypt
    isUpper: char -> bool
    toLower: char -> char
    toUpper: char -> char
    rec shift: char -> char
    rotify: string -> string
```

I have split Rot down into two submodules. Rot.Alphabet and Rot.Encrypt. Rot.Alphabet handles transforming the incoming strings to the appropriate char

Lists while Rot.Encrypt carries out the cyphering of the incoming text, utilising Rot.Alphabet

module Alphabet

```
let explode (s:string) =
        [for char in s -> char]

[<Test>]
let ''Expand a string into a list with each
   character as an element''() =
      let i = "Hello"
      let list = ['H';'e';'l';'l';'o']
      Assert.AreEqual(list, Rot.Alphabet.explode i)
```

Explode takes a string argument and will expand it out into a List of Chars. For every **char** found in the string **s** it will yield the **char** at that position.

```
let implode (xs:char list) =
      let sb = System.Text.StringBuilder(xs.Length)
      xs |> List.iter (sb.Append >> ignore)
      sb.ToString()

[<Test>]
let ''Implode the created list into a string
   containing the characters in the list''() =
      let list = ['H';'e';'l';'l';'o']
      Assert.AreEqual("Hello", Rot.Alphabet.implode
         list)
```

Implode as the name suggests does the opposite of explode. It takes a List of Chars and returns back a string of those chars. Unfortunately I couldn't not find a way around piping the output of sb.Append to ignore so this function is not as declarative as it could be.

module Encrypt

```
let isUpper (c:char) =
    System.Char.IsUpper c

let toLower (c:char) =
    System.Char.ToLower c

let toUpper (c:char) =
    System.Char.ToUpper c
```

These functions exists purely as syntactic sugar to make it easier to use them in the next function call.

```
let rec shift (c:char) =
     if(System.Char.IsLetter c) then
         let num = int(c) - int('A')
         let offsetNum = (num+13)%26
         if(isUpper c) then
             char(offsetNum + int('A'))
         else
             toUpper(c)
             |> shift
             |> toLower
     else
         c
[<Test>]
let ''Handle non text shifting''() =
   let i = '@'
   let o = '@'
   Assert.AreEqual(o, Rot.Encrypt.shift(i))

[<Test>]
let ''Shift 13 characters allong alphabet''() =
   let i = 'A'
   let o = 'N'
   Assert.AreEqual(o, Rot.Encrypt.shift(i))

[<Test>]
let ''Shift 13 characters allong alphabet lowercase
   ''() =
   let i = 'a'
   let o = 'n'
   Assert.AreEqual(o, Rot.Encrypt.shift(i))
```

Shift is what takes care of the cyphering process. It accepts a character and will then check to see if the character is uppercase. If it is uppercase it will get the raw character value and then subtract it from the character value for A.

In order to wrap around the alphabet our offsetNum will be a the remainder of the length of the alphabet, 26. The resulting character code is offsetNum + our starting character A. It then returns the appropriate character for the value

In order to make our logic work without writing too much extra code I chose to pass the lowercase case letter back through shift recursively as uppercase and then back to lowercase. This is an area that could be improved on.

There is also a check to see if the character is in fact a letter. If it is not a letter we cannot cypher it so we pass it back untouched.

```
let rotify(str:string) =
    Alphabet.explode(str)
    |> List.map (fun c -> shift(c))
```

```
    |> Alphabet.implode

[<Test>]
let ''Rot13 this string''() =
    let i = "
        ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
        "
    let o = "
        NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm
        "
    Assert.AreEqual(o, Rot.Encrypt.rotify(i))

[<Test>]
let ''Handle non text inputs''() =
    let i = "ABCD!FGHI-KLMNO4QRSTUV6XYZabcdefghijk-
        mnopqrs@uvwxyz"
    let o = "NOPQ!STUV-XYZAB4DEFGHI6KLMnopqrstuvwx-
        zabcdef@hijklm"
    Assert.AreEqual(o, Rot.Encrypt.rotify(i))
```

The rotify function takes all our independent function logic and creates a pipeline. We take in a string and explode it into a list of characters. We then pipe that list into a map function which applies the shift on each character. Finally we return back a string of the rotified text.

1.3.2 File.fs

File.fs handles our incoming file from the system. We want to do some checks to make sure the file actually exists before we start passing it to our rotify functions so we handle that here.

```
module FileIO
    Open: string -> string list
```

module FileIO

```
  let Open(path:string) =
      match File.Exists(path) with
      | true ->
          File.ReadAllLines(path)
          |> Array.toList
      | _ -> invalidArg "file" "The file specified
         does not exist"
```

When supplying the utility with a file argument we want to check that the file actually exists. We do a pattern match on File.Exists() and if it is true we can continue and read out all the lines to an Array.

If not we return an error

1.3.3 Main.fs

```
namespace Rot13
module Main
    entry: string[] -> int
```

module Main

```
[<EntryPoint>]
let entry argv =
  if (argv.Length = 0) then
        printfn("Please provide the route to a file
           on the system to process")
        1 //No file found so return an error exit
           code
  else
    FileIO.Open argv.[0]
    |> List.map(fun line -> Rot.Encrypt.rotify(line)
        |> printfn("%s") )
    |> ignore
  0
```

This is where our utility entry point exists. Before we even attempt to read a file we need to check that the user has given us a path to look in first. If not we will let the user know via a printfn and exit with an error code of 1.

If we do have a path we can attempt to open the file. Once opened FileIO gives us an Array of lines. We can map over that array and rotify each of them and subsequently print them to stdout. We don't expect an output from printfn so we have to send it to ignore.

The utility then exits with a code 0.

1.4 Solution Analysis

2 Pangram

2.1 Brief

Write a program which takes in a path to an input file, reads that file and then divides it into a set of elements representing each sentence within the file. Each sentence within the file should then be checked as to whether it is a Pangram.

A pangram is a sentence that contains all the letters of the English alphabet at least once.

2.2 Planning and Design

2.2.1 Defining behaviour

We will once again be working on a utility program in which we pass a file to the utility and the utility will tell us about the file. We want to know how many

pangrams are found in the file if at all, and what they are. Ideally we will also say where they were found in the file.

2.2.2 Approach

We will also be utilising a test driven development method with this utility in order to know when we have achieved the features and usability we intend to have.

The definition of a pangram is "all letters of the English alphabet at least once". **At least once** leads me to believe that we can utilise the unique nature of Sets to our advantage. Sets can be compared with each other and only contain a value once. So if we can convert all our sentences to sets and compare them with a base alphabet set, we can determine if it is a pangram.

2.2.3 Technical Requirements

After learning more about NUnit working on the previous application we will be adopting a slightly different project structure this time round.

So that we can package up a binary with nothing but the code to run and to be able to test all files including Main.fs, we will split this solution file into two project.

Pangram.fsproj and Pangram.Tests.fsproj

The files in Pangram will be (in execution order):

1. File.fs - File IO operations
2. Pangram.fs - Pangram checking
3. Main.fs - Entry point where File meets Pangram function and feedback to user

The files in Pangram.Tests will be:

1. Tests.fs - Unit tests

2.3 Implementation

The full source code is available in the appendix. Please note that unit tests will be shown inline with their accompanying function where applicable. In reality all tests exist in Tests.fs

Please also note that double single quotes (") are used in test function names as there were problems displaying them as double back ticks (`)

2.3.1 Pangram.fs

Because we are doing similar IO and text operations as the previous project we can reuse and adapt some of those well tested and reliable functions in this application. That is the benefit of writing small reusable and testable chunks.

We can also use some of what we learnt to write better pipelines and cleaner functions.

```
module Pangram
module Alphabet
    onlyLetters: char -> bool
    allLower: char -> char
    explode: string -> Set<char>

matchToString: Match -> string
sentences: string -> seq<string>
alphabet: Set<char>
isPangram: string -> bool
findPangrams: seq<string> -> seq<string>
Find: string -> seq<string>
```

We have kept the Alphabet module and adjusted it to our needs. However this time rather than organising into another module for the rest of the functions, the Pangram module scope will suffice.

module Alphabet

```
let onlyLetters = fun l -> System.Char.IsLetter(l)

let allLower = fun l -> System.Char.ToLower(l)

let explode (s:string) =
    set[for char in s -> char]
    |> Set.filter onlyLetters
    |> Set.map allLower

[<Test>]
let ''String to lowercase letter only set''() =
    let i = "HeL27&@lo00"
    let o = set['h';'e';'l';'l';'o']
    Assert.AreEqual(o, Pangram.Alphabet.explode i)

[<Test>]
let ''Set equality''() =
    let i = Pangram.Alphabet.explode "
        ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
        "
    let o = set['a'..'z']
    Assert.AreEqual(o, i)

[<Test>]
```

```
let ''Set inequality''() =
    let i = Pangram.Alphabet.explode "abhoisud"
    let o = set['a'..'z']
    Assert.AreNotSame(o, i)

[<Test>]
let ''Set equality even with random chars''() =
    let i = Pangram.Alphabet.explode "
        ABCDEFGHIJKLMNO7638&*62786187yS&*Dg78@198@873
        **@37PQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
    let o = set['a'..'z']
    Assert.AreEqual(o, i)
```

onlyLetters and **allLower** are helpers for a Filter that we will carry out during exploding of text.

Our explode function this time round does an extra step. It uses the higher order Filter and Map to get our set into the right format for comparison with the alphabet.

module Pangram

```
let matchToString (m : Match) = m.Value

let sentences (s:string) =
    //Sentence regex
    let re = new Regex("[^.!?\s
        ][^.!?]*(?:[.!?](?!['\"]?\s|$)[^.!?]*)
        *[.!?]?['\"]?(?=\s|$)")
    re.Matches(s)
    |> Seq.cast
    |> Seq.map matchToString

[<Test>]
let ''Convert string of sentences to list''() =
    let incoming = "Edit the Expression & Text to
        see matches. Roll over matches or the
        expression for details. Undo mistakes with
        cmd-z. Save Favorites & Share expressions
        with friends or the Community. Explore your
        results with Tools. A full Reference & Help
        is available in the Library, or watch the
        video Tutorial."
    let list = ["Edit the Expression & Text to see
        matches.";"Roll over matches or the
        expression for details."; "Undo mistakes with
         cmd-z."; "Save Favorites & Share expressions
```

```
        with friends or the Community."; "Explore
       your results with Tools."; "A full Reference
       & Help is available in the Library, or watch
       the video Tutorial."]
    Assert.AreEqual(list, Pangram.sentences incoming
       )
```

In order to get a Sequence of sentences out of our file we use a very elaborate Regex. The regex allows for the following

- `[^.!?\\s]}` - First character is neither punctuation not whitespace
- `[^.!?]*}` - Match anything up to punctuation
- `[.!?] (?!['\"]?\\s|$)` - Match inner punctuation if not followed by whitespace of end of stream
- `[.!?]?` - Optionally end with punctuation
- `['\"]?` - Match quotations with inner punctuation if that applies

The **Regex.Matches** class returns a **System.Text.RegularExpressions.MatchCollection** which we can cast to a sequence and iterate over with a map. The map gets the underlying value from the regex class. The matched sentence. We now have a sequence of sentences that appear in the string.

```
let alphabet = set['a' .. 'z']
let isPangram =
    fun s -> alphabet.Equals( Alphabet.explode s )
```

Utilizing the unique nature of Sets isPangram takes a set and checks if it matches the alphabet set that we defined. If it matches, then we must have a pangram.

```
let findPangrams ls =
    ls |> Seq.filter isPangram

let Find (s:string) =
    s |> sentences |> findPangrams
    [<Test>]
let ''Pangram detection are same''() =
    let list = set["Edit the Expression & Text to
       see matches.";"Roll over matches or the
       expression for details."; "Undo mistakes with
        cmd-z."; "Save Favorites & Share expressions
        with friends or the Community."; "Explore
       your results with Tools."; "the quick brown
       fox jumps over the lazy dog."; "A full
       Reference & Help is available in the Library,
        or watch the video Tutorial."]
```

```
        let o = set["the quick brown fox jumps over the
            lazy dog."]
        Assert.AreEqual(o, Pangram.findPangrams list)

    [<Test>]
    let ''Multiple pangram detection''() =
        let list = set["Edit the Expression & Text to
            see matches."; "This Pangram contains four
            a  s , one b, two c  s , one d, thirty e  s ,
             six f  s , five g  s , seven h  s , eleven
            i  s , one j, one k, two l  s , two m  s ,
            eighteen n  s , fifteen o  s , two p  s , one
             q, five r  s , twenty-seven s  s , eighteen
            t  s , two u  s , seven v  s , eight w  s ,
            two x  s , three y  s , & one z.";"Roll over
            matches or the expression for details."; "
            Undo mistakes with cmd-z."; "Save Favorites &
             Share expressions with friends or the
            Community."; "Explore your results with Tools
            ."; "the quick brown fox jumps over the lazy
            dog."; "A full Reference & Help is available
            in the Library, or watch the video Tutorial."
            ]
        let o = set["the quick brown fox jumps over the
            lazy dog."; "This Pangram contains four a  s
            , one b, two c  s , one d, thirty e  s , six
            f  s , five g  s , seven h  s , eleven i  s ,
             one j, one k, two l  s , two m  s , eighteen
             n  s , fifteen o  s , two p  s , one q, five
             r  s , twenty-seven s  s , eighteen t  s ,
            two u  s , seven v  s , eight w  s , two
            x  s , three y  s , & one z."]
        Assert.AreEqual(o, Pangram.findPangrams list)

    [<Test>]
    let ''Pangram detection fails''() =
        let list = set["Edit the Expression & Text to
            see matches."; "This Pangram contains four
            a  s , one b, two c  s , one d, thirty e  s ,
             six f  s , five g  s , seven h  s , eleven
            i  s , one j, one k, two l  s , two m  s ,
            eighteen n  s , fifteen o  s , two p  s , one
             q, five r  s , twenty-seven s  s , eighteen
            t  s , two u  s , seven v  s , eight w  s ,
            two x  s , three y  s , & one z.";"Roll over
            matches or the expression for details."; "
```

```
        Undo mistakes with cmd-z."; "Save Favorites &
         Share expressions with friends or the
        Community."; "Explore your results with Tools
        ."; "the quick brown fox jumps over the lazy
        dog."; "A full Reference & Help is available
        in the Library, or watch the video Tutorial."
        ]
    let o = set["the quick brown fox jumps over the
        lazy dog."]
    Assert.AreNotEqual(o, Pangram.findPangrams list)

[<Test>]
let''Find in string''() =
    let i ="Lots of sentences. THat arent right. the
         quick brown fox jumps over the lazy dog.
        lalallala"
    let o = set["the quick brown fox jumps over the
        lazy dog."]
    Assert.AreEqual(o, Pangram.Find i)
```

Finally we can put this all together with our final two functions. The Find method is the only function we will need to call externally and all the plumbing has been setup. When we pass a string of the document we want to find pangrams into Find, it will break it down into a Squence of sentences, filter out anything that doesnt match against the alphabet and we will be left with a sequence of strings that will be the pangrams found in the text.

2.4 File.fs

```
module FileIO
Open: string -> string
```

module FileIO

```
let Open(path:string) =
    match File.Exists(path) with
    | true ->
        File.ReadAllText(path)
    | _ -> invalidArg "file" "The file specified
        does not exist"
```

Instead of reading all the lines to a list instead we take in all the file as a string so that we can run the regex against it.

2.5 Main.fs

We simplified here heavily since our last project. Main now only does user interface and feedback to the user as we have taken care of all the plumbing in

our logic portion.

```
[<EntryPoint>]
main: string[] -> int


let main argv =
    if (argv.Length = 0) then
          printfn("Please provide the route to a
              file on the system to process")
          1 //No file found so return an error exit
              code
    else
      let foundPangrams = FileIO.Open argv.[0] |>
          Pangram.Find
      let numberFound = foundPangrams |> Seq.length


      let printFound n =
          if (n = 0) then
            printfn "No pangrams found in %s" argv
                .[0]
            exit 0 //exit the program
          else if (n = 1) then
            printfn "Found 1 pangram"
          else
            printfn "Found %i pangrams" n

      printFound numberFound

      //List the found pangrams to stdout
      let showPangrams found =
        found
        |> List.ofSeq
        |> List.map(fun line -> printfn "%s\n" line
            )
        |> ignore


      showPangrams foundPangrams

      0
```

Once again as we call the program we will need to give feedback to the user if he has not supplied a valid argument or the location to the file does not exist.

Once we have opened the file however we can pipe it to our Pangram.Find function to get out the pangrams.

We then print out one of three options to the screen, if no pangrams were found we will let the user know. Otherwise we list how many followed by each pangram printed to their screen with a new line seperating each result.

I had to cast to a List in showPangrams because Seq.map(fun line -¿ printfn line) would not print to stdout.

Program then exits with a code 0

2.6 Solution Analysis

3 Essay

How are Functional Programming techniques currently related to web development and what connections does it have to the development of web services?

There are many requirements that web services and applications need to fulfill which often leads to sites becoming very hard to maintain in the long term. As features get added and new users begin to use the services developed the project can become enormously monolithic.

Within the last decade web developers are trying to solve these challenges of constantly developing new features but avoiding a monolithic code base by splitting their services into *micro-services*. These are smaller more manageable chunks of an overall web service or product which can exist on their own without any dependency on the rest of the codebase.

This new shift in the way web developers are working and creating services means they can use any toolset or language they desire to build out a micro-service. This has allowed web developers to try out new methods and techniques integrating more functional and declarative techniques into the services they are producing.

Functional languages and the techniques used also make it significantly easier to secure a web service and maintain high uptime. This is because each component of the code only ever needs to mutate requests and data. There are less complex data structures and objects to pass around. (Silva Jr, Lins, and Santos, 2013)

This high degree of performance and reliability is what helps modern web business such as Facebook maintain the enormous amount of data throughput they achieve in their datacenter. Most of their content is served from Haskell backed servers. (Chris Piro, 2009)

There has also been a shift in the way web developers think about and develop new services. They are increasingly using more and more declarative syntax making small but highly reusable functions that can be picks out of and moved from project to project. This is a side effect of the scale of some engineering teams needing to maintain the readability and maintainability of their codebase. This process of defining reusable components makes designing and writing new features extremely easy and quick as most of the work is already done and validated.

Furthermore there is a trend towards cloud computing which is shaking up the way services interact with each other and process data internally. This is leading to a shift towards highly distributed workloads which traditional OOP based languages and paradigms cant keep up with. We are begging to see functional languages excel in this space because of their highly parallelized nature.(Chechina et al., 2012)

To conclude there is a culmination of factors that have been adding up over the years in conjunction with many new younger programmers introducing new techniques and tools to the web development ecosystem, that is leading to more functional and declarative systems. More developers are choosing to work with functional for reliability, security and scalability reasons.

References

Chechina, Natalia et al. (2012). "The design of scalable distributed Erlang". In: *Proceedings of the Symposium on Implementation and Application of Functional Languages, Oxford, UK*, p. 85.

Chris Piro, Eugene Letuchy (2009). "Functional Programming at Facebook". In: Commercial Users of Functional Programming (CUFP).

Silva Jr, Jucimar Maia da, Rafael Dueire Lins, and Lanier Menezes dos Santos (2013). "Comparing the Performance of Java, Erlang and Scala in Web 2.0 Applications". In: *COMPARING THE PERFORMANCE OF JAVA, ERLANG AND SCALA IN WEB 2.0 APPLICATIONS*, p. 137.