Functional Programming

CMPT 331 - Spring 2023 | Dr. Labouseur

Josh Seligman | joshua.seligman1@marist.edu

April 21, 2023

1 Log

1.1 Predicton

I am predicting that it will take me around **15 hours (average of 3 hours per programming language)** for me to learn LISP, ML, Erlang, functional JavaScipt, and functional Scala and write a Caesar cipher in each langage. Although I know that some of this assignment will be more challenging than Programming in the Past because functional programming requires a completely different mindset, I am already familiar with functional JavaScipt and can use some of that knowledge to hopefully be able to quickly write the Caesar cipher in each language. Also, since I have already written the Caesar cipher in 5 languages, I know this assignment will be translating that work into a functional manner rather than trying to do it entirely from scratch.

1.2 Progress Log

Date	Hours Spent	Tasks / Accomplishments / Issues / Thoughts
March 6	3.5 hours	I wrote the Caesar cipher in LISP. It was a bit challenging because the syntax is not the cleanest and it is very difficult to find things online about it because every dialect does basic tasks different from one another.

1.3 Final Results and Analysis

2 Commentary

2.1 LISP

2.1.1 My Thoughts

LISP was a language I do not want to remember because it was nothing special relative to most other programming languages and had some features that were really frustrating to deal with. More specifically, the syntax for the language is too simplistic with intense operator overloading, confusing code organization, and poor online resources.

First, the syntax of LISP is extremely simple because parentheses are used everywhere. In LISP, lists are the main data structure and are denoted with parentheses. However, lists are used to make function calls, to define functions, and to store data, which makes it really confusing what is being stored within a list as there is nothing to differentiate these structures. Readability is seriously hurt from this as I still struggle to make the distinction between function calls and data after writing the code. Additionally, I found it difficult to write LISP code because the parentheses were challenging to manage and to keep clean. I tried my absolute best to keep parentheses lined up similar to how braces would be lined up in C family languages, but it still was not perfect and had some inconsistencies. Furthermore, messing up on the parentheses was a pain to debug because the LISP interpreter would throw a confusing error that meant absolutely nothing in helping me resolve the real problem.

Next, LISP has a very frustrating way to represent math operations. Since everything is a function in LISP, all math functions will take in 2 arguments, which can be the result of other math function calls. Similar to how a compiler works, the operations you want to do first have to be encapsulated so their output will be used by other functions. Hovever, I had to learn this idea the hard way when I did (- 90 (+ diff 1) instead of (+ (- 90 diff) 1)) to represent the formula 90 - diff + 1. Naturally, I wanted to represent the subtraction first, so I naively placed it in the outermost function call, which was a really bad idea. Although minor, this change in operation order screwed up my program for a while and my friends had to remind me of the order of operations for why the 2 code chunks weere different from each other. This syntax for specifying the order of operations for a formula does not make logical sense as the code no longer reads from left to right. Rather, one has to read it middle out (sadly not related to middle out compression in the TV show Silicon Valley) and build a mini syntax tree in their head to be able to actually understand what the code is doing.

Lastly, LISP is a really old language, and, similar to BASIC, its many dialects made good online resources hard to come by. When searching for how to do something in LISP, whatever a website had as an answer was most likely wrong becouse the dialect that I was using (newLISP) did not support functionalities specific to some of the other dialects. Unfortunately, no website specified what dialect they were referring to, and doing the same search but with "newLISP" instead of "lisp" would just bring up the documentation for newLISP. At the end of the day, this documentation and the corresponding Wikibook were my best friends for writing the Caesar cipher in LISP as they provided me with the basics to get started, which I was able to modify for my own purposes.

Overall, LISP was not a terrible programming language as I really liked the succinctness of its functional style, but its overly simplistic syntax, annoying math functions, and limited resources made it challenging to work with. Relative to the Programming in the Past assignment, I would rank LISP similar to Fortran as they are both the grandfather languages of their respective domains and were not terrible and somewhat useable, but I would never want to use them again if I do not have to.

2.1.2 Google Search History

- lisp hello world
- comments in lisp
- math functions lisp
- lambda in lisp
- and in lisp

2.1.3 Code and Tests

```
(define (encryptStr str shiftAmt)
    ; Convert the input string to be upper case and split by character
    (set 'realStr (explode (upper-case str)))
    ; Get the mod because only need to work within -25 and 25
    (set 'realShift (mod shiftAmt 26))

    ; Map the transformation to each character
    (set 'newStr (map (lambda (strChar)
        ; Begin by getting the ASCII code
        (set 'newChar (char strChar))
        ; Only work with letters now
        (cond ((and (>= newChar 65) (<= newChar 90))
            ; Perform the shift
            (set 'newChar (+ newChar realShift))

            ; Check for the Z wraparound
            (set 'diff (- newChar 90))
            (cond
                ((> diff 0)
                    ; Do wraparound so anything beyound Z picks up at A
                    (set 'newChar (- (+ 65 diff) 1))
                )
                (true
                    ; Check for A wraparound
                    (set 'diff (- 65 newChar))
                    (cond
                        ((> diff 0)
                            ; Do wraparound so anything beyound A picks up at Z
                            (set 'newChar (+ (- 90 diff) 1))
                        )
                    )
                )
            )
        ))
        ; Convert to a character and return it
        (char newChar)
    ;  This is the input to the map function
    ) realStr))
    ; Join the exploded string and put it back together
    (join newStr "")
)

(define (decryptStr str shiftAmt)
  ; Decrypt is a negative encrypt
  (encryptStr str (* -1 shiftAmt))
)

(define (solve str maxShift)
  ; Make sure the shift is between 0 and 26
  (set 'realMaxShift maxShift)
  ; If negative, take absolute value
  (cond ((< realMaxShift 0) (set 'realMaxShift (* -1 realMaxShift))))
  ; If greater than 26, then take the mod
  (cond ((> realMaxShift 26) (set 'realMaxShift (mod realMaxShift 26))))
  (map (lambda (curShift)
         ; Call encrypt with the current shift amount
         (set 'out (encryptStr str curShift))
         (println "Caesar␣" curShift ":␣" out)
       )
       ; Generate a sequence from the max down to 0 (inclusive)
       (sequence realMaxShift 0)
  )
)

(println "Alan␣tests:")
(set 'encryptOut (encryptStr "This␣is␣a␣test␣string␣from␣Alan" 8))
(println encryptOut)
(set 'decryptOut (decryptStr encryptOut 8))
```

```
(println decryptOut)
(solve "HAL" 26)

(println "")
(println "Encrypt and decrypt tests:")
; Negative shift amount
(set 'encryptOut (encryptStr "This is a test string from Alan" -1))
(println encryptOut)
(set 'decryptOut (decryptStr encryptOut -1))
(println decryptOut)

; Modulus
(set 'encryptOut (encryptStr "This is a test string from Alan" 27))
(println encryptOut)
(set 'decryptOut (decryptStr encryptOut 27))
(println decryptOut)

; Empty string
(set 'encryptOut (encryptStr "" 7))
(println encryptOut)
(set 'decryptOut (decryptStr encryptOut 7))
(println decryptOut)

; Symbols and no letters
(set 'encryptOut (encryptStr "1234567890!@#$%^&*(){}" 7))
(println encryptOut)
(set 'decryptOut (decryptStr encryptOut 7))
(println decryptOut)

; Solve tests
(println "")
(println "Solve tests:")
; Negative shift amount
(solve "HAL" -26)
(println "")
; Modulus
(solve "HAL" 30)

; Needed for newlisp
(exit)
```

Listing 1: Caesar Cipher (LISP)

```
Alan tests:
BPQA QA I BMAB ABZQVO NZWU ITIV
THIS IS A TEST STRING FROM ALAN
Caesar 26: HAL
Caesar 25: GZK
Caesar 24: FYJ
Caesar 23: EXI
Caesar 22: DWH
Caesar 21: CVG
Caesar 20: BUF
Caesar 19: ATE
Caesar 18: ZSD
Caesar 17: YRC
Caesar 16: XQB
Caesar 15: WPA
Caesar 14: VOZ
Caesar 13: UNY
Caesar 12: TMX
Caesar 11: SLW
Caesar 10: RKV
Caesar 9: QJU
Caesar 8: PIT
Caesar 7: OHS
Caesar 6: NGR
Caesar 5: MFQ
Caesar 4: LEP
Caesar 3: KDO
Caesar 2: JCN
Caesar 1: IBM
Caesar 0: HAL

Encrypt and decrypt tests:
SGHR HR Z SDRS RSQHMF EQNL ZKZM
THIS IS A TEST STRING FROM ALAN
UIJT JT B UFTU TUSJOH GSPN BMBO
THIS IS A TEST STRING FROM ALAN


1234567890!@#$%^&*(){}
1234567890!@#$%^&*(){}

Solve tests:
Caesar 26: HAL
Caesar 25: GZK
Caesar 24: FYJ
Caesar 23: EXI
Caesar 22: DWH
Caesar 21: CVG
Caesar 20: BUF
Caesar 19: ATE
Caesar 18: ZSD
Caesar 17: YRC
Caesar 16: XQB
Caesar 15: WPA
Caesar 14: VOZ
Caesar 13: UNY
Caesar 12: TMX
Caesar 11: SLW
Caesar 10: RKV
Caesar 9: QJU
Caesar 8: PIT
Caesar 7: OHS
Caesar 6: NGR
Caesar 5: MFQ
Caesar 4: LEP
Caesar 3: KDO
Caesar 2: JCN
Caesar 1: IBM
Caesar 0: HAL

Caesar 4: LEP
Caesar 3: KDO
Caesar 2: JCN
Caesar 1: IBM
Caesar 0: HAL
```

Listing 2: LISP Output

2.2 ML

2.2.1 My Thoughts

2.2.2 Google Search History

2.2.3 Code and Tests

2.3 Erlang

2.3.1 My Thoughts

2.3.2 Google Search History

2.3.3 Code and Tests

2.4 Functional JavaScipt

2.4.1 My Thoughts

2.4.2 Google Search History

2.4.3 Code and Tests

2.5 Functional Scala

2.5.1 Google Search History

2.5.2 Code and Tests