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.
March 7	3.5 hours	I wrote the Caesar cipher in ML. It wos better documented than LISP, but more challenging to write because of its steeper learning curve.
March 8	0.25 hours	I had a lightbulb moment regarding everything I did poorly in ML and fixed it all up. I am much happier now that the code is more concise and flows a lot better.
March 8	2.5 hours	I wrote the Caesar cipher in Erlang and was pleasantly surprised by how easy it was relative to LISP and ML.
March 10	0.25 hours	I wrote the Caesar cipher in JavaScript. I already had experience with functional JS, so this was a breeze for me to do.
March 10	1 hour	I wrote the Caesar cipher in Scala. Similar to the last assigment, this was surprisingly really frustrating to do, but the compiler was helpful in getting me the answers I needed.

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 do a depth-first in-order traversal of a mini syntax tree in their head to be able to actually understand what the code is doing. This amount of brain damage is completely overkill as one already has to treat functional logic completely differently from procedural logic, and this math function ordering does not help LISP in being user-friendly.

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 (https://en.wikibooks.org/wiki/Introduction_to_newLISP)
- 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 beyond 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
```

```
67 Caesar 2: JCN
68 Caesar 1: IBM
69 Caesar 0: HAL
70
71 Caesar 4: LEP
72 Caesar 3: KDO
73 Caesar 2: JCN
74 Caesar 1: IBM
75 Caesar 0: HAL
```

Listing 2: LISP Output

2.2 ML

2.2.1 My Thoughts

My Caesar cipher in ML will forever be remembered as the hackiest solution to a Caesar cipher. The language itself is just a cleaned up version of LISP with better documentation, but lack of sequential statements within functions and some weird math quirks made it one of the most rewarding challenges I have had in recent memory.

From my knowledge, ML functions do not support chaining multiple statements together. Rather, they only support returning the output from a single function call, which can have many nested function calls used as the input to the main function whose value is being returned. This made modularization difficult because it was always a big dilemma as to which chunks of logic could be put together versus separating them out into their own functions. As a fan of procedural programming, this was difficult to wrap my brain around as I struggled to match the data types together to be able to use the output from one function as the input to another. Furthermore, as an inexperienced functional programmer, I initially forced together a solution that worked but was very inefficient as I made a list of shift amounts and then ziped that list with the characters to create a list of tuples to pass into the performShift function. It was not until after a good night's rest when I realized that I could have just used an anonymous function and only map through the characters because then I could pass the shift amount in consistently without my crazy solution. Although my initial solution worked, I did make the change for code conciseness and because it was a better translation of the logic that was being done. This was also a lesson learned as the first solution that comes to mind when writing a functional program is not always the best answer because the logic has to be thought of in a more abstract way to come up with the best possible solution.

The other weird quirk was the syntax for negative numbers. When I was initially writing my ML program, I did not know how to represent a negative number in ML. Every time I wanted to use a - to represent negation, I was presented with a syntax error, most likely because - was a function for subtraction behind the scenes. Since I was already in the mindset to hack a solution together, I put a Band Aid on the problem by writing a function that negated a number by subtracting itself twice. I originally did not care about this function because it worked, but it did bother me because I knew that there was no way a programming language would make me write a negate function just to be able to represent a negative number. Similar to the previous problem with the hacky map function, I played around in ML's interactive mode for a couple minutes the next day to find out that it uses $\sim$ instead of - to represent negative values. Although this is a tiny quirk of ML, it really bothered me because the writability was harmed due to the language not able to fully represent standand mathematical notation. Fortunately, this did not cause me nearly as much brain dammage as the order of operations in LISP or trying to force multiple arguments into a map function in ML, but it was something to note because it was initially a key part in my program until I fixed it for clarity in the code.

Despite my annoyed experience writing the Caesar cipher in ML, I did have some good moments writing it. First, I love the type safety of the language as every variable and function is strictly defined. This is crucial for functional languages because it makes the code self-documenting and I know exactly what data type a function expects and what I will be receiving back as a return value. My other positive note was

specific to the Standard ML of New Jersey, but its interactive mode proved to be extremely useful when I was trying to figure out what functions were available to me. For instance, if I typed the command "open String;", I was presented with a full list of every string function in ML with definitions for the expected data types of the arguments and the return type. I did not dive deeper to find formal function definitions with verbose descriptions of what a given function did, but the basic definitions were enough for me as I was able to find what I was looking for and quickly added it to my code.

ML is in a weird place for me because it was more challenging than LISP because of the lack of sequential statements within a function, but I preferred its syntax and tooling over LISP. In other words, although my experience in LISP was better than ML, I would pick ML over LISP if I had to write a program in a functional language because its sytax is cleaner and significantly more readable and has better tooling and documentation compared to LISP.

2.2.2 Google Search History

- standard ml of nj (https://www.smlnj.org/doc/literature.html#tutorials)
- ML programming language hello world (https://www.cs.nmsu.edu/~rth/cs/cs471/sml.html) (Amazing resource)
- sml merge 2 lists into a list of tuples (https://cs.wellesley.edu/~cs251/s19/slides/sml-lists-solns_4up.pdf) (Another great resource)
- else if sml

2.2.3 Code and Tests

```
(* Creates a sequence from num down to 0, inclusive *)
fun createSequence(num: int): int list =    if num <= 0 then
                                                (* Base case is just a 0 *)
                                                [0]
                                            else
                                                (* Append the current number to the rest of
                                                    the sequence *)
                                                [num]@createSequence(num - 1);

(* Function that deals with the wraparounds *)
fun handleWraparond(charValue: int): int =  if charValue > 90 then
                                                (* Handle Z wraparound *)
                                                65 + charValue - 90 - 1
                                            else if charValue < 65 then
                                                (* Handle A wraparound *)
                                                90 - 65 + charValue + 1
                                            else charValue; (* No wraparound *)

(* Performs a shift on the given character *)
fun performShift(c: char, shiftAmt: int): char =    if Char.isAlpha(c) then
                                                        (* Convert to uppercase, get the
                                                            ordinal value, add the shift,
                                                         * manage the wraparound, and then
                                                            convert back to a character *)
                                                        chr(handleWraparond(ord(Char.toUpper
                                                            (c)) + shiftAmt))
                                                    else c; (* Non-letters stay the same *)

(* Runs the encrypt with the given shift on the string *)
fun encrypt(inStr: string, shiftAmt: int): string = implode( (* Puts the list back together
    into a string *)
                                                        map (* Run performShift on every
                                                            element of the list *)
```

```
28                                                  (fn (c) => performShift(c, shiftAmt
                                                        mod 26))
29                                                  (
30                                                      (* Create list of characters *)
31                                                      explode(inStr)
32                                                  )
33                                              );
34
35  (* Runs the decrypt (negative encrypt) on a string *)
36  fun decrypt(inStr: string, shiftAmt: int): string = encrypt(inStr, ~shiftAmt);
37
38  (* Creates the shift value for solve *)
39  fun cleanSolveShift(shift: int): int =  if Int.abs(shift) > 26 then
40                                              (* Take mod if greater than 26 *)
41                                              Int.abs(shift) mod 26
42                                          else Int.abs(shift); (* Just take abs otherwise *)
43
44  (* Solves the Caesar cipher *)
45  fun solve(inStr: string, maxShiftAmt: int): string list =   map
46                                                              (* Lambda function that runs
                                                                    encrypt with the current
                                                                    shift amount *)
47                                                              (fn (shift) => "Caesar:␣"^Int.
                                                                    toString(shift)^":␣"^encrypt
                                                                    (inStr, shift))
48                                                              (
49                                                                  (* Create a sequence from
                                                                      the max shift down to 0
                                                                      *)
50                                                                  createSequence(
                                                                      cleanSolveShift(
                                                                      maxShiftAmt))
51                                                              );
52
53  print("Alan␣tests:\n");
54  val encryptOut: string = encrypt("This␣is␣a␣test␣string␣from␣Alan", 8);
55  val decryptOut: string = decrypt(encryptOut, 8);
56  val solveOut: string list = solve("HAL", 26);
57  print(String.concatWith("\n")(solveOut)^"\n");
58
59  print("Encrypt␣and␣decrypt␣tests:\n");
60  (* Negative shift *)
61  val encryptOut: string = encrypt("This␣is␣a␣test␣string␣from␣Alan", ~1);
62  val decryptOut: string = decrypt(encryptOut, ~1);
63  (* Mod shift *)
64  val encryptOut: string = encrypt("This␣is␣a␣test␣string␣from␣Alan", 27);
65  val decryptOut: string = decrypt(encryptOut, 27);
66  (* Empty string *)
67  val encryptOut: string = encrypt("", 7);
68  val decryptOut: string = decrypt(encryptOut, 7);
69  (* All numbers and symbols (no letters) *)
70  val encryptOut: string = encrypt("1234567890!@#$%^&*(){}", 7);
71  val decryptOut: string = decrypt(encryptOut, 7);
72
73  print("Solve␣tests:\n");
74  (* Negative shift *)
75  val solveOut: string list = solve("HAL", ~26);
76  print(String.concatWith("\n")(solveOut)^"\n");
77  (* Mod shift *)
78  val solveOut: string list = solve("HAL", 30);
79  print(String.concatWith("\n")(solveOut)^"\n");
80
81  (* Needed for SMLNJ *)
```

```
82 OS.Process.exit(OS.Process.success);
```

Listing 3: Caesar Cipher (ML)

```
1  Standard ML of New Jersey (64-bit) v110.99.3 [built: Thu Jul 28 00:35:16 2022]
2  [opening caesar.sml]
3  val createSequence = fn : int -> int list
4  val handleWraparond = fn : int -> int
5  [autoloading]
6  [library $SMLNJ-BASIS/basis.cm is stable]
7  [library $SMLNJ-BASIS/(basis.cm):basis-common.cm is stable]
8  [autoloading done]
9  val performShift = fn : char * int -> char
10 val encrypt = fn : string * int -> string
11 val decrypt = fn : string * int -> string
12 val cleanSolveShift = fn : int -> int
13 val solve = fn : string * int -> string list
14 Alan tests:
15 val it = () : unit
16 val encryptOut = "BPQA QA I BMAB ABZQVO NZWU ITIV" : string
17 val decryptOut = "THIS IS A TEST STRING FROM ALAN" : string
18 val solveOut =
19   ["Caesar: 26: HAL","Caesar: 25: GZK","Caesar: 24: FYJ","Caesar: 23: EXI",
20    "Caesar: 22: DWH","Caesar: 21: CVG","Caesar: 20: BUF","Caesar: 19: ATE",
21    "Caesar: 18: ZSD","Caesar: 17: YRC","Caesar: 16: XQB","Caesar: 15: WPA",
22    "Caesar: 14: VOZ","Caesar: 13: UNY","Caesar: 12: TMX","Caesar: 11: SLW",
23    ...] : string list
24 Caesar: 26: HAL
25 Caesar: 25: GZK
26 Caesar: 24: FYJ
27 Caesar: 23: EXI
28 Caesar: 22: DWH
29 Caesar: 21: CVG
30 Caesar: 20: BUF
31 Caesar: 19: ATE
32 Caesar: 18: ZSD
33 Caesar: 17: YRC
34 Caesar: 16: XQB
35 Caesar: 15: WPA
36 Caesar: 14: VOZ
37 Caesar: 13: UNY
38 Caesar: 12: TMX
39 Caesar: 11: SLW
40 Caesar: 10: RKV
41 Caesar: 9: QJU
42 Caesar: 8: PIT
43 Caesar: 7: OHS
44 Caesar: 6: NGR
45 Caesar: 5: MFQ
46 Caesar: 4: LEP
47 Caesar: 3: KDO
48 Caesar: 2: JCN
49 Caesar: 1: IBM
50 Caesar: 0: HAL
51 val it = () : unit
52 Encrypt and decrypt tests:
53 val it = () : unit
54 val encryptOut = "SGHR HR Z SDRS RSQHMF EQNL ZKZM" : string
55 val decryptOut = "THIS IS A TEST STRING FROM ALAN" : string
56 val encryptOut = "UIJT JT B UFTU TUSJOH GSPN BMBO" : string
57 val decryptOut = "THIS IS A TEST STRING FROM ALAN" : string
58 val encryptOut = "" : string
59 val decryptOut = "" : string
60 val encryptOut = "1234567890!@#$%^&*(){}" : string
61 val decryptOut = "1234567890!@#$%^&*(){}" : string
```

```
Solve tests:
val it = () : unit
val solveOut =
  ["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",
   ...] : string list
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
val it = () : unit
val solveOut =
  ["Caesar: 4: LEP","Caesar: 3: KDO","Caesar: 2: JCN","Caesar: 1: IBM",
   "Caesar: 0: HAL"] : string list
Caesar: 4: LEP
Caesar: 3: KDO
Caesar: 2: JCN
Caesar: 1: IBM
Caesar: 0: HAL
val it = () : unit
caesar.sml:82.1-82.36 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
```

Listing 4: ML Output

2.3 Erlang

2.3.1 My Thoughts

Erlang was a language that I was not expecting to like, but I really enjoyed my time working in Erlang. The langugae does come with a some problems, such as it not being strongly typed and performs implicit type conversions and variables are all immutable, but the syntax is overall really clean and easy to follow.

My only serious issue with Erlang was the implicit type conversion from a string to an integer. Erlang does not support strings and, instead, treats a string as a list of characters, which are just numbers behind the scenes. Thus, when I wrote the performShift method, I passed in the string/list and it automatically treated the data as an integer instead of a character. This is extremely dangerous because I never told it to make the conversion and was expecting to see individual characters. Unfortunately, this problem is empha-

sized more because Erlang is not strongly typed, so I never had an opportunity to state what data type I wanted each variable to be. This hurts Erlang in all 3 aspects: readability, writability, and reliability because the code is not self-documenting, it is harder to write when the variables are not strongly typed, and the language cannot be reliable if it does not tell the programmer that it is doing its own thing by making the conversions on its own.

One minor issue I had with Erlang was that all variables are immutable. Since Erlang supports sequential statements, I assumed that I would be able to combine a few statements to help create the logic for determining the value of a returned variable. Unfortunately, since variables are immutable, I had to be a little creative for how I could assign the proper value in one statement compared to in multiple steps. This is an extremely minor complaint on my part as it did not seriously impact my experience with Erlang.

Aside from the issues with the type conversions, Erlang's syntax is extremely clean and very easy to follow. I also appreciated how everything in Erlang requires recursion. When writing the Caesar cipher in LISP and ML, I was extremely caught up in the notion of using the map functionality to call a function many times and feel that I did not use recursion enough for those solutions. Erlang forces you to use recursion for literally everything as the functions to loop through the string and the solve shift amounts were done via recursion. Recursion is extremely easy to read, and the writability is stronger for Erlang for the consistency and minimal feature multiplicity with recursion being the way. This is the way, the Erlang way.

Overall, Erlang was actually really fun to use. Its syntax was really easy to learn, which helped me quickly and efficiently write my Caesar cipher. Despite the weird type conversions, Erlang is at least consistent about its weirdness and does not do anything unexpected once you start to learn how the language really works. The best part of Erlang was undoubtedly all of the recursion and how easy it was to be able to implement this form of logic within the program.

2.3.2 Google Search History

- https://www.tutorialspoint.com/erlang/ (From Alan's Language Study page)
- strings in erlang
- map function over list erlang (https://www.erlang.org/doc/programming_examples/funs.html)
- erlang if statement
- and in erlang
- modulo in erlang
- erlang empty true statement in if statement

2.3.3 Code and Tests

```
1  -module(caesar).
2  -export([performShift/2, encrypt/2, decrypt/2, solveHelper/2, solve/2, start/0]).
3
4  % Performs a shift on a list/string that is uppercase
5  performShift([Char | Str], ShiftAmt) ->
6      [
7          if
8              % Only shift if there is a letter
9              (Char >= 65) and (Char =< 90) ->
10                 NewChar = Char + ShiftAmt,
11                 if
12                     % Compute the Z wraparound
13                     NewChar > 90 ->
```

```
14                          65 + NewChar - 90 - 1;
15                      % Compute the A wraparound
16                      NewChar < 65 ->
17                          90 - 65 + NewChar + 1;
18                      true ->
19                          % If no wraparound then return the shifted character
20                          NewChar
21                  end;
22              true ->
23                  % No change so return the original character
24                  Char
25          end
26      % Recursively call performShift on the rest of the string
27      | performShift(Str, ShiftAmt)];
28 % Base case for the recursion. Empty string gets returned as an empty list.
29 performShift([], ShiftAmt) -> [].
30
31 % Encrypts a string with the given shift amount
32 encrypt(InStr, ShiftAmt) ->
33     % Get the uppercase string
34     NewStr = string:to_upper(InStr),
35     % Clean up the shift amount
36     RealShift = ShiftAmt rem 26,
37     % Perform the shift on the string
38     performShift(NewStr, RealShift).
39
40 % Decrypts a string
41 decrypt(InStr, ShiftAmt) ->
42     % Decrypt is negative encrypt
43     encrypt(InStr, -ShiftAmt).
44
45 % Recursively solves a Caesar cipher
46 solveHelper(InStr, ShiftAmt) ->
47     % Print out the output from the encrypt function
48     io:fwrite("Caesar ~w: ~p~n", [ShiftAmt, encrypt(InStr, ShiftAmt)]),
49     if
50         % Contine until we have no more shifts
51         ShiftAmt > 0 ->
52             solveHelper(InStr, ShiftAmt - 1);
53         true ->
54             % We are done so return void
55             void
56     end.
57
58 % Solves a Caesar cipher
59 solve(InStr, MaxShift) ->
60     if
61         abs(MaxShift) > 26 ->
62             % Can take mod if greater than 26
63             RealShift = abs(MaxShift) rem 26;
64         true ->
65             % Just take abs here
66             RealShift = abs(MaxShift)
67     end,
68     % Call the recursive solve helper function starting with the max shift
69     solveHelper(InStr, RealShift).
70
71 % Entrypoint for the program
72 start() ->
73     % Basic Alan tests
74     io:fwrite("Alan tests:~n"),
75     EncryptOut1 = encrypt("This is a test string from Alan", 8),
76     io:fwrite("~p~n", [EncryptOut1]),
77     DecryptOut1 = decrypt(EncryptOut1, 8),
78     io:fwrite("~p~n", [DecryptOut1]),
```

```
    solve("HAL", 26),
    io:fwrite("~nEncrypt and decrypt tests:~n"),
    % Negative shift
    EncryptOut2 = encrypt("This is a test string from Alan", -1),
    io:fwrite("~p~n", [EncryptOut2]),
    DecryptOut2 = decrypt(EncryptOut2, -1),
    io:fwrite("~p~n", [DecryptOut2]),
    % Mod shift
    EncryptOut3 = encrypt("This is a test string from Alan", 27),
    io:fwrite("~p~n", [EncryptOut3]),
    DecryptOut3 = decrypt(EncryptOut3, 27),
    io:fwrite("~p~n", [DecryptOut3]),
    % Empty string
    EncryptOut4 = encrypt("", 7),
    io:fwrite("~p~n", [EncryptOut4]),
    DecryptOut4 = decrypt(EncryptOut4, 7),
    io:fwrite("~p~n", [DecryptOut4]),
    % No letters
    EncryptOut5 = encrypt("1234567890!@#$%^&*(){}", 7),
    io:fwrite("~p~n", [EncryptOut5]),
    DecryptOut5 = decrypt(EncryptOut5, 7),
    io:fwrite("~p~n", [DecryptOut5]),
    io:fwrite("~nSolve tests:~n"),
    % Negative shift
    solve("HAL", -26),
    io:fwrite("~n"),
    % Mod shift
    solve("HAL", 30).
```

Listing 5: Caesar Cipher (Erlang)

```
Erlang/OTP 25 [erts-13.1.5] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [jit:
    ns] [dtrace]

Eshell V13.1.5  (abort with ^G)
1> c(caesar).
caesar.erl:29:18: Warning: variable 'ShiftAmt' is unused
%   29| performShift([], ShiftAmt) -> [].
%     |                  ^

{ok,caesar}
2> caesar:start().
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"
void
```

Listing 6: Erlang Output

2.4 Functional JavaScipt

2.4.1 My Thoughts

JavaScript is a wonderful language and has a clean syntax for writing functional code that, unlike some other functional languages, scales well with complexity.

One of the main problems with some functional languages, most notably ML, is that they only support

functions and sequential statements may be challenging to implement. However, since JavaScript is a procedural language that happens to support the functional philosophy, its functions can be as long as the programmer wants them to be. Although it is best to keep functions short, I felt that ML was too restrictive on this, and JavaScript allowed me to express the entire logic for a function rather than forcing me to split it up into multiple smaller functions. This was really important for me as I felt that I was working in a normal procedural manner even though I was really writing functional code.

The one downside with JavaScript, however, is that it is not strongly typed and the programmer cannot explictly define the data type of a variable. This can be challenging to both read and write as the data types are not a part of the syntax, so the code is not entirely self-documenting. Unlike Erlang, JavaScript does not perform implicit type conversions and there were never any surprises when writing my code, which made my life really easy with no brain damage.

JavaScript is one of the world's most popular programming languages for a reason. It has a clean and consistent syntax that allows programmers to get a best of both worlds with procedural and functional programming philosophies without being extremely restrictive. JavaScript's lack of a strongly typed syntax is a drawback for the language, but is quickly negated by the ease of use of JavaScript and the flexibility the language provides to its programmers.

2.4.2 Google Search History

- javascript string functions

2.4.3 Code and Tests

```
1  // Encrypts a string with the given shift
2  function encrypt(inStr, shiftAmt) {
3      // Take the mod because only woking between -25 and 25
4      shiftAmt = shiftAmt % 26;
5      // Convert the string to uppercase and split by character
6      // And create a new array using the map function
7      return inStr.toUpperCase().split('').map(c => {
8          // Get the char code of the character
9          let charNum = c.charCodeAt(0);
10         // Only shift on letters
11         if (charNum >= 65 && charNum <= 90) {
12             // Do the shift
13             let newChar = charNum + shiftAmt;
14             if (newChar > 90) {
15                 // Handle Z wraparound
16                 newChar = 65 + newChar - 90 - 1;
17             } else if (newChar < 65) {
18                 // Handle A wraparound
19                 newChar = 90 - 65 + newChar + 1;
20             }
21             // Set the charNum to be newChar if a shift was done
22             charNum = newChar;
23         }
24         // Return the character based on the code
25         return String.fromCharCode(charNum);
26     // Join the array into one big string with no separation of characters
27     }).join('');
28 }
29
30 // Function that decrypts a string with the given shift amount
31 function decrypt(inStr, shiftAmt) {
32     // Decrypt is negative encrypt
33     return encrypt(inStr, -shiftAmt);
34 }
```

```
35
36 // Function that recursively solves a caesar cipher
37 function solveHelper(inStr, shift) {
38     // Print out the encrypt result and the shift amount
39     console.log(`Caesar ${shift}: ${encrypt(inStr, shift)}`);
40     // Only continue if the shift > 0
41     if (shift > 0) {
42         solveHelper(inStr, shift - 1);
43     }
44 }
45
46 // Function that solves a caesar cipher
47 function solve(inStr, maxShift) {
48     // Take the absolute value for the max shift
49     maxShift = Math.abs(maxShift);
50     if (maxShift > 26) {
51         // Take mod if greater than 26 so we work with 0 - 26 inclusive
52         maxShift = maxShift % 26;
53     }
54     // Start the helper function at the shift amount
55     solveHelper(inStr, maxShift);
56 }
57
58 // Basic tests
59 console.log("Alan tests:");
60 let encryptOut = encrypt("This is a test string from Alan", 8);
61 console.log(encryptOut);
62 let decryptOut = decrypt(encryptOut, 8);
63 console.log(decryptOut);
64 solve("HAL", 26);
65
66 console.log("\nEncrypt and decrypt tests:");
67 // Negative shift
68 encryptOut = encrypt("This is a test string from Alan", -1);
69 console.log(encryptOut);
70 decryptOut = decrypt(encryptOut, -1);
71 console.log(decryptOut);
72 // Mod shift
73 encryptOut = encrypt("This is a test string from Alan", 27);
74 console.log(encryptOut);
75 decryptOut = decrypt(encryptOut, 27);
76 console.log(decryptOut);
77 // Empty string
78 encryptOut = encrypt("", 7);
79 console.log(encryptOut);
80 decryptOut = decrypt(encryptOut, 7);
81 console.log(decryptOut);
82 // No letters
83 encryptOut = encrypt("1234567890!@#$%^&*(){}", 7);
84 console.log(encryptOut);
85 decryptOut = decrypt(encryptOut, 7);
86 console.log(decryptOut);
87
88 console.log("\nSolve tests:");
89 // Negative max shift
90 solve("HAL", -26);
91 console.log();
92 // Mod shift
93 solve("HAL", 30);
```

Listing 7: Caesar Cipher (JavaScript)

```
1 Alan tests:
2 BPQA QA I BMAB ABZQVO NZWU ITIV
3 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
```

```
69 Caesar 0: HAL
70
71 Caesar 4: LEP
72 Caesar 3: KDO
73 Caesar 2: JCN
74 Caesar 1: IBM
75 Caesar 0: HAL
```

Listing 8: JavaScript Output

2.5 Functional Scala

2.5.1 My Thoughts

Since writing a Caesar cipher in Scala in a procedural manner, doing the same task in a functional manner did not improve my opinion of the language. Its identity is still completely mixed between a procedural and functional language and does not do either particularly well. The syntax of the language has always felt clunky and I always struggled to write code that would compile. Luckily, the Scala compiler is good and helped me overcome the many obstacles I faced.

The Scala code for the Caesar cipher was not much different than the code for the JavaScript solution. However, I found it much more challenging to write compared to the JavaScript solution. The largest roadblock I encontered was getting the map function to work on the character array. Since Scala is strongly typed, the compiler is extremely careful with the data types being inputted and returned to any given function. However, the Scala compiler was not completely clear about how the data types from the map function were being handled as I was consistently confused whether the map function was sending a Char type or an Int type. It did tell me what it was receiving and what was defined, but it took me a while to figure out how to make sense of the messages and fix it in my code. Unfortunately, I spent a ton of time trying to get the functions to line up in data types through trial and error and was relieved once the code decided to compile. Strongly typed code is really good for readability and writability, but it should not be a puzzle to get the data types to line up between functions.

Another issue I had with Scala with the map function was that I was unable to get it to work like the JavaScript code in that it only supported a single statement in the body. This forced me to modularize the performShift logic into its own function, but it just did not feel right in a hybrid language. It is either completely functional or completely procedural, which does not really allow for programs to be flexible and for developers to really express themselves because of the restrictions the language enforces through its syntax.

Overall, Scala is still not a preferred language of mine due to its surprisingly frustrating and restrictive syntax that prevented me from easily expressing the logic I wanted to represent in the code. Although not as useful as the last assignment, the compiler continued to be a redeeming factor for the language, and slightly improved my experience with the language and greatly decreased development time. After 2 assignments in Scala, the compiler has become a crutch for me, which emphasizes how much of a challenge it has been for me to write the Scala programs and how I have struggled to learn and understand the language.

2.5.2 Google Search History

- scala string functions
- array methods scala
- map function in scala
- scala convert char to int
- scala void method

2.5.3 Code and Tests

```
object Caesar {

    // Function to perform a shift on a caesar cipher
    def performShift(c: Char, shift: Int): Char = {
        // Have to work with integers, so do the conversion
        var newChar: Int = c.toInt;
        if (newChar >= 65 && newChar <= 90) {
            // Only perform shift on letters
            newChar = newChar + shift;

            if (newChar > 90) {
                // Handle Z wraparound
                newChar = 65 + newChar - 90 - 1;
            } else if (newChar < 65) {
                // Handle A wraparound
                newChar = 90 - 65 + newChar + 1;
            }
        }
        // Convert the int back to a char to be returned
        return newChar.toChar;
    }

    // Function to encrypt a string by the given shift amount
    def encrypt(inputStr: String, shiftAmount: Int): String = {
        // Compute the actual shift amount being used
        val realShift: Int = shiftAmount % 26;

        // Get the array of characters
        var stringChars: Array[Char] = inputStr.toUpperCase().toCharArray();

        // Perform the shift on each character in the array
        var newChars: Array[Char] = stringChars.map((c: Char) => performShift(c, realShift))
            ;

        // Create a new string with the shifted characters
        return String.copyValueOf(newChars);
    }

    // Function to decrypt a string by the given shift amount
    def decrypt(inputStr: String, shiftAmount: Int): String = {
        // Decrypt is negative encrypt
        return encrypt(inputStr, -shiftAmount);
    }

    /// Helper function for solve that returns nothing in practice
    def solveHelper(inputStr: String, shiftAmount: Int): Unit = {
        // Print the result of the encrypt function
        println(s"Caesar ${shiftAmount}: ${encrypt(inputStr, shiftAmount)}");
        if (shiftAmount > 0) {
            // Only continue if there is still more to do
            solveHelper(inputStr, shiftAmount - 1);
        }
    }

    // Function to solve a Caesar cipher
    def solve(inputStr: String, maxShiftAmount: Int) = {
        // Clean up the max shift amount
        var realMaxShift: Int = Math.abs(maxShiftAmount);
        if (realMaxShift > 26) {
            realMaxShift = realMaxShift % 26;
        }

        // Call the helper function starting with the max shift
```

```
63          solveHelper(inputStr, realMaxShift);
64      }
65
66      def main(args: Array[String]) = {
67          println("Alan tests:");
68          var encryptOut: String = encrypt("This is a test string from Alan", 8);
69          println(encryptOut);
70          var decryptOut: String = decrypt(encryptOut, 8);
71          println(decryptOut);
72          solve("HAL", 26);
73
74          println("\nEncrypt and decrypt tests:");
75          // Test negative shift amount
76          encryptOut = encrypt("This is a test string from Alan", -1);
77          println(encryptOut);
78          decryptOut = decrypt(encryptOut, -1);
79          println(decryptOut);
80
81          // Test modulus
82          encryptOut = encrypt("This is a test string from Alan", 27);
83          println(encryptOut);
84          decryptOut = decrypt(encryptOut, 27);
85          println(decryptOut);
86
87          // Test empty string
88          encryptOut = encrypt("", 27);
89          println(encryptOut);
90          decryptOut = decrypt(encryptOut, 27);
91          println(decryptOut);
92
93          // Test no letters
94          encryptOut = encrypt("1234567890!@#$%^&*(){}", 27);
95          println(encryptOut);
96          decryptOut = decrypt(encryptOut, 27);
97          println(decryptOut);
98
99          println("\nSolve tests:");
100         // Test absolute value
101         solve("HAL", -26);
102         println();
103         // Test modulus
104         solve("HAL", 30);
105     }
106 }
```

Listing 9: Caesar Cipher (Scala)

```
1  Alan tests:
2  BPQA QA I BMAB ABZQVO NZWU ITIV
3  THIS IS A TEST STRING FROM ALAN
4  Caesar 26: HAL
5  Caesar 25: GZK
6  Caesar 24: FYJ
7  Caesar 23: EXI
8  Caesar 22: DWH
9  Caesar 21: CVG
10 Caesar 20: BUF
11 Caesar 19: ATE
12 Caesar 18: ZSD
13 Caesar 17: YRC
14 Caesar 16: XQB
15 Caesar 15: WPA
16 Caesar 14: VOZ
17 Caesar 13: UNY
18 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 10: Scala Output

3 Conclusion

Here are my final rankings for the 5 functional programming languages:

5. JavaScript

4. Erlang
3. Scala
2. ML
1. LISP

Starting from the bottom, LISP was just alright in terms of features, but its syntax was confusing to follow because of all the paretheses. ML, despite being more challenging than LISP, was much more readable and a bit more writable once I got into the flow of functional programming. Next, Scala was just so frustrating to use that its compiler was unable to bail it out again to be preferred over Erlang. Speaking of Erlang, it was overall really clean and easy to write despite its weird implicit type conversions and unique syntax relative to other languages. Lastly, JavaScript was amazing in every facet and demonstrated its dominance in the industry with its simple and powerful syntax and flexibility to maximize developer productivity.