# 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

The assignment took me a total of 11 hours to complete. This was only 4 hours shorter than my prediction and a lot more accurate than my estimation in the previous assignment. Despite knowing how fast I would be able to complete the JavaScript and Scala programs, I was unsure of how I would fare with the older languages considering it took me a long time to complete the assignment in Fortran and COBOL in a procedural manner. Similar to procedural programming, functional programming has its own conceptual foundation that once mastered in one language, can easily be tranlated to other functional languages with the only difference being the syntax of the new language. This made it really easy to write the programs in the various languages as I just had to overcome the obstacle of writing a program that would compile and adheres to each language's grammar. Although easier said than done, the assignment was overall more bearable knowing that I had to figure out the functional logic once instead of 5 separate times because doing it once was hard enough with the strict restrictions of some of the languages.

# 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

```
1  (define (encryptStr str shiftAmt)
2      ; Convert the input string to be upper case and split by character
3      (set 'realStr (explode (upper-case str)))
4      ; Get the mod because only need to work within -25 and 25
5      (set 'realShift (mod shiftAmt 26))
6
7      ; Map the transformation to each character
8      (set 'newStr (map (lambda (strChar)
9          ; Begin by getting the ASCII code
10         (set 'newChar (char strChar))
11         ; Only work with letters now
12         (cond ((and (>= newChar 65) (<= newChar 90))
13             ; Perform the shift
14             (set 'newChar (+ newChar realShift))
15
16             ; Check for the Z wraparound
17             (set 'diff (- newChar 90))
18             (cond
19                 ((> diff 0)
20                     ; Do wraparound so anything beyond Z picks up at A
21                     (set 'newChar (- (+ 65 diff) 1))
22                 )
23                 (true
24                     ; Check for A wraparound
25                     (set 'diff (- 65 newChar))
26                     (cond
27                         ((> diff 0)
28                             ; Do wraparound so anything beyound A picks up at Z
29                             (set 'newChar (+ (- 90 diff) 1))
30                         )
31                     )
32                 )
```

```
33                  )
34              ))
35              ; Convert to a character and return it
36              (char newChar)
37          ;   This is the input to the map function
38          ) realStr))
39          ; Join the exploded string and put it back together
40          (join newStr "")
41  )
42
43  (define (decryptStr str shiftAmt)
44      ; Decrypt is a negative encrypt
45      (encryptStr str (* -1 shiftAmt))
46  )
47
48  (define (solve str maxShift)
49      ; Make sure the shift is between 0 and 26
50      (set 'realMaxShift maxShift)
51      ; If negative, take absolute value
52      (cond ((< realMaxShift 0) (set 'realMaxShift (* -1 realMaxShift))))
53      ; If greater than 26, then take the mod
54      (cond ((> realMaxShift 26) (set 'realMaxShift (mod realMaxShift 26))))
55      (map (lambda (curShift)
56              ; Call encrypt with the current shift amount
57              (set 'out (encryptStr str curShift))
58              (println "Caesar_" curShift ":_" out)
59          )
60          ; Generate a sequence from the max down to 0 (inclusive)
61          (sequence realMaxShift 0)
62      )
63  )
64
65  (println "Alan_tests:")
66  (set 'encryptOut (encryptStr "This_is_a_test_string_from_Alan" 8))
67  (println encryptOut)
68  (set 'decryptOut (decryptStr encryptOut 8))
69  (println decryptOut)
70  (solve "HAL" 26)
71
72  (println "")
73  (println "Encrypt_and_decrypt_tests:")
74  ; Negative shift amount
75  (set 'encryptOut (encryptStr "This_is_a_test_string_from_Alan" -1))
76  (println encryptOut)
77  (set 'decryptOut (decryptStr encryptOut -1))
78  (println decryptOut)
79
80  ; Modulus
81  (set 'encryptOut (encryptStr "This_is_a_test_string_from_Alan" 27))
82  (println encryptOut)
83  (set 'decryptOut (decryptStr encryptOut 27))
84  (println decryptOut)
85
86  ; Empty string
87  (set 'encryptOut (encryptStr "" 7))
88  (println encryptOut)
89  (set 'decryptOut (decryptStr encryptOut 7))
90  (println decryptOut)
91
92  ; Symbols and no letters
93  (set 'encryptOut (encryptStr "1234567890!@#$%^&*(){}" 7))
94  (println encryptOut)
95  (set 'decryptOut (decryptStr encryptOut 7))
96  (println decryptOut)
97
```

```
98  ; Solve tests
99  ( println "")
100 ( println "Solve_tests:")
101 ; Negative shift amount
102 ( solve "HAL" −26)
103 ( println "")
104 ; Modulus
105 ( solve "HAL" 30)
106
107 ; Needed for newlisp
108 ( exit )
```

Listing 1: Caesar Cipher (LISP)

```
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
19 Caesar 11: SLW
20 Caesar 10: RKV
21 Caesar 9: QJU
22 Caesar 8: PIT
23 Caesar 7: OHS
24 Caesar 6: NGR
25 Caesar 5: MFQ
26 Caesar 4: LEP
27 Caesar 3: KDO
28 Caesar 2: JCN
29 Caesar 1: IBM
30 Caesar 0: HAL
31
32 Encrypt and decrypt tests:
33 SGHR HR Z SDRS RSQHMF EQNL ZKZM
34 THIS IS A TEST STRING FROM ALAN
35 UIJT JT B UFTU TUSJOH GSPN BMBO
36 THIS IS A TEST STRING FROM ALAN
37
38
39 1234567890!@#$%^&*(){}
40 1234567890!@#$%^&*(){}
41
42 Solve tests:
43 Caesar 26: HAL
44 Caesar 25: GZK
45 Caesar 24: FYJ
46 Caesar 23: EXI
47 Caesar 22: DWH
48 Caesar 21: CVG
49 Caesar 20: BUF
50 Caesar 19: ATE
51 Caesar 18: ZSD
```

```
52  Caesar  17:  YRC
53  Caesar  16:  XQB
54  Caesar  15:  WPA
55  Caesar  14:  VOZ
56  Caesar  13:  UNY
57  Caesar  12:  TMX
58  Caesar  11:  SLW
59  Caesar  10:  RKV
60  Caesar  9:  QJU
61  Caesar  8:  PIT
62  Caesar  7:  OHS
63  Caesar  6:  NGR
64  Caesar  5:  MFQ
65  Caesar  4:  LEP
66  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 ∼ 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

```
1  (* Creates a sequence from num down to 0, inclusive  *)
2  fun createSequence(num: int): int list =    if num <= 0 then
3                                                  (* Base case is just a 0 *)
4                                                  [0]
5                                              else
6                                                  (* Append the current number to the rest of
                                                        the sequence *)
7                                                  [num] @createSequence(num − 1);
8
9  (* Function that deals with the wraparounds *)
10 fun handleWraparond(charValue: int): int =  if charValue > 90 then
11                                                  (* Handle Z wraparound  *)
12                                                  65 + charValue − 90 − 1
13                                              else if charValue < 65 then
14                                                  (* Handle A wraparound  *)
15                                                  90 − 65 + charValue + 1
16                                              else charValue; (* No wraparound *)
17
```

```sml
18 (* Performs a shift on the given character *)
19 fun performShift(c: char, shiftAmt: int): char =    if Char.isAlpha(c) then
20                                                         (* Convert to uppercase, get the
                                                            ordinal value, add the shift,
21                                                          * manage the wraparound, and then
                                                            convert back to a character *)
22                                                        chr(handleWraparond(ord(Char.toUpper
                                                           (c)) + shiftAmt))
23                                                      else c; (* Non-letters stay the same *)
24
25 (* Runs the encrypt with the given shift on the string  *)
26 fun encrypt(inStr: string, shiftAmt: int): string = implode( (* Puts the list back together
     into a string *)
27                                                      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
62 Solve tests:
63 val it = () : unit
64 val solveOut =
65   ["Caesar: 26: HAL","Caesar: 25: GZK","Caesar: 24: FYJ","Caesar: 23: EXI",
66    "Caesar: 22: DWH","Caesar: 21: CVG","Caesar: 20: BUF","Caesar: 19: ATE",
67    "Caesar: 18: ZSD","Caesar: 17: YRC","Caesar: 16: XQB","Caesar: 15: WPA",
68    "Caesar: 14: VOZ","Caesar: 13: UNY","Caesar: 12: TMX","Caesar: 11: SLW",
69    ...] : string list
70 Caesar: 26: HAL
71 Caesar: 25: GZK
72 Caesar: 24: FYJ
73 Caesar: 23: EXI
74 Caesar: 22: DWH
75 Caesar: 21: CVG
76 Caesar: 20: BUF
77 Caesar: 19: ATE
78 Caesar: 18: ZSD
79 Caesar: 17: YRC
80 Caesar: 16: XQB
81 Caesar: 15: WPA
82 Caesar: 14: VOZ
83 Caesar: 13: UNY
84 Caesar: 12: TMX
85 Caesar: 11: SLW
86 Caesar: 10: RKV
87 Caesar: 9: QJU
88 Caesar: 8: PIT
89 Caesar: 7: OHS
90 Caesar: 6: NGR
91 Caesar: 5: MFQ
92 Caesar: 4: LEP
93 Caesar: 3: KDO
94 Caesar: 2: JCN
95 Caesar: 1: IBM
96 Caesar: 0: HAL
97 val it = () : unit
98 val solveOut =
99   ["Caesar: 4: LEP","Caesar: 3: KDO","Caesar: 2: JCN","Caesar: 1: IBM",
100   "Caesar: 0: HAL"] : string list
101 Caesar: 4: LEP
102 Caesar: 3: KDO
103 Caesar: 2: JCN
104 Caesar: 1: IBM
105 Caesar: 0: HAL
106 val it = () : unit
107 caesar.sml:82.1-82.36 Warning: type vars not generalized because of
108    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 emphasized 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

```erlang
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;
```

```erlang
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]),
79     solve("HAL", 26),
80     io:fwrite("~nEncrypt and decrypt tests:~n"),
81     % Negative shift
82     EncryptOut2 = encrypt("This is a test string from Alan", -1),
83     io:fwrite("~p~n", [EncryptOut2]),
84     DecryptOut2 = decrypt(EncryptOut2, -1),
85     io:fwrite("~p~n", [DecryptOut2]),
86     % Mod shift
87     EncryptOut3 = encrypt("This is a test string from Alan", 27),
88     io:fwrite("~p~n", [EncryptOut3]),
89     DecryptOut3 = decrypt(EncryptOut3, 27),
90     io:fwrite("~p~n", [DecryptOut3]),
91     % Empty string
92     EncryptOut4 = encrypt("", 7),
93     io:fwrite("~p~n", [EncryptOut4]),
94     DecryptOut4 = decrypt(EncryptOut4, 7),
95     io:fwrite("~p~n", [DecryptOut4]),
96     % No letters
97     EncryptOut5 = encrypt("1234567890!@#$%^&*(){}", 7),
98     io:fwrite("~p~n", [EncryptOut5]),
99     DecryptOut5 = decrypt(EncryptOut5, 7),
100    io:fwrite("~p~n", [DecryptOut5]),
101    io:fwrite("~nSolve tests:~n"),
102    % Negative shift
103    solve("HAL", -26),
104    io:fwrite("~n"),
105    % Mod shift
106    solve("HAL", 30).
```

Listing 5: Caesar Cipher (Erlang)

```
1 Erlang/OTP 25 [erts -13.1.5] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [jit:
      ns] [dtrace]
2
3 Eshell V13.1.5  (abort with ^G)
4 1> c(caesar).
5 caesar.erl:29:18: Warning: variable 'ShiftAmt' is unused
6 %    29| performShift([], ShiftAmt) -> [].
7 %      |                  ^
8
9 {ok,caesar}
10 2> caesar:start().
11 Alan tests:
12 "BPQA QA I BMAB ABZQVO NZWU ITIV"
13 "THIS IS A TEST STRING FROM ALAN"
14 Caesar 26: "HAL"
15 Caesar 25: "GZK"
16 Caesar 24: "FYJ"
17 Caesar 23: "EXI"
18 Caesar 22: "DWH"
```

```
19  Caesar  21:  "CVG"
20  Caesar  20:  "BUF"
21  Caesar  19:  "ATE"
22  Caesar  18:  "ZSD"
23  Caesar  17:  "YRC"
24  Caesar  16:  "XQB"
25  Caesar  15:  "WPA"
26  Caesar  14:  "VOZ"
27  Caesar  13:  "UNY"
28  Caesar  12:  "TMX"
29  Caesar  11:  "SLW"
30  Caesar  10:  "RKV"
31  Caesar  9:  "QJU"
32  Caesar  8:  "PIT"
33  Caesar  7:  "OHS"
34  Caesar  6:  "NGR"
35  Caesar  5:  "MFQ"
36  Caesar  4:  "LEP"
37  Caesar  3:  "KDO"
38  Caesar  2:  "JCN"
39  Caesar  1:  "IBM"
40  Caesar  0:  "HAL"
41
42  Encrypt  and  decrypt  tests:
43  "SGHR  HR  Z  SDRS  RSQHMF  EQNL  ZKZM"
44  "THIS  IS  A  TEST  STRING  FROM  ALAN"
45  "UIJT  JT  B  UFTU  TUSJOH  GSPN  BMBO"
46  "THIS  IS  A  TEST  STRING  FROM  ALAN"
47  []
48  []
49  "1234567890!@#$%^&*(){}"
50  "1234567890!@#$%^&*(){}"
51
52  Solve  tests:
53  Caesar  26:  "HAL"
54  Caesar  25:  "GZK"
55  Caesar  24:  "FYJ"
56  Caesar  23:  "EXI"
57  Caesar  22:  "DWH"
58  Caesar  21:  "CVG"
59  Caesar  20:  "BUF"
60  Caesar  19:  "ATE"
61  Caesar  18:  "ZSD"
62  Caesar  17:  "YRC"
63  Caesar  16:  "XQB"
64  Caesar  15:  "WPA"
65  Caesar  14:  "VOZ"
66  Caesar  13:  "UNY"
67  Caesar  12:  "TMX"
68  Caesar  11:  "SLW"
69  Caesar  10:  "RKV"
70  Caesar  9:  "QJU"
71  Caesar  8:  "PIT"
72  Caesar  7:  "OHS"
73  Caesar  6:  "NGR"
74  Caesar  5:  "MFQ"
75  Caesar  4:  "LEP"
76  Caesar  3:  "KDO"
77  Caesar  2:  "JCN"
78  Caesar  1:  "IBM"
79  Caesar  0:  "HAL"
80
81  Caesar  4:  "LEP"
82  Caesar  3:  "KDO"
83  Caesar  2:  "JCN"
```

```
84  Caesar  1:  "IBM"
85  Caesar  0:  "HAL"
86  void
```

<div align="center">Listing 6: Erlang Output</div>

## 2.4 Functional JavaScript

### 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
 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
19  Caesar 11: SLW
20  Caesar 10: RKV
21  Caesar 9: QJU
22  Caesar 8: PIT
23  Caesar 7: OHS
24  Caesar 6: NGR
25  Caesar 5: MFQ
26  Caesar 4: LEP
27  Caesar 3: KDO
28  Caesar 2: JCN
29  Caesar 1: IBM
30  Caesar 0: HAL
31
32  Encrypt and decrypt tests:
33  SGHR HR Z SDRS RSQHMF EQNL ZKZM
34  THIS IS A TEST STRING FROM ALAN
35  UIJT JT B UFTU TUSJOH GSPN BMBO
36  THIS IS A TEST STRING FROM ALAN
37
38
39  1234567890!@#$%^&*(){}
40  1234567890!@#$%^&*(){}
41
42  Solve tests:
43  Caesar 26: HAL
44  Caesar 25: GZK
45  Caesar 24: FYJ
46  Caesar 23: EXI
47  Caesar 22: DWH
48  Caesar 21: CVG
49  Caesar 20: BUF
50  Caesar 19: ATE
51  Caesar 18: ZSD
52  Caesar 17: YRC
53  Caesar 16: XQB
```

```
54  Caesar  15: WPA
55  Caesar  14: VOZ
56  Caesar  13: UNY
57  Caesar  12: TMX
58  Caesar  11: SLW
59  Caesar  10: RKV
60  Caesar  9: QJU
61  Caesar  8: PIT
62  Caesar  7: OHS
63  Caesar  6: NGR
64  Caesar  5: MFQ
65  Caesar  4: LEP
66  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
```

<div align="center">Listing 8: JavaScript Output</div>

## 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

```scala
 1  object Caesar {
 2
 3      // Function to perform a shift on a caesar cipher
 4      def performShift(c: Char, shift: Int): Char = {
 5          // Have to work with integers, so do the conversion
 6          var newChar: Int = c.toInt;
 7          if (newChar >= 65 && newChar <= 90) {
 8              // Only perform shift on letters
 9              newChar = newChar + shift;
10
11              if (newChar > 90) {
12                  // Handle Z wraparound
13                  newChar = 65 + newChar - 90 - 1;
14              } else if (newChar < 65) {
15                  // Handle A wraparound
16                  newChar = 90 - 65 + newChar + 1;
17              }
18          }
19          // Convert the int back to a char to be returned
20          return newChar.toChar;
21      }
22
23      // Function to encrypt a string by the given shift amount
24      def encrypt(inputStr: String, shiftAmount: Int): String = {
25          // Compute the actual shift amount being used
26          val realShift: Int = shiftAmount % 26;
27
28          // Get the array of characters
29          var stringChars: Array[Char] = inputStr.toUpperCase().toCharArray();
30
31          // Perform the shift on each character in the array
32          var newChars: Array[Char] = stringChars.map((c: Char) => performShift(c, realShift))
                ;
33
34          // Create a new string with the shifted characters
35          return String.copyValueOf(newChars);
36      }
37
38      // Function to decrypt a string by the given shift amount
39      def decrypt(inputStr: String, shiftAmount: Int): String = {
40          // Decrypt is negative encrypt
41          return encrypt(inputStr, -shiftAmount);
42      }
43
44      /// Helper function for solve that returns nothing in practice
45      def solveHelper(inputStr: String, shiftAmount: Int): Unit = {
46          // Print the result of the encrypt function
```

```scala
47            println(s"Caesar_${shiftAmount}:_${encrypt(inputStr,_shiftAmount)}");
48            if (shiftAmount > 0) {
49                // Only continue if there is still more to do
50                solveHelper(inputStr, shiftAmount - 1);
51            }
52        }
53
54        // Function to solve a Caesar cipher
55        def solve(inputStr: String, maxShiftAmount: Int) = {
56            // Clean up the max shift amount
57            var realMaxShift: Int = Math.abs(maxShiftAmount);
58            if (realMaxShift > 26) {
59                realMaxShift = realMaxShift % 26;
60            }
61
62            // 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
19  Caesar 11: SLW
20  Caesar 10: RKV
21  Caesar 9: QJU
22  Caesar 8: PIT
23  Caesar 7: OHS
24  Caesar 6: NGR
25  Caesar 5: MFQ
26  Caesar 4: LEP
27  Caesar 3: KDO
28  Caesar 2: JCN
29  Caesar 1: IBM
30  Caesar 0: HAL
31
32  Encrypt and decrypt tests:
33  SGHR HR Z SDRS RSQHMF EQNL ZKZM
34  THIS IS A TEST STRING FROM ALAN
35  UIJT JT B UFTU TUSJOH GSPN BMBO
36  THIS IS A TEST STRING FROM ALAN
37
38
39  1234567890!@#$%^&*(){}
40  1234567890!@#$%^&*(){}
41
42  Solve tests:
43  Caesar 26: HAL
44  Caesar 25: GZK
45  Caesar 24: FYJ
46  Caesar 23: EXI
47  Caesar 22: DWH
48  Caesar 21: CVG
49  Caesar 20: BUF
50  Caesar 19: ATE
51  Caesar 18: ZSD
52  Caesar 17: YRC
53  Caesar 16: XQB
54  Caesar 15: WPA
55  Caesar 14: VOZ
56  Caesar 13: UNY
57  Caesar 12: TMX
58  Caesar 11: SLW
59  Caesar 10: RKV
60  Caesar 9: QJU
61  Caesar 8: PIT
62  Caesar 7: OHS
63  Caesar 6: NGR
64  Caesar 5: MFQ
65  Caesar 4: LEP
66  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 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.