

Programming in the Past

CMPT 331 - Spring 2023 | Dr. Labouseur

Josh Seligman | joshua.seligman1@marist.edu

February 17, 2023

1 LOG

1.1 PREDICTON

I am predicting that it will take me around **20 hours (average of 4 hours per programming language)** for me to learn Fortran, COBOL, BASIC, Pascal, and Scala and build a Caesar cipher in each language. This is an extremely rough estimate as I have never used any of these programming languages before and will have to learn each of them starting with "hello world." I am sure that I will have moments of staring at my computer screen for extended periods of time due to a weird nuance or gimmick in at least one of these programming languages that is not common in more modern languages. However, despite my lack of familiarity with these languages, I am hoping that there will be some nice similarities between them, so my approach to writing the Caesar cipher does not drastically change between them, and each implementation can easily be compared to each other on an even playing field.

1.2 PROGRESS LOG

Date	Hours Spent	Tasks / Accomplishments / Issues / Thoughts
January 18	3 hours	I built the entire Caesar cipher in Fortran, which was a bit frustrating but not the worst thing in the world. The language had some weird quirks, but was doable considering that it only took me a few hours.
January 19	3 hours	I started working on the Caesar cipher in COBOL. This has been exponentially more difficult than Fortran as some of the language's tiny details made me stare at a computer screen for hours without any resources because there were no error messages or warnings.
January 20	0.25 hours	I modified my Fortran implementation to account for negative shift amounts and shift amounts greater than 26.
January 20	1.5 hours	I finished writing the Caesar cipher in COBOL and dealt with more of COBOL's small nuances that did not provide error messages or warnings. Luckily, it did not take me nearly as long to find the issues today as it did for me yesterday.
January 21	1.5 hours	I wrote the Caesar cipher in BASIC without much trouble. The code was quite simple, and I enjoyed my time writing the BASIC program.

January 26	0.75 hours	I wrote the Caesar cipher in Pascal with even less trouble than BASIC. I was genuinely surprised by how quickly I completed the Caesar cipher in Pascal and found the language to be really nice and better than some modern languages.
January 27	0.75 hours	I wrote the Caesar cipher in Scala, which was unsurprisingly almost identical to how it would have been written in Java. Scala was a bit frustrating at times because of its quirks to try to simplify Java, but its tooling made my experience better compared to some of the older languages.

1.3 FINAL RESULTS AND ANALYSIS

Overall, this assignment took me 10.75 hours to complete, which is about half the time I initially expected. Although I was on pace for 20 hours after Fortran and COBOL, the time required for me to write the Caesar cipher in BASIC, Pascal, and Scala was much shorter than originally anticipated. I initially predicted about 20 hours because I did not know what to expect from the older programming languages and was unsure how long it would take for me to learn enough of each language to put together the respective programs. Furthermore, the results from Google searches were much more detailed and useful for the last 3 languages because they are not as old and have some good online resources that demonstrate the basics of the languages. Also, since they are newer, BASIC, Pascal, and Scala all have grammars that are much closer to other popular programming languages today, so more similarities were able to be recognized and taken advantage of. On the other hand, since there is no language today like COBOL, it is in a league of its own and had to be figured out in its entirety with no assumptions of similarities to other languages. Lastly, as described in my prediction, I wanted the solutions to be as close to each other as possible. Thus, I did not try to do anything crazy with the solutions and kept it simple throughout. In other words, once I wrote the Caesar cipher in Fortran, the assignment became a task of trying to translate the Fortran code into other languages rather than trying to create a new solution from scratch 5 times.

2 COMMENTARY

2.1 FORTRAN

2.1.1 MY THOUGHTS

Building the Caesar cipher in Fortran was a challenging experience due to its strict rules regarding code organization and how functions and subroutines work. First, regarding code organization, I did not like having to declare all used variables at the top of a subroutine. In many other languages, the purpose of variables can oftentimes be inferred by the location in which they are declared. For instance, a variable declared by a loop often means it will have an important role in the loop, usually as a loop increment variable. However, in Fortran, since the variables were declared at the top of each subroutine and program, the readability of the language is hurt because the variables are all clumped together, and the language is tougher to write because you have to be careful in making sure that comments are written to explain what each variable will be used for. In the end, this organization did clean up the rest of the subroutine, but I would have rather declared the variables in more logical spots to go along with the flow of the program.

Another issue that I had with Fortran was with functions and returning values. I initially wanted to use functions to take in the original string and return the output string for encrypt and decrypt. However, functions and character objects were really not working nice as I received a bunch of errors, for which Google searches were not able to help me as the language has not been widely used since the inception of the internet. Some

of these errors included but were not limited to an "entity with assumed character length at (1) must be a dummy argument or a PARAMETER" and functions that were explicitly marked to return a character (string) were returning a REAL and caused type mismatches at compile time. In the end, the best resource on the internet was the quick start guide on Fortran's official website, whose advanced examples provided me with some inspiration to use a subroutine and pass in the variable that would be modified and used as an output. Additionally, I wanted to make the function/subroutine support characters of an unknown length to work with all possible inputs. However, despite a lot of forums online having examples with this as the situation, the code just never compiled or worked for me. Therefore, I chose to have one of the parameters of my subroutines to be the length of the string, so the variable declarations at the top of the subroutine would be able to provide a known length.

Overall, aside from the variables being declared up top, Fortran is quite a clean programming language in terms of readability as a lot of the control structures are similar to those in more modern languages. However, the lack of clarity for how functions work with strings is extremely frustrating and significantly hurts the writability of Fortran as I had to find a way to finesse a solution using subroutines and pass in the output variable, even though I would have preferred to have better organized code that returns a result from a function.

2.1.2 GOOGLE SEARCH HISTORY

- fortran hello world
- function in fortran
- string type in fortran
- get length of string in fortran
- Entity at (1) has a deferred type parameter and requires either the POINTER or ALLOCATABLE attribute
- pass in character to a function fortran
- Entity with assumed character length at (1) must be a dummy argument or a PARAMETER
- return string from function fortran
- iterate through characters in a string fortran
- fortran print string to include value of variable
- fortran mod

2.1.3 CODE AND TESTS

```

1 ! Entrypoint for the program
2 program caesar
3     ! We need to make sure that all variable types are explicitly defined
4     implicit none
5
6     ! We will need 3 strings , the original , the output for encrypt , and the output for
      decrypt
7     character(31) :: original
8     character(31) :: encryptOutput
9     character(31) :: decryptOutput
10    character(31) :: emptyString
11    character(31) :: noLetters
12

```

```

13     original = 'This_is_a_test_string_from_Alan'
14     emptyString = ''
15     noLetters = '1234567890!@#$$%^&*(){}'
16
17     print *, 'Alan_tests:'
18
19     ! Call encrypt on the original string with a shift of 8
20     call encrypt(original, 31, 8, encryptOutput)
21     print *, encryptOutput
22
23     ! Decrypt the encrypted value with the shift of 8
24     call decrypt(encryptOutput, 31, 8, decryptOutput)
25     print *, decryptOutput
26
27     ! Call the solve subroutine
28     call solve('HAL', 3, 26)
29
30     print *, ''
31     print *, 'Encrypt_and_decrypt_tests:'
32
33     ! Test going back one character
34     call encrypt(original, 31, -1, encryptOutput)
35     print *, encryptOutput
36     call decrypt(encryptOutput, 31, -1, decryptOutput)
37     print *, decryptOutput
38
39     ! Test the modulus operator to go forward 1
40     call encrypt(original, 31, 27, encryptOutput)
41     print *, encryptOutput
42     call decrypt(encryptOutput, 31, 27, decryptOutput)
43     print *, decryptOutput
44
45     ! Test empty string
46     call encrypt(emptyString, 31, 7, encryptOutput)
47     print *, encryptOutput
48     call decrypt(encryptOutput, 31, 7, decryptOutput)
49     print *, decryptOutput
50
51     ! Make sure string stays the same
52     call encrypt(noLetters, 31, 7, encryptOutput)
53     print *, encryptOutput
54     call decrypt(encryptOutput, 31, 7, decryptOutput)
55     print *, decryptOutput
56
57     print *, ''
58     print *, 'Solve_tests:'
59     ! Test absolute value
60     call solve('HAL', 3, -26)
61     print *, ''
62     ! Test modulus
63     call solve('HAL', 3, 30)
64
65 contains
66
67     ! Subroutine to encrypt a string of size stringLength by shiftAmount and store the
68     ! result in outputString
69     subroutine encrypt(inputString, stringLength, shiftAmount, outputString)
70         implicit none
71
72         ! stringLength is needed so the type of inputString and outputString can be
73         ! explicitly defined by their length
74         ! shiftAmount is the amount to shift the string by
75         integer, intent(in) :: stringLength, shiftAmount
76
77         ! inputString will not be modified and is a character type of size stringLength

```

```

76     character(stringLength), intent(in) :: inputString
77     ! outputString will be modified and has the same length as inputString
78     character(stringLength) :: outputString
79
80     ! newShift will be used to make sure the shift amount is between -25 and 25
81     ! index will be used to iterate through the string character by character
82     ! charValue is the ASCII value of a given character
83     ! diff will be used to determine if the wraparound calculation is needed
84     integer :: newShift, index, charValue, diff
85
86     ! Start off by setting outputString to the same value as inputString
87     outputString = inputString
88
89     ! Convert the shift amount to be between -25 and 25
90     newShift = mod(shiftAmount, 26)
91
92     ! Iterate through the string one character at a time
93     ! Looping and modifying individual characters found at the link below
94     ! http://computer-programming-forum.com/49-fortran/4075a24f74fcc9ce.htm
95     do index = 1, len(outputString)
96         ! Get the original character value at the given position
97         charValue = ichar(outputString(index:index))
98
99         ! Check to see if it is a lowercase letter ('a' - 'z')
100        if (charValue >= 97 .and. charValue <= 122) then
101            ! If so, make it capital by subtracting 32 ('a' - 'A')
102            charValue = charValue - 32
103        endif
104
105        ! Make sure the character is a letter ('A' - 'Z') because already converted to
106        ! uppercase
107        if (charValue >= 65 .and. charValue <= 90) then
108            ! Perform the shift by the given amount
109            charValue = charValue + newShift
110
111            ! diff is how much over the character is relative to 'Z' (90)
112            diff = charValue - 90
113
114            ! If it went over, wraparound is needed
115            if (diff > 0) then
116                ! Finish the wraparound by adding the diff to 'A' (65)
117                ! The -1 is needed because a diff of 1 means that the character is 1
118                ! beyond 'Z', which will be 'A'
119                charValue = 65 + diff - 1
120            else
121                ! diff is how much under the character is relative to 'A' (65)
122                diff = 65 - charValue
123
124                ! If it went under, wraparound is needed
125                if (diff > 0) then
126                    ! Finish the wraparound by subtracting the diff from 'Z' (90)
127                    ! The +1 is needed because a diff of 1 means that the character is 1
128                    ! beyond 'A', which will be 'Z'
129                    charValue = 90 - diff + 1
130                endif
131            endif
132
133            ! Update the output string at the index with the new character value
134            ! If the character was not originally a letter, then no change is needed,
135            ! which is why this is inside of the if block
136            outputString(index:index) = char(charValue)
137        endif
138    enddo
139 end subroutine encrypt

```

```

137      ! Subroutine to decrypt a string of size stringLength that has been encrypted by
138      ! shiftAmount and store the result in outputString
139      subroutine decrypt(inputString, stringLength, shiftAmount, outputString)
140      implicit none
141
142      ! stringLength is needed so the type of inputString and outputString can be
143      ! explicitly defined by their length
144      ! shiftAmount is the amount to unshift by
145      integer, intent(in) :: stringLength, shiftAmount
146
147      ! inputString will not be modified and is a character type of size stringLength
148      character(stringLength), intent(in) :: inputString
149      ! outputString will be modified and has the same length as inputString
150      character(stringLength) :: outputString
151
152      ! newShift will be used to make sure the shift amount is between -25 and 25
153      ! index will be used to iterate through the string character by character
154      ! charValue is the ASCII value of a given character
155      ! diff will be used to determine if the wraparound calculation is needed
156      integer :: newShift, index, charValue, diff
157
158      ! Start off by setting outputString to the same value as inputString
159      outputString = inputString
160
161      ! Convert the shift amount to be between -25 and 25
162      newShift = mod(shiftAmount, 26)
163
164      ! Iterate through the string one character at a time
165      ! Looping and modifying individual characters found at the link below
166      ! http://computer-programming-forum.com/49-fortran/4075a24f74fcc9ce.htm
167      do index = 1, len(outputString)
168          ! Get the original character value at the given position
169          charValue = ichar(outputString(index:index))
170
171          ! Check to see if it is a lowercase letter ('a' - 'z')
172          if (charValue >= 97 .and. charValue <= 122) then
173              ! If so, make it capital by subtracting 32 ('a' - 'A')
174              charValue = charValue - 32
175          endif
176
177          ! Make sure the character is a letter ('A' - 'Z') because already converted to
178          ! uppercase
179          if (charValue >= 65 .and. charValue <= 90) then
180              ! Perform the shift by the given amount
181              charValue = charValue - newShift
182
183              ! diff is how much over the character is relative to 'Z' (90)
184              diff = charValue - 90
185
186              ! If it went over, wraparound is needed
187              if (diff > 0) then
188                  ! Finish the wraparound by adding the diff to 'A' (65)
189                  ! The -1 is needed because a diff of 1 means that the character is 1
190                  ! beyond 'Z', which will be 'A'
191                  charValue = 65 + diff - 1
192              else
193                  ! diff is how much under the character is relative to 'A' (65)
194                  diff = 65 - charValue
195
196                  ! If it went under, wraparound is needed
197                  if (diff > 0) then
198                      ! Finish the wraparound by subtracting the diff from 'Z' (90)
199                      ! The +1 is needed because a diff of 1 means that the character is 1
200                      ! beyond 'A', which will be 'Z'
201                      charValue = 90 - diff + 1

```

```

197         endif
198     endif
199
200     ! Update the output string at the index with the new character value
201     ! If the character was not originally a letter, then no change is needed,
202     ! which is why this is inside of the if block
203     outputString(index:index) = char(charValue)
204     endif
205 enddo
206 end subroutine decrypt
207
208 ! Subroutine to solve a caesar cipher of the input string of size stringLength with a
209 ! set number of tries (maxShiftValue)
210 subroutine solve(inputString, stringLength, maxShiftValue)
211     implicit none
212
213     ! stringLength is needed so the type of inputString and outputString can be
214     ! explicitly defined by their length
215     ! shiftAmount is the amount to unshift by
216     integer, intent(in) :: stringLength, maxShiftValue
217
218     ! inputString will not be modified and is a character type of size stringLength
219     character(stringLength), intent(in) :: inputString
220
221     ! outputString is a temporary variable that will be used as the result from each
222     ! solve attempt
223     character(stringLength) :: outputString
224
225     ! shiftAmount is the current amount being shifted
226     integer :: shiftAmount
227
228     ! Make sure 0 <= shiftAmount <= 26
229     shiftAmount = abs(maxShiftValue)
230     if (shiftAmount > 26) then
231         shiftAmount = mod(shiftAmount, 26)
232     endif
233
234     ! Try all shift amounts from the max down to 0
235     do shiftAmount = shiftAmount, 0, -1
236         ! Call the decrypt subroutine with the current shiftAmount and store in
237         ! outputString
238         call decrypt(inputString, stringLength, -shiftAmount, outputString)
239
240         ! Print out the results
241         ! Formatting strings found at the link below
242         ! https://pages.mtu.edu/~shene/COURSES/cs201/NOTES/chap05/format.html
243         print '(a,I0,a,a)', 'Caesar_', shiftAmount, ':_', outputString
244     enddo
245 end subroutine solve
246 end program caesar

```

Listing 1: Caesar Cipher (Fortran)

```

1 > gfortran caesar.f90 -o caesar
2 > ./caesar
3 Alan tests:
4 BPQA QA I BMAB ABZQVO NZWU ITIV
5 THIS IS A TEST STRING FROM ALAN
6 Caesar 26: HAL
7 Caesar 25: GZK
8 Caesar 24: FYJ
9 Caesar 23: EXI
10 Caesar 22: DWH
11 Caesar 21: CVG
12 Caesar 20: BUF

```

```

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

```


2.2 COBOL

2.2.1 MY THOUGHTS

Based on my experience with writing the Caesar cipher in COBOL, I have determined that it is not a user-friendly programming language in the slightest of ways. COBOL is extremely readable and self-documenting as every operation and section is explicitly defined by the programmer. This is the only decent thing going for it as the rest of my experience was dreadful due to its poor implicit type conversions and its lack of stack frames for functions.

First, the implicit type conversions for passing in parameters to functions was a nightmare to deal with by itself. In each function, all variables are defined with their types, and numeric types need to state the number of digits the variable will take up. However, oftentimes as a programmer you want to pass in a raw number to a function as the number will not be used anywhere else, so storing it in a variable would not be very useful. However, COBOL does not interpret these values well against defined types. For instance, an unsigned integer parameter that needs to store 2 digits must take in a number like "08" and not just "8." The main problem here is that the compiler does not say anything is wrong with the input. Instead, it can print out that it has an "8," but all mathematical operations with the value will be completely off. The same goes with signed numbers as well, but they basically do not even work as "-001" in a 3-digit signed integer comes back as "-00" and the "1" is missing, not to mention wild math results as well. Unfortunately, the only way to really deal with parameters is to pass in a variable that has the exact data type that the function is requesting. This ensures type safety as an "8" will be the same as "08," which contradicts the problems stated before. It almost reminds me of truthy and falsy values in JavaScript as you sometimes cannot tell what the output will be because the compiler just does its own thing without telling you what it is going to do.

Another serious issue that I faced was the lack of stack frames with function calls. In my experiences with many other programming languages, they all seem to use stack frames for function calls. That is, any variable defined in a function exists for no longer than the duration of the function. This means that the next time the function is called, the variable will be reinitialized with its starting value every time. Unfortunately, this is not the case for COBOL. Based on what I am able to tell, all variables defined in data divisions exist for the entirety of the program's lifespan. Therefore, the state of the variables for a function will be preserved across multiple function calls. COBOL lets you initialize variables in the data division with a value, but it is basically useless for functions that need to make sure that the variables get reset at the start of the function call anyway. This is another example of COBOL making itself really difficult to use because it provides functionality for features that do not make sense, at least relative to today's standards and paradigms.

Writing my Caesar cipher in COBOL was by far one of the most challenging things I have done as a programmer. In fact, COBOL's poor writability made Fortran look good, which demonstrates how bad my COBOL experience was because I did not originally think highly of Fortran. Although COBOL's code is self-documenting, it was nothing that could not be done with comments in Fortran and could not make up for the excruciatingly difficult task of writing COBOL code.

2.2.2 GOOGLE SEARCH HISTORY

- cobol hello world
- cobol column rules
- functions in cobol

- user-defined functions in cobol
- data types in cobol
- iterate through characters in a string cobol
- string copy cobol
- cobol function with both linkage section and working storage
- convert character to ascii code cobol
- get character from ordinal value cobol
- cobol addition is not right
- negative numbers in cobol
- string length cobol
- working storage section variables are not wiped after a function call and their state persists between function calls

2.2.3 CODE AND TESTS

```

1 000100 Identification division.
2 ***** This function encrypts a string based on the given shift amount
3 000101     Function-id. encrypt.
4 000102 Data division.
5 000103     Working-storage section.
6 ***** Represents the current character being analyzed
7 000104         1 curChar pic 999.
8 ***** Represents the index of the string being worked on
9 000105         1 idx pic 99.
10 ***** Represents the difference to determine wraparound
11 000106         1 diff pic S9(2).
12 ***** Shift value between -25 and 25
13 000107         1 newShift pic S9(2).
14 000108 Linkage section.
15 ***** The input string
16 000109         1 inStr pic x(32).
17 ***** The amount to shift by
18 000110         1 shiftAmt pic S999.
19 ***** The shifted string
20 000111         1 res pic x(32).
21 000112 Procedure division
22 000113     using by reference inStr shiftAmt
23 000114     returning res.
24 ***** Begin by copying the input to the result and set the index to 1
25 000115     Move inStr to res
26 000116     move 1 to idx
27 ***** Adjust the shift amount to be in the range -25 to 25
28 000117     compute newShift = function mod(shiftAmt 26)
29 ***** Repeat the work for all characters in the string
30 000118     perform encrypt-work until idx > function length(inStr)
31 000119     goback.
32 000120     encrypt-work.
33 ***** Get the current character's ordinal value (ASCII + 1)
34 000121     compute curChar = function ord(res(idx:1))
35 ***** Convert to uppercase if between 'a' (98) and 'z' (123)
36 000122     if curChar >= 98 and curChar <= 123 then
37 000123         compute curChar = curChar - 32
38 000124     end-if

```

```

39 ***** Only need to modify the character if it is a letter ('A' (66) to 'Z' (91))
40 000125 if curChar >= 66 and curChar <= 91 then
41 ***** Perform the shift
42 000126 compute curChar = curChar + newShift
43 ***** Check for wraparound on the 'Z' end
44 000127 compute diff = curChar - 91
45 000128 if diff > 0 then
46 ***** Do the wraparound. -1 at the end because diff of 1 means it should be 'A'
47 000129 compute curChar = 66 + diff - 1
48 000130 else
49 ***** Check wraparound on the 'A' end
50 000131 compute diff = 66 - curChar
51 000132 if (diff > 0) then
52 ***** Do the wraparound. +1 at the end because diff of 1 means it should be 'Z'
53 000133 compute curChar = 91 - diff + 1
54 000134 end-if
55 000135 end-if
56 ***** Update the character in the result string
57 000136 move function char(curChar) to res(idx:1)
58 000137 end-if
59 000138 add 1 to idx.
60 000139 End function encrypt.
61 000200 Identification division.
62 ***** This function decrypts a string based on the given shift amount
63 000201 Function-id. decrypt.
64 000202 Data division.
65 000203 Working-storage section.
66 ***** Represents the current character being analyzed
67 000204 1 curChar pic 999.
68 ***** Represents the index of the string being worked on
69 000205 1 idx pic 99.
70 ***** Represents the difference to determine wraparound
71 000206 1 diff pic S9(2).
72 ***** Shift value between -25 and 25
73 000207 1 newShift pic S9(2).
74 000208 Linkage section.
75 ***** The input string
76 000209 1 inStr pic x(32).
77 ***** The amount to shift by
78 000210 1 shiftAmt pic S999.
79 ***** The shifted string
80 000211 1 res pic x(32).
81 000212 Procedure division
82 000213 using by reference inStr shiftAmt
83 000214 returning res.
84 ***** Begin by copying the input to the result and set the index to 1
85 000215 Move inStr to res
86 000216 move 1 to idx
87 ***** Adjust the shift amount to be in the range -25 to 25
88 000217 compute newShift = function mod(shiftAmt 26)
89 ***** Repeat the work for all characters in the string
90 000218 perform decrypt-work until idx > function length(inStr)
91 000219 goback.
92 000220 decrypt-work.
93 ***** Get the current character's ordinal value (ASCII + 1)
94 000221 compute curChar = function ord(res(idx:1))
95 ***** Convert to uppercase if between 'a' (98) and 'z' (123)
96 000222 if curChar >= 98 and curChar <= 123 then
97 000223 compute curChar = curChar - 32
98 000224 end-if
99 ***** Only need to modify the character if it is a letter ('A' (66) to 'Z' (91))
100 000225 if curChar >= 66 and curChar <= 91 then
101 ***** Perform the shift
102 000226 compute curChar = curChar - newShift
103 ***** Check for wraparound on the 'Z' end

```

```

104 000227      compute diff = curChar - 91
105 000228      if diff > 0 then
106 *****      Do the wraparound. -1 at the end because diff of 1 means it should be 'A'
107 000229          compute curChar = 66 + diff - 1
108 000230      else
109 *****      Check wraparound on the 'A' end
110 000231          compute diff = 66 - curChar
111 000232          if (diff > 0) then
112 *****      Do the wraparound. +1 at the end because diff of 1 means it should be 'Z'
113 000233              compute curChar = 91 - diff + 1
114 000234          end-if
115 000235      end-if
116 *****      Update the character in the result string
117 000236          move function char(curChar) to res(idx:1)
118 000237      end-if
119 000238      add 1 to idx.
120 000239 End function decrypt.
121 000300 Identification division.
122 *****      Function to try to break a Caesar cipher
123 000301      Function-id. solve.
124 000302 Environment division.
125 000303      Configuration section.
126 000304          Repository.
127 *****      Have to import the decrypt function
128 000305          Function decrypt.
129 000306 Data division.
130 000307      Working-storage section.
131 *****      The current amount to shift by
132 000308          1 shiftAmt pic S999.
133 *****      Negated shiftAmt
134 000309          1 realShiftAmt pic S999.
135 *****      The result string for each call to decrypt
136 000310          1 outputStr pic x(32).
137 000311 Linkage section.
138 *****      The input string
139 000312          1 inStr pic x(32).
140 *****      The max shift amount to try
141 000313          1 maxShiftAmt pic S999.
142 *****      Have to return something, so will return 0
143 000314          1 res pic 9.
144 000315 Procedure division
145 000316      using by reference inStr maxShiftAmt
146 000317      returning res.
147 000318      Move 0 to res
148 *****      Get the absolute value for the shift amount to make sure it is positive
149 *****      In the end, -26 will be the same as +26, and 27 will be the same as 1
150 000319      move function abs(maxShiftAmt) to shiftAmt
151 000320      if (shiftAmt > 26) then
152 000320          compute shiftAmt = function mod(shiftAmt 26)
153 000321      end-if
154 *****      Repeat for all possible shift amounts
155 000322      perform solve-work until shiftAmt < 0
156 000323      goback.
157 000324      solve-work.
158 *****      Negate the shift amount
159 000325      compute realShiftAmt = shiftAmt * -1
160 *****      Try to decrypt the string and display the result
161 000326      move function decrypt(inStr realShiftAmt) to outputStr
162 000327      display "Caesar_" shiftAmt ":_" outputStr
163 000328      subtract 1 from shiftAmt.
164 000329 End function solve.
165 000000 Identification division.
166 000001      Program-id. caesar.
167 000002 Environment division.
168 000003      Configuration section.

```

```

169 000004      Repository.
170 000005      Function encrypt
171 000006      function decrypt
172 000007      function solve.
173 000008 Data division.
174 000009      Working-storage section.
175 000010      1 inStr pic x(32) value "This_is_a_test_string_from_Alan".
176 000011      1 shift pic S999 value 8.
177 000012      1 encryptRes pic x(32).
178 000013      1 decryptRes pic x(32).
179 000014      1 solveStr pic x(32) value "HAL".
180 000015      1 solveShift pic S999 value 26.
181 000016      1 emptyStr pic x(32) value "_".
182 000017      1 noLetters pic x(32) value "1234567890!@#$%^&*(){}".
183 000018 Procedure division.
184 000019      display "Alan_tests:"
185 000020      Move encrypt(inStr shift) to encryptRes
186 000021      display encryptRes
187 000022      move decrypt(encryptRes shift) to decryptRes
188 000023      display decryptRes
189 000024      display solve(solveStr solveShift)
190 000025      display "_"
191 000026      display "Encrypt_and_decrypt_tests:"
192 ***** Test negative shifts
193 000027      move -1 to shift
194 000028      Move encrypt(inStr shift) to encryptRes
195 000029      display encryptRes
196 000030      move decrypt(encryptRes shift) to decryptRes
197 000031      display decryptRes
198 ***** Test modulus
199 000032      move 27 to shift
200 000033      Move encrypt(inStr shift) to encryptRes
201 000034      display encryptRes
202 000035      move decrypt(encryptRes shift) to decryptRes
203 000036      display decryptRes
204 ***** Test empty string
205 000037      move 7 to shift
206 000038      Move encrypt(emptyStr shift) to encryptRes
207 000039      display encryptRes
208 000040      move decrypt(encryptRes shift) to decryptRes
209 000041      display decryptRes
210 ***** Test no letters
211 000042      Move encrypt(noLetters shift) to encryptRes
212 000043      display encryptRes
213 000044      move decrypt(encryptRes shift) to decryptRes
214 000045      display decryptRes
215 000046      display "_"
216 000047      display "Solve_tests:"
217 ***** Test absolute value
218 000048      move -26 to solveShift
219 000049      display solve(solveStr solveShift)
220 000050      display "_"
221 ***** Test modulus
222 000051      move 30 to solveShift
223 000052      display solve(solveStr solveShift)
224 000053      goback.
225 000054 End program caesar.

```

Listing 3: Caesar Cipher (COBOL)

```

1 > cobc -x caesar.cbl
2 > ./caesar
3 Alan tests:
4 BPQA QA I BMAB ABZQVO NZWU ITIV
5 THIS IS A TEST STRING FROM ALAN

```

```

6 Caesar +026: HAL
7 Caesar +025: GZK
8 Caesar +024: FYJ
9 Caesar +023: EXI
10 Caesar +022: DWH
11 Caesar +021: CVG
12 Caesar +020: BUF
13 Caesar +019: ATE
14 Caesar +018: ZSD
15 Caesar +017: YRC
16 Caesar +016: XQB
17 Caesar +015: WPA
18 Caesar +014: VOZ
19 Caesar +013: UNY
20 Caesar +012: TMX
21 Caesar +011: SLW
22 Caesar +010: RKV
23 Caesar +009: QJU
24 Caesar +008: PIT
25 Caesar +007: OHS
26 Caesar +006: NGR
27 Caesar +005: MFQ
28 Caesar +004: LEP
29 Caesar +003: KDO
30 Caesar +002: JCN
31 Caesar +001: IBM
32 Caesar +000: HAL
33 0
34
35 Encrypt and decrypt tests:
36 SGHR HR Z SDRS RSQHMF EQNL ZKZM
37 THIS IS A TEST STRING FROM ALAN
38 UIJT JT B UFTU TUSJOH GSPN BMBO
39 THIS IS A TEST STRING FROM ALAN
40
41
42 1234567890!@#$$%^&*(){}
43 1234567890!@#$$%^&*(){}
44
45 Solve tests:
46 Caesar +026: HAL
47 Caesar +025: GZK
48 Caesar +024: FYJ
49 Caesar +023: EXI
50 Caesar +022: DWH
51 Caesar +021: CVG
52 Caesar +020: BUF
53 Caesar +019: ATE
54 Caesar +018: ZSD
55 Caesar +017: YRC
56 Caesar +016: XQB
57 Caesar +015: WPA
58 Caesar +014: VOZ
59 Caesar +013: UNY
60 Caesar +012: TMX
61 Caesar +011: SLW
62 Caesar +010: RKV
63 Caesar +009: QJU
64 Caesar +008: PIT
65 Caesar +007: OHS
66 Caesar +006: NGR
67 Caesar +005: MFQ
68 Caesar +004: LEP
69 Caesar +003: KDO
70 Caesar +002: JCN

```

```
71 Caesar +001: IBM
72 Caesar +000: HAL
73 0
74
75 Caesar +004: LEP
76 Caesar +003: KDO
77 Caesar +002: JCN
78 Caesar +001: IBM
79 Caesar +000: HAL
80 0
```

Listing 4: COBOL Output

2.3 BASIC

2.3.1 MY THOUGHTS

BASIC, as implied by its name, is a very simple programming language that, at times, felt too orthogonal due to its lack of key features that are included in almost every other programming language. Despite these flaws, my experience writing the Caesar cipher in BASIC was much better than that for Fortran and COBOL.

User-defined program structure in BASIC was nice and flexible, but prevented any form of documentation. In BASIC, every line of code is preceded with a number that represents which line in the program the code should be inserted into. Although it was annoying to micromanage the code and make sure every line number was consistent, I did not mind this too much because I enjoy seeing these lower-level details. I was also thankful for a text editor because I know I would have struggled had I manually entered each line into an Apple II computer and rewrote the lines if I ran out of space or wanted to clean up the code organization. For subroutines, I used the practice of every 1000 lines was a new subroutine (i.e., encrypt started on line 1000, decrypt started on line 2000, and solve started on line 3000). I personally thought this was really cool because I was able to explicitly define how I wanted my code to be organized. However, calling these subroutines was through the line number, so the code was not self-documenting and comments were needed to label which subroutine was being called in each instance. This was really unfortunate as the rest of BASIC's syntax is extremely readable and self-documenting.

Similar to the code organization, I had mixed feelings about the simplicity of variables in BASIC. I really liked how all identifiers had to end with either a '%' to show it is a number or a '\$' to show that it is a string. This was really nice because variable declaration was simple as the data type is implied with the name and there was never a question of a variable's type when writing the program because it is self-documented in its name. Despite this really nice syntax, BASIC provides no variable checking at runtime. For instance, if you omit the '%' or '\$' or misspell a variable, the BASIC interpreter assumes that you are trying to reference a new variable. Thus, it creates a new variable and initializes it to 0 without telling you. When writing my Caesar cipher, this "feature" screwed up program execution, and it was very difficult to debug since there were no warnings or error messages.

Aside from some of these minor issues I had with BASIC, there were a few more major issues that deterred from the language's readability and writability. First, since each line of code starts with the line number in the program, there is no formatting in BASIC to line up when different blocks of code start and end. This became very challenging to debug when I forgot an endif as the code was not formatted, which hurt the language's readability and writability as well. The other serious issue with BASIC was its lack of scope for variables. All variables are global and can be accessed from anywhere in the program. This is very dangerous when dealing with subroutines and made the code quite messy when trying to reuse input and output variables that I dedicated to the various subroutines. Considering that the program is a Caesar cipher and was messy because of variable reuse, I cannot imagine writing more complex programs in BASIC and micromanaging all the variables and how and when they are used.

Overall, the nice syntax of BASIC made it much easier to both read and write code compared to Fortran and COBOL, but the lack of formatting and variable scope makes the language much more suited for smaller programs with few moving parts as it does not scale up well with increased complexity.

2.3.2 GOOGLE SEARCH HISTORY

- hello world basic programming language
- chipmunk basic
- <https://www.youtube.com/watch?v=7r83N3c2kPw>
- mid\$ basic

2.3.3 CODE AND TESTS

```
1 9 rem Define the strings for the encryption input and output
2 10 encryptinput$ = "This_is_a_test_string_from_Alan"
3 20 encryptoutput$ = ""
4 29 rem Define the strings for the decryption input and output
5 30 decryptinput$ = ""
6 40 decryptoutput$ = ""
7 49 rem Define the shift amount for encrypt and decrypt
8 50 shiftamount% = 8
9 60 print "Alan_tests:"
10 69 rem Call the encrypt subroutine and print the output
11 70 gosub 1000
12 80 print encryptoutput$
13 89 rem Set the decrypt input to be what the encrypt output was
14 90 decryptinput$ = encryptoutput$
15 99 rem Call decrypt with the same shift amount as encrypt
16 100 gosub 2000
17 110 print decryptoutput$
18 119 rem Set the solve variables for the string and the amount
19 120 solveInput$ = "HAL"
20 130 shiftAmount% = 26
21 139 rem Call solve with the given variables
22 140 gosub 3000
23 150 print ""
24 160 print "Encrypt_and_decrypt_tests:"
25 169 rem Test the negative shift amount
26 170 shiftAmount% = -1
27 180 gosub 1000
28 190 print encryptoutput$
29 200 decryptinput$ = encryptoutput$
30 210 gosub 2000
31 220 print decryptoutput$
32 229 rem Test modulus
33 230 shiftAmount% = 27
34 240 gosub 1000
35 250 print encryptoutput$
36 260 decryptinput$ = encryptoutput$
37 270 gosub 2000
38 280 print decryptoutput$
39 289 rem Test empty string
40 290 encryptinput$ = ""
41 300 shiftAmount% = 7
42 310 gosub 1000
43 320 print encryptoutput$
44 330 decryptinput$ = encryptoutput$
45 340 gosub 2000
```



```

46 350 print decryptoutput$
47 359 rem Test no letters
48 360 encryptinput$ = "1234567890!@#%^&*(){}"
49 370 gosub 1000
50 380 print encryptoutput$
51 390 decryptinput$ = encryptoutput$
52 400 gosub 2000
53 410 print decryptoutput$
54 420 print ""
55 430 print "Solve_tests:"
56 439 rem Test the absolute value
57 440 shiftAmount% = -26
58 450 gosub 3000
59 460 print ""
60 469 rem Test modulus
61 470 shiftAmount% = 30
62 480 gosub 3000
63 999 end
64 1000 rem Encrypt subroutine
65 1009 rem Encrypt output starts off as an uppercase version of the input
66 1010 encryptoutput$ = ucase$(encryptinput$)
67 1019 rem Shift amount should be between -25 and 25
68 1020 realShift% = shiftamount% mod 26
69 1029 rem Loop through every character in the string
70 1030 for index% = 1 to len(encryptoutput$)
71 1039 rem Get the ascii code
72 1040 char% = asc(mid$(encryptoutput$, index%, 1))
73 1050 if char% >= 65 and char% <= 90 then
74 1059 rem Do the shift if it is a letter
75 1060 char% = char% + realShift%
76 1069 rem Check to see if there is wraparound on the 'Z' end
77 1070 diff% = char% - 90
78 1080 if diff% > 0 then
79 1089 rem Wrap the character around to the 'A' side
80 1090 char% = 65 + diff% - 1
81 1100 else
82 1109 rem Check to see if there is wraparound on the 'A' end
83 1110 diff% = 65 - char%
84 1120 if diff% > 0 then
85 1129 rem Wrap the character around to the 'Z' side
86 1130 char% = 90 - diff% + 1
87 1140 endif
88 1150 endif
89 1159 rem Only have to replace the character if it is a letter
90 1160 mid$(encryptoutput$, index%, 1) = chr$(char%)
91 1170 endif
92 1180 next index%
93 1190 return
94 2000 rem Decrypt subroutine
95 2009 rem Decrypt output starts off as an uppercase version of the input
96 2010 decryptoutput$ = ucase$(decryptinput$)
97 2019 rem Shift amount should be between -25 and 25
98 2020 realShift% = shiftamount% mod 26
99 2029 rem Loop through every character in the string
100 2030 for index% = 1 to len(decryptoutput$)
101 2039 rem Get the ascii code
102 2040 char% = asc(mid$(decryptoutput$, index%, 1))
103 2050 if char% >= 65 and char% <= 90 then
104 2059 rem Do the shift if it is a letter
105 2060 char% = char% - realShift%
106 2069 rem Check to see if there is wraparound on the 'Z' end
107 2070 diff% = char% - 90
108 2080 if diff% > 0 then
109 2089 rem Wrap the character around to the 'A' side
110 2090 char% = 65 + diff% - 1

```

```

111 2100 else
112 2109 rem Check to see if there is wraparound on the 'A' end
113 2110 diff% = 65 - char%
114 2120 if diff% > 0 then
115 2129 rem Wrap the character around to the 'Z' side
116 2130 char% = 90 - diff% + 1
117 2140 endif
118 2150 endif
119 2159 rem Only have to replace the character if it is a letter
120 2160 mid$(decryptoutput$, index%, 1) = chr$(char%)
121 2170 endif
122 2180 next index%
123 2190 return
124 3000 rem Solve subroutine
125 3009 rem Only need positive shift amounts
126 3010 realShift% = abs(shiftAmount%)
127 3020 if realShift% > 26 then
128 3029 rem Take the mod of the shift is greater than 26
129 3030 realShift% = realShift% mod 26
130 3040 endif
131 3049 rem Decrypt takes in the solve input string
132 3050 decryptinput$ = solveInput$
133 3060 for shift% = realShift% to 0 step -1
134 3069 rem Update the shift amount according to the current step in the loop
135 3070 shiftAmount% = -shift%
136 3079 rem Call decrypt and print the result
137 3080 gosub 2000
138 3090 print "Caesar_" shift% ":_:" decryptoutput$
139 3100 next shift%
140 3110 return

```

Listing 5: Caesar Cipher (BASIC)

```

1 Chipmunk BASIC 368b2.02
2 >load "/Users/joshuaseligman/Documents/GitHub/cmpt331-s23/programming-in-the-past/basic/
  caesar.bas
3 >run
4 Alan tests:
5 BPQA QA I BMAB ABZQVO NZWU ITIV
6 THIS IS A TEST STRING FROM ALAN
7 Caesar 26 : HAL
8 Caesar 25 : GZK
9 Caesar 24 : FYJ
10 Caesar 23 : EXI
11 Caesar 22 : DWH
12 Caesar 21 : CVG
13 Caesar 20 : BUF
14 Caesar 19 : ATE
15 Caesar 18 : ZSD
16 Caesar 17 : YRC
17 Caesar 16 : XQB
18 Caesar 15 : WPA
19 Caesar 14 : VOZ
20 Caesar 13 : UNY
21 Caesar 12 : TMX
22 Caesar 11 : SLW
23 Caesar 10 : RKV
24 Caesar 9 : QJU
25 Caesar 8 : PIT
26 Caesar 7 : OHS
27 Caesar 6 : NGR
28 Caesar 5 : MFQ
29 Caesar 4 : LEP
30 Caesar 3 : KDO
31 Caesar 2 : JCN

```

```

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

```

Listing 6: BASIC Output

2.4 PASCAL

2.4.1 MY THOUGHTS

My experience in Pascal was overall really good and I genuinely have very little to complain about. I found the code to be a great example of elegant simplicity. Pascal contained all of the basic programming constructs, including some of the more modern ones, while being less verbose than the other programming languages so far.

First, Pascal's clean and simple grammar made it both really easy to read and write the code. One of the issues in languages like Fortran and COBOL was that they required all variables to be declared at the top of functions. Fortran did not have anything to separate these declarations and the actual program, and COBOL was extremely detailed and also did not exactly line up with the actual execution of the program.

Although Pascal also requires variables to be declared at the top of functions, it has 2 very clean sections to do so: "const" for constants and "var" for everything else. The inclusion of these sections makes the code more readable as it is self-documenting within the sections as well as extremely writable as I was able to write a lot less code to do the same thing as COBOL. Another part of the language's grammar that is unique relative to the languages used so far is the walrus assignment operator (`:=`). Rather than using the single equal sign (`=`) for assignment, Pascal uses the walrus operator, which increases readability as there is no confusion between assignments and comparisons and only comes at the cost of one additional character being typed, which is a miniscule cost relative to ease of which one can understand the code.

Another feature that I really appreciated was how functions deal with return variables. In most other programming languages, you can return from any point in a function as well as have multiple return points. In Pascal, however, the returned variable takes the same name as the function and automatically gets returned at the end of the function. This design pattern upholds the idea of functions being one-way in and one-way out and makes the code extremely easy to understand and follow. I do, however, wish that the returned variable had a built-in name that was not the same name as the function as recursive functions may be hard to read because you can have multiple instances of the same name in a single line of code that both mean completely different things. Despite this small complaint from me, the improved design pattern carries a much larger weight than the name of a variable and, therefore, functions are much more readable and writable than most other programming languages today.

Overall, my experience using Pascal to write a Caesar cipher was really enjoyable due to its simple, yet feature-filled, grammar that made the code really easy to both read and write. I wish the language was more popular today because I found my experience to be better than some more modern languages that are widely used throughout the world.

2.4.2 GOOGLE SEARCH HISTORY

- pascal hello world
- pascal function multiple parameters
- and operator pascal boolean expression

2.4.3 CODE AND TESTS

```

1 program Caesar;
2 (* Function that encrypts the input string by the shift amount *)
3 function Encrypt(inStr : string; shiftAmount : integer) : string;
4 var
5     curChar: Char; (* The current character being worked with *)
6     charValue: integer; (* The ASCII value of the current character *)
7     diff: integer; (* The difference between the current character and A and Z to detect
8         wraparound *)
9     idx: integer; (* The position in the string we are at *)
10 begin
11     (* Start off by normalizing the shift amount *)
12     shiftAmount := shiftAmount mod 26;
13     Encrypt := inStr;
14
15     (* Pascal uses 1-based indexing *)
16     idx := 1;
17     (* Loop through each character in the string *)
18     for curChar in Encrypt do
19         begin
20             (* Get the initial character ASCII code *)
21             charValue := ord(curChar);

```

```

22      (* Convert the character to uppercase if it is not already *)
23      if ((charValue >= 97) and (charValue <= 122)) then
24          charValue := charValue - 32;
25
26      (* Caesar cipher only impacts letters, so check that conditon first *)
27      if ((charValue >= 65) and (charValue <= 90)) then
28          begin
29              (* Perform the shift *)
30              charValue := charValue + shiftAmount;
31
32              (* Check wraparound on the Z end *)
33              diff := charValue - 90;
34              if diff > 0 then
35                  (* Perform the wraparound (-1 is needed because 91 should be 65 (A)) *)
36                  charValue := 65 + diff - 1
37              else
38                  begin
39                      (* Check the low end of the range *)
40                      diff := 65 - charValue;
41                      if diff > 0 then
42                          (* Perform the wraparound (+1 is needed because 64 should be 90 (Z)) *)
43                          charValue := 90 - diff + 1;
44                      end;
45                      (* Update the string accordingly *)
46                      Encrypt[idx] := chr(charValue);
47                  end;
48              (* Increment the index for our own reference *)
49              idx := idx + 1;
50          end;
51      end;
52
53      (* Function that decrypts the input string by the shift amount *)
54      function Decrypt(inStr : string; shiftAmount : integer) : string;
55      var
56          curChar: Char; (* The current character being worked with *)
57          charValue: integer; (* The ASCII value of the current character *)
58          diff: integer; (* The difference between the current character and A and Z to detect
59                          wraparound *)
59          idx: integer; (* The position in the string we are at *)
60      begin
61          (* Start off by normalizing the shift amount *)
62          shiftAmount := shiftAmount mod 26;
63          Decrypt := inStr;
64
65          (* Pascal uses 1-based indexing *)
66          idx := 1;
67          (* Loop through each character in the string *)
68          for curChar in Decrypt do
69              begin
70                  (* Get the initial character ASCII code *)
71                  charValue := ord(curChar);
72
73                  (* Convert the character to uppercase if it is not already *)
74                  if ((charValue >= 97) and (charValue <= 122)) then
75                      charValue := charValue - 32;
76
77                  (* Caesar cipher only impacts letters, so check that conditon first *)
78                  if ((charValue >= 65) and (charValue <= 90)) then
79                      begin
80                          (* Perform the shift *)
81                          charValue := charValue - shiftAmount;
82
83                          (* Check wraparound on the Z end *)
84                          diff := charValue - 90;
85                          if diff > 0 then

```

```

86         (* Perform the wraparound (-1 is needed because 91 should be 65 (A)) *)
87         charValue := 65 + diff - 1
88     else
89     begin
90         (* Check the low end of the range *)
91         diff := 65 - charValue;
92         if diff > 0 then
93             (* Perform the wraparound (+1 is needed because 64 should be 90 (Z)) *)
94             charValue := 90 - diff + 1;
95         end;
96         (* Update the string accordingly *)
97         Decrypt[idx] := chr(charValue);
98     end;
99     (* Increment the index for our own reference *)
100     idx := idx + 1;
101 end;
102 end;
103
104 (* Procedure to solve a Caesar cipher *)
105 procedure solve(inputString : string; maxShiftAmount : integer);
106 var
107     curShift: integer; (* The current shift amount *)
108 begin
109     (* Normalize the max shift amount being used *)
110     maxShiftAmount := abs(maxShiftAmount);
111     if maxShiftAmount > 26 then
112         maxShiftAmount := maxShiftAmount mod 26;
113     end;
114     (* Loop through everything *)
115     for curShift := maxShiftAmount downto 0 do
116         (* Perform the decrypt and print the result *)
117         writeln('Caesar_', curShift, ':_', Decrypt(inputString, -curShift));
118     end;
119
120 var
121     originalString: string; (* The original string *)
122     encryptOut: string; (* The output of encrypt *)
123     decryptOut: string; (* The output of decrypt *)
124 begin
125     originalString := 'This_is_a_test_string_from_Alan';
126     writeln('Alan_tests:');
127     encryptOut := Encrypt(originalString, 8);
128     decryptOut := Decrypt(encryptOut, 8);
129     writeln(encryptOut);
130     writeln(decryptOut);
131     solve('HAL', 26);
132
133     writeln();
134     writeln('Encrypt_and_decrypt_tests:');
135     (* Test negative shift *)
136     encryptOut := Encrypt(originalString, -1);
137     decryptOut := Decrypt(encryptOut, -1);
138     writeln(encryptOut);
139     writeln(decryptOut);
140     (* Test modulus *)
141     encryptOut := Encrypt(originalString, 27);
142     decryptOut := Decrypt(encryptOut, 27);
143     writeln(encryptOut);
144     writeln(decryptOut);
145     (* Test empty string *)
146     encryptOut := Encrypt('', 7);
147     decryptOut := Decrypt(encryptOut, 7);
148     writeln(encryptOut);
149     writeln(decryptOut);
150     (* Test no letters *)

```

```

151     encryptOut := Encrypt('1234567890!@#$$%^&*(){}', 7);
152     decryptOut := Decrypt(encryptOut, 7);
153     writeln(encryptOut);
154     writeln(decryptOut);
155
156     writeln();
157     writeln('Solve tests:');
158     (* Test absolute value *)
159     solve('HAL', -26);
160     writeln();
161     (* Test modulus *)
162     solve('HAL', 30);
163 end.

```

Listing 7: Caesar Cipher (Pascal)

```

1 > fpc caesar.pas
2 Free Pascal Compiler version 3.2.2 [2022/11/29] for x86_64
3 Copyright (c) 1993–2021 by Florian Klaempfl and others
4 Target OS: Darwin for x86_64
5 Compiling caesar.pas
6 Assembling caesar
7 Linking caesar
8 162 lines compiled, 0.3 sec
9 > ./caesar
10 Alan tests:
11 BPQA QA I BMAB ABZQVO NZWU ITIV
12 THIS IS A TEST STRING FROM ALAN
13 Caesar 26: HAL
14 Caesar 25: GZK
15 Caesar 24: FYJ
16 Caesar 23: EXI
17 Caesar 22: DWH
18 Caesar 21: CVG
19 Caesar 20: BUF
20 Caesar 19: ATE
21 Caesar 18: ZSD
22 Caesar 17: YRC
23 Caesar 16: XQB
24 Caesar 15: WPA
25 Caesar 14: VOZ
26 Caesar 13: UNY
27 Caesar 12: TMX
28 Caesar 11: SLW
29 Caesar 10: RKV
30 Caesar 9: QJU
31 Caesar 8: PIT
32 Caesar 7: OHS
33 Caesar 6: NGR
34 Caesar 5: MFQ
35 Caesar 4: LEP
36 Caesar 3: KDO
37 Caesar 2: JCN
38 Caesar 1: IBM
39 Caesar 0: HAL
40
41 Encrypt and decrypt tests:
42 SGHR HR Z SDRS RSQHMF EQNL ZKZM
43 THIS IS A TEST STRING FROM ALAN
44 UIJT JT B UFTU TUSJOH GSPN BMBO
45 THIS IS A TEST STRING FROM ALAN
46
47
48 1234567890!@#$$%^&*(){}
49 1234567890!@#$$%^&*(){}

```

```

50
51 Solve tests:
52 Caesar 26: HAL
53 Caesar 25: GZK
54 Caesar 24: FYJ
55 Caesar 23: EXI
56 Caesar 22: DWH
57 Caesar 21: CVG
58 Caesar 20: BUF
59 Caesar 19: ATE
60 Caesar 18: ZSD
61 Caesar 17: YRC
62 Caesar 16: XQB
63 Caesar 15: WPA
64 Caesar 14: VOZ
65 Caesar 13: UNY
66 Caesar 12: TMX
67 Caesar 11: SLW
68 Caesar 10: RKV
69 Caesar 9: QJU
70 Caesar 8: PIT
71 Caesar 7: OHS
72 Caesar 6: NGR
73 Caesar 5: MFQ
74 Caesar 4: LEP
75 Caesar 3: KDO
76 Caesar 2: JCN
77 Caesar 1: IBM
78 Caesar 0: HAL
79
80 Caesar 4: LEP
81 Caesar 3: KDO
82 Caesar 2: JCN
83 Caesar 1: IBM
84 Caesar 0: HAL

```

Listing 8: Pascal Output

2.5 PROCEDURAL SCALA

Scala, although very similar to Java, was a bit awkward to write in a procedural manner because its identity lies closer to a functional programming language than a procedural one. Although it tries to be a better version of Java, its increased simplicity makes the syntax feel a bit constrained and challenging to write at times.

First, Scala's simple syntax makes it a bit hard to both read and write in certain situations. Looping in Scala is similar to the other languages used in this assignment as it condenses the variable declaration, comparison, and post-iteration adjustment into a few words. However, the syntax is inconsistent with declaring variables everywhere else in the program, which I do not really understand. Speaking of variable declaration, Scala's syntax for declaring variables prevents one from understanding if a variable is a constant or not. Variable declaration for constants begin with "val" and variable declarations for mutable variables begin with "var." This one letter difference does not make it very readable as one can easily glance over this error (I did several times) and is not very writable because the change is so minor that I do not feel like I am expressing the purpose of a variable with a single letter.

Fortunately, since Scala is a newer programming language, it provides developers with a wide range of tools out of the box. More specifically, its compiler is designed to increase understanding of the problems that it finds rather than just building programs. For instance, when I was trying to assign a new value to a variable declared with "val," the compiler initially threw an unhelpful error of the problem and suggested that I use the "-explain" flag for more details. I then tried to recompile the project with the "-explain" flag

and the compiler not only showed me the line of code that had the error, but also explained why it was throwing the error and how to fix it. This was extremely useful as it helped me fix the problem as well as understand what I did wrong so I could learn from my mistakes. This feature is not uncommon with compilers of newer programming languages (Rust is a great example) as it helps boost the writability of the language by providing better tools for developers to quickly get the answers to their problems rather than having to perform a Google search on an error message and hope that someone else has previously had the error.

Overall, Scala was not a bad programming language for the Caesar cipher as it has a lot of similar features and syntax patterns as a lot of other programming languages. However, its oversimplified syntax hurts it at times and is quickly made up for by its compiler that was built to increase developer productivity.

2.5.1 GOOGLE SEARCH HISTORY

- scala hello world
- scala loop through a string
- scala string object
- absolute value scala
- scala convert int to char

2.5.2 CODE AND TESTS

```
1 object Caesar {
2   // Function to encrypt a string by the given shift amount
3   def encrypt(inputStr: String, shiftAmount: Int): String = {
4     // Compute the actual shift amount being used
5     val realShift: Int = shiftAmount % 26
6
7     // String builder to put together the final string without copying the data
8     var outputBuilder: StringBuilder = new StringBuilder()
9
10    // Loop through the input string
11    for (i <= 0 until inputStr.length) {
12      // Get the current character ASCII value
13      var charVal: Int = inputStr.charAt(i)
14
15      // Convert to uppercase
16      if (charVal >= 97 && charVal <= 122) {
17        charVal -= 32
18      }
19
20      // Only have to do work if it is uppercase ('A' - 'Z')
21      if (charVal >= 65 && charVal <= 90) {
22        // Compute the shift
23        charVal += realShift
24
25        // Check for wraparound on the 'Z' end
26        var diff: Int = charVal - 90
27        if (diff > 0) {
28          // Perform the wraparound
29          charVal = 65 + diff - 1
30        } else {
31          // Check for wraparound on the 'A' end
32          diff = 65 - charVal
33          if (diff > 0) {
34            // Perform the wraparound
```

```

35         charVal = 90 - diff + 1
36     }
37 }
38 }
39 // Add the character to the output
40 outputBuilder.append(charVal.toChar)
41 }
42
43 // Return the string
44 return outputBuilder.toString()
45 }
46
47 // Function to decrypt a string by the given shift amount
48 def decrypt(inputStr: String, shiftAmount: Int): String = {
49     // Compute the actual shift amount being used
50     val realShift: Int = shiftAmount % 26
51
52     // String builder to put together the final string without copying the data
53     var outputBuilder: StringBuilder = new StringBuilder()
54
55     // Loop through the input string
56     for (i <- 0 until inputStr.length) {
57         // Get the current character ASCII value
58         var charVal: Int = inputStr.charAt(i)
59
60         // Convert to uppercase
61         if (charVal >= 97 && charVal <= 122) {
62             charVal -= 32
63         }
64
65         // Only have to do work if it is uppercase ('A' - 'Z')
66         if (charVal >= 65 && charVal <= 90) {
67             // Compute the shift
68             charVal -= realShift
69
70             // Check for wraparound on the 'Z' end
71             var diff: Int = charVal - 90
72             if (diff > 0) {
73                 // Perform the wraparound
74                 charVal = 65 + diff - 1
75             } else {
76                 // Check for wraparound on the 'A' end
77                 diff = 65 - charVal
78                 if (diff > 0) {
79                     // Perform the wraparound
80                     charVal = 90 - diff + 1
81                 }
82             }
83         }
84         // Add the character to the output
85         outputBuilder.append(charVal.toChar)
86     }
87
88     // Return the string
89     return outputBuilder.toString()
90 }
91
92 // Function to solve a Caesar cipher
93 def solve(inputStr: String, maxShiftAmount: Int) = {
94     // Normalize the max shift to be between 0 and 26
95     var realMaxShiftAmount = maxShiftAmount.abs
96     if (realMaxShiftAmount > 26) {
97         realMaxShiftAmount = realMaxShiftAmount % 26
98     }
99 }

```

```

100     // Loop through each shift amount
101     for (curShift <- realMaxShiftAmount to 0 by -1) {
102         // Print the result of running decrypt on the string
103         println(s"Caesar_${curShift}: ${decrypt(inputStr, -curShift)}")
104     }
105 }
106
107 def main(args: Array[String]) = {
108     println("Alan_tests:")
109     var encryptOut: String = encrypt("This_is_a_test_string_from_Alan", 8)
110     println(encryptOut)
111     var decryptOut: String = decrypt(encryptOut, 8)
112     println(decryptOut)
113     solve("HAL", 26)
114
115     println("\nEncrypt_and_decrypt_tests:")
116     // Test negative shift amount
117     encryptOut = encrypt("This_is_a_test_string_from_Alan", -1)
118     println(encryptOut)
119     decryptOut = decrypt(encryptOut, -1)
120     println(decryptOut)
121
122     // Test modulus
123     encryptOut = encrypt("This_is_a_test_string_from_Alan", 27)
124     println(encryptOut)
125     decryptOut = decrypt(encryptOut, 27)
126     println(decryptOut)
127
128     // Test empty string
129     encryptOut = encrypt("", 27)
130     println(encryptOut)
131     decryptOut = decrypt(encryptOut, 27)
132     println(decryptOut)
133
134     // Test no letters
135     encryptOut = encrypt("1234567890!@#$$%^&*(){}", 27)
136     println(encryptOut)
137     decryptOut = decrypt(encryptOut, 27)
138     println(decryptOut)
139
140     println("\nSolve_tests:")
141     // Test absolute value
142     solve("HAL", -26)
143     println()
144     // Test modulus
145     solve("HAL", 30)
146
147 }
148 }

```

Listing 9: Caesar Cipher (Scala)

```

1 > scalac caesar.scala -explain
2 > scala Caesar
3 Alan tests:
4 BPQA QA I BMAB ABZQVO NZWU ITIV
5 THIS IS A TEST STRING FROM ALAN
6 Caesar 26: HAL
7 Caesar 25: GZK
8 Caesar 24: FYJ
9 Caesar 23: EXI
10 Caesar 22: DWH
11 Caesar 21: CVG
12 Caesar 20: BUF
13 Caesar 19: ATE

```

```

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

```

3 CONCLUSION

Here are my final rankings for the 5 programming languages:

1. Pascal
2. Scala
3. BASIC
4. Fortran
5. COBOL

Starting from the bottom, COBOL was a near nightmare to work with because it was extremely difficult to write and, despite its verbose code, did not follow the code that I wrote and, instead, decided to do its own thing at times. Next, Fortran was not a bad programming language by any means, but it was very difficult to write and had a compiler that produced confusing errors to understand. In second and third place, I have Scala and BASIC, respectively. Although I enjoyed writing BASIC more than Scala, the unsafe program structure of BASIC was tricky to work with at times, while the Scala compiler provided me with a great experience despite me not entirely liking its syntax. Lastly, Pascal was my favorite programming language in this assignment because it demonstrated the perfect balance of simplicity in its code while also being detailed enough for the code to document itself without me questioning what the code does.

4 APPENDIX: RISC-V ASSEMBLY

4.1 MY THOUGHTS

For an additional challenge (and some fun), I decided to also complete the assignment in RISC-V assembly. Since the solution in the other programming languages was quite simple, the logic of the RISC-V assembly was very similar and was easy to translate from the other languages. I was genuinely surprised by the writability of RISC-V assembly, but this increased writability hurts the readability from the short instructions.

RISC-V was surprisingly extremely expressive for an assembly language. RISC-V is unique in that it has a very small instruction set with not many features. However, it comes with a collection of pseudo instructions that all translate more common instructions in other assembly languages into the RISC-V equivalent. For instance, instead of loading a constant to a register by doing "ADDI rd, zero, immediate," RISC-V has a pseudo instruction of LI, which stands for load immediate and just takes in the destination register and the immediate. This expressiveness made it easier to write as the pseudo instruction is a mini higher-level abstraction that allowed me to focus on the logic and register management rather than dealing with any small nuances with RISC-V assembly. Additionally, the small instruction set made it quick to learn as I was able to build the entire program with only a small number of instructions. Next, the program structure of the RISC-V implementation reminded me the most of BASIC as it was just one big program and I had to manage the different subroutines that get run and the registers that are needed in each subroutine. However, one of the features of RISC-V that BASIC was lacking was a stack pointer. With this capability, I was able to modularize my code and not worry about a subroutine overwriting a register that was being used in a different subroutine. This is extremely important as RISC-V assembly is able to scale with increased complexity without needing to go crazy about making sure the data are safe when calling different subroutines.

Despite the impressive writability of RISC-V assembly, the terse instructions made the RISC-V assembly difficult to read. Since instructions are typically 2-4 letters long, I always felt that I was not being verbose enough to be able to understand the code that I wrote. Thus, comments were a necessity to explain what each chunk of code did so I could look back and understand the logic behind the various subroutines. Assembly is already hard enough to read because of its extreme low-level nature and the many steps needed to perform basic operations, so the very short instructions never helped with being able to read the code because I had to think to myself what each instruction represented.

Overall, I really enjoyed writing the Caesar cipher in RISC-V assembly and found it easier than Fortran and COBOL. However, assembly is not a reasonable choice for complex programs as one has to manage all of the registers by hand and micromanage every tiny detail of the program's execution. Regardless, I believe that RISC-V assembly is a really nice way to better understand both computer hardware as well as compilers and how they manage registers and the various resources in a computer.

4.2 RESOURCES I USED

As RISC-V assembly was purely for my own enjoyment, I did not time myself or keep track of my Google Search history. Instead, I did keep track of some of the most useful resources I found.

- Toolchain (Homebrew): <https://github.com/riscv-software-src/homebrew-riscv>
- RISC-V hello world (one of my favorite youtube channels for CS): <https://www.youtube.com/watch?v=GWIAQs4-UQ0>
- Instruction cheat sheet: <http://blog.translusion.com/images/posts/RISC-V-cheatsheet-RV32I-4-3.pdf>
- Stack and functions: [http://wla.berkeley.edu/cs61c/fa17/lec/06/L06%20RISCV%20Functions%20\(1up\).pdf](http://wla.berkeley.edu/cs61c/fa17/lec/06/L06%20RISCV%20Functions%20(1up).pdf)
- Linux system calls: <https://github.com/riscv-collab/riscv-gnu-toolchain/blob/master/linux-headers/include/asm-generic/unistd.h>
- RISC-V book: <https://riscv-programming.org/book/riscv-book.html>

4.3 CODE AND TESTS

```
1 .section .text
2 .global _start
3
4 _start:
5     nop        # Needed for gdb to work properly
6
7     # Alan encrypt and decrypt tests
8     la         a0, alan_test
9     li         a1, 31
10    li         a2, 8
11    call        encrypt
12
13    li         a7, 64
14    li         a0, 1
15    la         a1, alan_test
16    li         a2, 32
17    ecall
18
19    la         a0, alan_test
20    li         a1, 31
21    li         a2, 8
22    call        decrypt
23
24    li         a7, 64
```

```

25     li    a0, 1
26     la    a1, alan_test
27     li    a2, 32
28     ecall
29
30     # Alan solve test
31     la     a0, hal
32     li     a1, 3
33     li     a2, 26
34     call   solve
35
36     li    a7, 64
37     li    a0, 1
38     la    a1, empty_string
39     li    a2, 1
40     ecall
41
42     # Test negative shift amounts
43     la     a0, alan_test
44     li     a1, 31
45     li     a2, -1
46     call   encrypt
47
48     li    a7, 64
49     li    a0, 1
50     la    a1, alan_test
51     li    a2, 32
52     ecall
53
54     la     a0, alan_test
55     li     a1, 31
56     li     a2, -1
57     call   decrypt
58
59     li    a7, 64
60     li    a0, 1
61     la    a1, alan_test
62     li    a2, 32
63     ecall
64
65     # Test modulus with encrypt and decrypt
66     la     a0, alan_test
67     li     a1, 31
68     li     a2, 27
69     call   encrypt
70
71     li    a7, 64
72     li    a0, 1
73     la    a1, alan_test
74     li    a2, 32
75     ecall
76
77     la     a0, alan_test
78     li     a1, 31
79     li     a2, 27
80     call   decrypt
81
82     li    a7, 64
83     li    a0, 1
84     la    a1, alan_test
85     li    a2, 32
86     ecall
87
88     # Test an empty string
89     la     a0, empty_string

```

```

90     li      a1, 0
91     li      a2, 7
92     call    encrypt
93
94     li      a7, 64
95     li      a0, 1
96     la      a1, empty_string
97     li      a2, 1
98     ecall
99
100    la      a0, empty_string
101    li      a1, 0
102    li      a2, 7
103    call    decrypt
104
105    li      a7, 64
106    li      a0, 1
107    la      a1, empty_string
108    li      a2, 1
109    ecall
110
111    # Test with no letters (only numbers and symbols)
112    la      a0, no_letters
113    li      a1, 0
114    li      a2, 22
115    call    encrypt
116
117    li      a7, 64
118    li      a0, 1
119    la      a1, no_letters
120    li      a2, 23
121    ecall
122
123    la      a0, no_letters
124    li      a1, 22
125    li      a2, 7
126    call    decrypt
127
128    li      a7, 64
129    li      a0, 1
130    la      a1, no_letters
131    li      a2, 23
132    ecall
133
134    li      a7, 64
135    li      a0, 1
136    la      a1, empty_string
137    li      a2, 1
138    ecall
139
140    # Test abs with solve
141    la      a0, hal
142    li      a1, 3
143    li      a2, -26
144    call    solve
145
146    li      a7, 64
147    li      a0, 1
148    la      a1, empty_string
149    li      a2, 1
150    ecall
151
152    # Test modulus with solve
153    la      a0, hal
154    li      a1, 3

```



```

155     li      a2, 30
156     call    solve
157
158     addi     a7, zero, 93 # exit sys call
159     addi     a0, zero, 0 # exit status 0
160     ecall
161
162 # a0 = address of the string to encrypt
163 # a1 = length of the string
164 # a2 = shift amount
165 encrypt:
166     mv      t0, a0 # Copy string address into a temporary register
167     li      t1, 0 # Counter variable for the character we are on
168     mv      t2, a2 # Move the shift amount into a temporary register
169     li      t6, 26 # Used for getting the mod of the shift amount
170
171 # If shift > 26, then keep subtracting until < 26
172 encrypt_positive_mod:
173     blt     t2, t6, encrypt_change_limit
174     sub     t2, t2, t6
175     j       encrypt_positive_mod
176
177 # Change to -26 for next check
178 encrypt_change_limit:
179     li      t6, -26
180
181 # If shift < -26, then keep subtracting -26 (add 26) until > -26
182 encrypt_negative_mod:
183     blt     t6, t2, encrypt_loop
184     sub     t2, t2, t6
185     j       encrypt_negative_mod
186
187 # The actual encryption loop
188 encrypt_loop:
189     lb      t3, 0(t0) # Get the current character
190     li      t5, 97 # a
191     li      t6, 122 # z
192
193 # Convert to uppercase if needed
194     blt     t3, t5, encrypt_conversion
195     blt     t6, t3, encrypt_conversion
196     addi     t3, t3, -32 # Convert to uppercase
197
198 encrypt_conversion:
199     li      t5, 65 # A
200     li      t6, 90 # Z
201
202 # Only continue if there is a letter
203     blt     t3, t5, encrypt_check_loop
204     blt     t6, t3, encrypt_check_loop
205     add     t3, t3, t2
206
207     sub     t4, t3, t6 # character - 90 (check for Z wraparound)
208     bge     zero, t4, encrypt_lower_wraparound
209 # Char value is A (t5) + diff (t4) - 1
210     add     t3, t4, t5
211     addi     t3, t3, -1
212
213 # Can go straight to storing as we do not need to worry about A wraparound
214     j       encrypt_store_char
215
216 encrypt_lower_wraparound:
217     sub     t4, t5, t3 # 65 - char (check for A wraparound)
218     bge     zero, t4, encrypt_store_char
219 # Char value is Z (t6) - diff (t4) + 1

```

```

220         sub t3, t6, t4
221         addi t3, t3, 1
222
223     encrypt_store_char:
224         sb t3, 0(t0) # Store it in the output string
225
226     encrypt_check_loop:
227         addi t0, t0, 1 # Move to next byte
228         addi t1, t1, 1 # Increment the counter
229         ble t1, a1, encrypt_loop # Loop if there is still more to do
230     ret
231
232 # a0 = address of the string to decrypt
233 # a1 = length of the string
234 # a2 = shift amount
235 decrypt:
236     mv t0, a0 # Copy string address into a temporary register
237     li t1, 0 # Counter variable for the character we are on
238     mv t2, a2 # Move the shift amount into a temporary register
239     li t6, 26 # Used for getting the mod of the shift amount
240
241     # If shift > 26, then keep subtracting until < 26
242     decrypt_positive_mod:
243         blt t2, t6, decrypt_change_limit
244         sub t2, t2, t6
245         j decrypt_positive_mod
246
247     # Change to -26 for next check
248     decrypt_change_limit:
249         li t6, -26
250
251     # If shift < -26, then keep subtracting -26 (add 26) until > -26
252     decrypt_negative_mod:
253         blt t6, t2, decrypt_loop
254         sub t2, t2, t6
255         j decrypt_negative_mod
256
257     # The actual encryption loop
258     decrypt_loop:
259         lb t3, 0(t0) # Get the current character
260         li t5, 97 # a
261         li t6, 122 # z
262
263         # Convert to uppercase if needed
264         blt t3, t5, decrypt_conversion
265         blt t6, t3, decrypt_conversion
266         addi t3, t3, -32 # Convert to uppercase
267
268     decrypt_conversion:
269         li t5, 65 # A
270         li t6, 90 # Z
271
272         # Only continue if there is a letter
273         blt t3, t5, decrypt_check_loop
274         blt t6, t3, decrypt_check_loop
275         sub t3, t3, t2
276
277         sub t4, t3, t6 # character - 90 (check for Z wraparound)
278         bge zero, t4, decrypt_lower_wraparound
279         # Char value is A (t5) + diff (t4) - 1
280         add t3, t4, t5
281         addi t3, t3, -1
282
283         # Can go straight to storing as we do not need to worry about A wraparound
284         j decrypt_store_char

```

```

285
286     decrypt_lower_wraparound:
287         sub t4, t5, t3 # 65 - char (check for A wraparound)
288         bge zero, t4, decrypt_store_char
289         # Char value is Z (t6) - diff (t4) + 1
290         sub t3, t6, t4
291         addi t3, t3, 1
292
293     decrypt_store_char:
294         sb t3, 0(t0) # Store it in the output string
295
296     decrypt_check_loop:
297         addi t0, t0, 1 # Move to next byte
298         addi t1, t1, 1 # Increment the counter
299         ble t1, a1, decrypt_loop # Loop if there is still more to do
300     ret
301
302 # a0 = address of the string to solve
303 # a1 = string length
304 # a2 = max shift amount
305 solve:
306     mv t0, a2 # Store the current shift amount
307     li t1, 26 # 26 constant for use to take the modulus
308
309     solve_abs:
310         # Negate the value if needed to make it positive
311         bge t0, zero, solve_mod
312         neg t0, t0
313
314     # Take the mod to make the max shift <= 26
315     solve_mod:
316         ble t0, t1, solve_loop
317         sub t0, t0, t1
318         j solve_mod
319
320     solve_loop:
321         # Break when less than 0
322         blt t0, zero, solve_break
323
324         # Negate the current shift amount for the decrypt function
325         neg a2, t0
326
327         # Push the current shift amount and return address onto the stack
328         addi sp, sp, -8
329         sw ra, 4(sp)
330         sw t0, 0(sp)
331
332         call decrypt
333
334         # Pop the current shift amount from the stack
335         lw t0, 0(sp)
336         lw ra, 4(sp)
337         addi sp, sp, 8
338
339         # Print out the current shifted string
340         li a7, 64
341         addi a2, a1, 1
342         mv a1, a0
343         li a0, 1
344         ecall
345
346         # Reset the registers to work with decrypt
347         mv a0, a1
348         addi a1, a2, -1
349         mv a2, t0

```

```

350
351     # We now have to call decrypt with the non-negated shift amount
352     # because decrypt modifies the original string, so we need to fix that
353
354     # Push the current shift amount and return address onto the stack
355     addi    sp, sp, -8
356     sw      ra, 4(sp)
357     sw      t0, 0(sp)
358
359     call    decrypt
360
361     # Pop the current shift amount from the stack
362     lw      t0, 0(sp)
363     lw      ra, 4(sp)
364     addi    sp, sp, 8
365
366     # Decrement the shift amount
367     addi    t0, t0, -1
368
369     # Go back to the top
370     j       solve_loop
371
372 solve_break:
373     ret
374
375 alan_test:
376     .ascii "This is a test string from Alan\n"
377
378 hal:
379     .ascii "HAL\n"
380
381 empty_string:
382     .ascii "\n"
383
384 no_letters:
385     .ascii "1234567890!@#$$%^&*(){}\n"

```

Listing 11: Caesar Cipher (RISC-V Assembly)

```

1 > make
2 rm -f caesar.o
3 rm -f caesar
4 riscv64-unknown-elf-as -g -c caesar.s -o caesar.o
5 riscv64-unknown-elf-gcc -g caesar.o -o caesar -nostdlib -static
6 > make run
7 riscv64-unknown-elf-run caesar
8 BPQA QA I BMAB ABZQVO NZWU ITIV
9 THIS IS A TEST STRING FROM ALAN
10 HAL
11 GZK
12 FYJ
13 EXI
14 DWH
15 CVG
16 BUF
17 ATE
18 ZSD
19 YRC
20 XQB
21 WPA
22 VOZ
23 UNY
24 TMX
25 SLW
26 RKV

```

```

27 QJU
28 PIT
29 OHS
30 NGR
31 MFQ
32 LEP
33 KDO
34 JCN
35 IBM
36 HAL
37
38 SGHR HR Z SDRS RSQHMF EQNL ZKZM
39 THIS IS A TEST STRING FROM ALAN
40 UIJT JT B UFTU TUSJOH GSPN BMBO
41 THIS IS A TEST STRING FROM ALAN
42
43
44 1234567890!@#$$%^&*(){}
45 1234567890!@#$$%^&*(){}
46
47 HAL
48 GZK
49 FYJ
50 EXI
51 DWH
52 CVG
53 BUF
54 ATE
55 ZSD
56 YRC
57 XQB
58 WPA
59 VOZ
60 UNY
61 TMX
62 SLW
63 RKV
64 QJU
65 PIT
66 OHS
67 NGR
68 MFQ
69 LEP
70 KDO
71 JCN
72 IBM
73 HAL
74
75 LEP
76 KDO
77 JCN
78 IBM
79 HAL

```

Listing 12: RISC-V Assembly Output