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 (https://fortran-lang.org/en/learn/quickstart/)
- 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

```fortran
! Entrypoint for the program
program caesar
    ! We need to make sure that all variable types are explicitly defined
    implicit none

    ! We will need 3 strings, the original, the output for encrypt, and the output for
        decrypt
    character(31) :: original
    character(31) :: encryptOutput
    character(31) :: decryptOutput
    character(31) :: emptyString
    character(31) :: noLetters

```

```fortran
    original = 'This is a test string from Alan'
    emptyString = ''
    noLetters = '1234567890!@#$%^&*(){}'

    print *, 'Alan tests:'

    ! Call encrypt on the original string with a shift of 8
    call encrypt(original, 31, 8, encryptOutput)
    print *, encryptOutput

    ! Decrypt the encrypted value with the shift of 8
    call decrypt(encryptOutput, 31, 8, decryptOutput)
    print *, decryptOutput

    ! Call the solve subroutine
    call solve('HAL', 3, 26)

    print *, ''
    print *, 'Encrypt and decrypt tests:'

    ! Test going back one character
    call encrypt(original, 31, -1, encryptOutput)
    print *, encryptOutput
    call decrypt(encryptOutput, 31, -1, decryptOutput)
    print *, decryptOutput

    ! Test the modulus operator to go forward 1
    call encrypt(original, 31, 27, encryptOutput)
    print *, encryptOutput
    call decrypt(encryptOutput, 31, 27, decryptOutput)
    print *, decryptOutput

    ! Test empty string
    call encrypt(emptyString, 31, 7, encryptOutput)
    print *, encryptOutput
    call decrypt(encryptOutput, 31, 7, decryptOutput)
    print *, decryptOutput

    ! Make sure string stays the same
    call encrypt(noLetters, 31, 7, encryptOutput)
    print *, encryptOutput
    call decrypt(encryptOutput, 31, 7, decryptOutput)
    print *, decryptOutput

    print *, ''
    print *, 'Solve tests:'
    ! Test absolute value
    call solve('HAL', 3, -26)
    print *, ''
    ! Test modulus
    call solve('HAL', 3, 30)

contains

    ! Subroutine to encrypt a string of size stringLength by shiftAmount and store the result in outputString
    subroutine encrypt(inputString, stringLength, shiftAmount, outputString)
        implicit none

        ! stringLength is needed so the type of inputString and outputString can be explicitly defined by their length
        ! shiftAmount is the amount to shift the string by
        integer, intent(in) :: stringLength, shiftAmount

        ! inputString will not be modified and is a character type of size stringLength
```

```fortran
        character(stringLength), intent(in) :: inputString
        ! outputString will be modified and has the same length as inputString
        character(stringLength) :: outputString

        ! newShift will be used to make sure the shift amount is between -25 and 25
        ! index will be used to iterate through the string character by character
        ! charValue is the ASCII value of a given character
        ! diff will be used to determine if the wraparound calculation is needed
        integer :: newShift, index, charValue, diff

        ! Start off by setting outputString to the same value is inputString
        outputString = inputString

        ! Convert the shift amount to be between -25 and 25
        newShift = mod(shiftAmount, 26)

        ! Iterate through the string one character at a time
        ! Looping and modifying individual characters found at the link below
        ! http://computer-programming-forum.com/49-fortran/4075a24f74fcc9ce.htm
        do index = 1, len(outputString)
            ! Get the original character value at the given position
            charValue = ichar(outputString(index:index))

            ! Check to see if it is a lowercase letter ('a' - 'z')
            if (charValue >= 97 .and. charValue <= 122) then
                ! If so, make it capital by subtracting 32 ('a' - 'A')
                charValue = charValue - 32
            endif

            ! Make sure the character is a letter ('A' - 'Z') because already converted to uppercase
            if (charValue >= 65 .and. charValue <= 90) then
                ! Perform the shift by the given amount
                charValue = charValue + newShift

                ! diff is how much over the charater is relative to 'Z' (90)
                diff = charValue - 90

                ! If it went over, wraparound is needed
                if (diff > 0) then
                    ! Finish the wraparound by adding the diff to 'A' (65)
                    ! The -1 is needed because a diff of 1 means that the character is 1 beyond 'Z', which will be 'A'
                    charValue = 65 + diff - 1
                else
                    ! diff is how much under the charater is relative to 'A' (65)
                    diff = 65 - charValue

                    ! If it went under, wraparound is needed
                    if (diff > 0) then
                        ! Finish the wraparound by subtracting the diff from 'Z' (90)
                        ! The +1 is needed because a diff of 1 means that the character is 1 beyond 'A', which will be 'Z'
                        charValue = 90 - diff + 1
                    endif
                endif

                ! Update the output string at the index with the new character value
                ! If the character was not originally a letter, then no change is needed, which is why this is inside of the if block
                outputString(index:index) = char(charValue)
            endif
        enddo
    end subroutine encrypt

```

```fortran
! Subroutine to decrypt a string of size stringLength that has been encrypted by shiftAmount and store the result in outputString
subroutine decrypt(inputString, stringLength, shiftAmount, outputString)
    implicit none

    ! stringLength is needed so the type of inputString and outputString can be explicitly defined by their length
    ! shiftAmount is the amount to unshift by
    integer, intent(in) :: stringLength, shiftAmount

    ! inputString will not be modified and is a character type of size stringLength
    character(stringLength), intent(in) :: inputString
    ! outputString will be modified and has the same length as inputString
    character(stringLength) :: outputString

    ! newShift will be used to make sure the shift amount is between -25 and 25
    ! index will be used to iterate through the string character by character
    ! charValue is the ASCII value of a given character
    ! diff will be used to determine if the wraparound calculation is needed
    integer :: newShift, index, charValue, diff

    ! Start off by setting outputString to the same value is inputString
    outputString = inputString

    ! Convert the shift amount to be between -25 and 25
    newShift = mod(shiftAmount, 26)

    ! Iterate through the string one character at a time
    ! Looping and modifying individual characters found at the link below
    ! http://computer-programming-forum.com/49-fortran/4075a24f74fcc9ce.htm
    do index = 1, len(outputString)
        ! Get the original character value at the given position
        charValue = ichar(outputString(index:index))

        ! Check to see if it is a lowercase letter ('a' - 'z')
        if (charValue >= 97 .and. charValue <= 122) then
            ! If so, make it capital by subtracting 32 ('a' - 'A')
            charValue = charValue - 32
        endif

        ! Make sure the character is a letter ('A' - 'Z') because already converted to uppercase
        if (charValue >= 65 .and. charValue <= 90) then
            ! Perform the shift by the given amount
            charValue = charValue - newShift

            ! diff is how much over the charater is relative to 'Z' (90)
            diff = charValue - 90

            ! If it went over, wraparound is needed
            if (diff > 0) then
                ! Finish the wraparound by adding the diff to 'A' (65)
                ! The -1 is needed because a diff of 1 means that the character is 1 beyond 'Z', which will be 'A'
                charValue = 65 + diff - 1
            else
                ! diff is how much under the charater is relative to 'A' (65)
                diff = 65 - charValue

                ! If it went under, wraparound is needed
                if (diff > 0) then
                    ! Finish the wraparound by subtracting the diff from 'Z' (90)
                    ! The +1 is needed because a diff of 1 means that the character is 1 beyond 'A', which will be 'Z'
                    charValue = 90 - diff + 1
```

```fortran
                    endif
                endif

                ! Update the output string at the index with the new character value
                ! If the character was not originally a letter, then no change is needed, which is why this is inside of the if block
                outputString(index:index) = char(charValue)
            endif
        enddo
    end subroutine decrypt

    ! Subroutine to solve a caesar cipher of the input string of size stringLength with a set number of tries (maxShiftValue)
    subroutine solve(inputString, stringLength, maxShiftValue)
        implicit none

        ! stringLength is needed so the type of inputString and outputString can be explicitly defined by their length
        ! shiftAmount is the amount to unshift by
        integer, intent(in) :: stringLength, maxShiftValue

        ! inputString will not be modified and is a character type of size stringLength
        character(stringLength), intent(in) :: inputString

        ! outputString is a temporary variable that will be used as the result from each solve attempt
        character(stringLength) :: outputString

        ! shiftAmount is the current amount being shifted
        integer :: shiftAmount

        ! Make sure 0 <= shiftAmount <= 26
        shiftAmount = abs(maxShiftValue)
        if (shiftAmount > 26) then
            shiftAmount = mod(shiftAmount, 26)
        endif

        ! Try all shift amounts from the max down to 0
        do shiftAmount = shiftAmount, 0, -1
            ! Call the decrypt subroutine with the current shiftAmount and store in outputString
            call decrypt(inputString, stringLength, -shiftAmount, outputString)

            ! Print out the results
            ! Formatting strings found at the link below
            ! https://pages.mtu.edu/~shene/COURSES/cs201/NOTES/chap05/format.html
            print '(a,I0,a,a)', 'Caesar␣', shiftAmount, ':␣', outputString
        enddo
    end subroutine solve
end program caesar
```

Listing 1: Caesar Cipher (Fortran)

```
> gfortran caesar.f90 -o caesar
> ./caesar
 Alan tests:
 BPQA QA I BMAB ABZQVO NZWU ITIV
 THIS IS A TEST STRING FROM ALAN
Caesar 26: HAL
Caesar 25: GZK
Caesar 24: FYJ
Caesar 23: EXI
Caesar 22: DWH
Caesar 21: CVG
Caesar 20: BUF
```

```
Caesar 19: ATE
Caesar 18: ZSD
Caesar 17: YRC
Caesar 16: XQB
Caesar 15: WPA
Caesar 14: VOZ
Caesar 13: UNY
Caesar 12: TMX
Caesar 11: SLW
Caesar 10: RKV
Caesar 9: QJU
Caesar 8: PIT
Caesar 7: OHS
Caesar 6: NGR
Caesar 5: MFQ
Caesar 4: LEP
Caesar 3: KDO
Caesar 2: JCN
Caesar 1: IBM
Caesar 0: HAL

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


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

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

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

Listing 2: Fortran Output

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

```
000100 Identification division.
******  This function encrypts a string based on the given shift amount
000101   Function-id. encrypt.
000102 Data division.
000103   Working-storage section.
******     Represents the current character being analyzed
000104     1 curChar pic 999.
******     Represents the index of the string being worked on
000105     1 idx pic 99.
******     Represents the difference to determine wraparound
000106     1 diff pic S9(2).
******     Shift value between -25 and 25
000107     1 newShift pic S9(2).
000108   Linkage section.
******     The input string
000109     1 inStr pic x(32).
******     The amount to shift by
000110     1 shiftAmt pic S999.
******     The shifted string
000111     1 res pic x(32).
000112 Procedure division
000113   using by reference inStr shiftAmt
000114   returning res.
******   Begin by copying the input to the result and set the index to 1
000115   Move inStr to res
000116   move 1 to idx
******   Adjust the shift amount to be in the range -25 to 25
000117   compute newShift = function mod(shiftAmt 26)
******   Repeat the work for all characters in the string
000118   perform encrypt-work until idx > function length(inStr)
000119   goback.
000120   encrypt-work.
******     Get the current character's ordinal value (ASCII + 1)
000121     compute curChar = function ord(res(idx:1))
******     Convert to uppercase if between 'a' (98) and 'z' (123)
000122     if curChar >= 98 and curChar <= 123 then
000123       compute curChar = curChar - 32
000124     end-if
```

```
*******    Only need to modify the character if it is a letter ('A' (66) to 'Z' (91))
000125     if curChar >= 66 and curChar <= 91 then
*******      Perform the shift
000126       compute curChar = curChar + newShift
*******      Check for wraparound on the 'Z' end
000127       compute diff = curChar - 91
000128       if diff > 0 then
*******        Do the wraparound. -1 at the end because diff of 1 means it should be 'A'
000129         compute curChar = 66 + diff - 1
000130       else
*******        Check wraparound on the 'A' end
000131         compute diff = 66 - curChar
000132         if (diff > 0) then
*******        Do the wraparound. +1 at the end because diff of 1 means it should be 'Z'
000133           compute curChar = 91 - diff + 1
000134         end-if
000135       end-if
*******      Update the character in the result string
000136       move function char(curChar) to res(idx:1)
000137     end-if
000138     add 1 to idx.
000139 End function encrypt.
000200 Identification division.
*******  This function decrypts a string based on the given shift amount
000201   Function-id. decrypt.
000202 Data division.
000203   Working-storage section.
*******    Represents the current character being analyzed
000204     1 curChar pic 999.
*******    Represents the index of the string being worked on
000205     1 idx pic 99.
*******    Represents the difference to determine wraparound
000206     1 diff pic S9(2).
*******    Shift value between -25 and 25
000207     1 newShift pic S9(2).
000208   Linkage section.
*******    The input string
000209     1 inStr pic x(32).
*******    The amount to shift by
000210     1 shiftAmt pic S999.
*******    The shifted string
000211     1 res pic x(32).
000212 Procedure division
000213   using by reference inStr shiftAmt
000214   returning res.
*******  Begin by copying the input to the result and set the index to 1
000215   Move inStr to res
000216   move 1 to idx
*******  Adjust the shift amount to be in the range -25 to 25
000217   compute newShift = function mod(shiftAmt 26)
*******  Repeat the work for all characters in the string
000218   perform decrypt-work until idx > function length(inStr)
000219   goback.
000220   decrypt-work.
*******    Get the current character's ordinal value (ASCII + 1)
000221     compute curChar = function ord(res(idx:1))
*******    Convert to uppercase if between 'a' (98) and 'z' (123)
000222     if curChar >= 98 and curChar <= 123 then
000223       compute curChar = curChar - 32
000224     end-if
*******    Only need to modify the character if it is a letter ('A' (66) to 'Z' (91))
000225     if curChar >= 66 and curChar <= 91 then
*******      Perform the shift
000226       compute curChar = curChar - newShift
*******      Check for wraparound on the 'Z' end
```

```
000227       compute diff = curChar - 91
000228       if diff > 0 then
*******        Do the wraparound. -1 at the end because diff of 1 means it should be 'A'
000229         compute curChar = 66 + diff - 1
000230       else
*******        Check wraparound on the 'A' end
000231         compute diff = 66 - curChar
000232         if (diff > 0) then
*******        Do the wraparound. +1 at the end because diff of 1 means it should be 'Z'
000233           compute curChar = 91 - diff + 1
000234         end-if
000235       end-if
*******      Update the character in the result string
000236       move function char(curChar) to res(idx:1)
000237     end-if
000238     add 1 to idx.
000239 End function decrypt.
000300 Identification division.
*******  Function to try to break a Caesar cipher
000301   Function-id. solve.
000302 Environment division.
000303   Configuration section.
000304     Repository.
*******      Have to import the decrypt function
000305       Function decrypt.
000306 Data division.
000307   Working-storage section.
*******    The current amount to shift by
000308     1 shiftAmt pic S999.
*******    Negated shiftAmt
000309     1 realShiftAmt pic S999.
*******    The result string for each call to decrypt
000310     1 outputStr pic x(32).
000311   Linkage section.
*******    The input string
000312     1 inStr pic x(32).
*******    The max shift amount to try
000313     1 maxShiftAmt pic S999.
*******    Have to return something, so will return 0
000314     1 res pic 9.
000315 Procedure division
000316   using by reference inStr maxShiftAmt
000317   returning res.
000318   Move 0 to res
*******  Get the absolute value for the shift amount to make sure it is positive
*******  In the end, -26 will be the same as +26, and 27 will be the same as 1
000319   move function abs(maxShiftAmt) to shiftAmt
000320   if (shiftAmt > 26) then
000320     compute shiftAmt = function mod(shiftAmt 26)
000321   end-if
*******  Repeat for all possible shift amounts
000322   perform solve-work until shiftAmt < 0
000323   goback.
000324   solve-work.
*******    Negate the shift amount
000325     compute realShiftAmt = shiftAmt * -1
*******    Try to decrypt the string and display the result
000326     move function decrypt(inStr realShiftAmt) to outputStr
000327     display "Caesar " shiftAmt ": " outputStr
000328     subtract 1 from shiftAmt.
000329 End function solve.
000000 Identification division.
000001   Program-id. caesar.
000002 Environment division.
000003   Configuration section.
```

```
000004      Repository.
000005        Function encrypt
000006        function decrypt
000007        function solve.
000008 Data division.
000009   Working-storage section.
000010     1 inStr pic x(32) value "This is a test string from Alan".
000011     1 shift pic S999 value 8.
000012     1 encryptRes pic x(32).
000013     1 decryptRes pic x(32).
000014     1 solveStr pic x(32) value "HAL".
000015     1 solveShift pic S999 value 26.
000016     1 emptyStr pic x(32) value " ".
000017     1 noLetters pic x(32) value "1234567890!@#$%^&*(){}".
000018 Procedure division.
000019   display "Alan tests:"
000020   Move encrypt(inStr shift) to encryptRes
000021   display encryptRes
000022   move decrypt(encryptRes shift) to decryptRes
000023   display decryptRes
000024   display solve(solveStr solveShift)
000025   display " "
000026   display "Encrypt and decrypt tests:"
*******  Test negative shifts
000027   move -1 to shift
000028   Move encrypt(inStr shift) to encryptRes
000029   display encryptRes
000030   move decrypt(encryptRes shift) to decryptRes
000031   display decryptRes
*******  Test modulus
000032   move 27 to shift
000033   Move encrypt(inStr shift) to encryptRes
000034   display encryptRes
000035   move decrypt(encryptRes shift) to decryptRes
000036   display decryptRes
*******  Test empty string
000037   move 7 to shift
000038   Move encrypt(emptyStr shift) to encryptRes
000039   display encryptRes
000040   move decrypt(encryptRes shift) to decryptRes
000041   display decryptRes
*******  Test no letters
000042   Move encrypt(noLetters shift) to encryptRes
000043   display encryptRes
000044   move decrypt(encryptRes shift) to decryptRes
000045   display decryptRes
000046   display " "
000047   display "Solve tests:"
*******  Test absolute value
000048   move -26 to solveShift
000049   display solve(solveStr solveShift)
000050   display " "
*******  Test modulus
000051   move 30 to solveShift
000052   display solve(solveStr solveShift)
000053   goback.
000054 End program caesar.
```

Listing 3: Caesar Cipher (COBOL)

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

```
Caesar +026: HAL
Caesar +025: GZK
Caesar +024: FYJ
Caesar +023: EXI
Caesar +022: DWH
Caesar +021: CVG
Caesar +020: BUF
Caesar +019: ATE
Caesar +018: ZSD
Caesar +017: YRC
Caesar +016: XQB
Caesar +015: WPA
Caesar +014: VOZ
Caesar +013: UNY
Caesar +012: TMX
Caesar +011: SLW
Caesar +010: RKV
Caesar +009: QJU
Caesar +008: PIT
Caesar +007: OHS
Caesar +006: NGR
Caesar +005: MFQ
Caesar +004: LEP
Caesar +003: KDO
Caesar +002: JCN
Caesar +001: IBM
Caesar +000: HAL
0

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


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

Solve tests:
Caesar +026: HAL
Caesar +025: GZK
Caesar +024: FYJ
Caesar +023: EXI
Caesar +022: DWH
Caesar +021: CVG
Caesar +020: BUF
Caesar +019: ATE
Caesar +018: ZSD
Caesar +017: YRC
Caesar +016: XQB
Caesar +015: WPA
Caesar +014: VOZ
Caesar +013: UNY
Caesar +012: TMX
Caesar +011: SLW
Caesar +010: RKV
Caesar +009: QJU
Caesar +008: PIT
Caesar +007: OHS
Caesar +006: NGR
Caesar +005: MFQ
Caesar +004: LEP
Caesar +003: KDO
Caesar +002: JCN
```

```
Caesar +001: IBM
Caesar +000: HAL
0

Caesar +004: LEP
Caesar +003: KDO
Caesar +002: JCN
Caesar +001: IBM
Caesar +000: HAL
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 (https://en.wikibooks.org/wiki/BASIC_Programming)
- chipmunk basic
- https://www.youtube.com/watch?v=7r83N3c2kPw
- mid$ basic

2.3.3 Code and Tests

```
9 rem Define the strings for the encryption input and output
10 encryptinput$ = "This is a test string from Alan"
20 encryptoutput$ = ""
29 rem Define the strings for the decrytion input and output
30 decryptinput$ = ""
40 decryptoutput$ = ""
49 rem Define the shift amount for encrypt and decrypt
50 shiftamount% = 8
60 print "Alan tests:"
69 rem Call the encrypt subroutine and print the output
70 gosub 1000
80 print encryptoutput$
89 rem Set the decrypt input to be what the encrypt output was
90 decryptinput$ = encryptoutput$
99 rem Call decrypt with the same shift amount as encrypt
100 gosub 2000
110 print decryptoutput$
119 rem Set the solve variables for the string and the amount
120 solveInput$ = "HAL"
130 shiftAmount% = 26
139 rem Call solve with the given variables
140 gosub 3000
150 print ""
160 print "Encrypt and decrypt tests:"
169 rem Test the negative shift amount
170 shiftAmount% = -1
180 gosub 1000
190 print encryptoutput$
200 decryptinput$ = encryptoutput$
210 gosub 2000
220 print decryptoutput$
229 rem Test modulus
230 shiftAmount% = 27
240 gosub 1000
250 print encryptoutput$
260 decryptinput$ = encryptoutput$
270 gosub 2000
280 print decryptoutput$
289 rem Test empty string
290 encryptinput$ = ""
300 shiftAmount% = 7
310 gosub 1000
320 print encryptoutput$
330 decryptinput$ = encryptoutput$
340 gosub 2000
```

```
350 print decryptoutput$
359 rem Test no letters
360 encryptinput$ = "1234567890!@#$%^&*(){}"
370 gosub 1000
380 print encryptoutput$
390 decryptinput$ = encryptoutput$
400 gosub 2000
410 print decryptoutput$
420 print ""
430 print "Solve tests:"
439 rem Test the absolute value
440 shiftAmount% = -26
450 gosub 3000
460 print ""
469 rem Test modulus
470 shiftAmount% = 30
480 gosub 3000
999 end
1000 rem Encrypt subroutine
1009 rem Encrypt output starts off as an uppercase version of the input
1010 encryptoutput$ = ucase$(encryptinput$)
1019 rem Shift amount should be between -25 and 25
1020 realShift% = shiftamount% mod 26
1029 rem Loop through every character in the string
1030 for index% = 1 to len(encryptoutput$)
1039 rem Get the ascii code
1040 char% = asc(mid$(encryptoutput$, index%, 1))
1050 if char% >= 65 and char% <= 90 then
1059 rem Do the shift if it is a letter
1060 char% = char% + realShift%
1069 rem Check to see if there is wraparound on the 'Z' end
1070 diff% = char% - 90
1080 if diff% > 0 then
1089 rem Wrap the character around to the 'A' side
1090 char% = 65 + diff% - 1
1100 else
1109 rem Check to see if there is wraparound on the 'A' end
1110 diff% = 65 - char%
1120 if diff% > 0 then
1129 rem Wrap the character around to the 'Z' side
1130 char% = 90 - diff% + 1
1140 endif
1150 endif
1159 rem Only have to replace the character if it is a letter
1160 mid$(encryptoutput$, index%, 1) = chr$(char%)
1170 endif
1180 next index%
1190 return
2000 rem Decrypt subroutine
2009 rem Decrypt output starts off as an uppercase version of the input
2010 decryptoutput$ = ucase$(decryptinput$)
2019 rem Shift amount should be between -25 and 25
2020 realShift% = shiftamount% mod 26
2029 rem Loop through every character in the string
2030 for index% = 1 to len(decryptoutput$)
2039 rem Get the ascii code
2040 char% = asc(mid$(decryptoutput$, index%, 1))
2050 if char% >= 65 and char% <= 90 then
2059 rem Do the shift if it is a letter
2060 char% = char% - realShift%
2069 rem Check to see if there is wraparound on the 'Z' end
2070 diff% = char% - 90
2080 if diff% > 0 then
2089 rem Wrap the character around to the 'A' side
2090 char% = 65 + diff% - 1
```

```
2100 else
2109 rem Check to see if there is wraparound on the 'A' end
2110 diff% = 65 - char%
2120 if diff% > 0 then
2129 rem Wrap the character around to the 'Z' side
2130 char% = 90 - diff% + 1
2140 endif
2150 endif
2159 rem Only have to replace the character if it is a letter
2160 mid$(decryptoutput$, index%, 1) = chr$(char%)
2170 endif
2180 next index%
2190 return
3000 rem Solve subroutine
3009 rem Only need positive shift amounts
3010 realShift% = abs(shiftAmount%)
3020 if realShift% > 26 then
3029 rem Take the mod of the shift is greater than 26
3030 realShift% = realShift% mod 26
3040 endif
3049 rem Decrypt takes in the solve input string
3050 decryptinput$ = solveInput$
3060 for shift% = realShift% to 0 step -1
3069 rem Update the shift amount according to the current step in the loop
3070 shiftAmount% = -shift%
3079 rem Call decrypt and print the result
3080 gosub 2000
3090 print "Caesar " shift% ": " decryptoutput$
3100 next shift%
3110 return
```

Listing 5: Caesar Cipher (BASIC)

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

```
Caesar 1 : IBM
Caesar 0 : HAL

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


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

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

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

Listing 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 (https://wiki.freepascal.org/Basic_Pascal_Tutorial)
- pascal function multiple parameters
- and operator pascal boolean expression

2.4.3 Code and Tests

```
program Caesar;
(* Function that encrypts the input string by the shift amount *)
function Encrypt(inStr : string; shiftAmount : integer) : string;
var
    curChar: Char; (* The current character being worked with *)
    charValue: integer; (* The ASCII value of the current character *)
    diff: integer; (* The difference between the current character and A and Z to detect
        wraparound *)
    idx: integer; (* The position in the string we are at *)
begin
    (* Start off by normalizing the shift amount *)
    shiftAmount := shiftAmount mod 26;
    Encrypt := inStr;

    (* Pascal uses 1-based indexing *)
    idx := 1;
    (* Loop through each character in the string *)
    for curChar in Encrypt do
    begin
        (* Get the initial character ASCII code *)
        charValue := ord(curChar);

```

```
        (* Convert the character to uppercase if it is not already *)
        if ((charValue >= 97) and (charValue <= 122)) then
            charValue := charValue - 32;

        (* Caesar cipher only impacts letters, so check that conditon first *)
        if ((charValue >= 65) and (charValue <= 90)) then
        begin
            (* Perform the shift *)
            charValue := charValue + shiftAmount;

            (* Check wraparound on the Z end *)
            diff := charValue - 90;
            if diff > 0 then
                (* Perform the wraparound (-1 is needed because 91 should be 65 (A)) *)
                charValue := 65 + diff - 1
            else
            begin
                (* Check the low end of the range *)
                diff := 65 - charValue;
                if diff > 0 then
                    (* Perform the wraparound (+1 is needed because 64 should be 90 (Z)) *)
                    charValue := 90 - diff + 1;
            end;
            (* Update the string accordingly *)
            Encrypt[idx] := chr(charValue);
        end;
        (* Increment the index for our own reference *)
        idx := idx + 1;
    end;
end;

(* Function that decrypts the input string by the shift amount *)
function Decrypt(inStr : string; shiftAmount : integer) : string;
var
    curChar: Char; (* The current character being worked with *)
    charValue: integer; (* The ASCII value of the current character *)
    diff: integer; (* The difference between the current character and A and Z to detect wraparound *)
    idx: integer; (* The position in the string we are at *)
begin
    (* Start off by normalizing the shift amount *)
    shiftAmount := shiftAmount mod 26;
    Decrypt := inStr;

    (* Pascal uses 1-based indexing *)
    idx := 1;
    (* Loop through each character in the string *)
    for curChar in Decrypt do
    begin
        (* Get the initial character ASCII code *)
        charValue := ord(curChar);

        (* Convert the character to uppercase if it is not already *)
        if ((charValue >= 97) and (charValue <= 122)) then
            charValue := charValue - 32;

        (* Caesar cipher only impacts letters, so check that conditon first *)
        if ((charValue >= 65) and (charValue <= 90)) then
        begin
            (* Perform the shift *)
            charValue := charValue - shiftAmount;

            (* Check wraparound on the Z end *)
            diff := charValue - 90;
            if diff > 0 then
```

```
                (* Perform the wraparound (-1 is needed because 91 should be 65 (A)) *)
                charValue := 65 + diff - 1
            else
            begin
                (* Check the low end of the range *)
                diff := 65 - charValue;
                if diff > 0 then
                    (* Perform the wraparound (+1 is needed because 64 should be 90 (Z)) *)
                    charValue := 90 - diff + 1;
            end;
            (* Update the string accordingly *)
            Decrypt[idx] := chr(charValue);
        end;
        (* Increment the index for our own reference *)
        idx := idx + 1;
    end;
end;

(* Procedure to solve a Caesar cipher *)
procedure solve(inputString : string; maxShiftAmount : integer);
var
    curShift: integer; (* The current shift amount  *)
begin
    (* Normalize the max shift amount being used *)
    maxShiftAmount := abs(maxShiftAmount);
    if maxShiftAmount > 26 then
        maxShiftAmount := maxShiftAmount mod 26;

    (* Loop through everything *)
    for curShift := maxShiftAmount downto 0 do
        (* Perform the decrypt and print the result *)
        writeln('Caesar ', curShift, ': ', Decrypt(inputString, -curShift));
end;

var
    originalString: string; (* The original string *)
    encryptOut: string; (* The output of encrypt *)
    decryptOut: string; (* The output of decrypt *)
begin
    originalString := 'This is a test string from Alan';
    writeln('Alan tests:');
    encryptOut := Encrypt(originalString, 8);
    decryptOut := Decrypt(encryptOut, 8);
    writeln(encryptOut);
    writeln(decryptOut);
    solve('HAL', 26);

    writeln();
    writeln('Encrypt and decrypt tests:');
    (* Test negative shift *)
    encryptOut := Encrypt(originalString, -1);
    decryptOut := Decrypt(encryptOut, -1);
    writeln(encryptOut);
    writeln(decryptOut);
    (* Test modulus *)
    encryptOut := Encrypt(originalString, 27);
    decryptOut := Decrypt(encryptOut, 27);
    writeln(encryptOut);
    writeln(decryptOut);
    (* Test empty string *)
    encryptOut := Encrypt('', 7);
    decryptOut := Decrypt(encryptOut, 7);
    writeln(encryptOut);
    writeln(decryptOut);
    (* Test no letters *)
```

```
    encryptOut := Encrypt('1234567890!@#$%^&*(){}', 7);
    decryptOut := Decrypt(encryptOut, 7);
    writeln(encryptOut);
    writeln(decryptOut);

    writeln();
    writeln('Solve tests:');
    (* Test absolute value *)
    solve('HAL', -26);
    writeln();
    (* Test modulus *)
    solve('HAL', 30);
end.
```

Listing 7: Caesar Cipher (Pascal)

```
> fpc caesar.pas
Free Pascal Compiler version 3.2.2 [2022/11/29] for x86_64
Copyright (c) 1993-2021 by Florian Klaempfl and others
Target OS: Darwin for x86_64
Compiling caesar.pas
Assembling caesar
Linking caesar
162 lines compiled, 0.3 sec
> ./caesar
Alan tests:
BPQA QA I BMAB ABZQVO NZWU ITIV
THIS IS A TEST STRING FROM ALAN
Caesar 26: HAL
Caesar 25: GZK
Caesar 24: FYJ
Caesar 23: EXI
Caesar 22: DWH
Caesar 21: CVG
Caesar 20: BUF
Caesar 19: ATE
Caesar 18: ZSD
Caesar 17: YRC
Caesar 16: XQB
Caesar 15: WPA
Caesar 14: VOZ
Caesar 13: UNY
Caesar 12: TMX
Caesar 11: SLW
Caesar 10: RKV
Caesar 9: QJU
Caesar 8: PIT
Caesar 7: OHS
Caesar 6: NGR
Caesar 5: MFQ
Caesar 4: LEP
Caesar 3: KDO
Caesar 2: JCN
Caesar 1: IBM
Caesar 0: HAL

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


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

```

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

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

Listing 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 (https://docs.scala-lang.org/overviews/scala-book/introduction.html)
- scala loop through a string
- scala string object
- absolute value scala
- scala convert int to char

2.5.2 Code and Tests

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

        // String builder to put together the final string without copying the data
        var outputBuilder: StringBuilder = new StringBuilder()

        // Loop through the input string
        for (i <- 0 until inputStr.length) {
            // Get the current character ASCII value
            var charVal: Int = inputStr.charAt(i)

            // Convert to uppercase
            if (charVal >= 97 && charVal <= 122) {
                charVal -= 32
            }

            // Only have to do work if it is uppercase ('A' - 'Z')
            if (charVal >= 65 && charVal <= 90) {
                // Compute the shift
                charVal += realShift

                // Check for wraparound on the 'Z' end
                var diff: Int = charVal - 90
                if (diff > 0) {
                    // Perform the wraparound
                    charVal = 65 + diff - 1
                } else {
                    // Check for wraparound on the 'A' end
                    diff = 65 - charVal
                    if (diff > 0) {
                        // Perform the wraparound
```

```
                    charVal = 90 - diff + 1
                }
            }
        }
        // Add the character to the output
        outputBuilder.append(charVal.toChar)
    }

    // Return the string
    return outputBuilder.toString()
}

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

    // String builder to put together the final string without copying the data
    var outputBuilder: StringBuilder = new StringBuilder()

    // Loop through the input string
    for (i <- 0 until inputStr.length) {
        // Get the current character ASCII value
        var charVal: Int = inputStr.charAt(i)

        // Convert to uppercase
        if (charVal >= 97 && charVal <= 122) {
            charVal -= 32
        }

        // Only have to do work if it is uppercase ('A' - 'Z')
        if (charVal >= 65 && charVal <= 90) {
            // Compute the shift
            charVal -= realShift

            // Check for wraparound on the 'Z' end
            var diff: Int = charVal - 90
            if (diff > 0) {
                // Perform the wraparound
                charVal = 65 + diff - 1
            } else {
                // Check for wraparound on the 'A' end
                diff = 65 - charVal
                if (diff > 0) {
                    // Perform the wraparound
                    charVal = 90 - diff + 1
                }
            }
        }
        // Add the character to the output
        outputBuilder.append(charVal.toChar)
    }

    // Return the string
    return outputBuilder.toString()
}

// Function to solve a Caesar cipher
def solve(inputStr: String, maxShiftAmount: Int) = {
    // Normalize the max shift to be between 0 and 26
    var realMaxShiftAmount = maxShiftAmount.abs
    if (realMaxShiftAmount > 26) {
        realMaxShiftAmount = realMaxShiftAmount % 26
    }

```

```scala
        // Loop through each shift amount
        for (curShift <- realMaxShiftAmount to 0 by -1) {
            // Print the result of running decrypt on the string
            println(s"Caesar ${curShift}: ${decrypt(inputStr, -curShift)}")
        }
    }

    def main(args: Array[String]) = {
        println("Alan tests:")
        var encryptOut: String = encrypt("This is a test string from Alan", 8)
        println(encryptOut)
        var decryptOut: String = decrypt(encryptOut, 8)
        println(decryptOut)
        solve("HAL", 26)

        println("\nEncrypt and decrypt tests:")
        // Test negative shift amount
        encryptOut = encrypt("This is a test string from Alan", -1)
        println(encryptOut)
        decryptOut = decrypt(encryptOut, -1)
        println(decryptOut)

        // Test modulus
        encryptOut = encrypt("This is a test string from Alan", 27)
        println(encryptOut)
        decryptOut = decrypt(encryptOut, 27)
        println(decryptOut)

        // Test empty string
        encryptOut = encrypt("", 27)
        println(encryptOut)
        decryptOut = decrypt(encryptOut, 27)
        println(decryptOut)

        // Test no letters
        encryptOut = encrypt("1234567890!@#$%^&*(){}", 27)
        println(encryptOut)
        decryptOut = decrypt(encryptOut, 27)
        println(decryptOut)

        println("\nSolve tests:")
        // Test absolute value
        solve("HAL", -26)
        println()
        // Test modulus
        solve("HAL", 30)

    }
}
```

Listing 9: Caesar Cipher (Scala)

```
> scalac caesar.scala -explain
> scala Caesar
Alan tests:
BPQA QA I BMAB ABZQVO NZWU ITIV
THIS IS A TEST STRING FROM ALAN
Caesar 26: HAL
Caesar 25: GZK
Caesar 24: FYJ
Caesar 23: EXI
Caesar 22: DWH
Caesar 21: CVG
Caesar 20: BUF
Caesar 19: ATE
```

```
Caesar 18: ZSD
Caesar 17: YRC
Caesar 16: XQB
Caesar 15: WPA
Caesar 14: VOZ
Caesar 13: UNY
Caesar 12: TMX
Caesar 11: SLW
Caesar 10: RKV
Caesar 9: QJU
Caesar 8: PIT
Caesar 7: OHS
Caesar 6: NGR
Caesar 5: MFQ
Caesar 4: LEP
Caesar 3: KDO
Caesar 2: JCN
Caesar 1: IBM
Caesar 0: HAL

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


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

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

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

Listing 10: Scala Output

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
- 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

```
.section .text
.global _start

_start:
    nop     # Needed for gdb to work properly

    # Alan encrypt and decrypt tests
    la      a0, alan_test
    li      a1, 31
    li      a2, 8
    call    encrypt

    li  a7, 64
    li  a0, 1
    la  a1, alan_test
    li  a2, 32
    ecall

    la      a0, alan_test
    li      a1, 31
    li      a2, 8
    call    decrypt

    li  a7, 64
```

```
    li   a0, 1
    la   a1, alan_test
    li   a2, 32
    ecall

    # Alan solve test
    la      a0, hal
    li      a1, 3
    li      a2, 26
    call    solve

    li   a7, 64
    li   a0, 1
    la   a1, empty_string
    li   a2, 1
    ecall

    # Test negative shift amounts
    la      a0, alan_test
    li      a1, 31
    li      a2, -1
    call    encrypt

    li   a7, 64
    li   a0, 1
    la   a1, alan_test
    li   a2, 32
    ecall

    la      a0, alan_test
    li      a1, 31
    li      a2, -1
    call    decrypt

    li   a7, 64
    li   a0, 1
    la   a1, alan_test
    li   a2, 32
    ecall

    # Test modulus with encrypt and decrypt
    la      a0, alan_test
    li      a1, 31
    li      a2, 27
    call    encrypt

    li   a7, 64
    li   a0, 1
    la   a1, alan_test
    li   a2, 32
    ecall

    la      a0, alan_test
    li      a1, 31
    li      a2, 27
    call    decrypt

    li   a7, 64
    li   a0, 1
    la   a1, alan_test
    li   a2, 32
    ecall

    # Test an empty string
    la      a0, empty_string
```

```
    li      a1, 0
    li      a2, 7
    call    encrypt

    li  a7, 64
    li  a0, 1
    la  a1, empty_string
    li  a2, 1
    ecall

    la      a0, empty_string
    li      a1, 0
    li      a2, 7
    call    decrypt

    li  a7, 64
    li  a0, 1
    la  a1, empty_string
    li  a2, 1
    ecall

    # Test with no letters (only numbers and symbols)
    la      a0, no_letters
    li      a1, 0
    li      a2, 22
    call    encrypt

    li  a7, 64
    li  a0, 1
    la  a1, no_letters
    li  a2, 23
    ecall

    la      a0, no_letters
    li      a1, 22
    li      a2, 7
    call    decrypt

    li  a7, 64
    li  a0, 1
    la  a1, no_letters
    li  a2, 23
    ecall

    li  a7, 64
    li  a0, 1
    la  a1, empty_string
    li  a2, 1
    ecall

    # Test abs with solve
    la      a0, hal
    li      a1, 3
    li      a2, -26
    call    solve

    li  a7, 64
    li  a0, 1
    la  a1, empty_string
    li  a2, 1
    ecall

    # Test modulus with solve
    la      a0, hal
    li      a1, 3
```

```
    li      a2, 30
    call    solve

    addi    a7, zero, 93 # exit sys call
    addi    a0, zero, 0 # exit status 0
    ecall

# a0 = address of the string to encrypt
# a1 = length of the string
# a2 = shift amount
encrypt:
    mv  t0, a0 # Copy string address into a temporary register
    li  t1, 0 # Conter variable for the character we are on
    mv  t2, a2 # Move the shift amount into a temporary register
    li  t6, 26 # Used for getting the mod of the shift amount

    # If shift > 26, then keep subtracting until < 26
    encrypt_positive_mod:
        blt t2, t6, encrypt_change_limit
        sub t2, t2, t6
        j   encrypt_positive_mod

    # Change to -26 for next check
    encrypt_change_limit:
        li t6, -26

    # If shift < -26, then keep subtracting -26 (add 26) until > -26
    encrypt_negative_mod:
        blt t6, t2, encrypt_loop
        sub t2, t2, t6
        j encrypt_negative_mod

    # The actual encryption loop
    encrypt_loop:
        lb  t3, 0(t0) # Get the current character
        li  t5, 97 # a
        li  t6, 122 # z

        # Convert to uppercase if needed
        blt t3, t5, encrypt_conversion
        blt t6, t3, encrypt_conversion
        addi    t3, t3, -32 # Convert to uppercase

        encrypt_conversion:
            li  t5, 65 # A
            li  t6, 90 # Z

            # Only continue if there is a letter
            blt t3, t5, encrypt_check_loop
            blt t6, t3, encrypt_check_loop
            add t3, t3, t2

            sub t4, t3, t6 # character - 90 (check for Z wraparound)
            bge zero, t4, encrypt_lower_wraparound
            # Char value is A (t5) + diff (t4) - 1
            add t3, t4, t5
            addi t3, t3, -1

            # Can go straight to storing as we do not need to worry about A wraparound
            j encrypt_store_char

            encrypt_lower_wraparound:
                sub t4, t5, t3 # 65 - char (check for A wraparound)
                bge zero, t4, encrypt_store_char
                # Char value is Z (t6) - diff (t4) + 1
```

```
                sub t3, t6, t4
                addi t3, t3, 1

            encrypt_store_char:
                sb  t3, 0(t0) # Store it in the output string

            encrypt_check_loop:
                addi t0, t0, 1 # Move to next byte
                addi t1, t1, 1 # Increment the counter
                ble t1, a1, encrypt_loop # Loop if there is still more to do
    ret

# a0 = address of the string to decrypt
# a1 = length of the string
# a2 = shift amount
decrypt:
    mv  t0, a0 # Copy string address into a temporary register
    li  t1, 0 # Conter variable for the character we are on
    mv  t2, a2 # Move the shift amount into a temporary register
    li  t6, 26 # Used for getting the mod of the shift amount

    # If shift > 26, then keep subtracting until < 26
    decrypt_positive_mod:
        blt t2, t6, decrypt_change_limit
        sub t2, t2, t6
        j   decrypt_positive_mod

    # Change to -26 for next check
    decrypt_change_limit:
        li t6, -26

    # If shift < -26, then keep subtracting -26 (add 26) until > -26
    decrypt_negative_mod:
        blt t6, t2, decrypt_loop
        sub t2, t2, t6
        j decrypt_negative_mod

    # The actual encryption loop
    decrypt_loop:
        lb  t3, 0(t0) # Get the current character
        li  t5, 97 # a
        li  t6, 122 # z

        # Convert to uppercase if needed
        blt t3, t5, decrypt_conversion
        blt t6, t3, decrypt_conversion
        addi    t3, t3, -32 # Convert to uppercase

        decrypt_conversion:
            li  t5, 65 # A
            li  t6, 90 # Z

            # Only continue if there is a letter
            blt t3, t5, decrypt_check_loop
            blt t6, t3, decrypt_check_loop
            sub t3, t3, t2

            sub t4, t3, t6 # character - 90 (check for Z wraparound)
            bge zero, t4, decrypt_lower_wraparound
            # Char value is A (t5) + diff (t4) - 1
            add t3, t4, t5
            addi t3, t3, -1

            # Can go straight to storing as we do not need to worry about A wraparound
            j decrypt_store_char
```

```
            decrypt_lower_wraparound:
                sub t4, t5, t3 # 65 - char (check for A wraparound)
                bge zero, t4, decrypt_store_char
                # Char value is Z (t6) - diff (t4) + 1
                sub t3, t6, t4
                addi t3, t3, 1

            decrypt_store_char:
                sb  t3, 0(t0) # Store it in the output string

            decrypt_check_loop:
                addi t0, t0, 1 # Move to next byte
                addi t1, t1, 1 # Increment the counter
                ble t1, a1, decrypt_loop # Loop if there is still more to do
    ret

# a0 = address of the string to solve
# a1 = string length
# a2 = max shift amount
solve:
    mv  t0, a2 # Store the current shift amount
    li  t1, 26 # 26 constant for use to take the modulus

    solve_abs:
        # Negate the value if needed to make it positive
        bge t0, zero, solve_mod
        neg t0, t0

    # Take the mod to make the max shift <= 26
    solve_mod:
        ble t0, t1, solve_loop
        sub t0, t0, t1
        j   solve_mod

    solve_loop:
        # Break when less than 0
        blt     t0, zero, solve_break

        # Negate the current shift amount for the decrypt function
        neg     a2, t0

        # Push the current shift amount and return address onto the stack
        addi    sp, sp, -8
        sw      ra, 4(sp)
        sw      t0, 0(sp)

        call    decrypt

        # Pop the current shift amount from the stack
        lw      t0, 0(sp)
        lw      ra, 4(sp)
        addi    sp, sp, 8

        # Print out the current shifted string
        li      a7, 64
        addi    a2, a1, 1
        mv      a1, a0
        li      a0, 1
        ecall

        # Reset the registers to work with decrypt
        mv      a0, a1
        addi    a1, a2, -1
        mv      a2, t0
```

```

        # We now have to call decrypt with the non-negated shift amount
        # because decrypt modifies the original string, so we need to fix that

        # Push the current shift amount and return address onto the stack
        addi    sp, sp, -8
        sw      ra, 4(sp)
        sw      t0, 0(sp)

        call    decrypt

        # Pop the current shift amount from the stack
        lw      t0, 0(sp)
        lw      ra, 4(sp)
        addi    sp, sp, 8

        # Decrement the shift amount
        addi    t0, t0, -1

        # Go back to the top
        j   solve_loop

    solve_break:
        ret

alan_test:
    .ascii "This is a test string from Alan\n"

hal:
    .ascii "HAL\n"

empty_string:
    .ascii "\n"

no_letters:
    .ascii "1234567890!@#$%^&*(){}\n"
```

Listing 11: Caesar Cipher (RISC-V Assembly)

```
> make
rm -f caesar.o
rm -f caesar
riscv64-unknown-elf-as -g -c caesar.s -o caesar.o
riscv64-unknown-elf-gcc -g caesar.o -o caesar -nostdlib -static
> make run
riscv64-unknown-elf-run caesar
BPQA QA I BMAB ABZQVO NZWU ITIV
THIS IS A TEST STRING FROM ALAN
HAL
GZK
FYJ
EXI
DWH
CVG
BUF
ATE
ZSD
YRC
XQB
WPA
VOZ
UNY
TMX
SLW
RKV
```

```
QJU
PIT
OHS
NGR
MFQ
LEP
KDO
JCN
IBM
HAL

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


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

HAL
GZK
FYJ
EXI
DWH
CVG
BUF
ATE
ZSD
YRC
XQB
WPA
VOZ
UNY
TMX
SLW
RKV
QJU
PIT
OHS
NGR
MFQ
LEP
KDO
JCN
IBM
HAL

LEP
KDO
JCN
IBM
HAL
```

Listing 12: RISC-V Assembly Output