Gianna Julio

CMPT331 - Theory of Programming Languages

Programming in the Past

Contents

1 Fortran

1.1 Program

```fortran
program caesar_cipher
    implicit none

    ! declare variables
    character(len=100) :: input
    integer :: shift, i

    ! prompt user for input text
    print *, 'Enter text to encrypt: '
    read(*, '(A)') input
    ! prompt user for shift value
    print *, 'Enter shift value: '
    read(*,*) shift

    call encrypt(input, shift)

    ! prompt user for input text
    print *, 'Enter text to decrypt: '
    read(*, '(A)') input

    ! prompt user for shift value
    print *, 'Enter shift value: '
    read(*,*) shift

    call decrypt(input, shift)

    ! prompt user for input text
    print *, 'Enter text for brute-force solve: '
    read(*, '(A)') input

    call solve(input)

    contains

    ! subroutine for encryption
    subroutine encrypt(input, shift)
        character(len=100) :: input     ! input text
        integer, intent(in) :: shift    ! shift value
        character(len=100) :: output    ! output encrypted text
        integer :: i, len_text, ascii   ! loop index, text length, and ASCII values

        len_text = len_trim(input)      ! get length
        output = input                  ! initialize output with input text

        ! loop through each character in the input text
        do i = 1, len_text
            ! get ASCII value and convert lowercase to uppercase
            ascii = ichar(input(i:i))
            if (ascii >= ichar('a') .and. ascii <= ichar('z')) then
                ascii = ascii - ichar('a') + ichar('A')
```

```
            end if
            ! encrypt uppercase letters
            if (ascii >= ichar('A') .and. ascii <= ichar('Z')) then
                ascii = mod(ascii - ichar('A') + shift, 26) + ichar('A')
            end if
            ! convert ASCII back to character and store
            output(i:i) = char(ascii)
        end do
        ! print output
        print *, 'Encrypted text: ', trim(output)
    end subroutine encrypt

    ! subroutine for decryption
    subroutine decrypt(input, shift)
        character(len=100) :: input     ! input text
        integer, intent(in) :: shift    ! shift value
        character(len=100) :: output    ! output decrypted text
        integer :: i, len_text, ascii   ! loop index, text length, and ASCII values

        len_text = len_trim(input)      ! get length
        output = input                  ! initialize output with encrypted text

        ! loop through each character in the text
        do i = 1, len_text
            ! get ASCII value and convert lowercase to uppercase
            ascii = ichar(input(i:i))
            if (ascii >= ichar('a') .and. ascii <= ichar('z')) then
                ascii = ascii - ichar('a') + ichar('A')
            end if
            ! decrypt uppercase letters
            if (ascii >= ichar('A') .and. ascii <= ichar('Z')) then
                ascii = mod(ascii - ichar('A') - shift + 26, 26) + ichar('A')
            end if
            ! convert ASCII back to character and store
            output(i:i) = char(ascii)
        end do
        ! print output
        print *, 'Decrypted text: ', trim(output)
    end subroutine decrypt

    ! subroutine to solve will all possible shifts
    subroutine solve(input)
        character(len=100):: input
        character(len=100) :: output
        integer :: i, j, len_text, ascii

        print *, 'Solving the cipher with all 26 possible shifts:'

        len_text = len_trim(input)

        ! loop through all possible shift values
        do i = 1, 26
            output = input  ! copy input to output

```

```
            ! perform manual decryption for shift value i
            do j = 1, len_text
                ascii = ichar(input(j:j))
                if (ascii >= ichar('a') .and. ascii <= ichar('z')) then
                    ascii = ascii - ichar('a') + ichar('A')
                end if
                if (ascii >= ichar('A') .and. ascii <= ichar('Z')) then
                    ascii = mod(ascii - ichar('A') - i + 26, 26) + ichar('A')
                end if
                output(j:j) = char(ascii)
            end do

            ! display decrypted text for this shift
            print *, 'Shift ', i, ': ', trim(output)
        end do
    end subroutine solve

end program caesar_cipher
```

1.2 Sample Output

```
Enter text to encrypt: Hello World!
Enter shift value: 5
Encrypted text: MJQQT BTWQI!

Enter text to decrypt: Vszzc Kcfzr!
Enter shift value: 14
Decrypted text: HELLO WORLD!

Enter text for brute-force solve: Ebiil Tloia!
Solving the cipher with all 26 possible shifts:
Shift            1 : DAHHK SKNHZ!
Shift            2 : CZGGJ RJMGY!
Shift            3 : BYFFI QILFX!
Shift            4 : AXEEH PHKEW!
Shift            5 : ZWDDG OGJDV!
Shift            6 : YVCCF NFICU!
Shift            7 : XUBBE MEHBT!
Shift            8 : WTAAD LDGAS!
Shift            9 : VSZZC KCFZR!
Shift           10 : URYYB JBEYQ!
Shift           11 : TQXXA IADXP!
Shift           12 : SPWWZ HZCWO!
Shift           13 : ROVVY GYBVN!
Shift           14 : QNUUX FXAUM!
Shift           15 : PMTTW EWZTL!
Shift           16 : OLSSV DVYSK!
Shift           17 : NKRRU CUXRJ!
Shift           18 : MJQQT BTWQI!
Shift           19 : LIPPS ASVPH!
Shift           20 : KHOOR ZRUOG!
Shift           21 : JGNNQ YQTNF!
Shift           22 : IFMMP XPSME!
```

```
Shift           23 : HELLO WORLD!
Shift           24 : GDKKN VNQKC!
Shift           25 : FCJJM UMPJB!
Shift           26 : EBIIL TLOIA!
```

1.3 Work Log

Predicted Time: 4 hours **Actual Time:** 5 hours **Ranking:** #3

- **Similarities/Differences:** Fortran felt more modern compared to some of the other older languages due to its use of subroutines and functions. However, its variable declaration rules were harder to grasp. The implicit typing rules based on the first character of the variable name seem illogical to me, especially since it is standard to begin a Fortran program with the `implicit none` statement.
- **Readability/Writability:** The syntax is structured similarly to modern languages, making it relatively readable once you understand the specifics. Writing wasn't too difficult, though formatting variable declarations and understanding the syntax of the `print` and `read` cost me some time debugging.
- **What I Loved:** The logical structure and use of functions felt intuitive, especially compared to languages like COBOL and BASIC.
- **What I Hated:** Variable declaration syntax was kind of a hassle and felt unintuitive compared to modern languages. Debugging took longer than expected because of the strict type rules.
- **Why the Time Discrepancy?** Some debugging setbacks with variable declarations and minor syntax mistakes added to the development time. I also had to do some quick research on the difference between subroutines and functions in Fortran, and eventually settled on using subroutines as it felt like more of an exploration of the language.
- **AI/Google Searches Used:**
  - "Fortran variable declaration rules"
  - "Fortran character handling and string operations"
  - "Fortran subroutines vs functions"
  - "Fortran print and read syntax"

2 COBOL

2.1 Program

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CaesarCipher.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 inText           PIC X(100).
       01 outText          PIC X(100).
       01 shift            PIC 99.
       01 decrypted        PIC X(100).
       01 i                PIC 99.
       01 j                PIC 99.
       01 encoded             PIC 99.
       01 decoded          PIC X.

       PROCEDURE DIVISION.

           DISPLAY "Enter text to encrypt: ".
           ACCEPT inText.
           DISPLAY "Enter shift value: ".
           ACCEPT shift.
           PERFORM ENCRYPT.

           DISPLAY "Enter text to decrypt: ".
           ACCEPT inText.
           DISPLAY "Enter shift value: ".
           ACCEPT shift.
           PERFORM DECRYPT.

           DISPLAY "Enter text for brute-force solve: ".
           ACCEPT inText.
           PERFORM BRUTE-FORCE.

           STOP RUN.

      * ENCRYPT subroutine to perform Caesar cipher encryption
       ENCRYPT.
           MOVE SPACES TO outText.
           PERFORM VARYING i FROM 1 BY 1 UNTIL i > LENGTH OF inText
               IF inText(i:1) >= "A" AND inText(i:1) <= "Z"
                   COMPUTE encoded = FUNCTION ORD(inText(i:1)) + shift
                   IF encoded > FUNCTION ORD("Z")
                       COMPUTE encoded = encoded - 26
                   END-IF
                   MOVE FUNCTION CHAR(encoded) TO outText(i:1)
               ELSE IF inText(i:1) >= "a" AND inText(i:1) <= "z"
                   COMPUTE encoded = FUNCTION ORD(inText(i:1)) + shift
                   IF encoded > FUNCTION ORD("z")
                       COMPUTE encoded = encoded - 26
                   END-IF
                   MOVE FUNCTION CHAR(encoded) TO outText(i:1)
```

```
                ELSE
                    MOVE inText(i:1) TO outText(i:1)
                END-IF
            END-PERFORM.
            DISPLAY "Encrypted String: " outText.
            EXIT.

       * DECRYPT subroutine to perform Caesar cipher decryption
        DECRYPT.
            MOVE SPACES TO outText.
            PERFORM VARYING i FROM 1 BY 1 UNTIL i > LENGTH OF inText
                IF inText(i:1) >= "A" AND inText(i:1) <= "Z"
                    COMPUTE encoded = FUNCTION ORD(inText(i:1)) - shift
                    IF encoded < FUNCTION ORD("A")
                        COMPUTE encoded = encoded + 26
                    END-IF
                    MOVE FUNCTION CHAR(encoded) TO outText(i:1)
                ELSE IF inText(i:1) >= "a" AND inText(i:1) <= "z"
                    COMPUTE encoded = FUNCTION ORD(inText(i:1)) - shift
                    IF encoded < FUNCTION ORD("a")
                        COMPUTE encoded = encoded + 26
                    END-IF
                    MOVE FUNCTION CHAR(encoded) TO outText(i:1)
                ELSE
                    MOVE inText(i:1) TO outText(i:1)
                END-IF
            END-PERFORM.
            DISPLAY "Decrypted String: " outText.
            EXIT.

       * BRUTE-FORCE subroutine to try all possible shift values (1-25)
        BRUTE-FORCE.
            PERFORM VARYING shift FROM 1 BY 1 UNTIL shift > 25
                MOVE SPACES TO decrypted
                PERFORM VARYING i FROM 1 BY 1 UNTIL i > LENGTH OF inText
                    IF inText(i:1) >= "A" AND inText(i:1) <= "Z"
                        COMPUTE encoded = FUNCTION ORD(inText(i:1))-shift
                        IF encoded < FUNCTION ORD("A")
                            COMPUTE encoded = encoded + 26
                        END-IF
                        MOVE FUNCTION CHAR(encoded) TO decrypted(i:1)
                    ELSE IF inText(i:1) >= "a" AND inText(i:1) <= "z"
                        COMPUTE encoded = FUNCTION ORD(inText(i:1))-shift
                        IF encoded < FUNCTION ORD("a")
                            COMPUTE encoded = encoded + 26
                        END-IF
                        MOVE FUNCTION CHAR(encoded) TO decrypted(i:1)
                    ELSE
                        MOVE inText(i:1) TO decrypted(i:1)
                    END-IF
                END-PERFORM.
                DISPLAY "Shift Value: "shift" Decrypted Text: " decrypted
            END-PERFORM.
            EXIT.
```

2.2 Sample Output

```
Enter text to encrypt: You go tell that vapid existentialist quack Freddy Nietzsche that he can just bite me, twice.
Enter shift value: 12
Encrypted text: KAG SA FQXX FTMF HMBUP QJUEFQZFUMXUEF CGMOW RDQPPK ZUQFLEOTQ FTMF TQ OMZ VGEF NUFQ YQ, FIUOQ.

Enter text to decrypt: DTZ LT YJQQ YMFY AFUNI JCNXYJSYNFQNXY VZFHP KWJIID SNJYEXHMJ YMFY MJ HFS OZXY GNYJ RJ, YBNHJ.
Enter shift value: 5
Decrypted text: YOU GO TELL THAT VAPID EXISTENTIALIST QUACK FREDDY NIETZSCHE THAT HE CAN JUST BITE ME, TWICE.

 Enter text for brute-force solve:
 Solving the cipher with all 26 possible shifts:
 Shift  1 : XNT FN SDKK SGZS UZOHC DWHRSDMSHZKHRS PTZBJ EQDCCX MHDSYRBGD SGZS GD BZM ITRS AHSD LD, SVHBD.
 Shift  2 : WMS EM RCJJ RFYR TYNGB CVGQRCLRGYJGQR OSYAI DPCBBW LGCRXQAFC RFYR FC AYL HSQR ZGRC KC, RUGAC.
 Shift  3 : VLR DL QBII QEXQ SXMFA BUFPQBKQFXIFPQ NRXZH COBAAV KFBQWPZEB QEXQ EB ZXK GRPQ YFQB JB, QTFZB.
 Shift  4 : UKQ CK PAHH PDWP RWLEZ ATEOPAJPEWHEOP MQWYG BNAZZU JEAPVOYDA PDWP DA YWJ FQOP XEPA IA, PSEYA.
 Shift  5 : TJP BJ OZGG OCVO QVKDY ZSDNOZIODVGDNO LPVXF AMZYYT IDZOUNXCZ OCVO CZ XVI EPNO WDOZ HZ, ORDXZ.
 Shift  6 : SIO AI NYFF NBUN PUJCX YRCMNYHNCUFCMN KOUWE ZLYXXS HCYNTMWBY NBUN BY WUH DOMN VCNY GY, NQCWY.
 Shift  7 : RHN ZH MXEE MATM OTIBW XQBLMXGMBTEBLM JNTVD YKXWWR GBXMSLVAX MATM AX VTG CNLM UBMX FX, MPBVX.
 Shift  8 : QGM YG LWDD LZSL NSHAV WPAKLWFLASDAKL IMSUC XJWVVQ FAWLRKUZW LZSL ZW USF BMKL TALW EW, LOAUW.
 Shift  9 : PFL XF KVCC KYRK MRGZU VOZJKVEKZRCZJK HLRTB WIVUUP EZVKQJTYV KYRK YV TRE ALJK SZKV DV, KNZTV.
 Shift  10 : OEK WE JUBB JXQJ LQFYT UNYIJUDJYQBYIJ GKQSA VHUTTO DYUJPISXU JXQJ XU SQD ZKIJ RYJU CU, JMYSU.
 Shift  11 : NDJ VD ITAA IWPI KPEXS TMXHITCIXPAXHI FJPRZ UGTSSN CXTIOHRWT IWPI WT RPC YJHI QXIT BT, ILXRT.
 Shift  12 : MCI UC HSZZ HVOH JODWR SLWGHSBHWOZWGH EIOQY TFSRRM BWSHNGQVS HVOH VS QOB XIGH PWHS AS, HKWQS.
 Shift  13 : LBH TB GRYY GUNG INCVQ RKVFGRAGVNYVFG DHNPX SERQQL AVRGMFPUR GUNG UR PNA WHFG OVGR ZR, GJVPR.
 Shift  14 : KAG SA FQXX FTMF HMBUP QJUEFQZFUMXUEF CGMOW RDQPPK ZUQFLEOTQ FTMF TQ OMZ VGEF NUFQ YQ, FIUOQ.
 Shift  15 : JZF RZ EPWW ESLE GLATO PITDEPYETLWTDE BFLNV QCPOOJ YTPEKDNSP ESLE SP NLY UFDE MTEP XP, EHTNP.
 Shift  16 : IYE QY DOVV DRKD FKZSN OHSCDOXDSKVSCD AEKMU PBONNI XSODJCMRO DRKD RO MKX TECD LSDO WO, DGSMO.
 Shift  17 : HXD PX CNUU CQJC EJYRM NGRBCNWCRJURBC ZDJLT OANMMH WRNCIBLQN CQJC QN LJW SDBC KRCN VN, CFRLN.
```

```
Shift  18 : GWC OW BMTT BPIB DIXQL MFQABMVBQITQAB YCIKS NZMLLG VQMBHAKPM BPIB PM KIV
    RCAB JQBM UM, BEQKM.
Shift  19 : FVB NV ALSS AOHA CHWPK LEPZALUAPHSPZA XBHJR MYLKKF UPLAGZJOL AOHA OL JHU
    QBZA IPAL TL, ADPJL.
Shift  20 : EUA MU ZKRR ZNGZ BGVOJ KDOYZKTZOGROYZ WAGIQ LXKJJE TOKZFYINK ZNGZ NK IGT
    PAYZ HOZK SK, ZCOIK.
Shift  21 : DTZ LT YJQQ YMFY AFUNI JCNXYJSYNFQNXY VZFHP KWJIID SNJYEXHMJ YMFY MJ HFS
    OZXY GNYJ RJ, YBNHJ.
Shift  22 : CSY KS XIPP XLEX ZETMH IBMWXIRXMEPMWX UYEGO JVIHHC RMIXDWGLI XLEX LI GER
    NYWX FMXI QI, XAMGI.
Shift  23 : BRX JR WHOO WKDW YDSLG HALVWHQWLDOLVW TXDFN IUHGGB QLHWCVFKH WKDW KH FDQ
    MXVW ELWH PH, WZLFH.
Shift  24 : AQW IQ VGNN VJCV XCRKF GZKUVGPVKCNKUV SWCEM HTGFFA PKGVBUEJG VJCV JG ECP
    LWUV DKVG OG, VYKEG.
Shift  25 : ZPV HP UFMM UIBU WBQJE FYJTUFOUJBMJTU RVBDL GSFEEZ OJFUATDIF UIBU IF DBO
    KVTU CJUF NF, UXJDF.
Shift  26 : YOU GO TELL THAT VAPID EXISTENTIALIST QUACK FREDDY NIETZSCHE THAT HE CAN
    JUST BITE ME, TWICE.
```

2.3 Work Log

Predicted Time: 5 hours **Actual Time:** 8.5 hours **Ranking:** #5

- **Similarities/Differences:** COBOL has the strangest and in my opinion the most illogical structure with rigid column-based indentation. Unlike modern languages, it's extremely verbose, prioritizing human/non-programmer readability over conciseness.
- **Readability/Writability:** Ironically, its high readability for non-programmers made it harder for me to read as someone who programs in more modern languages. The verbosity made coding feel tedious.
- **What I Loved:** I didn't really like programming in COBOL at all. The indentation made it look cool and felt really outdated - like I was programming in the 60's. That's about it.
- **What I Hated:** The indentation rules were extremely frustrating, especially since the online compiler I used didn't enforce them. I would have to alter the indentation of my program to run it, and debugging was slow because there were so many keywords to worry about.
- **Why the Time Discrepancy?** Most of my extra time was spent adjusting to the indentation rules and debugging errors related to formatting. Every online compiler I used did not enforce the indentation rules so it made it so much harder to run and debug.
- **AI/Google Searches Used:**
  - "COBOL indentation rules"
  - "COBOL string manipulation"
  - "COBOL free online compiler"

3 BASIC

3.1 Program

```
10 INPUT "Enter text to encrypt: "; IN_TEXT$
20 INPUT "Enter shift value: "; SHIFT
30 GOSUB 120
40 PRINT OUT_TEXT$

50 INPUT "Enter text to decrypt: "; IN_TEXT$
60 INPUT "Enter shift value: "; SHIFT
70 GOSUB 210
80 PRINT OUT_TEXT$

90 INPUT "Enter text for brute-force solve: "; IN_TEXT$
100 GOSUB 300
110 END

120 REM ENCRYPTION SUBROUTINE
130 OUT_TEXT$ = ""
140 FOR I = 1 TO LEN(IN_TEXT$)
150     ASCII = ASC(MID(IN_TEXT$, I, 1))
160     IF ASCII >= 97 AND ASCII <= 122 THEN ASCII = ASCII - 32
170     IF ASCII >= 65 AND ASCII <= 90 THEN ASCII = (ASCII - 65 + SHIFT - (26 * INT((
    ASCII - 65 + SHIFT)/26))) + 65
180     OUT_TEXT$ = OUT_TEXT$ + CHR(ASCII)
190 NEXT I
200 RETURN

210 REM DECRYPTION SUBROUTINE
220 OUT_TEXT$ = ""
230 FOR I = 1 TO LEN(IN_TEXT$)
240     ASCII = ASC(MID(IN_TEXT$, I, 1))
250     IF ASCII >= 97 AND ASCII <= 122 THEN ASCII = ASCII - 32
260     IF ASCII >= 65 AND ASCII <= 90 THEN ASCII = (ASCII - 65 - SHIFT + 26 - (26 * INT
    ((ASCII - 65 - SHIFT + 26)/26))) + 65
270     OUT_TEXT$ = OUT_TEXT$ + CHR(ASCII)
280 NEXT I
290 RETURN

300 REM BRUTE-FORCE SOLVING SUBROUTINE
310 PRINT "Solving the cipher with all 26 possible shifts:"
320 FOR S = 1 TO 26
330     OUT_TEXT$ = ""
340     FOR I = 1 TO LEN(IN_TEXT$)
350         ASCII = ASC(MID(IN_TEXT$, I, 1))
360         IF ASCII >= 97 AND ASCII <= 122 THEN ASCII = ASCII - 32
370         IF ASCII >= 65 AND ASCII <= 90 THEN ASCII = (ASCII - 65 - S + 26 - (26 * INT
    ((ASCII - 65 - S + 26)/26))) + 65
380         OUT_TEXT$ = OUT_TEXT$ + CHR(ASCII)
390     NEXT I
400     PRINT S " " OUT_TEXT$
410 NEXT S
420 RETURN
```

3.2 Sample Output

```
Enter text to encrypt: The quick, brown fox jumps over the lazy dog.
Enter shift value: 9
Encrypted text: CQN ZDRLT, KAXFW OXG SDVYB XENA CQN UJIH MXP.

Enter text to decrypt: Ymj vznhp, gwtbs ktc ozrux tajw ymj qfed itl.
Enter shift value: 5
Decrypted text: THE QUICK, BROWN FOX JUMPS OVER THE LAZY DOG.

Enter text for brute-force solve: Ocz lpdxf wmjri ajs ephkn jqzm ocz gvut yjb.
Solving the cipher with all 26 possible shifts:
Shift            1 : NBY KOCWE, VLIQH ZIR DOGJM IPYL NBY FUTS XIA.
Shift            2 : MAX JNBVD, UKHPG YHQ CNFIL HOXK MAX ETSR WHZ.
Shift            3 : LZW IMAUC, TJGOF XGP BMEHK GNWJ LZW DSRQ VGY.
Shift            4 : KYV HLZTB, SIFNE WFO ALDGJ FMVI KYV CRQP UFX.
Shift            5 : JXU GKYSA, RHEMD VEN ZKCFI ELUH JXU BQPO TEW.
Shift            6 : IWT FJXRZ, QGDLC UDM YJBEH DKTG IWT APON SDV.
Shift            7 : HVS EIWQY, PFCKB TCL XIADG CJSF HVS ZONM RCU.
Shift            8 : GUR DHVPX, OEBJA SBK WHZCF BIRE GUR YNML QBT.
Shift            9 : FTQ CGUOW, NDAIZ RAJ VGYBE AHQD FTQ XMLK PAS.
Shift           10 : ESP BFTNV, MCZHY QZI UFXAD ZGPC ESP WLKJ OZR.
Shift           11 : DRO AESMU, LBYGX PYH TEWZC YFOB DRO VKJI NYQ.
Shift           12 : CQN ZDRLT, KAXFW OXG SDVYB XENA CQN UJIH MXP.
Shift           13 : BPM YCQKS, JZWEV NWF RCUXA WDMZ BPM TIHG LWO.
Shift           14 : AOL XBPJR, IYVDU MVE QBTWZ VCLY AOL SHGF KVN.
Shift           15 : ZNK WAOIQ, HXUCT LUD PASVY UBKX ZNK RGFE JUM.
Shift           16 : YMJ VZNHP, GWTBS KTC OZRUX TAJW YMJ QFED ITL.
Shift           17 : XLI UYMGO, FVSAR JSB NYQTW SZIV XLI PEDC HSK.
Shift           18 : WKH TXLFN, EURZQ IRA MXPSV RYHU WKH ODCB GRJ.
Shift           19 : VJG SWKEM, DTQYP HQZ LWORU QXGT VJG NCBA FQI.
Shift           20 : UIF RVJDL, CSPXO GPY KVNQT PWFS UIF MBAZ EPH.
Shift           21 : THE QUICK, BROWN FOX JUMPS OVER THE LAZY DOG.
Shift           22 : SGD PTHBJ, AQNVM ENW ITLOR NUDQ SGD KZYX CNF.
Shift           23 : RFC OSGAI, ZPMUL DMV HSKNQ MTCP RFC JYXW BME.
Shift           24 : QEB NRFZH, YOLTK CLU GRJMP LSBO QEB IXWV ALD.
Shift           25 : PDA MQEYG, XNKSJ BKT FQILO KRAN PDA HWVU ZKC.
Shift           26 : OCZ LPDXF, WMJRI AJS EPHKN JQZM OCZ GVUT YJB.
```

3.3 Work Log

Predicted Time: 4 hours **Actual Time:** 5.5 hours **Ranking:** #3

- **Similarities/Differences:** Basic reminds me of assembly in some ways, mainly the reliance on line numbers. Although most online compilers including the one I used did not enforce line numbers, I still used them as it felt more like an exploration of the original language. The `GOSUB` statements made flow control unique compared to more structured modern languages. Even though modern languages like C have similar `GOTO` statements, its not common practice to use them anymore.

- **Readability/Writability:** The line numbers made the code somewhat easier to follow at first but also more difficult when trying to navigate jumps in execution when debugging later. Adjusting the line numbers after some edits was a pain.
- **What I Loved:** I had fun coding in Basic. I enjoyed writing code in assembly (every CS major's rite of passage) for other classes, so programming in Basic was like a more cleaned-up version of assembly. It felt retro but simple enough to grasp quickly.
- **What I Hated:** The `GOSUB` and line numbers made code organization more cumbersome, especially after editing and debugging, but I kind of did that to myself.
- **Why the Time Discrepancy?** Small debugging issues, mostly related to execution flow and tracking down logic errors due to jumps.
- **AI/Google Searches Used:**
  - "BASIC programming language GOSUB example"
  - "BASIC programming language programs structure"
  - "BASIC programming language string manipulation"

4 Pascal

4.1 Program

```pascal
program CaesarCipher;

uses crt;

var
  input: string;
  shift: integer;

procedure Encrypt(var text: string; shift: integer);
var
  i, ascii: integer;
  output: string;
begin
  output := text;
  for i := 1 to Length(text) do
  begin
    ascii := Ord(text[i]);
    if (ascii >= Ord('a')) and (ascii <= Ord('z')) then
      ascii := ascii - Ord('a') + Ord('A');
    if (ascii >= Ord('A')) and (ascii <= Ord('Z')) then
      ascii := ((ascii - Ord('A') + shift) mod 26) + Ord('A');
    output[i] := Chr(ascii);
  end;
  writeln('Encrypted text: ', output);
end;

procedure Decrypt(var text: string; shift: integer);
var
  i, ascii: integer;
  output: string;
begin
  output := text;
  for i := 1 to Length(text) do
  begin
    ascii := Ord(text[i]);
    if (ascii >= Ord('a')) and (ascii <= Ord('z')) then
      ascii := ascii - Ord('a') + Ord('A');
    if (ascii >= Ord('A')) and (ascii <= Ord('Z')) then
      ascii := ((ascii - Ord('A') - shift + 26) mod 26) + Ord('A');
    output[i] := Chr(ascii);
  end;
  writeln('Decrypted text: ', output);
end;

procedure Solve(var text: string);
var
  i, j, ascii: integer;
  output: string;
begin
  writeln('Solving the cipher with all 26 possible shifts:');
```

```
  for i := 1 to 26 do
  begin
    output := text;
    for j := 1 to Length(text) do
    begin
      ascii := Ord(text[j]);
      if (ascii >= Ord('a')) and (ascii <= Ord('z')) then
        ascii := ascii - Ord('a') + Ord('A');
      if (ascii >= Ord('A')) and (ascii <= Ord('Z')) then
        ascii := ((ascii - Ord('A') - i + 26) mod 26) + Ord('A');
      output[j] := Chr(ascii);
    end;
    writeln('Shift ', i, ': ', output);
  end;
end;

begin
  clrscr;
  writeln('Enter text to encrypt: ');
  readln(input);
  writeln('Enter shift value: ');
  readln(shift);
  Encrypt(input, shift);

  writeln('Enter text to decrypt: ');
  readln(input);
  writeln('Enter shift value: ');
  readln(shift);
  Decrypt(input, shift);

  writeln('Enter text for brute-force solve: ');
  readln(input);
  Solve(input);

  readln;
end.
```

4.2 Sample Output

```
Enter text to encrypt: rats live on no evil star.
Enter shift value: 18
Encrypted text: JSLK DANW GF FG WNAD KLSJ.

Enter text to decrypt: AJCB UREN XW WX NERU BCJA.
Enter shift value: 9
Decrypted text: RATS LIVE ON NO EVIL STAR.

Enter text for brute-force solve:
Solving the cipher with all 26 possible shifts:
Shift 1: BKDC VSFO YX XY OFSV CDKB.
Shift 2: AJCB UREN XW WX NERU BCJA.
Shift 3: ZIBA TQDM WV VW MDQT ABIZ.
Shift 4: YHAZ SPCL VU UV LCPS ZAHY.
```

```
Shift 5: XGZY ROBK UT TU KBOR YZGX.
Shift 6: WFYX QNAJ TS ST JANQ XYFW.
Shift 7: VEXW PMZI SR RS IZMP WXEV.
Shift 8: UDWV OLYH RQ QR HYLO VWDU.
Shift 9: TCVU NKXG QP PQ GXKN UVCT.
Shift 10: SBUT MJWF PO OP FWJM TUBS.
Shift 11: RATS LIVE ON NO EVIL STAR.
Shift 12: QZSR KHUD NM MN DUHK RSZQ.
Shift 13: PYRQ JGTC ML LM CTGJ QRYP.
Shift 14: OXQP IFSB LK KL BSFI PQXO.
Shift 15: NWPO HERA KJ JK AREH OPWN.
Shift 16: MVON GDQZ JI IJ ZQDG NOVM.
Shift 17: LUNM FCPY IH HI YPCF MNUL.
Shift 18: KTML EBOX HG GH XOBE LMTK.
Shift 19: JSLK DANW GF FG WNAD KLSJ.
Shift 20: IRKJ CZMV FE EF VMZC JKRI.
Shift 21: HQJI BYLU ED DE ULYB IJQH.
Shift 22: GPIH AXKT DC CD TKXA HIPG.
Shift 23: FOHG ZWJS CB BC SJWZ GHOF.
Shift 24: ENGF YVIR BA AB RIVY FGNE.
Shift 25: DMFE XUHQ AZ ZA QHUX EFMD.
Shift 26: CLED WTGP ZY YZ PGTW DELC.
```

4.3 Work Log

Predicted Time: 3 hours **Actual Time:** 3 hours **Ranking:** #1

- **Similarities/Differences:** Pascal felt the most "modern" out of the languages explored. It had an easy-to-understand structure, strong typing, and a more familiar syntax. It reminded me of structured languages like C. It felt like a combination of pseudocode and the best parts of modern languages.
- **Readability/Writability:** Very readable and straightforward to write. The structured approach reduced programming and debugging time significantly.
- **What I Loved:** I liked the syntax and structure of Pascal the best overall. It felt like I was just programming in pseudocode and it would actually run. It was relatively easy to debug, which made it my favorite language in this comparison.
- **What I Hated:** Not much! I think it would have been much more fun to learn programming with Pascal instead of Python.
- **Why No Time Discrepancy?** Pascal was intuitive, and I didn't run into major debugging issues. I had the least issues with my Pascal version of the caesar cipher, so I may be biased in ranking this as #1.
- **AI/Google Searches Used:**
  - "Pascal string manipulation"
  - "Pascal procedure vs function"
  - "Pascal ord function"

5 Scala (procedural)

5.1 Program

```scala
object CaesarCipher {
  def main(args: Array[String]): Unit = {
    print("Enter text to encrypt: ")
    val inputEncrypt = scala.io.StdIn.readLine()
    print("Enter shift value: ")
    val shiftEncrypt = scala.io.StdIn.readInt()
    println("Encrypted text: " + encrypt(inputEncrypt, shiftEncrypt))

    print("Enter text to decrypt: ")
    val inputDecrypt = scala.io.StdIn.readLine()
    print("Enter shift value: ")
    val shiftDecrypt = scala.io.StdIn.readInt()
    println("Decrypted text: " + decrypt(inputDecrypt, shiftDecrypt))

    print("Enter text for brute-force solve: ")
    val inputSolve = scala.io.StdIn.readLine()
    solve(inputSolve)
  }

  def encrypt(input: String, shift: Int): String = {
    val upperInput = input.toUpperCase
    val encrypted = new StringBuilder

    for (char <- upperInput) {
      if (char.isLetter) {
        val shifted = ((char - 'A' + shift) % 26 + 'A').toChar
        encrypted.append(shifted)
      } else {
        encrypted.append(char)
      }
    }
    encrypted.toString()
  }

  def decrypt(input: String, shift: Int): String = {
    encrypt(input, 26 - (shift % 26))
  }

  def solve(input: String): Unit = {
    println("Solving the cipher with all 26 possible shifts:")
    for (shift <- 1 to 26) {
      println(s"Shift $shift: ${decrypt(input, shift)}")
    }
  }
}
```

5.2 Sample Output

```
Enter text to encrypt: I love lambda calculus
Enter shift value: 17
Encrypted text: Z CFMV CRDSUR TRCTLCLJ

Enter text to decrypt: S vyfo vkwlnk mkvmevec!
Enter shift value: 10
Decrypted text: I LOVE LAMBDA CALCULUS!

Enter text for brute-force solve: J MPWF MBNCEB DBMDVMVT!
Solving the cipher with all 26 possible shifts:
Shift 1: I LOVE LAMBDA CALCULUS!
Shift 2: H KNUD KZLACZ BZKBTKTR!
Shift 3: G JMTC JYKZBY AYJASJSQ!
Shift 4: F ILSB IXJYAX ZXIZRIRP!
Shift 5: E HKRA HWIXZW YWHYQHQO!
Shift 6: D GJQZ GVHWYV XVGXPGPN!
Shift 7: C FIPY FUGVXU WUFWOFOM!
Shift 8: B EHOX ETFUWT VTEVNENL!
Shift 9: A DGNW DSETVS USDUMDMK!
Shift 10: Z CFMV CRDSUR TRCTLCLJ!
Shift 11: Y BELU BQCRTQ SQBSKBKI!
Shift 12: X ADKT APBQSP RPARJAJH!
Shift 13: W ZCJS ZOAPRO QOZQIZIG!
Shift 14: V YBIR YNZOQN PNYPHYHF!
Shift 15: U XAHQ XMYNPM OMXOGXGE!
Shift 16: T WZGP WLXMOL NLWNFWFD!
Shift 17: S VYFO VKWLNK MKVMEVEC!
Shift 18: R UXEN UJVKMJ LJULDUDB!
Shift 19: Q TWDM TIUJLI KITKCTCA!
Shift 20: P SVCL SHTIKH JHSJBSBZ!
Shift 21: O RUBK RGSHJG IGRIARAY!
Shift 22: N QTAJ QFRGIF HFQHZQZX!
Shift 23: M PSZI PEQFHE GEPGYPYW!
Shift 24: L ORYH ODPEGD FDOFXOXV!
Shift 25: K NQXG NCODFC ECNEWNWU!
Shift 26: J MPWF MBNCEB DBMDVMVT!
```

5.3 Work Log

Predicted Time: 3 hours **Actual Time:** 4 hours **Ranking:** #2

- **Similarities/Differences:** Very similar to Java in many ways. Coming from a procedural mindset rather than functional, it wasn't too difficult to pick up. However, debugging was just as annoying as it is in Java.
- **Readability/Writability:** Readable once you know what's going on especially if you know Java, but Scala's flexibility makes debugging harder.
- **What I Loved:** Because it was so similar to Java it was easy to pick up and felt more intuitive than other older languages. The functions in this program were much shorter - I

could do the same things in fewer lines. Scala felt more modern and more powerful compared to the others.

- **What I Hated:** I spent a decent amount of time debugging the user input portion. It takes longer and is less intuitive similar to Java or C.
- **Why the Time Discrepancy?** Debugging took longer than expected, due to unfamiliarity with some Scala-specific syntax.
- **AI/Google Searches Used:**
  - "Scala string handling functions"
  - "Scala read user input"
  - "Scala objects and functions"

6 Summary

I had a lot of fun exploring old languages, and it gave me a better perspective on where modern languages get their roots. It was cool to see the different iterations over time, and I really liked the "hands-on" approach to programming history. An assignment like this would also be a unique talking point to bring up in a job interview - not many people even code in COBOL anymore (for good reason).

7 Resources

7.1 Compilers and IDEs

- OneCompiler: free online compiler to run programs
- IDEone: another free online IDE