Joshua Dunne
Assignment 5

## Abstract

We're going to demonstrate a basic understanding of shell scripts, and using tr, we'll provide a basic command line interface whereby which a user might use our scripts functionality of ROT13 and Caesar encoding and decoding.

## Introduction

We use our virtual machine as was set up in previous assignments. Using only command line tools, we'll develop a simple script that enables a user to

## Summary of Results

First, let's examine the encrypt and decrypt functions we've produced.

```
#!/bin/bash

encrypt() {
        local encrypted="$1"
        local offset="$2"
        local decrypted=""

        upper="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        lower="abcdefghijklmnopqrstuvwxyz"
        # The logic here is simple, we're transpoing by a specific offset
        decrypted=$(echo "$encrypted" | tr "\[$upper$lower\]" "\[${upper:$offset}${upper::$offset}${lower:$offset}${lower::$offset}\]")

        echo "$decrypted"
}
```

As it was intended that we use tr, we can see its use here, using substring expansion, we map the alphabet to an offset alphabet.

```
decrypt() {
        local encrypted="$1"
        local offset="$2"
        local decrypted=""

        upper="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        lower="abcdefghijklmnopqrstuvwxyz"
        # All we need to do is swap the ordering of our parameters to work backwords from where we started
        decrypted=$(echo "$encrypted" | tr "\[${upper:$offset}${upper::$offset}${lower:$offset}${lower::$offset}\]" "\[$upper$lower\]")

        echo "$decrypted"
}
```

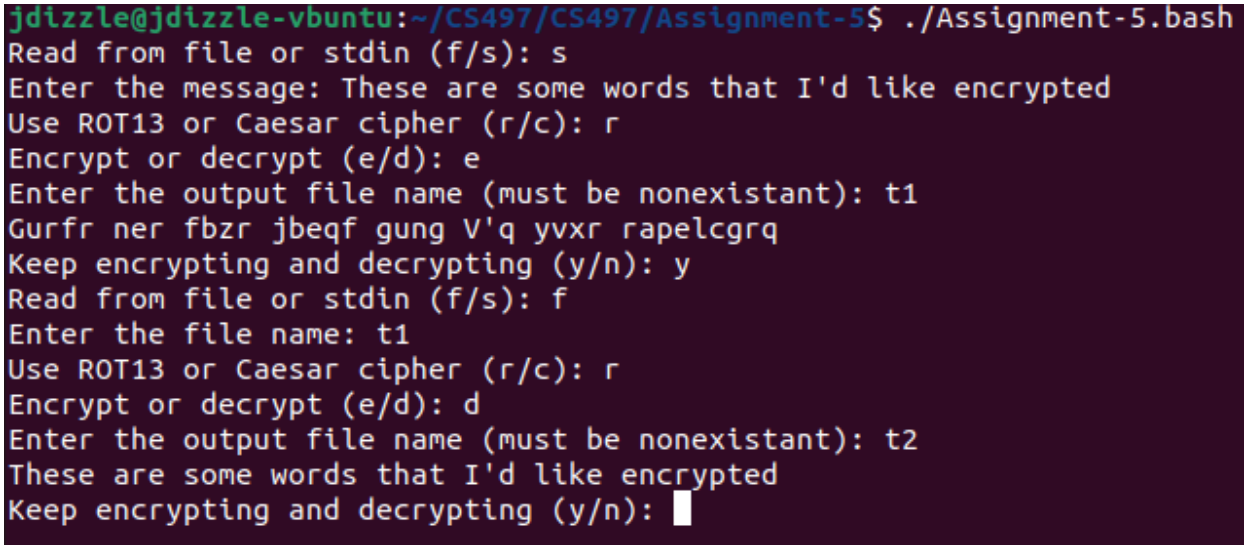
Likewise, because we're using a simple methodology, in order to transpose the elements back to where they were, we simply transpose the parameters we give to tr.

```
keep_going_string=""
while [[ ! $keep_going_string =~ ^[nN]$ ]]; do
        # There's a simple thing we keep doing here, we'll keep looping until the user enters output
        # in the format that we're expecting
        file_or_stdin=""
        while [[ ! $file_or_stdin =~ ^[fF]$ && ! $file_or_stdin =~ ^[sS]$ ]]; do
                read -p "Read from file or stdin (f/s): " file_or_stdin
        done
        if [[ $file_or_stdin =~ ^[fF]$ ]]; then
                file_name=""
                while [[ ! -f $file_name ]]; do
                        read -p "Enter the file name: " file_name
                done
                message=$(cat "$file_name")
        else
                read -p "Enter the message: " message
        fi
        use_rot13_or_caesar=""
        while [[ ! $use_rot13_or_caesar =~ ^[rR]$ && ! $use_rot13_or_caesar =~ ^[cC]$ ]]; do
                read -p "Use ROT13 or Caesar cipher (r/c): " use_rot13_or_caesar
        done
        if [[ $use_rot13_or_caesar =~ ^[rR]$ ]]; then
                offset=13
        else
                offset=""
                while [[ ! $offset =~ ^[0-9]+$ ]]; do
                        read -p "Enter the offset: " offset
                done
        fi
```

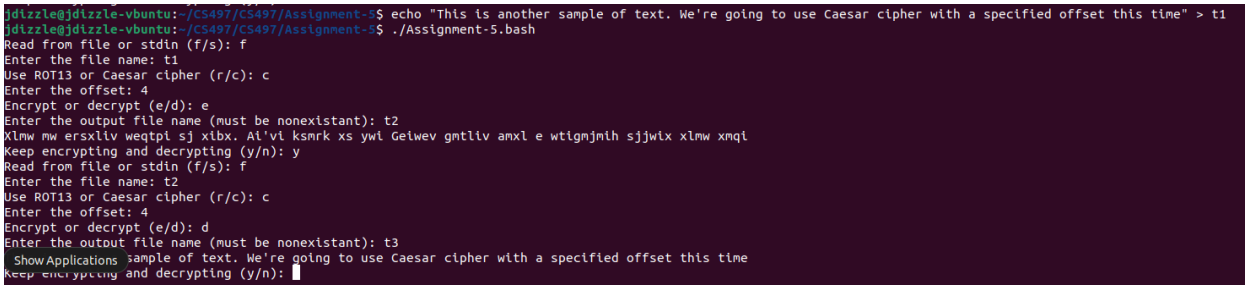
Because it was intended that we loop while the user keeps wanting to encrypt and decrypt our main loop does this. We suppose the user wants to continue for the first iteration, then continue, asking the user repeatedly until their input conforms to the options we're asking them to specify.

```
        # Determine whether to encrypt or decrypt
        encrypt_or_decrypt=""
        while [[ ! $encrypt_or_decrypt =~ ^[eE]$ && ! $encrypt_or_decrypt =~ ^[dD]$ ]]; do
                read -p "Encrypt or decrypt (e/d): " encrypt_or_decrypt
        done

        # Enter name of output file
        output_file_name=""
        while [[ -z "$output_file_name" ]]; do
                read -p "Enter the output file name (must be nonexistant): " output_file_name
        done

        # Perform the encryption or decryption
        if [[ $encrypt_or_decrypt =~ ^[eE]$ ]]; then
                encrypted_message=$(encrypt "$message" "$offset")
                echo "$encrypted_message"
                echo "$encrypted_message" > "$output_file_name"
        else
                decrypted_message=$(decrypt "$message" "$offset")
                echo "$decrypted_message"
                echo "$decrypted_message" > "$output_file_name"
        fi

        keep_going_string=""
        while [[ ! $keep_going_string =~ ^[yYnN]$ ]];do
                read -p "Keep encrypting and decrypting (y/n): " keep_going_string
        done
done
```

We continue with the same methodology, asking the user to specify their options, and parameterizing our encryption and decryption methods to their specifications. Output is collected in specified files, and our input can come from either a file or STDIN.




Now to demonstrate usage and proper function, we can see that through the first iteration, we take a line of text from the user, encrypt that using ROT13, then output it into a file “t1”. On the next iteration, we ask the program to read from “t1”, then use ROT13 decryption to produce the initial text.




On a second time through, we’re going to use a Caesar cipher with a specified offset. This time, we first place our input text into a file, then work from there. We read in from our file “t1”, encrypt that using a Caesar cipher with a shift of 4, then output that in to a file “t2”. We then take the file “t2” and decrypt it using a Caesar shift of 4, outputting the result in to t3. Having done this, we can see the proper functioning of the shift, as, we’re presenting with the same text we started with. Important to note, is that all non alphabetic characters are simply left in place.

## Conclusion:

This is our first time writing any shell scripting in this class. We can see that, despite having incredible powers when it comes to things like substrings, there are many

differences between the methods we're used to. Inside our shell script we have access to the myriad tools that come standard with most unix distros. Using tr, we were able to transpose alphabetic text, then return the original text having been encrypted then decrypted using the functions we've developed. Too, we demonstrated the basic use of loops within bash to ensure that users entered proper input, and that the program would continue to run while wanted.