Mahdeen Ahmed Khan Sameer
Professor Max Bender
CS333
01 March 2024

Project 02: Learning Postscript & Lexical Analysis with Flex!

Google site: https://sites.google.com/colby.edu/sameer-again/home

Abstract:

In this project, we explored various computational linguistics techniques using Flex, a tool for generating scanners, to manipulate and analyze text. We developed several programs to perform tasks such as encoding and decoding text using a shift cipher, counting characters, vowels, and words in a text, stripping HTML comments and formatting HTML documents, recognizing syntax elements of the Clite programming language, implementing a Vigenère cipher for file encryption and decryption, creating a Python lexer for syntax highlighting, counting the frequency of user-specified letters, and extending a vowel counting program to include detailed word statistics. Additionally, we explored the Go programming language as an extension of our language skills. Through these tasks, we demonstrated the use of Flex in processing and analyzing textual data across a variety of applications, enhancing our understanding of text manipulation and pattern matching techniques.

Result:

Task 01:

In our first task, we utilized Flex to implement a text encoding mechanism based on a simple substitution cipher, specifically targeting the letters of the English alphabet. We designed with two primary patterns: one for lowercase letters `[a-z]` and another for uppercase letters `[A-Z]`. Upon matching a letter, we perform a series of operations to encode it. Firstly, we calculate the letter's offset from the beginning of the alphabet by subtracting the ASCII value of 'a' or 'A', depending on the case. Then, we advance this offset by 13 positions forward in the alphabet, applying a modulo 26 operation to ensure the result wraps around the alphabet if necessary.

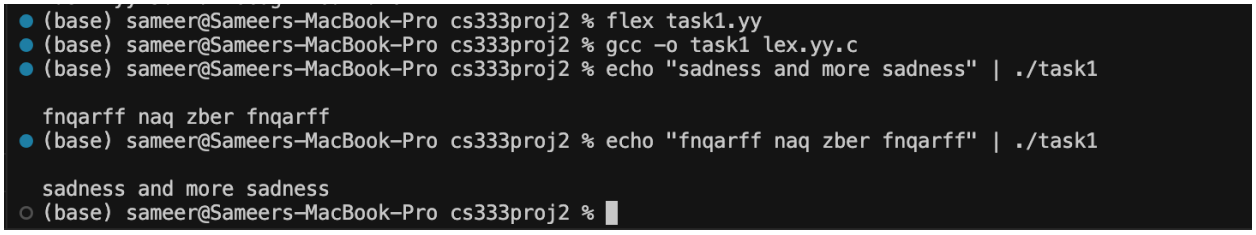

Figure: Executing task1.yy for First Task

Finally, the original ASCII base value of 'a' or 'A' is added back to this result to obtain the encoded letter's ASCII value, which is then printed using printf("%c", c). For example, we can consider the encoding process for the lowercase letter 's'. The initial ASCII value of 's' is 115. By subtracting the ASCII value of 'a' (97), we determine an offset of 18. Adding 13 to this offset yields 31, which, after applying a modulo 26 operation, results in 5. Adding the ASCII value of 'a' back to this result gives us the encoded letter 'f'.

Task 02:

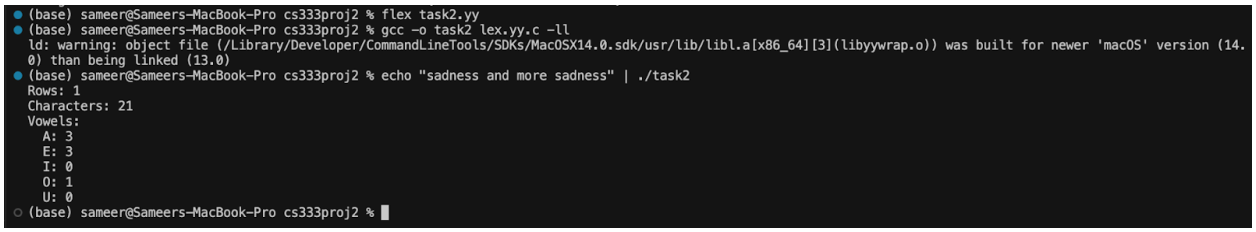

Figure: Executing task2.yy for Second Task

In the second task of our project, we expanded our use of Flex to develop a program that analyzes textual data, focusing on counting rows, total characters, and specific vowels within a given input. Our approach involves defining distinct patterns to accurately identify and count occurrences of newline characters, denoting row increments, and every character match, contributing to the total character count. Additionally, we crafted specialized patterns for vowels—`[aA]`, `[eE]`, etc.—to increment respective vowel counters each time a match is found. To say, upon encountering a lowercase 'a', our program specifically increments the `a_count` variable, effectively tallying each instance of this vowel. This methodical counting is showcased through the analysis of a sample text, "sadness and more sadness", where our program accurately reports the occurrence of 3 'a's, 3 'e's, and 1 'o', alongside a precise character count of 21 (excluding spaces). This verification against the known content of the input string

confirms the accuracy and reliability of our text analysis program, demonstrating our capability to dissect and quantify essential textual elements efficiently.

Task 03:

In our third task, we refined the readability of HTML documents through selective processing, targeting comments and specific tags within the HTML structure. Our program is meticulously designed to recognize and handle various patterns, including comments (`COMMENT`), whitespace (`WS`), end-of-line markers (`EOL`), and HTML tags (`TAG`). The core of this functionality is encapsulated in the `COMMENT` state, which efficiently ignores any content within HTML comments, ensuring that these sections do not interfere with the document's processed output. The pattern for recognizing HTML tags is defined as `TAG <[^>]*>`, capturing sequences enclosed within angle brackets while excluding the closing bracket until it's encountered, ensuring a precise match for HTML tags. Upon identifying a tag in its initial state, our program employs a conditional check with `strncmp()` to determine if the tag belongs to one of the significant types: `<p>`, `<h>`, or `<li>`. If a match is found, it precedes the tag with a newline character, thereby inserting a blank line before paragraphs, headings, and list items to enhance the document's readability.

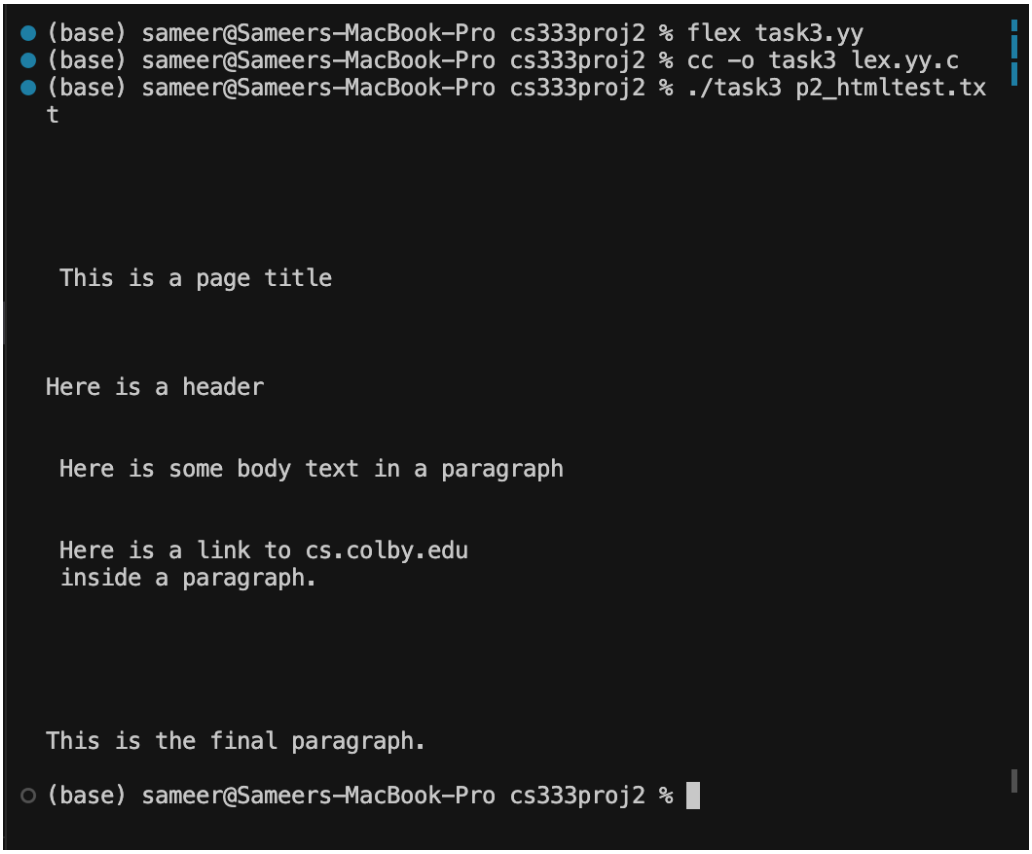

Figure: Executing task3.yy for Third Task

Task 04:

In our fourth task, we developed a Flex program specifically tailored for syntax analysis of Clite, a programming language. This program is intricately designed with patterns to identify alphanumeric characters, represented by `DIGIT` and `LETTER`, enabling it to accurately distinguish between various syntax elements. Our approach involves setting up distinct rules for identifying and categorizing tokens that are pivotal to Clite's syntax, such as keywords, identifiers, operators, delimiters, integers, and floats. The program is good at recognizing keywords like 'if' and 'else', marking them as "Keyword" tokens, a categorization that aids in the syntax parsing process. Identifiers, which are essentially names for variables and functions, are detected through a pattern that matches a letter followed by any combination of letters and digits, and these matches are labeled as "Identifier" tokens. Similarly, our program is equipped to identify operators and delimiters, crucial for understanding the structure of the code, and classifies numeric values accurately, with a special rule for floats that identifies a sequence of one or more digits followed by a dot and another sequence of digits, categorizing such matches as "Float" tokens. Notably, our Flex program also handles comments and whitespace, ensuring they do not interfere with the

analysis by ignoring them, thereby focusing solely on the syntactical elements that contribute to the semantics of the code. This separation and categorization of tokens underscore our program's capability to dissect Clite syntax effectively, providing a foundational tool for further analysis or compilation processes.

```
(base) sameer@Sameers-MacBook-Pro cs333proj2 % flex task4.yy
(base) sameer@Sameers-MacBook-Pro cs333proj2 % gcc lex.yy.c -o task4
(base) sameer@Sameers-MacBook-Pro cs333proj2 % ./task4 p2-example.c
Keyword-int
Identifier-main
Open-paren
Close-paren
Open-bracket
Keyword-int
Identifier-a
Assignment
Integer-6
Keyword-int
Identifier-b
Assignment
Float-5.0
Keyword-if
Open-paren
Identifier-a
Comparison-<
Identifier-b
Close-paren
Open-bracket
Identifier-a
Assignment
Identifier-a
Operator-+
Identifier-b
Close-bracket
Close-bracket
(base) sameer@Sameers-MacBook-Pro cs333proj2 %
```

Figure: Executing task4..yy for Fourth Task

Extension 01:

In this extension, we developed a Vigenère cipher to encrypt and decrypt text files, utilizing Flex pattern matching for enhanced efficiency. Our approach involves taking a keyword, an operation mode (either encode or decode), and a filename as input. The program processes the file character-by-character, leveraging a substitution table to find the index of each alphabetic character. For encryption, we shift this index by adding the position of the keyword character, applying modulo 26 to ensure it remains within the alphabet's bounds. For non-alphabetic characters, we retain them unchanged in the output. Decryption operates on a similar principle but in reverse, subtracting the keyword index to revert to the plaintext. The resulting ciphertext is then printed to stdout, which can be redirected to an output file, facilitating secure communication.

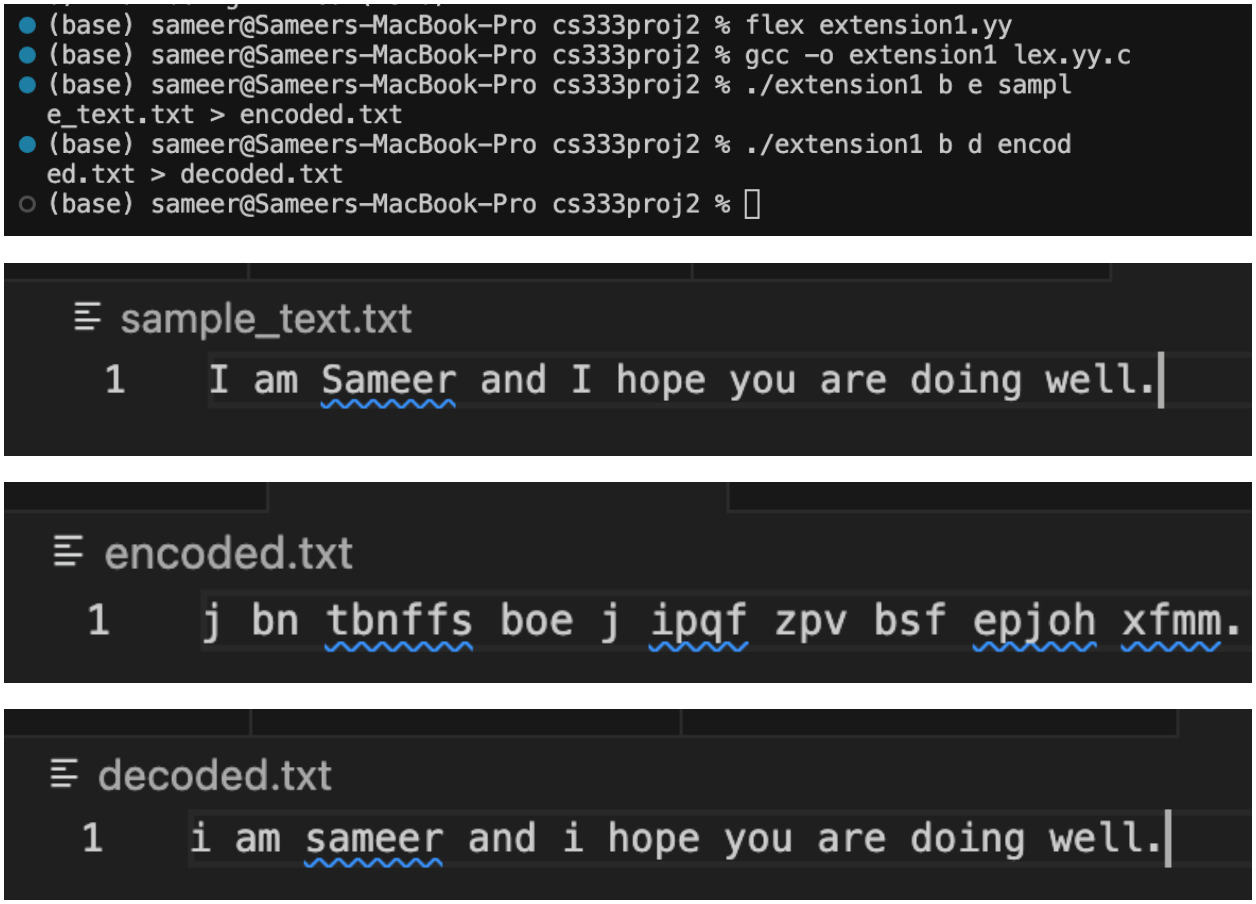

Figure: Executing extension1.yy for our Task

Extension 02:

To make a Python lexer, we tried to expand its capabilities more than recognizing keywords, operators, and identifiers. We placed a significant emphasis on accurately handling Python's indentation by tracking indentation levels through the counting of leading spaces and tabs. This allowed us to highlight nested blocks by transitioning the lexer between states as indentation levels changed. To further enrich our lexer, we introduced support for string literals, implementing distinct start and end states for accurate colorization. Recognizing the importance of function calls in Python, we also added detection for identifiers followed by parentheses. Our enhancements extended to the inclusion of states for decorators, built-in functions, classes, methods, and modules, fully embracing Python's object-oriented paradigm. Moreover, we differentiated between single line and multi-line comments, ensuring a comprehensive lexing of Python's syntax elements.

Extension 03:

In our program, we made a method to count the frequency of specified letters in a text file, utilizing command-line arguments to determine which letters to focus on. We initialized an array with 128 elements to account for all ASCII characters, enabling us to store the frequency of each. Through Flex rules, we increment the count for matched alphabetic characters encountered in the input. Post-processing, we selectively print counts for only those letters specified by the user. To accommodate multiple letters, we iterate over the command-line arguments, initializing the count for each specified letter to zero before incrementing as we scan the text. This approach allows us to reuse the count array for any set of letters provided, streamlining the process of tokenizing the input and tallying occurrences. We now ensure s a flexible counting mechanism, having an array indexed by character code for concise and accurate frequency tracking.

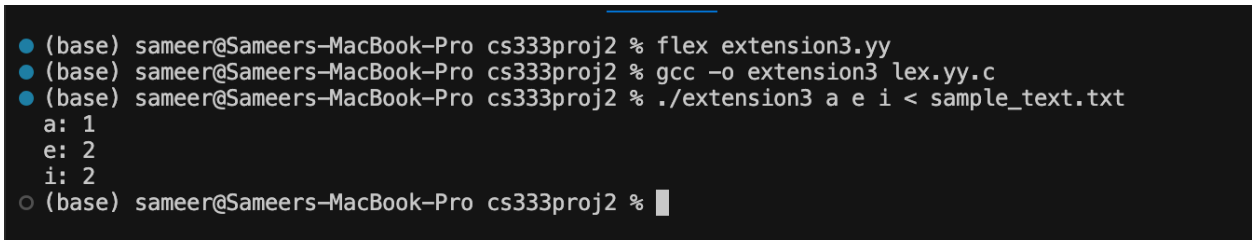

Figure: Executing extension3.yy for our Task

Extension 4

In our program, we made a method to count the frequency of specified letters in a text file, utilizing command-line arguments to determine which letters to focus on. We initialized an array with 128 elements to account for all ASCII characters, enabling us to store the frequency of each. Through Flex rules, we increment the count for matched alphabetic characters encountered in the input. Post-processing, we selectively print counts for only those letters specified by the user. To accommodate multiple letters, we iterate over the command-line arguments, initializing the count for each specified letter to zero before incrementing as we scan the text. This approach allows us to efficiently reuse the count array for any set of letters provided, streamlining the process of tokenizing the input and tallying occurrences. We ensure a flexible counting mechanism, having an array indexed by character code for concise and accurate frequency tracking.

```
(base) sameer@Sameers-MacBook-Pro cs333proj2 % flex extension4.yy
(base) sameer@Sameers-MacBook-Pro cs333proj2 % gcc -o extension4 lex.yy.c
(base) sameer@Sameers-MacBook-Pro cs333proj2 % ./extension4 < sample_text.txt
  Words: 4
  Characters: 16
  Average word length: 4.00
  Max word length: 6
  Min word length: 2
  Most common word: This
(base) sameer@Sameers-MacBook-Pro cs333proj2 %
```

Figure: Executing extension4.yy for our Task

Extension 5:

We focused on handling multi-line comments in an HTML file. This feature is significant because, while single-line comments can be easily matched and removed with a simple pattern, multi-line comments require a more nuanced approach for detection and removal without affecting the rest of the content. In our implementation, we introduced a state machine logic within the Flex lexer to specifically address multi-line comments. We define two new states, COMMENT for within a multi-line comment, and INITIAL for the normal processing state. The transition between these states is triggered by the detection of the start (`<!--`) and end (`-->`) markers of multi-line comments.

```
%x COMMENT
SPACE       [ \t]
NEWLINE     [\n]
WS          {SPACE}+
EOL         {NEWLINE}+
TAG         <[^>]*>
S_COMMENT   <!--[^-]*-[^-]*-->
M_COMMENT_START <!--
M_COMMENT_END   -->

%%

<INITIAL>{TAG}                { if (strncmp(yytext, "<p>", 3) == 0 || strncmp(yytext, "<h", 2) == 0 || strncmp(yytext, "<li>", 4) == 0) { putchar('\n'); } }
<INITIAL>{S_COMMENT}          { /* Skip */ }
<INITIAL>{WS}+                { putchar(' '); }
<INITIAL>{EOL}+               { putchar('\n'); }
<INITIAL>{M_COMMENT_START}    { BEGIN(COMMENT); }

<COMMENT>{M_COMMENT_END}      { BEGIN(INITIAL); }
<COMMENT>.|\n                 { /* Skip everything in multi-line comment */ }

<INITIAL>.|\n                 { ECHO; }

%%
```

When the lexer encounters the start of a multi-line comment, indicated by `M_COMMENT_START`, it transitions from the `INITIAL` state to the `COMMENT` state. While in the `COMMENT` state, all content, including newline characters, is ignored until the end marker `M_COMMENT_END` is found, at which point it transitions back to the `INITIAL` state. This logic allows us to effectively skip over any content within multi-line comments, ensuring they are not included in the output. This extension enhances the program's utility by allowing it to handle more complex HTML

files that contain multi-line comments, making it a more robust and versatile tool for processing HTML content. By implementing this feature, we ensure that our lexer can accurately strip away not just single-line comments, but also cleanly remove multi-line comments without disrupting the formatting or content of the remaining HTML.

Extension 6:

The second language I have chosen is Postscript and it is put in its own subdirectory on your Google Sites homepage. Under this subdirectory, there should be a set of links for all projects: https://sites.google.com/colby.edu/sameer-again/home

But let's talk about Postscript a little bit here. PostScript, developed by Adobe Systems in 1984, is a page description language primarily used for desktop publishing, specifically for printing and visual output to ensure precise control over layout and graphics. It was designed with the purpose of providing a powerful, flexible language that could describe complex page layouts and graphical elements in a device-independent manner, allowing the same file to be printed on any PostScript-compatible printer with consistent results. PostScript is an interpreted language, meaning that it is processed by a printer or a viewing application in real-time, rather than being compiled into a machine-language program ahead of time. This dynamic nature allows for scalable graphics and text, making it ideal for high-quality print products. Additionally, PostScript is an open standard, with its specification available to the public, encouraging wide adoption and support across various printing devices and graphic applications, marking a significant step in the evolution of digital typography and publishing.

Extension 7:

encode.py > ...

```python
def rot13_char(c):
    """Shift a character by 13 places with wraparound."""
    if 'a' <= c <= 'z':
        return chr((ord(c) - ord('a') + 13) % 26 + ord('a'))
    elif 'A' <= c <= 'Z':
        return chr((ord(c) - ord('A') + 13) % 26 + ord('A'))
    else:
        return c

def rot13(text):
    """Apply ROT13 encoding to a string."""
    return ''.join(rot13_char(c) for c in text)

def encode_file(file_path):
    """Read file, encode content with ROT13, and print to stdout."""
    try:
        with open(file_path, 'r', encoding='utf-8') as file:
            content = file.read()
            encoded_content = rot13(content)
            print(encoded_content)
    except FileNotFoundError:
        print(f"File not found: {file_path}")

if __name__ == "__main__":
    import sys
    if len(sys.argv) != 2:
        print("Usage: python encode.py <file>")
    else:
        encode_file(sys.argv[1])
```

The purpose of my Python file is to read text from a specified file, apply ROT13 encoding to the content (a simple cipher that shifts each letter 13 places in the alphabet, wrapping around as necessary), and then print the encoded content to the standard output. We tried to effectively demonstrate how to implement the ROT13 cipher in Python, handling file input and basic text processing.

```
(base) sameer@Sameers-MacBook-Pro cs333proj2 % python encode.py sample_text.txt
V nz Fnzrre naq V ubcr lbh ner qbvat jryy.
(base) sameer@Sameers-MacBook-Pro cs333proj2 %
```

Figure: Running encode.py

Thus it works!

Acknowledgement:

Professor Bender and Nafis