Name: Prathamesh Prabhakar Thakare
Class: TY CS D
Batch: 2
Roll number: 58
PRN: 12220016

Assignment number 1: Count the number of indentifiers

Theory:

Lexical Analysis and Lex:
Lex is a powerful tool used to create lexical analyzers or scanners, which are essential for processing and tokenizing text or source code.

Structure of Lex Code
Lex code is typically organized into three main sections:

Definitions Section (%{ ... %}):

In this section, you can include C code that will be directly incorporated into the generated lexer code. For example, you can define variables and functions here that you want to use throughout your lexer.

Rules Section (%% ... %%):

This section is where you define patterns and their corresponding actions.
These patterns are used to identify and process tokens from the input text.
Actions specify what should be done when a particular pattern is matched.
These patterns and actions together constitute the core logic of the lexer.

User Code Section:

Typically, the int main() function serves as the entry point for the generated lexer code.
In this section, you invoke yylex() to initiate the scanning process on the input text.
The user code section can include any additional code or functions required to interact with the lexer or handle the results.

Definitions Section Example:

- Within the definitions section, you can define variables and functions that assist in the lexer's operation.
- For instance, you might declare variables to keep track of counts or other state information.

Rules Section Example:

- In the rules section, you define regular expressions or patterns that specify the tokens you want to identify.
- For each pattern, you provide an associated action or code snippet that is executed when the pattern is matched.
- Patterns and actions collectively define how the lexer tokenizes and processes the input.

User Code Section Example:

- The main() function in the user code section is where the lexer's execution begins.
- It often involves setting up the input source, invoking yylex() to start scanning, and possibly handling the lexer's output or results.

Maths:

Character Classes:

1. digit[0-9]: This defines a character class that matches any single digit (0-9).
2. letter[a-zA-Z_]: This defines a character class that matches any single letter (uppercase or lowercase) or an underscore (_).

Regular Expression Pattern for Identifiers:

1. {letter}({letter}|{digit})*: This pattern starts with a letter (a-zA-Z_) and is followed by zero or more occurrences of either a letter or a digit (0-9).
2. This regular expression captures valid identifiers in the input.

Counting Identifiers:

1. When the regular expression pattern for identifiers is matched, it signifies the recognition of an identifier.
2. Each time this pattern is matched, the program increments the count variable by 1.

Algorithm:

1. Initialize count to 0
2. Initialize ch to 0
3. Define character classes digit as [0-9] and letter as [a-zA-Z_]
4. Define Lex patterns in the Rules Section:

   - {letter}({letter}|{digit})* {
     Increment count by 1
   }

5. Define the yywrap() function, which always returns 1 (indicating the end of input).

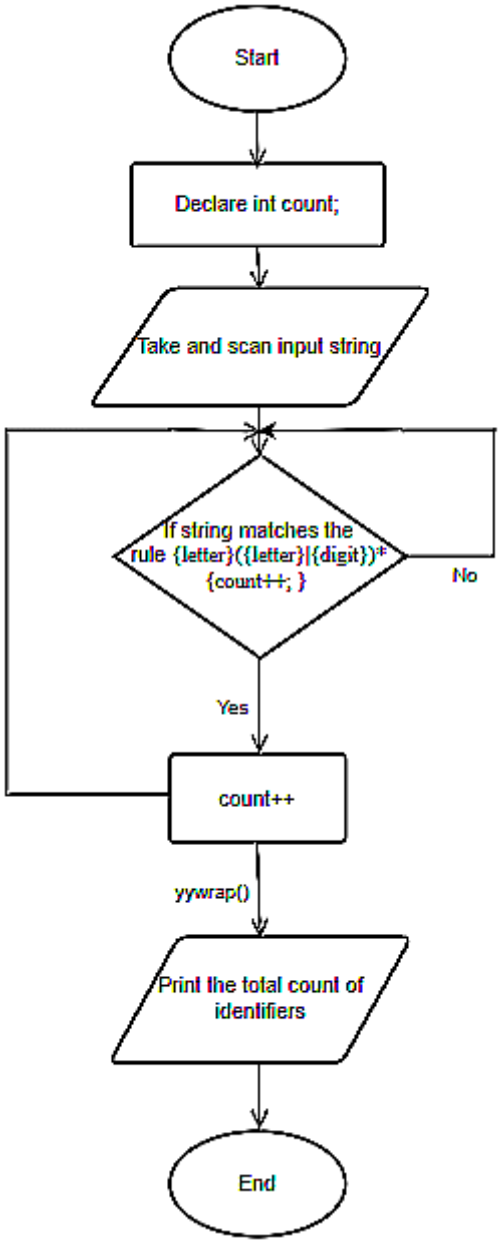
6. In the main() function:

   - Call yylex() to start scanning the input.

   - After scanning is complete, print "Number of identifiers: " followed by the value of count.

7. End the program.

Flow Chart



Code

```
%{#include<iostream.h>
int count=0;
char ch=0;
%}

digit[0-9]
letter[a-zA-Z_]

%%

{letter}({letter}|{digit})* {
 count++;
}
```

```
%%

int yywrap(){return (1);}

int main()
{
 yylex();
 printf("Number of identifiers: %d",count);
 return 0;
}
```

Output

Conclusion

In this assignment, we utilized Lex, a lexical analyzer generator, to develop a program focused on counting identifiers within text. Lex operates by identifying specific patterns in text based on defined rules and incrementing a counter for each occurrence of these patterns. This experience illustrated Lex's practicality as a tool for text analysis, demonstrating its ability to streamline tasks like tokenization and code processing. Overall, this assignment enhanced our understanding of pattern recognition, regular expressions, and code generation, underscoring the value of computational tools in text analysis.