**Compiler Design Lab Report**

**Name:** Vaka Tharun Kumar

**Roll no:** CH.EN.U4CSE22158

**Course Code:** 19CSE401

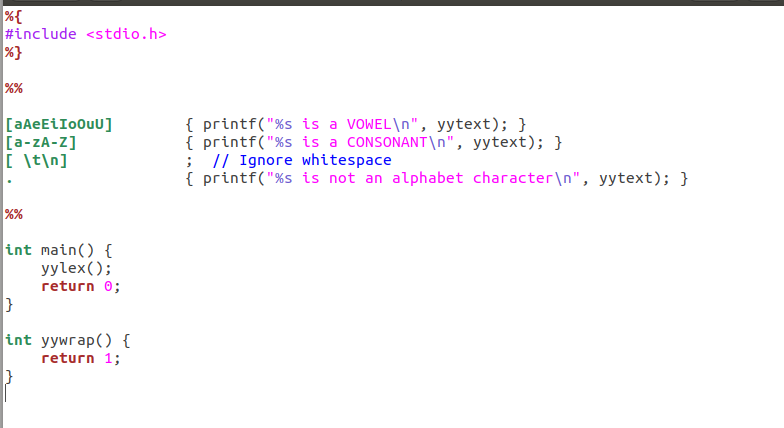
**Basic Programs**

1. **Aim:** Program to Identify Vowels and Consonants

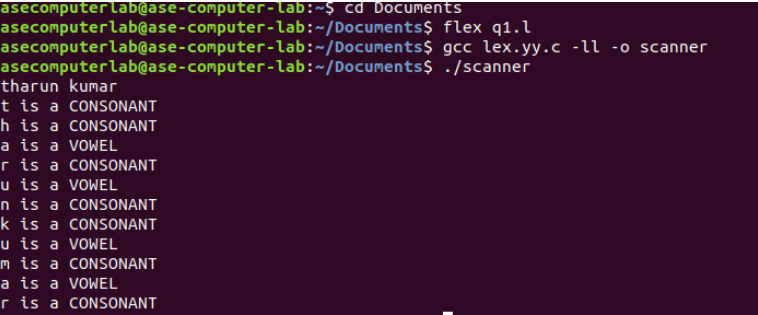
**Algorithm:**

* Open the gedit text editor from Accessories under Applications menu.
* Specify the header file <stdio.h> between %{ and %}.
* Define the character patterns for vowels [aAeEiIoOuU], alphabets [a-zA-Z], whitespaces [ \t\n], and other characters ..
* Use translation rules to print whether the character is a vowel, consonant, or not an alphabet character.
* Call yylex() inside the main() function to begin lexical analysis.
* Save the program as vowelconsonant.l using the LEX language.
* Run the program using the LEX compiler to generate lex.yy.c.
* The generated lex.yy.c contains tables and routines to match input characters.
* Compile lex.yy.c using a C compiler to create an executable file.
* Run the executable to check each character in the input and classify it.

**Code:**



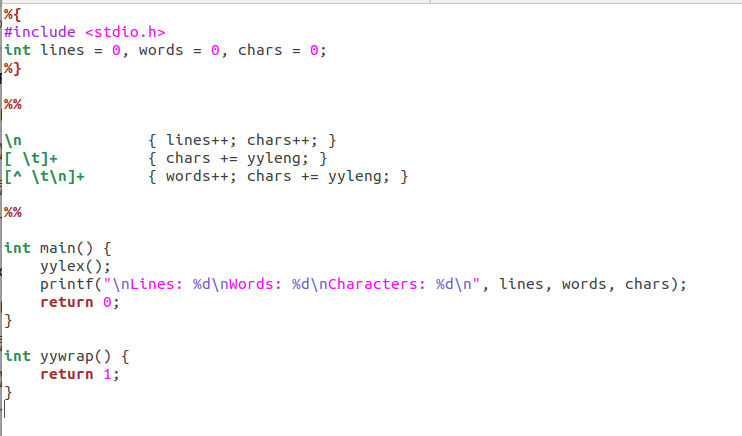
**Output:**

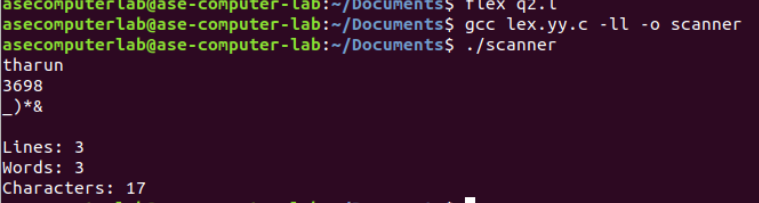


2. **Aim:** Program to Count Lines, Words, and Characters  
 **Algorithm:**

* Open the gedit text editor from Accessories under Applications menu.
* Include the header file <stdio.h> between %{ and %}.
* Declare and initialize line, word, and character counters.
* Define regular expressions for newline, whitespace, and words.
* Use translation rules to update the respective counters.
* Call yylex() inside the main() function.
* Print the final count of lines, words, and characters.
* Save the program as counter.l.
* Run the program using the LEX compiler to generate lex.yy.c.
* Compile lex.yy.c using a C compiler to produce the executable.
* Run the executable to perform the counting operation on input.

**Code:**

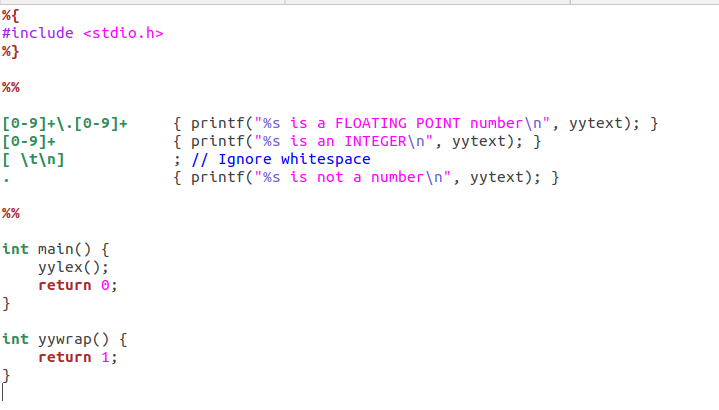


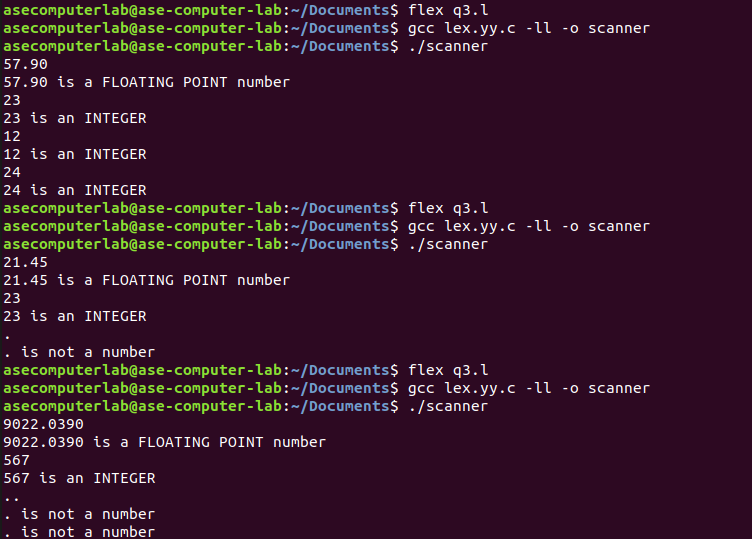
**Output:** 

3. **Aim:** Program to Recognize Integers and Floating-Point Numbers  
  **Algorithm:**

* Open the gedit text editor from Accessories under Applications menu.
* Include the header file <stdio.h> between %{ and %}.
* Define patterns for floating point numbers, integers, whitespaces, and other characters.
* Use translation rules to identify and print whether input is float, integer, or not a number.
* Ignore whitespaces like tab, space, and newline.
* Call yylex() inside the main() function to start lexical analysis.
* Save the program as numcheck.l.
* Run the program using the LEX compiler to generate lex.yy.c.
* Compile lex.yy.c using a C compiler to get the executable.
* Run the executable to test inputs and identify the type of number.

**Code:**

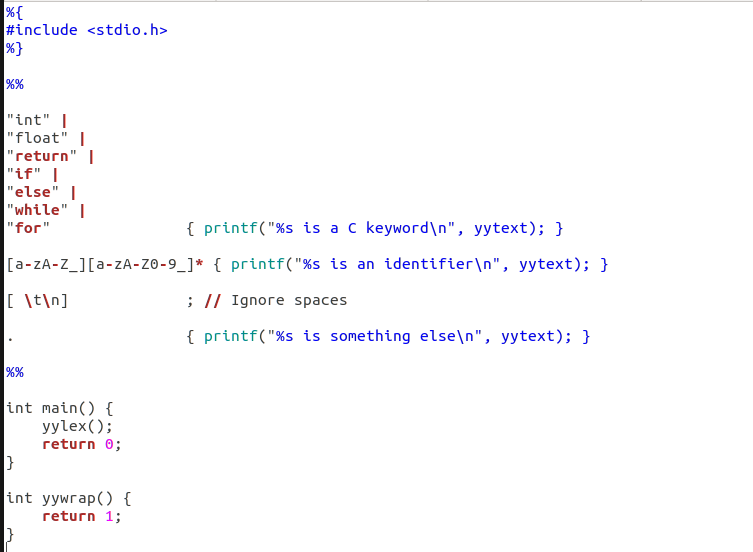


**Output:** 

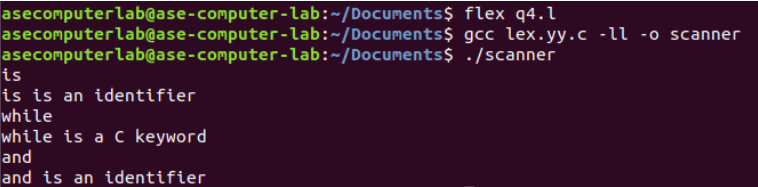
4. **Aim:** Program to Recognize C Keywords  
 **Algorithm:**

* Open the gedit text editor from Accessories under Applications menu.
* Include the header file <stdio.h> between %{ and %}.
* Define regular expressions for C keywords, identifiers, whitespaces, and other characters.
* Use translation rules to print whether input is a C keyword, identifier, or something else.
* Ignore spaces, tabs, and newline characters.
* Call yylex() in the main() function to begin lexical analysis.
* Save the program as keywordid.l.
* Run the program through the LEX compiler to generate lex.yy.c.
* Compile lex.yy.c using a C compiler to get the final executable.
* Run the executable to classify each token as keyword, identifier, or other.

**Code:**



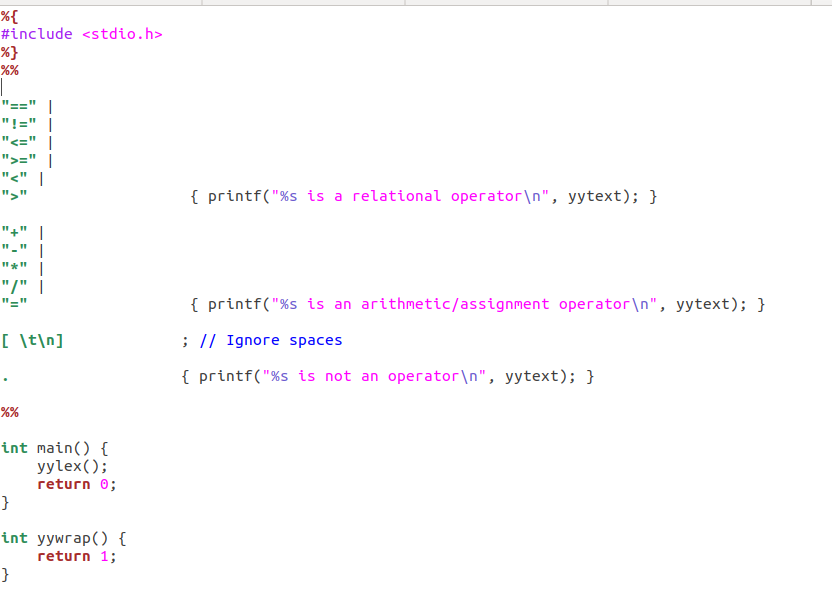
**Output:**



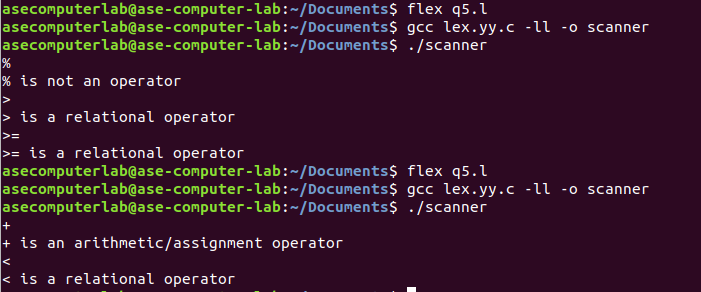
5. **Aim:** Program to Recognize Operators  
  **Algorithm:**

* Open the gedit text editor from Accessories under Applications menu.
* Include the header file <stdio.h> between %{ and %}.
* Define regular expressions for relational operators, arithmetic/assignment operators, whitespaces, and other characters.
* Use translation rules to check and print whether input is a relational operator, arithmetic/assignment operator, or not an operator.
* Ignore whitespaces like tab and newline characters.
* Call yylex() inside the main() function to begin lexical analysis.
* Save the program as operatorcheck.l.
* Run the program through the LEX compiler to generate lex.yy.c.
* Compile lex.yy.c using a C compiler to get the executable.
* Run the executable to test and classify the input operators.

**Code:**



**Output:**



**EXPERIMENT NO – 1**

**Aim:** To implement Lexical Analyzer Using Lex Tool

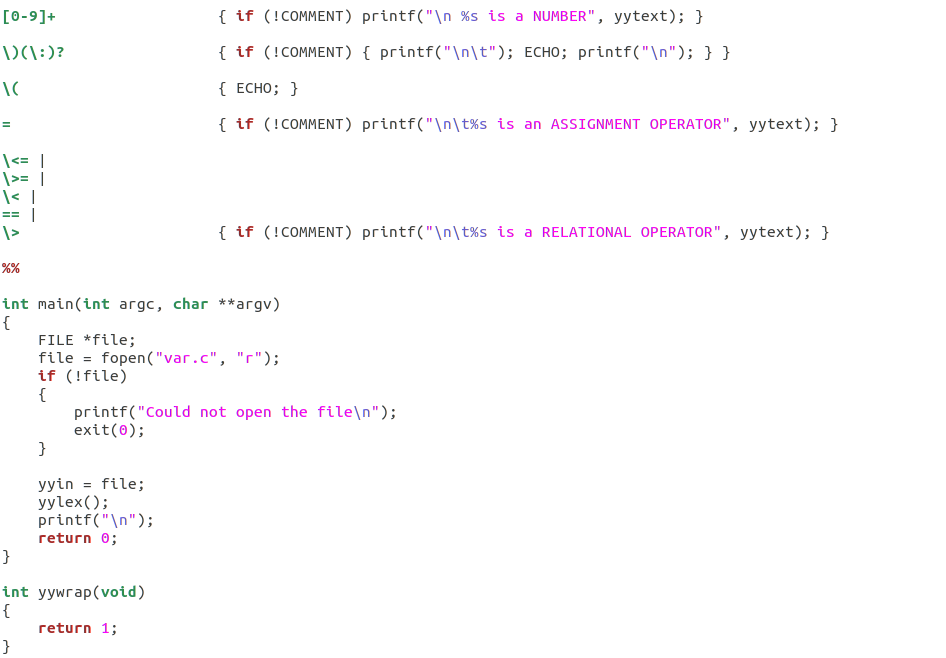
**Algorithm:**

* Open gedit text editor from Accessories in Applications.
* Specify the header files to be included inside the declaration part (i.e. between %{ and %}).
* Define the digits 0-9 and identifiers a-z and A-Z.
* Using translation rules, define the regular expressions for digit, keywords, identifiers, operators, header files etc. If matched with the input, store and display using yytext.
* Inside procedure main (), use yyin() to point to the current file being passed by the lexer.
* The specification of the lexical analyzer is prepared by creating a program lab1.l in the LEX language.
* The lab1.l program is run through the LEX compiler to produce equivalent C code named lex.yy.c.
* The program lex.yy.c consists of a table constructed from the regular expressions of lab1.l, along with standard routines that use the table to recognize lexemes.
* Finally, the lex.yy.c program is run through a C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

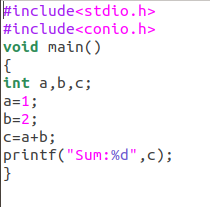
**Code:**

**Lab1.l:**

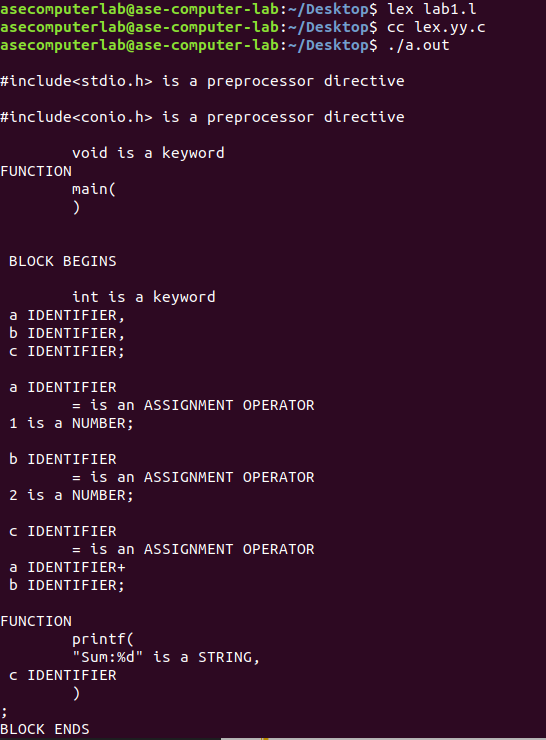




**Var.c:**



**Output:**



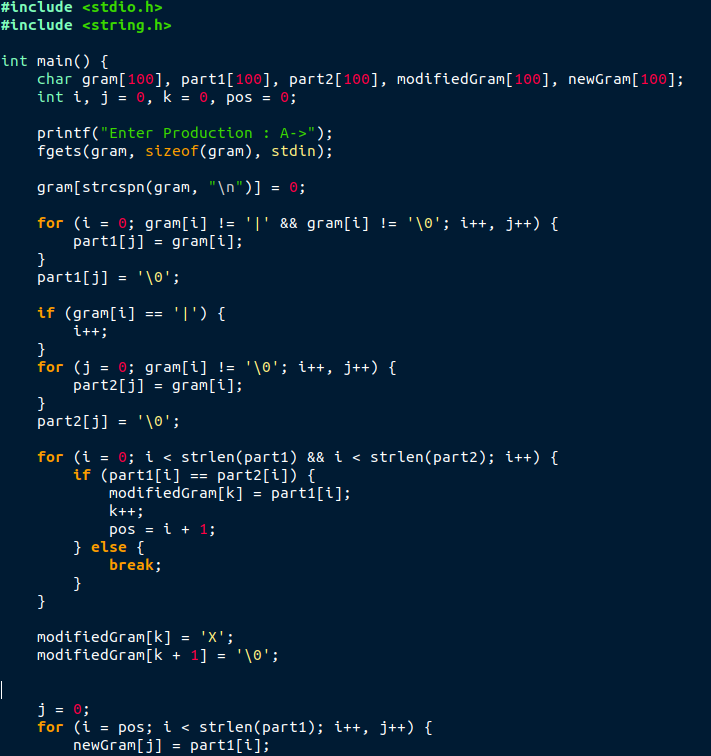
**EXPERIMENT NO – 2**

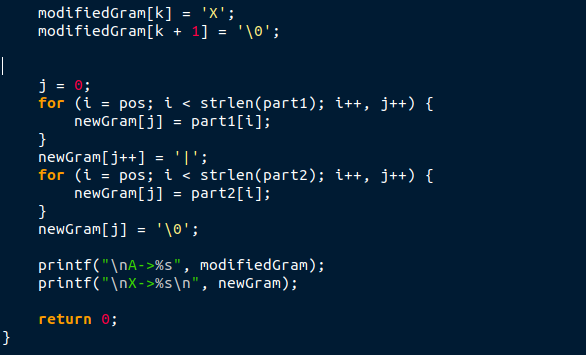
**Aim:** Program to eliminate left recursion and factoring from the given grammar

**Algorithm:**

* Open any text editor and start writing a C program.
* Include the necessary header files: stdio.h and string.h.
* Declare required character arrays for grammar parts and variables for loop counters and positions.
* Prompt the user to enter a production in the form A->alpha|beta.
* Use fgets() to read the entire input line, removing the trailing newline.
* Extract the portion before the | into part1 and the portion after into part2.
* Find the longest common prefix between part1 and part2 and store it in modifiedGram.
* After the common part, append 'X' to modifiedGram to denote the new non-terminal.
* Create newGram to store the restructured productions from the remaining suffixes of part1 and part2.
* Display the final left-factored productions using printf().

**Code:**





**Output:**

