

## Laboratory 1

Title of the Laboratory Exercise: Program to count the number of vowels and consonants in a given string

### 1. Introduction and Purpose of Experiment

Students learn to use Lex program to find out vowels and consonants in a given string

### 2. Aim and Objectives

Aim

- To write a program to count the number of vowels and consonants in a given string

Objectives

At the end of this lab, the student will be able to

- Define regular expression for vowels and consonants
- Count the number of vowels and consonants

### 3. Experimental Procedure

Students are required to carry out the following steps:

- Algorithm
- Write the Lex program
- Compile and execute the program (steps)
- Complete the documentation for the given problem

### 4. Presentation of Results

Algorithm:

**count\_vowels\_consonants() :**

1. if `regex_match([aeiouAEIOU])`
2.     `vowel++`
3. else if `regex_match([a-zA-Z])`
4.     `consonant++`

## 5. print vowels and consonants

**main.cpp**

```
#include <iostream>

int vowel_cnt = 0;
int consonant_cnt = 0;

extern int yylex();

int main(int argc, char* argv[]) {
    std::cout << "Enter your stream of characters"
               << "\n";

    yylex();
    std::cout << "Found " << vowel_cnt << " Vowels and " << consonant_cnt << " Consonants"
               << "\n";

    return 0;
}
```

**tokens.l**

```
%{
#include <iostream>

extern int vowel_cnt;
extern int consonant_cnt;

void yyerror(const char *s);

}%

%option noyywrap

%%

[aeiouAEIOU]    { std::cout << "[VOWEL: " << yytext << "]; vowel_cnt++; }
[a-zA-Z]        { std::cout << "[CONSONANT: " << yytext << "]; consonant_cnt++; }
\n              { std::cout << "\nEND PARSE [NEWLINE]\n\n"; return 1; }
.               { std::cout << "[UNRECOGNIZED: "<< yytext << " ]"; }

%%
```

**CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.0.0)
project(Lab01 VERSION 0.1.0)

include(CTest)
enable_testing()

# setup FLEX
FIND_PACKAGE(FLEX)
```

```

FLEX_TARGET(Scanner tokens.l ${CMAKE_CURRENT_BINARY_DIR}/tokens.cpp)

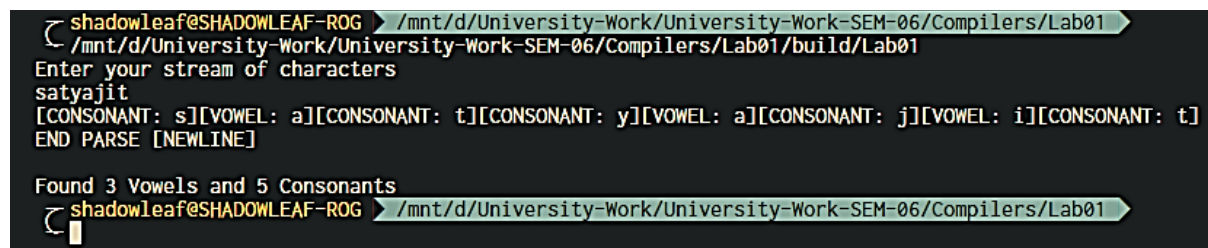
add_executable(Lab01 main.cpp tokens.l ${FLEX_Scanner_OUTPUTS})
target_include_directories(Lab01 PUBLIC ${CMAKE_CURRENT_BINARY_DIR})

set(CPACK_PROJECT_NAME ${PROJECT_NAME})
set(CPACK_PROJECT_VERSION ${PROJECT_VERSION})
include(CPack)

```

## 5. Analysis and Discussions

### OUTPUT:



```

shadowleaf@SHADOWLEAF-ROG > /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab01
/mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab01/build/Lab01
Enter your stream of characters
satyajit
[CONSONANT: s][VOWEL: a][CONSONANT: t][CONSONANT: y][VOWEL: a][CONSONANT: j][VOWEL: i][CONSONANT: t]
END PARSE [NEWLINE]

Found 3 Vowels and 5 Consonants
shadowleaf@SHADOWLEAF-ROG > /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab01

```

Figure 0-1 Output for "satyajit"

### Explanation:

The regex for matching the vowels is `[aeiouAEIOU]` which will match all upper case and lower-case vowels, and the rest of the letters `[a-zA-Z]` will be consonants, i.e. if the vowel regex does not match and the `a-zA-Z` does, then it is a consonant. Once the regex matches, the corresponding count is incremented. When a newline is encountered, the parsing is stopped and the output is displayed.

## 6. Conclusions

Flex is a tool for generating scanners: programs which recognize lexical patterns in text. The flex input file consists of three sections, separated by a line with just `%%` in it:

```

definitions
%%
rules
%%
user code

```

The definitions section contains declarations of simple name definitions to simplify the scanner specification, and declarations of start conditions, which are explained in a later section.

Name definitions have the form:

```
name definition
```

The rules section of the flex input contains a series of rules of the form:

```
pattern  action
```

where the pattern must be unindented and the action must begin on the same line.

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the flex input file is chosen.

Once the match is determined, the text corresponding to the match (called the token) is made available in the global character pointer `yytext`, and its length in the global integer `yylen`. The action corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.

If no match is found, then the default rule is executed: the next character in the input is considered matched and copied to the standard output.

For a really detailed documentation refer to: <http://dinosaur.compilertools.net/flex/manpage.html>

## 7. Comments

### a. Limitations of Experiments

The experiment does not define the input file i.e. either STDIN or some other user file, the buffer size isn't specified either.

### b. Limitations of Results

The results are limited to very few test cases, which could have been increased to justify the regex used in the program.

### c. Learning happened

The concept of regex to detect patterns was learnt, along with lex, which was used to perform some action with the matched strings.

### d. Recommendations

Perform extensive testing on different forms of regex to better understand the different forms of regex that match the same string.

Component	Max Marks	Marks Obtained
Viva	6	
Results	7	
Documentation	7	
Total	20	