COLLEGE OF ENGINEERING, GOTHAM
GOTHAM

COMPUTER SCIENCE AND ENGINEERING
WAYNE ENTERPRISES LAB

G013

LAB REPORT

CERTIFIED BONAFIDE RECORD OF WORK DONE BY

BRUCE WAYNE

University Reg. No: GTM16CSXXX

Roll No. : 13

Class : S8 - CSE

Staff In-charge:
Lucius Fox
Professor
Department of Computer Science & Engineering

Contents

1 Lexical Analyser

1.1 Aim

Design and Implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new line.

1.2 Theory

The very first phase of a compiler deals with lexical analysis. A lexical analyzer, also known as scanner, converts the high level input program into a sequence of **tokens**. A lexical token is a sequence of characters which is treated as a unit in the grammar of the programming languages. The common type of tokens include:

- **Keywords**: A keyword is a word reserved by a programming language having a special meaning.
- **Identifiers**: It is a user-defined name used to uniquely identify a program element. It can be a class, method, variable, namespace etc.
- **Operators**: It is a symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operation and produce final result.
- **Separators**: Separators are used to separate one programming element from the other.
- **Literals**: A literal is a notation for representing a fixed value and do not change during the course of execution of the program.

1.3 Algorithm

Algorithm 1: Algorithm for the Client

```
START
Get the input file and read from the file word by word.
Split the word into meaningful tokens with the help of delimiters
Read each token one by one
    If token is a keyword
        print <token, keyword>
    If token is an operator
        print <token, operator>
    If token is a seperator/ delimiter
        print <token, delimiter>
    If token is a literal
        print <token, literal>
    If token is an identifier
        print <token, identifier>
STOP
```

1.4 Code

Lexical Analyser - Code

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define keyword_count 13
#define operator_count 7
#define symbol_count 17

char *keywords[20] = {"int", "float", "double", "char", "return", "
   switch", "for", "if",
                      "while", "do", "continue", "break", "goto"};
char operators[15] = {'+', '-', '*', '/', '^', '%', '='};
char symbols[25] = {';', ':', '{', '}', '(', ')', ',', '?', '+', '-
   ', '*', '/', '^', '%', '=',
                   '[', ']'};

int isOperator(char s[]) {
        for(int i = 0; i < operator_count; i++) {
                if(operators[i] == s[0])
                        return 1;
        }
        return 0;
}

int isSymbol(char s[], int len) {
        if(len == 1) {
                for(int i = 0; i < symbol_count; i++) {
                        if(symbols[i] == s[0])
                                return 1;
                }
        }
        return 0;
}

int isSymbolChar(char s) {
        for(int i = 0; i < symbol_count; i++) {
                if(symbols[i] == s)
                        return 1;
        }
        return 0;
}

void preprocess(char *s, char **tokens, int *t_c) {
        int curr_len = 0;
        for(int i = 0; s[i] != '\0'; i++) {
                if(isSymbolChar(s[i])) {
                        tokens[*t_c][curr_len++] = '\0';

                        curr_len = 0;   (*t_c)++;

                        tokens[*t_c][curr_len++] = s[i];
                        tokens[*t_c][curr_len] = '\0';
                        curr_len = 0;   (*t_c)++;
                }

                else {
                        tokens[*t_c][curr_len++] = s[i];
                }
        }
        (*t_c)++;
        return;
}

int isKeyword(char s[]) {
        for(int i = 0; i < keyword_count; i++) {
                if(strcmp(keywords[i],s) == 0) {
                        return 1;
                }
        }
        return 0;
}

int isIdentifier(char s[]) {
        if(!isalpha(s[0]) && s[0] != '_') {
                return 0;
        }
        for(int i = 1; i < s[i] != '\0'; i++) {
                if(!isalpha(s[i]) && !isdigit(s[i]) && s[i] != '_')
    {
                        return 0;
                }
        }
        return 1;
}

int isLiteral(char s[], int len) {

        if(s[0] == '\'' && s[len-1] == '\'' || s[0] == '"' && s[len
   -1] == '"') {
                return 1;
        }
        else if(isdigit(s[0])) {
                for(int i = 0; s[i] != '\0'; i++) {
                        if(!isdigit(s[i])) {
                                return 0;
                        }
                }
                return 1;
        }
}

int main() {
        char **tokens = (char**)malloc(sizeof(char*)*50);
        for(int i = 0; i < 50; i++) {
                tokens[i] = (char*)malloc(sizeof(char)*10);
        }

        int token_count = 0;
        char s[100];
        FILE *fp = fopen("input.c", "r");

        while(fscanf(fp, "%s", s) == 1) {
                //preprocess the input line
                preprocess(s, tokens, &token_count);
        }

        for(int i = 0; i < token_count; i++) {
                if(isKeyword(tokens[i])) {
                        printf("< %s\t\t, keyword\t>\n", tokens[i])
   ;
                }
                else if(isOperator(tokens[i])) {
                        printf("< %s\t\t, operator\t>\n", tokens[i
   ]);
                }
                else if(isSymbol(tokens[i], strlen(tokens[i]))) {
                        printf("< %s\t\t, symbol\t>\n", tokens[i]);
                }
                else if(isLiteral(tokens[i], strlen(tokens[i]))) {
                        printf("< %s\t\t, literal\t>\n", tokens[i])
   ;
                }
                else if(isIdentifier(tokens[i])) {
                        printf("< %s\t\t, identifier\t>\n", tokens[
   i]);
                }
        }
        return 0;
}
```

1.5 Input and Output

1.5.1 Input

```
{
int a, b;
printf(a);
printf("Hey_Mr");
return 0;
}
```

1.5.2 Output

```
< int    , keyword >
< main    , identifier >
< (    , symbol     >
< )    , symbol     >
< {    , symbol     >
< int      , keyword >
< a    , identifier >
< ,   , symbol     >
< b    , identifier >
< ;    , symbol     >
< printf   , identifier >
< (    , symbol     >
< a    , identifier >
< )    , symbol     >
< ;    , symbol     >
< printf   , identifier >
< (    , symbol     >
< "Hey_Mr" , literal    >
< )    , symbol     >
< ;    , symbol     >
< return   , keyword >
< 0    , literal >
< ;    , symbol     >
< }    , symbol     >
```

1.6 Result

Implemented the program for implemeting lexical analyser using C and was compiled using gcc version 5.4.0, and executed in ubuntu 16.04 with kernel and the above output was obtained.