Department of CSE
SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VI

22 January 2021

UCS 1602 - Compiler Design

Exercise 1: Lexical Analyser Using C

Aim:

To write a program using C to perform the basic functionalities of a **Lexical Analyser**.

Code:

```c
/*  C Program that performs a basic lexical analysis of a given string  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>

int isOperator(char ch);
int isSeparator(char ch);
int isDelimiter(char ch);
int isValidIdentifier(char *str);
int isInteger(char *str);
int isKeyword(char *str);
int isPreprocessorDirective(char ch);
char *subString(char *str, int start, int end);
int printOperator(char ch1, char ch2);
int lexicalParse(char *str);

int main(void){
    int status = 0, len, fp;
    char text[10000], file[100];

    printf("\n\t\t\tLexical Analyser Using C\n");
    printf("\n\t\tEnter file name to parse: ");
    scanf("%[^\n]", file);

    fp = open(file, O_RDONLY);

    if(fp < 0){
        printf("\nError: File does not exist.\n");
        return 0;
    }

    len = read(fp, text, 10000);
    close(fp);

    printf("\nText to be parsed:\n\n%s\n", text);

    status = lexicalParse(text);

    if(status){
        printf("\n\n\t\tThe given expression is lexically valid.\n");
    }

    else{
        printf("\n\n\t\tThe given expression is lexically invalid.\n");
    }

    return 0;
}

int isOperator(char ch){
    //Checks if the character is a valid operator

    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=' || ch == '%' || ch == '!' ){
            return 1;
        }

    return 0;
}

int isSeparator(char ch){
    //Checks if the character is a valid separator

    if (ch == ';'|| ch == '{' || ch == '}' || ch == ','){
            return 1;
        }

    return 0;
}

int isDelimiter(char ch){
    //Checks if the character is a valid delimiter

    if (ch == ' ' || ch == '(' || ch == ')'
        || isSeparator(ch) == 1 || isOperator(ch) == 1){
            return 1;
        }

    return 0;
}

int isValidIdentifier(char *str){
    //Checks if the character is a valid identifier

    if(isdigit(str[0]) > 0 || isDelimiter(str[0]) == 1){
        //First character shouldn't be a digit or a special character
        return 0;
    }

    return 1;
}

int isInteger(char *str){
    //Checks if the string is a valid integer

    int i = 0, len = strlen(str);

    if(!len){
        return 0;
    }

    for(i = 0; i < len; i++){
        if(!isdigit(str[i])){
            return 0;
        }
    }

    return 1;
}

int isKeyword(char *str){
    //Checks if the string is a valid keyword

    if(!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while") ||
        !strcmp(str, "for") || !strcmp(str, "do") || !strcmp(str, "break") ||
        !strcmp(str, "switch") || !strcmp(str, "continue") || !strcmp(str, "return") ||
        !strcmp(str, "case") || !strcmp(str, "default") || !strcmp(str, "void") ||
        !strcmp(str, "int") || !strcmp(str, "char") || !strcmp(str, "bool") ||
        !strcmp(str, "struct") || !strcmp(str, "goto") || !strcmp(str, "typedef") ||
        !strcmp(str, "unsigned") || !strcmp(str, "long") || !strcmp(str, "short") ||
        !strcmp(str, "float") || !strcmp(str, "double") || !strcmp(str, "sizeof")){
            return 1;
        }

    return 0;
}

int isPreprocessorDirective(char ch){
    //Checks if the string is a valid preprocessor directive

    if(ch == '#'){
        //Basic check, works for header files, macros and const declarations
        return 1;
    }
    return 0;
```

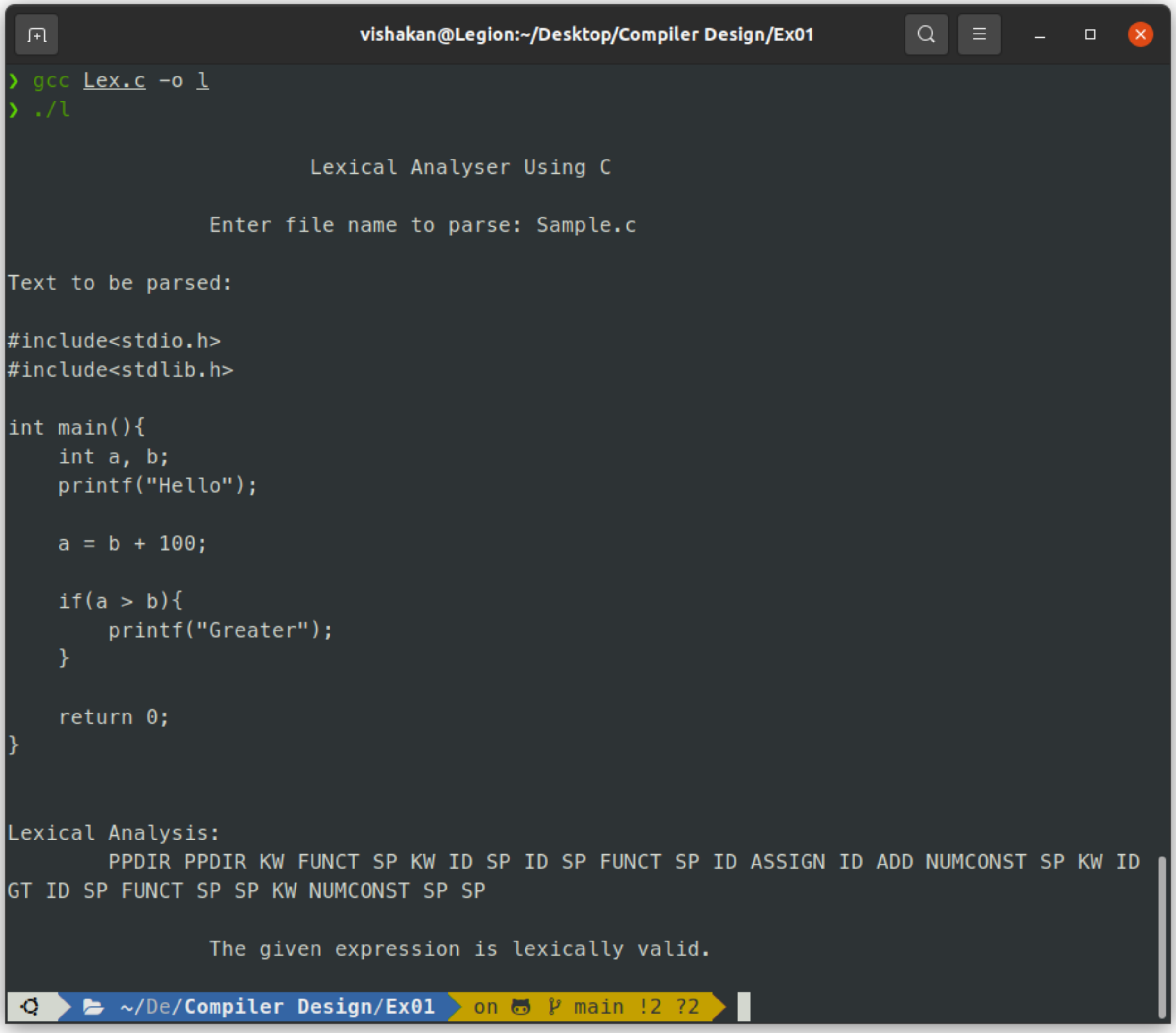
```
}

char *subString(char *str, int start, int end){
    //Get a substring from the given string
    int i = 0;
    char *sub = (char *)malloc(sizeof(char) * (end - start + 2));

    for(i = start; i <= end; i++){
        sub[i - start] = str[i];
    }

    sub[end - start + 1] = ’\0’;

    return sub;
}

int printOperator(char ch1, char ch2){
    //Print the details of the parsed operator

    switch(ch1){
        case ’+’:
            if(ch2 == ’=’){
                printf("ASSIGN ");
            }
            else if(ch2 == ’ ’){
                printf("ADD ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;


        case ’-’:
            if(ch2 == ’=’){
                printf("SUB-ASSIGN ");
            }
            else if(ch2 == ’ ’){
                printf("SUB ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case ’*’:
            if(ch2 == ’=’){
                printf("PRODUCT-ASSIGN ");
            }
            else if(ch2 == ' '){
                printf("PRODUCT ");
            }
            else{
                printf("INVALID-OP");
                return 0;
            }
            break;

        case '/':
            if(ch2 == '='){
                printf("DIVISION-ASSIGN ");
            }
            else if(ch2 == ' '){
                printf("DIVISION ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case '%':
            if(ch2 == '='){
                printf("MODULO-ASSIGN ");
            }
            else if(ch2 == ' '){
                printf("MODULO ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case '=':
            if(ch2 == '='){
                printf("EQUALITY ");
            }
            else if(ch2 == ' '){
                printf("ASSIGN ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case '>':
            if(ch2 == '='){
                printf("GT-EQ ");
            }
            else if(ch2 == ' '){
                printf("GT ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case '<':
            if(ch2 == '='){
                printf("LT-EQ ");
            }
            else if(ch2 == ' '){
                printf("LT ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case '!':
            printf("NOT ");
            break;

        default:
            printf("INVALID-OP ");
            return 0;
    }

    return 1;
}

int lexicalParse(char *str){
    //Parse the given string to check for validity
    int left = 0, right = 0, len = strlen(str), status = 1, i;

    printf("\nLexical Analysis:\n\t");

    while(right <= len && left <= right){
        //While we are within the valid bounds of the string, check:

        while(isPreprocessorDirective(str[right]) == 1){
                //Check if string is preprocessor directive
                printf("PPDIR ");

                for(right; str[right] != '\n'; right++);
                right++;
                left = right;
```

```c
    }

    for(i = right; i < len; i++){
        //Clearing linebreaks & tabs to spaces for efficient processing
        if(str[i] == '\n' || str[i] == '\t'){
            str[i] = ' ';
        }
    }

    if(isDelimiter(str[right]) == 0){
        //If we do not encounter a delimiter, keep moving forward
        //"right" points to the next character
        right++;
    }

    else if(isDelimiter(str[right]) == 1 && left == right){
        //If it is a delimiter, and we haven't parsed it yet

        if(isSeparator(str[right]) == 1){
            //Check if the delimiter is a separator
            printf("SP ");
        }

        else if(isOperator(str[right]) == 1){
            //Check if the delimiter is an operator
            if((right + 1) <= len && isOperator(str[right + 1]) == 1){
                //Check if the next character is also an operator
                status = status & printOperator(str[right], str[right + 1]);
                right++;
            }

            else{
                //Next character is not an operator
                status = status & printOperator(str[right], ' ');
            }

            //printf("\n\t\t'%c' is an operator.", str[right]);
        }

        right++;
        left = right;
    }

    else if(str[right] == '(' && left != right || (right == len && left != right)){
        //Special case, to check for functions

        char *sub = subString(str, left, right - 1);

        if(isKeyword(sub) == 1){
            //Check if the function is a keyword based function, like "if" & "for"
            printf("KW ");
            left = right;
            continue;   //Go ahead with the next check
        }

        //Otherwise, its some other function, parse it.

        for(i = right + 1; i < len; i++){
            if(str[i] == ')'){
                //Finish parsing till the end of the block and break
                printf("FUNCT ");
                right = i + 1;
                left = right;
                status = status & 1;
                break;
            }
        }
    }

    else if(isDelimiter(str[right]) == 1 && left != right || (right == len && left != right)){
        //We encountered a delimiter in the "right" position, but left != right
        //thus a chunk of unparsed characters exist between left and right

        //Make a substring of the unparsed characters
        char *sub = subString(str, left, right - 1);

        if(isInteger(sub) == 1){
            //Check if substring is an integer
            printf("NUMCONST ");
        }
        else if(isKeyword(sub) == 1){
            //Check if substring is a keyword
            printf("KW ");
        }
        else if(isValidIdentifier(sub) == 1){
            //Check if substring is a valid identifier
            printf("ID ");
        }
        else if(isValidIdentifier(sub) == 0 && isDelimiter(str[right - 1]) == 0){
            //Otherwise, print that it is not a valid identifier
            status = status & 0;
            printf("INVALID-ID");
        }

```

```
            left = right;   //We have parsed the chunk, thus "left" = "right"
        }

    }

    return status;
}
```

Output - Valid Case:

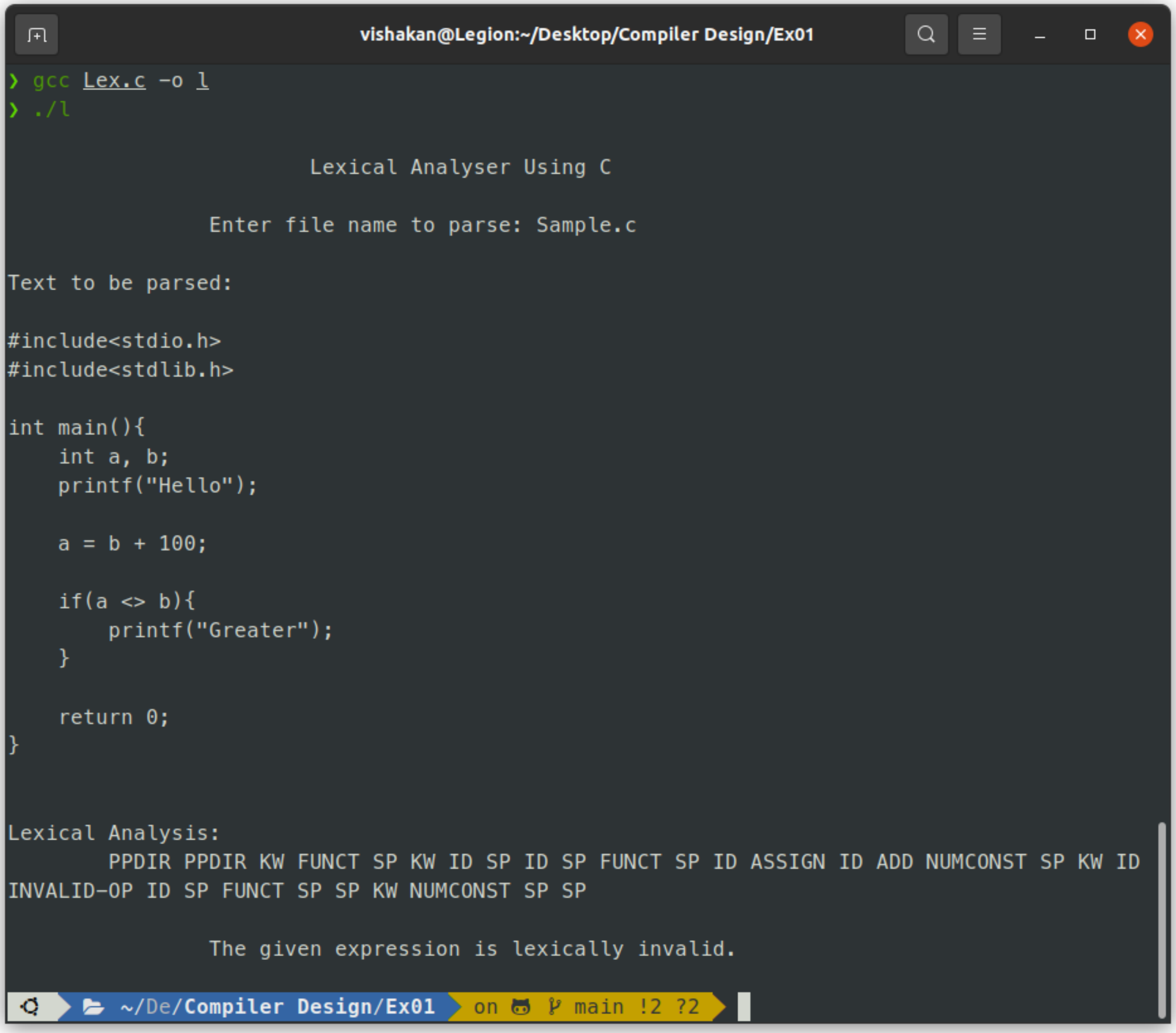
Figure 1: Console Output for a Valid Program.



```
❯ gcc Lex.c -o l
❯ ./l

                        Lexical Analyser Using C

                Enter file name to parse: Sample.c

Text to be parsed:

#include<stdio.h>
#include<stdlib.h>

int main(){
    int a, b;
    printf("Hello");

    a = b + 100;

    if(a > b){
        printf("Greater");
    }

    return 0;
}


Lexical Analysis:
        PPDIR PPDIR KW FUNCT SP KW ID SP ID SP FUNCT SP ID ASSIGN ID ADD NUMCONST SP KW ID
GT ID SP FUNCT SP SP KW NUMCONST SP SP

                The given expression is lexically valid.

 ~/De/Compiler Design/Ex01  on  main !2 ?2
```

Output - Invalid Case:

Figure 2: Console Output for an Invalid Program.



```
❯ gcc Lex.c -o l
❯ ./l

                         Lexical Analyser Using C

                 Enter file name to parse: Sample.c

Text to be parsed:

#include<stdio.h>
#include<stdlib.h>

int main(){
    int a, b;
    printf("Hello");

    a = b + 100;

    if(a <> b){
        printf("Greater");
    }

    return 0;
}


Lexical Analysis:
        PPDIR PPDIR KW FUNCT SP KW ID SP ID SP FUNCT SP ID ASSIGN ID ADD NUMCONST SP KW ID
INVALID-OP ID SP FUNCT SP SP KW NUMCONST SP SP

                 The given expression is lexically invalid.

  ~/De/Compiler Design/Ex01  on   main !2 ?2
```

Learning Outcome:

- From the experiment, I understood how a basic **Lexical Analyser** works.
- I was able to formulate ideas on how to implement recognition of specific tokens in programs for identification by the Lexical Analyser.
- I was able to implement simple regular expressions in C.
- I learnt how to parse a program for lexical validity, utilising the concept of **lexemes**.
- I was able to visualize the complexity that goes behind the compilation process and the significance of a Lexical Analyser phase in the compilation flow.