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>

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

int main(void){
    int status = 0;
    char str[100];

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

    status = lexicalParse(str);

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

    if (ch == ' ' || ch == ';' || ch == '(' || ch == ')'
        || ch == '{' || ch == '}' || ch == '=' || 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")
    ||
```

```c
        !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")){
            return 1;
        }

    return 0;
}

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

    if(str[0] == '#'){
        //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("\n\t\t'%c%c' is ADD/ASSIGNMENT operator.", ch1, ch2);
            }
            else if(ch2 == ' '){
                printf("\n\t\t'%c' is ADD operator.", ch1);
            }
            else{
                printf("\n\t\t'%c' is not a valid operator.", ch1);
                return 0;
            }
            break;


        case '-':
            if(ch2 == '='){
                printf("\n\t\t'%c%c' is SUBTRACT/ASSIGNMENT operator.", ch1, ch2);
            }
            else if(ch2 == ' '){
                printf("\n\t\t'%c' is SUBTRACT operator.", ch1);
            }
            else{
                printf("\n\t\t'%c' is not a valid operator.", ch1);
                return 0;
            }
            break;

        case '*':
            if(ch2 == '='){
                printf("\n\t\t'%c%c' is PRODUCT/ASSIGNMENT operator.", ch1, ch2);
            }
            else if(ch2 == ' '){
                printf("\n\t\t'%c' is PRODUCT operator.", ch1);
            }
            else{
                printf("\n\t\t'%c' is not a valid operator.", ch1);
                return 0;
            }
            break;

        case '/':
            if(ch2 == '='){
                printf("\n\t\t'%c%c' is DIVISION/ASSIGNMENT operator.", ch1, ch2);
            }
            else if(ch2 == ' '){
                printf("\n\t\t'%c' is DIVISION operator.", ch1);
            }
            else{
                printf("\n\t\t'%c' is not a valid operator.", ch1);
                return 0;
            }
            break;

        case '%':
            if(ch2 == '='){
                printf("\n\t\t'%c%c' is MODULO/ASSIGNMENT operator.", ch1, ch2);
            }
            else if(ch2 == ' '){
                printf("\n\t\t'%c' is MODULO operator.", ch1);
            }
            else{
                printf("\n\t\t'%c' is not a valid operator.", ch1);
                return 0;
            }
            break;

        case '=':
            if(ch2 == '='){
                printf("\n\t\t'%c%c' is EQUALITY operator.", ch1, ch2);
            }
            else if(ch2 == ' '){
                printf("\n\t\t'%c' is ASSIGNMENT operator", ch1);
            }
            else{
                printf("\n\t\t'%c' is not a valid operator.", ch1);
                return 0;
            }
            break;

        case '>':
            if(ch2 == '='){
                printf("\n\t\t'%c%c' is GREATER THAN/EQUAL TO operator.", ch1, ch2);
            }
            else if(ch2 == ' '){
                printf("\n\t\t'%c' is GREATER THAN operator.", ch1);
            }
            else{
                printf("\n\t\t'%c%c' is not a valid operator.", ch1, ch2);
                return 0;
            }
            break;

        case '<':
            if(ch2 == '='){
                printf("\n\t\t'%c%c' is LESSER THAN/EQUAL TO operator.", ch1, ch2);
            }
            else if(ch2 == ' '){
                printf("\n\t\t'%c' is LESSER THAN operator.", ch1);
            }
            else{
                printf("\n\t\t'%c%c' is not a valid operator.", ch1, ch2);
                return 0;
            }
            break;

        case ’!’:
            printf("\n\t\t’%c’ is a NOT operator.", ch1);
            break;

        default:
            printf("\n\t\t’%c’ is a not a valid operator.", ch1);
            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;

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

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

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

            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 = printOperator(str[right], str[right + 1]);
                    right++;
                }

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

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

            right++;
            left = right;
        }
```

```c

        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(isPreprocessorDirective(sub) == 1){
                //Check if substring is preprocessor directive
                printf("\n\t\t'%s' is a valid preprocessor directive.", sub);
            }
            else if(isValidIdentifier(sub) == 1){
                //Check if substring is a valid identifier
                printf("\n\t\t'%s' is a valid identifier.", sub);
            }
            else if(isInteger(sub) == 1){
                //Check if substring is an integer
                printf("\n\t\t'%s' is an integer.", sub);
            }
            else if(isKeyword(sub) == 1){
                //Check if substring is a keyword
                printf("\n\t\t'%s' is a valid keyword.", sub);
            }
            else if(isValidIdentifier(sub) == 0 && isDelimiter(str[right - 1]) == 0){
                //Otherwise, print that it is not a valid identifier
                status = 0;
                printf("\n\t\t'%s' is not a valid identifier.", sub);
            }

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

    }

    return status;
}
```

Output:

```
gcc Lex.c -o l
./l

            Lexical Analyser Using C

        Enter a string to parse: a + b = c

        'a' is a valid identifier.
        '+' is ADD operator.
        'b' is a valid identifier.
        '=' is ASSIGNMENT operator
        'c' is a valid identifier.

        The given expression is lexically valid.

gcc Lex.c -o l
./l

            Lexical Analyser Using C

        Enter a string to parse: a >! b == 2c

        'a' is a valid identifier.
        '>!' is not a valid operator.
        'b' is a valid identifier.
        '==' is EQUALITY operator.
        '2c' is not a valid identifier.

        The given expression is lexically invalid.
```