

Experiment No. 05

Class: TECMPNA
Name: Rebecca Dias

Date:27/03/21
Rollno: 19

AIM: TO DESIGN LEXICAL ANALYZER FOR A LANGUAGE WHOSE GRAMMAR IS KNOWN

THEORY:

- **Role of a lexical analyzer**

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

- **Compare Token, Patterns and Lexemes.**

- Tokens:

Sequence of characters that have a collective meaning.

- Patterns:

There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token

- Lexeme:

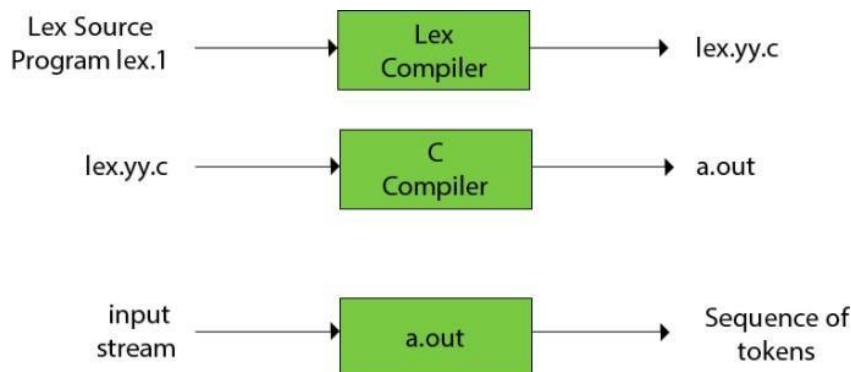
A sequence of characters in the source program that is matched by the pattern for a token.

- **Design of a lexical analyzer**

Lex is commonly used with the YACC parser generator. Lex is a program generator designed for lexical processing of character input streams. The code written in lex recognize these expressions in an input stream and partitions the

input stream in two strings matching the expressions. Lex helps in writing the program whose control flow is directed by instances of regular expressions in the input stream. The regular expressions are given to the lex and it associates the regular expressions and the program fragments. Lex is not a complete language, but rather a generation representing a new language feature which can be added two different programming languages called host languages. Lex can write code in different hosts languages.

The host language is used for the output code generated by lex and also for the program fragments added by the user. As lex is adaptable to various environments and users so it provides the component table runtime libraries for the different host languages. It accepts a high level, problem-oriented specification for character string matching and produces a program in a general-purpose language which recognizes regular expressions.



The function of Lex is as follows:

Firstly, lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.

Finally, C compiler runs the lex.yy.c program and produces an object program a.out. a.out is lexical analyzer that transforms an input stream into a sequence of tokens.

Given Grammar G:

LINE - If PHRASE then ACTION. LINE / ϵ

PHRASE - NOUN VERB NOUN

NOUN - (a-z)*

VERB - hate / like

ACTION - they NOUN

Input: "If dogs hate cats then they chase. If cats like milk then they drink. \$"

Output: (k) (n,1) (v) (n,2) (k) (a) (n,3) (op) (k) (n,2) (v) (n,4) (k) (a) (n,5) (op)

<eof>

Symbol table:

dogs	cats	chase	milk	drink
------	------	-------	------	-------

[1] [2] [3] [4] [5]

IMPLEMENTATION:

- **Pseudo code (Algorithm)**

1. Divide the program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error messages by providing row numbers and column numbers.

- **CODE (C)**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    char input[100]=""; // "If dogs hate cats then they chase. If cats like milk the
n they drink.";
    char noun[50][50], temp[50]="", op[]=".,!,:;?", output[50]="", t[5]="";
    int i = 0, j, n = 0, index, f = -1, k;
    // clrscr();
    for(index = 0; index < 50; index++)
    {
        strncpy(noun[index], "", 50);
```

```

}
index = 0;
printf("Input: ");
gets(input);
strlwr(input);
while(input[i] != '\0')
{
    j = 0;
    // If
    if(input[i] == 'i' && input[i + 1] == 'f')
    {
        output[n] = 'k';
        n++;
        while(input[i] != 'l' && input[i] != '\0' && input[i] != 'h')
        {
            i++;
        }
        // like
        if(input[i] == 'l' && input[i + 1] == 'i' && input[i + 2] == 'k' && input[i +
3] == 'e')
        {
            i -= 2;
            while(input[i] != ' ' && i != 0)
            {
                i--;
            }
            if(i != 0)
                i++;
            // before like
            while(input[i] != ' ' && input[i] != '\0')
            {
                temp[j] = input[i];
                i++;
                j++;
            }
            for(k = 0; k < index; k++)
            {
                if(!strcmp(noun[k], temp))
                {
                    f = k;
                    break;
                }
            }
            else
            {
                f = -1;
            }
        }
    }
}

```

```

    }
}
if(f == -1)
{
    strcat(noun[index], temp);
    strcat(noun[index], "\0");
    output[n] = index + '1';
    n++;
    index++;
    i += 6;
    strcat(temp, "\0");
}
else
{
    output[n] = f + '1';
    n++;
    i += 6;
}
strncpy(temp, " ", 50);
j = 0;
output[n] = 'v';
n++;
// after like
while(input[i] != ' ' && input[i] != '\0')
{
    temp[j] = input[i];
    i++;
    j++;
}
strcat(temp, "\0"); for(k
= 0; k < index; k++)
{
    if(!strcmp(noun[k], temp))
    {
        f = k;
        break;
    }
    else
    {
        f = -1;
    }
}
if(f == -1)
{
    strcat(noun[index], temp);

```

```

    strcat(noun[index], "\0");
    output[n] = index + '1';
    n++;
    index++;
}
else
{
    output[n] = f + '1';
    n++;
}
strncpy(temp, "", 50);
j = 0;
}
//hate
if(input[i] == 'h' && input[i + 1] == 'a' && input[i + 2] == 't' && input[i +
3] == 'e')
{
    i -= 2;
    while(input[i] != ' ' && i != 0)
    {
        i--;
    }
    if(i != 0)
        i++;
    // before hate
    while(input[i] != ' ' && input[i] != '\0')
    {
        temp[j] = input[i];
        i++;
        j++;
    }
    strcat(temp, "\0"); for(k
= 0; k < index; k++)
    {
        if(!strcmp(noun[k], temp))
        {
            f = k;
            break;
        }
        else
        {
            f = -1;
        }
    }
}
if(f == -1)

```

```

{
    strcat(noun[index], temp);
    strcat(noun[index], "\0");
    output[n] = index + '1';
    n++;
    index++;
    i += 6;
}
else
{
    output[n] = f + '1';
    n++;
    i += 6;
}
strncpy(temp, " ", 50);
j = 0;
output[n] = 'v';
n++;
while(input[i] != ' ' && input[i] != '\0')
{
    temp[j] = input[i];
    i++;
    j++;
}
strcat(temp, "\0"); for(k
= 0; k < index; k++)
{
    if(!strcmp(noun[k], temp))
    {
        f = k;
        break;
    }
    else
    {
        f = -1;
    }
}
if(f == -1)
{
    strcat(noun[index], temp);
    strcat(noun[index], "\0");
    output[n] = index + '1';
    n++;
    index++;
}

```

```

else
{
    output[n] = f + '1';
}
strncpy(temp, "", 50);
j = 0;
}
}
// then
if(input[i] == 't' && input[i + 1] == 'h' && input[i + 2] == 'e' && input[i + 3
] == 'n')
{
    i += 5;
    output[n] = 'k';
    n++;
    if(input[i] == 't' && input[i + 1] == 'h' && input[i + 2] == 'e' && input[i +
3] == 'y')
    {
        i += 5;
        output[n] = 'a';
        n++;
        while(input[i] != ' ' && input[i] != '\0')
        {
            t[0] = input[i];
            if(strstr(op, t) == NULL)
            {
                temp[j] = input[i];
                i++;
                j++;
            }
            else
            {
                i++;
            }
        }
        strcat(temp, "\0");
        for(k = 0; k < index; k++)
        {
            if(!strcmp(noun[k], temp))
            {
                f = k;
                break;
            }
        }
        else
        {

```



```

        f = -1;
    }
}
if(f == -1)
{
    strcat(noun[index], temp);
    //strcat(noun[index], "|");
    strcat(noun[index], "\0");
    output[n] = index + '1';
    n++;
    output[n] = 'o';
    n++;
    output[n] = 'p';
    n++;
    index++;
}
else
{
    output[n] = f + '1';
    n++;
}
strncpy(temp, " ", 50);
j = 0;
}
}
i++;
}
i = 0;
printf("Output: ");
while(output[i] != '\0')
{
    if(output[i] == 'o')
    {
        printf("(%c%c)", output[i], output[i+1]);
        i += 2;
    }
    else if(output[i] == 'k' || output[i] == 'v' || output[i] == 'a')
    {
        printf("(%c)", output[i]);
        i++;
    }
    else
    {
        printf("(n-%c)", output[i]);
        i++;
    }
}

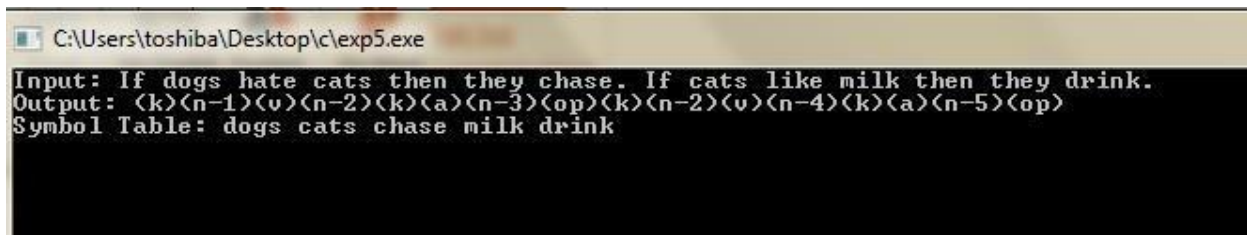
```

```

    }
}
printf("\n");
printf("Symbol Table: ");
for(i = 0; i < index; i++)
printf("%s ", noun[i]);
getch();
}

```

- **OUTPUT:**



```

C:\Users\toshiba\Desktop\c\exp5.exe
Input: If dogs hate cats then they chase. If cats like milk then they drink.
Output: <k><n-1><v><n-2><k><a><n-3><op><k><n-2><v><n-4><k><a><n-5><op>
Symbol Table: dogs cats chase milk drink

```

CONCLUSION:

A lexical analyzer is designed to create tokens for strings generated by the given grammar.