# Name :- DIWANSHU KUMAR Registration No :- RA1911003010473

**Date: 28-01-2022**

**COMPILER DESIGN LAB**

**EXPERIMENT- 1**

**1.1 OBJECTIVE:**

Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Simulate the same in C language.

**1.2 RESOURCE:**

Turbo C ++

**1.3 PROGRAM LOGIC:**

1. Read the input Expression

2. Check whether input is alphabet or digits then store it as identifier

3. If the input is is operator store it as symbol

4. Check the input for keywords

**1.4 PROCEDURE:**

Go to debug -> run or press CTRL + F9 to run the program

**1.5 PROGRAM:**

#include <fstream>

#include <iostream>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

using namespace std;

bool isPunctuator(char ch)

{

if (ch == ' ' || ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == ',' || ch == ';' || ch == '>' ||

ch == '<' || ch == '=' || ch == '(' || ch == ')' ||

ch == '[' || ch == ']' || ch == '{' || ch == '}' ||

ch == '&' || ch == '|' || ch == '^')

{

return true;

}

return false;

}

bool validIdentifier(char\* str)

{

if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||

str[0] == '3' || str[0] == '4' || str[0] == '5' ||

str[0] == '6' || str[0] == '7' || str[0] == '8' ||

str[0] == '9' || isPunctuator(str[0]) == true)

{

return false;

}

int i,len = strlen(str);

if (len == 1)

{

return true;

}

else

{

for (i = 1 ; i < len ; i++)

{

if (isPunctuator(str[i]) == true)

{

return false;

}

}

}

return true;

}

bool isOperator(char ch)

{

if (ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == '>' || ch == '<' ||

ch == '=' || ch == '|' || ch == '&' ||

ch == '^')

{

return true;

}

return false;

}

bool isKeyword(char \*str)

{

if (!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while") || !strcmp(str, "do") || !strcmp(str, "break") || !strcmp(str, "continue") || !strcmp(str, "int") || !strcmp(str, "double") || !strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str, "char") || !strcmp(str, "case")|| !strcmp(str, "long") || !strcmp(str, "short")|| !strcmp(str, "typedef") || !strcmp(str, "switch")|| !strcmp(str, "unsigned") || !strcmp(str, "void")|| !strcmp(str, "static") || !strcmp(str, "struct")|| !strcmp(str, "sizeof") || !strcmp(str,"long")|| !strcmp(str, "volatile") || !strcmp(str, "typedef")|| !strcmp(str, "enum") || !strcmp(str, "const")|| !strcmp(str, "union") || !strcmp(str, "extern")|| !strcmp(str,"bool"))

{

return true;

}

else

{

return false;

}

}

bool isNumber(char\* str)

{

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

if (len == 0)

{

return false;

}

for (i = 0 ; i < len ; i++)

{

if (numOfDecimal > 1 && str[i] == '.')

{

return false;

} else if (numOfDecimal <= 1)

{

numOfDecimal++;

}

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' || (str[i] == '-' && i > 0))

{

return false;

}

}

return true;

}

char\* subString(char\* realStr, int l, int r)

{

int i;

char\* str = (char\*) malloc(sizeof(char) \* (r - l + 2));

for (i = l; i <= r; i++)

{

str[i - l] = realStr[i];

str[r - l + 1] = '\0';

}

return str;

}

void parse(char\* str)

{

int left = 0, right = 0;

int len = strlen(str);

while (right <= len && left <= right) {

if (isPunctuator(str[right]) == false)

{

right++;

}

if (isPunctuator(str[right]) == true && left == right)

{

if (isOperator(str[right]) == true)

{

std::cout<< str[right] <<" is an operator\n";

}

right++;

left = right;

} else if (isPunctuator(str[right]) == true && left != right

|| (right == len && left != right))

{

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

if (isKeyword(sub) == true)

{

cout<< sub <<" is a keyword\n";

}

else if (isNumber(sub) == true)

{

cout<< sub <<" is a number\n";

}

else if (validIdentifier(sub) == true

&& isPunctuator(str[right - 1]) == false)

{

cout<< sub <<" is an identifier\n";

}

else if (validIdentifier(sub) == false

&& isPunctuator(str[right - 1]) == false)

{

cout<< sub <<" is not valid identifier\n";

}

left = right;

}

}

return;

}

int main()

{

char str[100];

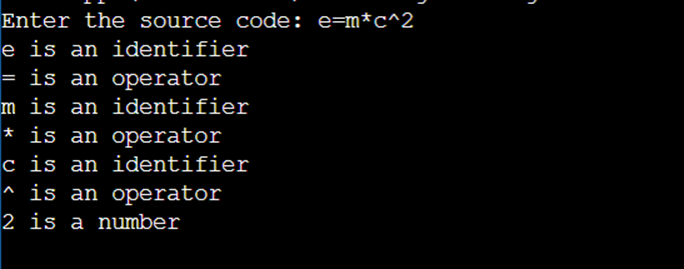
gets(str);

parse(str);

return 0;

}

**Output:**

****

**1.6 PRE LAB QUESTIONS**

**1. What is token?**

Ans- Tokens are the smallest elements of a program, which are meaningful to the compiler.The following are the types of tokens: Keywords, Identifiers, Constant, Strings, Operators, etc..

**2. What is lexeme?**

Ans- A lexeme is a sequence of alphanumeric characters in a token.

**3. What is the difference between token and lexeme?**

Ans-.A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.

**4. Define phase and pass?**

Ans- Pass : A pass refers to the traversal of a compiler through the entire program.

Phase : A phase of a compiler is a distinguishable stage, which takes input from the previous stage,

Processes and yields output that can be used as input for the next stage.

**5. What is the difference between phase and pass?**

Ans- Pass : A pass refers to the traversal of a compiler through the entire program.

Phase : A phase of a compiler is a distinguishable stage, which takes input from the previous stage,

Processes and yields output that can be used as input for the next stage

**6. What is the difference between compiler and interpreter?**

Ans- Compiler Interpreter

1. Compiler scans the whole program in one go. Translates program one statement at a time.

2. As it scans the code in one go, the errors (if any) Considering it scans code one

are shown at the end together. line at a time, errors are shown line by line.

**1.7 LAB ASSIGNMENT**

1. Write a program to recognize identifiers.

#include<stdio.h>

#include<conio.h>

#include<ctype.h>

void main()

{

char a[10];

int flag, i=1;

clrscr();

printf("\n Enter an identifier:");

gets(a);

if(isalpha(a[0]))

flag=1;

else

printf("\n Not a valid identifier");

while(a[i]!='\0')

{

if(!isdigit(a[i])&&!isalpha(a[i]))

{

flag=0;

break;

}

i++;

}

if(flag==1)

printf("\n Valid identifier");

getch();

}

2. Write a program to recognize constants.

bool isRealNumber(char\* str)

{

int i, len = strlen(str);

bool hasDecimal = false;

if (len == 0)

return (false);

for (i = 0; i < len; i++) {

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' && str[i] != '.' ||

(str[i] == '-' && i > 0))

return (false);

if (str[i] == '.')

hasDecimal = true;

}

return (hasDecimal);

}

bool isInteger(char\* str)

{

int i, len = strlen(str);

if (len == 0)

return (false);

for (i = 0; i < len; i++) {

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' || (str[i] == '-' && i > 0))

return (false);

}

3. Write a program to recognize keywords and identifiers.

The function below would check for keywords:

bool isKeyword(char\* str)

{

if (!strcmp(str, "if") || !strcmp(str, "else") ||

!strcmp(str, "while") || !strcmp(str, "do") ||

!strcmp(str, "break") ||

!strcmp(str, "continue") || !strcmp(str, "int")

|| !strcmp(str, "double") || !strcmp(str, "float")

|| !strcmp(str, "return") || !strcmp(str, "char")

|| !strcmp(str, "case") || !strcmp(str, "char")

|| !strcmp(str, "sizeof") || !strcmp(str, "long")

|| !strcmp(str, "short") || !strcmp(str, "typedef")

|| !strcmp(str, "switch") || !strcmp(str, "unsigned")

|| !strcmp(str, "void") || !strcmp(str, "static")

|| !strcmp(str, "struct") || !strcmp(str, "goto"))

return (true);

return (false);

}

The function below would check for identifiers

bool validIdentifier(char\* str)

{

if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||

str[0] == '3' || str[0] == '4' || str[0] == '5' ||

str[0] == '6' || str[0] == '7' || str[0] == '8' ||

str[0] == '9' || isDelimiter(str[0]) == true)

return (false);

return (true);

}

4. Write a program to ignore the comments in the given input source program.

#include <iostream>

using namespace std;

string removeComments(string prgm)

{

int n = prgm.length();

string res;

// Flags to indicate that single line and multiple line comments

// have started or not.

bool s\_cmt = false;

bool m\_cmt = false;

// Traverse the given program

for (int i=0; i<n; i++)

{

// If single line comment flag is on, then check for end of it

if (s\_cmt == true && prgm[i] == '\n')

s\_cmt = false;

// If multiple line comment is on, then check for end of it

else if (m\_cmt == true && prgm[i] == '\*' && prgm[i+1] == '/')

m\_cmt = false, i++;

// If this character is in a comment, ignore it

else if (s\_cmt || m\_cmt)

continue;

// Check for beginning of comments and set the appropriate flags

else if (prgm[i] == '/' && prgm[i+1] == '/')

s\_cmt = true, i++;

else if (prgm[i] == '/' && prgm[i+1] == '\*')

m\_cmt = true, i++;

// If current character is a non-comment character, append it to res

else res += prgm[i];

}

return res;

}

// Driver program to test above functions

int main()

{

string prgm = " /\* Test program \*/ \n"

" int main() \n"

" { \n"

" // variable declaration \n"

" int a, b, c; \n"

" /\* This is a test \n"

" multiline \n"

" comment for \n"

" testing \*/ \n"

" a = b + c; \n"

" } \n";

cout << "Given Program \n";

cout << prgm << endl;

cout << " Modified Program ";

cout << removeComments(prgm);

return 0;

}

**1.8 POST LAB QUESTIONS**

**1. What is a lexical analyzer?**

Ans- Lexical analysis is the first phase of a compiler. It takes

modified source code from language preprocessors that are

written in the form of sentences. The lexical analyzer breaks

these syntaxes into a series of tokens, by removing any

whitespace or comments in the source code,If the lexical analyzer finds a token invalid, it generates an

error. The lexical analyzer works closely with the syntax

analyzer. It reads character streams from the source code,

checks for legal tokens, and passes the data to the syntax

analyzer when it demands.

**2. Which phase of the compiler is used for lexical analyzer?**

The first phase of the compiler is the lexical analyzer, also known as the scanner, which recognizes the basic language units, called tokens. The exact characters in a token is called its lexeme.

**3. What is the output of Lexical analyzer?**

The output of the lexical analyzer is tokens

**4. What is the LEX source Program?**

Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). Lex is commonly used with the yacc parser generator. Lex reads an input stream specifying the lexical analyzer and writes source code which implements the lexical analyzer in the C programming language.

**1.9 INPUT & OUTPUT:**

**Input:**

Enter Program $ for termination:

{

int a[3],t1,t2;

t1=2; a[0]=1; a[1]=2; a[t1]=3;

t2=-(a[2]+t1\*6)/(a[2]-t1);

if t2>5 then

print(t2);

else {

int t3;

t3=99;

t2=-25;

print(-t1+t2\*t3); /\* this is a comment on 2 lines \*/

} endif

}

$

**Output:**

Variables : a[3] t1 t2 t3

Operator : - + \* / >

Constants : 2 1 3 6 5 99 -25

Keywords : int if then else endif

Special Symbols : , ; ( ) { }

Comments : this is a comment on 2 lines

3