LAB SESSION 2: LEXICAL ANALYSIS WITH ERROR REPORTING USING LEX

DATE:

AIM: To implement a lexical analyzer for a given programming language using Lex tool..

PROBLEM DEFINITION: Design and implement a Lexical Analyzer (Lexer) using the Lex tool for a new: **ExtendC programming language**, which extends the standard C language by introducing two additional datatypes:

1. phone: Represents a phone number.
2. email: Represents an email address.

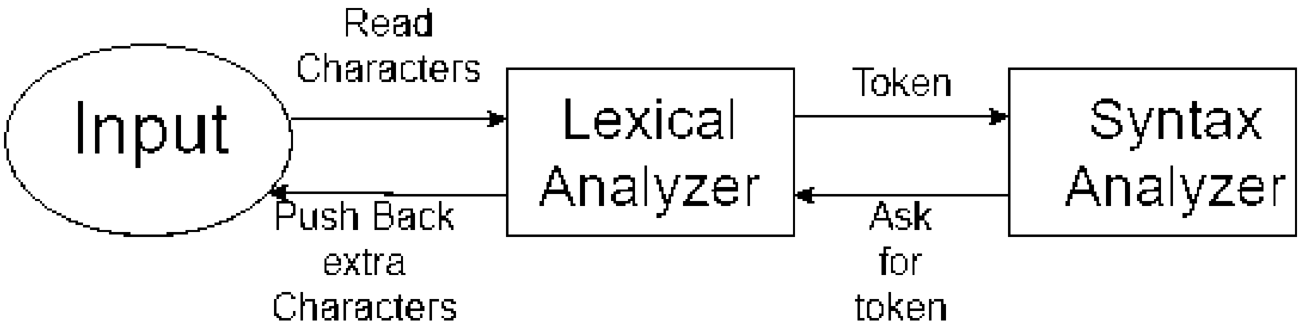
The lexical analyzer should correctly identify and tokenize the following components from an ExtendC source code file:

- Keywords: All standard C keywords (e.g., int, char, if, else, for, while, return, etc.), plus the new data types:
  - Phone
  - Email
- Identifiers: Names of variables, functions, etc., following standard C rules.
- Constants:
  - Integer constants
  - Floating-point constants
  - String constants (delimited by double quotes)
  - Character constants (single quotes)
- Special Tokens:
  - Phone Number: Should match the pattern of valid phone numbers (e.g., country code optional, 10 digits). Example valid formats:+91-9876543210, 9876543210
  - Email Address: Should match a valid email pattern as per standard email rules. Example valid formats:user@example.com, first.last@domain.co.in
  - Operators and Punctuators: +, -, *, /, =, ==, !=, <, >, <=, >=, {, }, (,), ;, ,
  - Comments:
    - Single-line comments: // comment text

        - Multi-line comments: /* comment text */
    - Whitespace and Newlines: Should be ignored except for counting line numbers for error reporting.

Expected Output: For each token identified in the input ExtendC source file, the lexical analyzer should print the token type and its lexeme.

THEORY: Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens.
Lexical Analysis can be implemented with the Deterministic finite Automata. The output is a sequence of tokens that is sent to the parser for syntax analysis



A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages. Example of tokens:

Type token (id, number, real, . . .)
Punctuation tokens (IF, void, return, . . .)
Alphabetic tokens (keywords)
Keywords; Examples-for, while, if etc.
Identifier; Examples-Variable name, function name, etc.
Operators; Examples '+', '++', '-' etc.
Separators; Examples ',' ';' etc
Example of Non-Tokens:Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

Steps in Lexical Analysis:

Input preprocessing: This stage involves cleaning up the input text and preparing it for lexical analysis. This may include removing comments, whitespace, and other non-essential characters from the input text.

Tokenization: This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.

Token classification: In this stage, the lexer determines the type of each token. For example, in a programming language, the lexer might classify keywords, identifiers, operators, and punctuation symbols as separate token types.

Token validation: In this stage, the lexer checks that each token is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.

Output generation: In this final stage, the lexer generates the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.

The lexical analyzer identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error.
Suppose we pass a statement through lexical analyzer – a = b + c ; It will generate token sequence like this: id=id+id;

Advantages:

Efficiency: Lexical analysis improves the efficiency of the parsing process because it breaks down the input into smaller, more manageable chunks. This allows the parser to focus on the structure of the code, rather than the individual characters.

Flexibility: Lexical analysis allows for the use of keywords and reserved words in programming languages. This makes it easier to create new programming languages and to modify existing ones.

Error Detection: The lexical analyzer can detect errors such as misspelled words, missing semicolons, and undefined variables. This can save a lot of time in the debugging process.

Code Optimization: Lexical analysis can help optimize code by identifying common patterns and replacing them with more efficient code. This can improve the performance of the program.

Disadvantages:

Complexity: Lexical analysis can be complex and require a lot of computational power. This can make it difficult to implement in some programming languages.

Limited Error Detection: While lexical analysis can detect certain types of errors, it cannot detect all errors. For example, it may not be able to detect logic errors or type errors.

Increased Code Size: The addition of keywords and reserved words can increase the size of the code, making it more difficult to read and understand.

Reduced Flexibility: The use of keywords and reserved words can also reduce the flexibility of a programming language. It may not be possible to use certain words or phrases in a way that is intuitive to the programmer.

PROGRAM:

Lex Code:

```
%{
#include <stdio.h>
#include <string.h>
int lineno = 1;
%}

%x COMMENT

%%
("auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"phone"|"
email"|"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|"register"|"retur
n"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|"typedef"|"union"|"unsigned"
|"void"|"volatile"|"while") { printf("<KEYWORD, %s>\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]*   { printf("<IDENTIFIER, %s >\n", yytext); }

[0-9]+\.[0-9]+          { printf("<FLOAT, %s >\n", yytext); }
[0-9]+                  { printf("<INTEGER, %s >\n", yytext); }
\"([^"\n]|(\\.))*\"     { printf("<STRING, %s >\n", yytext); }
\'([^'\n]|(\\.))\'      { printf("<CHAR, %s >\n", yytext); }

(\+91-)?[0-9]{10}       { printf("<PHONE, %s >\n", yytext); }
[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,} { printf("<EMAIL, %s >\n", yytext); }

"=="|"!="|"<="|">="|"="|"+"|"-"|"*"|"/"|">"|"<"       {   printf("<OPERATOR,   %s   >\n",
yytext); }
"{"|"}"|"("|")"|";"|","               { printf("<PUNCTUATOR, %s >\n", yytext); }

"//".*                 { /* Ignore single-line comments */ }

"/*"                  { BEGIN(COMMENT); }
<COMMENT>"*/"           { BEGIN(INITIAL); }
<COMMENT>\n            { lineno++; }
<COMMENT>.             { /* Ignore inside comment */ }

[ \t\r]+              { /* Ignore whitespace (including CR) */ }
\n                  { lineno++; }
.                 { printf("ERROR: Unknown symbol '%s' at line %d\n", yytext, lineno); }
```

```
%%

int main() {
   yylex();
   return 0;
}

int yywrap() { return 1; }
```

ex.c file:

```
int x = 10;
float y = 20.5;
char ch = 'A';
phone p = +91-9876543210;
email e = user@example.com;

if (x < y) {
   x = x + 1;
}
return 0;
```

OUTPUT:

```
PS C:\Users\Joseph\Desktop\compiler design\expt1> cd ..
PS C:\Users\Joseph\Desktop\compiler design> cd expt2
PS C:\Users\Joseph\Desktop\compiler design\expt2> flex expt2a.l
PS C:\Users\Joseph\Desktop\compiler design\expt2> gcc lex.yy.c -o lex.exe
PS C:\Users\Joseph\Desktop\compiler design\expt2> Get-Content ex.c | ./lex.exe
<KEYWORD, int>
<IDENTIFIER, x >
<OPERATOR, = >
<INTEGER, 10 >
<PUNCTUATOR, ; >
<KEYWORD, float>
<IDENTIFIER, y >
<OPERATOR, = >
<FLOAT, 20.5 >
<PUNCTUATOR, ; >
```

```
<KEYWORD, char>
<IDENTIFIER, ch >
<OPERATOR, = >
<CHAR, 'A' >
<PUNCTUATOR, ; >
<KEYWORD, phone>
<IDENTIFIER, p >
<OPERATOR, = >
<PHONE, +91-9876543210 >
<PUNCTUATOR, ; >
<KEYWORD, email>
<IDENTIFIER, e >
<OPERATOR, = >
<EMAIL, user@example.com >
<PUNCTUATOR, ; >
<KEYWORD, if>
<PUNCTUATOR, ( >
<IDENTIFIER, x >
<OPERATOR, < >
<IDENTIFIER, y >
<PUNCTUATOR, ) >
<PUNCTUATOR, { >
<IDENTIFIER, x >
<OPERATOR, = >
<IDENTIFIER, x >
<OPERATOR, + >
<INTEGER, 1 >
<PUNCTUATOR, ; >
<PUNCTUATOR, } >
<KEYWORD, return>
<INTEGER, 0 >
<PUNCTUATOR, ; >
```

CONCLUSION:

In this experiment, we implemented a Lexical Analyzer for the extended C language (ExtendC) using Lex. It successfully identified keywords, identifiers, constants, operators, punctuators, and the new data types phone and email, recognizing valid phone numbers and email addresses as special tokens while reporting errors for invalid symbols with line numbers.

We learned how Lex converts regular expressions into finite state automata (FSA) for token recognition, how to extend a language with new tokens and patterns, and how to classify tokens effectively. The experiment also emphasized handling whitespace, line numbers, and errors, highlighting the importance of lexical analysis as the foundation for syntax and semantic analysis. Additionally, practical skills like processing source code files using file redirection were reinforced.