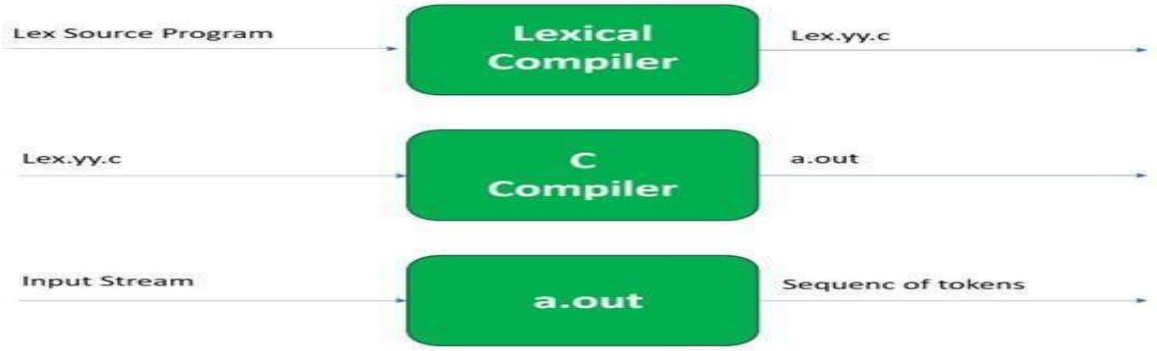
Experiment No.4

Aim : Implementation of lexical analyzer for identifiers , keywords , variables , constants using lex

Theory :

Lex :

Lex, also known as Flex (Fast Lexical Analyzer Generator), is a tool used for generating lexical analyzers, which are also known as tokenizers or scanners, for use in compiler construction. Lexical analyzers are the initial stage of a compiler, responsible for converting the input source code into a stream of tokens, each representing a meaningful unit such as keywords, identifiers, literals, and operators.



Working of Lex :

- Lex Specification: The process begins with the creation of a Lex specification file (often with a .l extension).
- Lex Compiler: The Lex specification file is processed by the Lex compiler (lex or flex), which generates C code for the lexical analyzer based on the rules defined in the specification.
- Generated C Code: The Lex compiler produces a C source file containing the code for the lexical analyzer.
- Compilation: The generated C source file is then compiled using a C compiler (e.g., GCC, Clang) to produce an executable program.
- Tokenization: When the generated lexical analyzer program is executed, it reads the input source code character by character. It matches the input characters against the patterns defined in the Lex specification file to identify tokens such as keywords, identifiers, literals, and punctuation symbols.
- Action Execution: Upon matching a pattern, the corresponding action specified in the Lex rules is executed.
- Output: As the lexical analyzer scans the input source code, it generates a stream of tokens as output.
- Error Handling: Lex provides mechanisms for error handling, allowing developers to define rules for handling unrecognized input, syntax errors, and other exceptional conditions encountered during lexical analysis.

Lex File Format

A Lex program consists of three parts and is separated by %% delimiters:- Declarations
%%

Translation rule
%%

Auxiliary procedures

exp4.l

```
%{
#include<stdio.h>
#include<string.h>
int count = 0;
%}

%%
([a-zA-Z0-9])*    {count++;}

"\n" {printf("Total Number of Words : %d\n", count); count = 0;}
%%

int yywrap(void){}

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

```
umit@umit-OptiPlex-990:~$ lex exp4.l
"exp4.l", line 2: bad character: #
"exp4.l", line 2: unknown error processing section 1
"exp4.l", line 2: unknown error processing section 1
"exp4.l", line 2: bad character: <
"exp4.l", line 2: unknown error processing section 1
"exp4.l", line 2: bad character: .
"exp4.l", line 2: unknown error processing section 1
"exp4.l", line 2: bad character: >
"exp4.l", line 4: bad character: %
"exp4.l", line 4: bad character: }
umit@umit-OptiPlex-990:~$ cc lex.yy.c
cc: error: lex.yy.c: No such file or directory
cc: fatal error: no input files
compilation terminated.
umit@umit-OptiPlex-990:~$ ./a.out

#include<stdio.h> is a preprocessor directive

FUNCTION
        main(
        )


 BLOCK BEGINS

        int is a keyword
 a IDENTIFIER,
 b IDENTIFIER;
BLOCK ENDS
```

If in case of installation following commands should follow :

1. Sudo.su apt-get update
2. Sudo.su apt-get install byacc
3. Sudo.su apt-get install bison
4. Sudo.su apt-get install bison++
5. Sudo.su apt-get byacc-j

Conclusion: Implementing a lexical analyzer for identifiers, keywords, variables, and constants using Lex provides a practical hands-on experience in compiler construction.