EXPERIMENT No: 2

5/10/23 LEX PROGRAMS (KEYWORDS, UPPERCASE, VOWELS, COUNT) 6

AIM

To implement lex programs to identify keywords, uppercase words, vowels in a word & vowel count in a word.

THEORETICAL BACKGROUND

Lex tool is a vital tool in compiler construction & text processing used to create lexical analyzers or lexers. Lexical analyzers break down input streams into tokens, which are fundamentals units of information. These tokens serve as the basis for subsequent phases of a compiler or for text processing tasks.

Lex programs are structured into three sections: definitions, rules & user-defined functions. In the definition section, constants, datatypes & regular expressions are declared. The rules section defines patterns to match against the input, & each pattern is associated with actions to be executed when a match occurs. User-defined functions can also be used to customize the lexers behaviour.

The core of Lex's functionality lies in pattern matching using regular expression. These expression describe the structure of tokens & enable to identify & separate language constructs like keywords, identifiers, operators & literals in programming languages. When a pattern matches, an associated action is executed. These actions typically result in the creation of tokens with attributes. For example, lex can recognize keywords, assign them a type, & provide them to the parser for further analysis.

Lex tools are particularly essential in compiler development. They simplify the task of creating lexers

/ /

which are the first phase of compilers. Lexers convert source code into tokens, which are subsequently passed to a parser for further analysis & code generation.

CONCLUSION-

The program was successfully implemented & executed.

ALGORITHM 2.1

1. Start
2. Include necessary header files.
3. Define patterns to recognize keywords, numbers, words & any other char.
4. In the main function, print a prompt.
5. Invoke the lexer ('yylex')
6. In the lexer, recognize tokens & print them
7. If the lexer returns, end the program.
8. Stop

ALGORITHM 2.2

1. Start.
2. Include necessary header file
3. Define patterns to recognize vowels & consonants
4. In the main function, print a prompt.
5. In the lexer, recognize vowels & consonants & print them
6. If the lexer returns, end the program.
7. Stop

ALGORITHM 2.3

1. Start
2. Include necessary header files.
3. Define pattern to recognize small letters & capital letters.
4. In the main function, print a prompt.
5. Invoke the lexer.
6. In the lexer, recognize small letters & capital letters & print them.
7. If the lexer returns, end the program.
8. Stop

ALGORITHM 2.4

1. Start
2. Include necessary header files & declare counter for vowels
3. Define pattern to recognize vowels, consonants & newline characters
4. Invoke the lexer ("yylex")
5. In the main function, print a prompt.
6. In the lexer, count vowels & consonants
7. When a newline is encountered, print the counts & end the program.

EXPERIMENT NO: 3

12/10/23 | YACC specification | 10

AIM

Generate YACC specification for a few syntactic categories.

a) Implementation of calculator using LEX and YACC

b) Program to recognize a valid arithmetic expression that uses operators +, -, * & /.

c) Program to recognize a valid variable which starts with a letter followed by any number of letters & digits.

THEORETICAL BACKGROUND

Yacc, or Yet another Compiler Compiler, is a powerful tool for generating parsers & syntax analyzer. It plays a pivotal role in the construction of compilers, interpreters & other software systems that require the analysis of complex structured languages.

At its core, Yacc operates on context-free grammar, making it suitable for recognizing & processing the syntax of wide range of programming languages & data formats. Yacc's theoretical foundation is based on formal language & automata theory, specifically context-free grammar & pushdown automata.

The main concepts of a Yacc specification include grammar rules, tokens (terminals) & actions. Grammar rules define the syntax of the language being parsed, tokens represent the language's basic building blocks, & actions define the behaviour associated with each rule. Yacc generates parsers based on the provided specifications that can parse input according to the defined grammar.

Yacc parsers use a bottom-up parsing technique, constructing parse trees or abstract syntax trees from the input. These parse trees represent the structure of the input language, allowing for further processing, optimization, or code generation.

Conclusion -

The programs were successfully implemented & executed.

ALGORITHM 3.1

1. Start
2. Include the necessary headers for Lex & Yacc
3. In the lex file, define patterns to match numbers, identifiers, whitespaces, newline & other characters
4. Associate actions with Lex patterns to return tokens & attribute values
5. In the Yacc file, specify grammar rules to parse & evaluate mathematical expressions.
6. Define actions with Yacc rules to perform calculations.
7. In the main function, print a prompt.
8. Invoke the Yacc parser (yyparse)
9. Handle errors using 'yyerror' function
10. Compile the Lex & Yacc files & create executable.

ALGORITHM 3.2