**LIST OF EXPERIMENTS:**  
1. Implementation of Symbol Table  
2. Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.)  
3. Implementation of Lexical Analyzer using  Lex Tool  
4. Generate YACC specification for a few syntactic categories.  
a) Program to recognize a valid arithmetic expression that uses operator +, - , \* and /.  
5. Convert the BNF rules into Yacc form and write code to generate Abstract Syntax Tree.  
6. Implement type checking  
7. Implement control flow analysis and Data flow Analysis  
8. Implement any one storage allocation strategies(Heap,Stack,Static)  
9. Construction of DAG  
10. Implementation of Simple Code Optimization Techniques (Constant Folding., etc.)

|  |  |
| --- | --- |
| **Ex. No. 1** | **IMPLENTATION OF SYMBOL TABLE** |
| **Date:** |

**AIM:**  
To write a program for implementing Symbol Table using C.  
**ALGORITHM:**  
**Step1:** Start the program for performing insert, display, delete, search and modify option in symbol table  
**Step2:**Define the structure of the Symbol Table  
**Step3:** Enter the choice for performing the operations in the symbol Table  
**Step4:** If the entered choice is 1, search the symbol table for the symbol to be inserted. If the symbol is  
already present, it displays “Duplicate Symbol”. Else, insert the symbol and the corresponding address in  
the symbol table.  
**Step5:** If the entered choice is 2, the symbols present in the symbol table are displayed.  
**Step6:**If the entered choice is 3, the symbol to be deleted is searched in the symbol table.  
**Step7:**If it is not found in the symbol table it displays “Label Not found”. Else, the symbol is deleted.  
**Step8:** If the entered choice is 5, the symbol to be modified is searched in the symbol table.

Result:

Thus the program for Symbol table is created and executed successfully.

|  |  |
| --- | --- |
| **Ex. No. 2** | Develop a lexical analyzer to recognize a few patterns in C. |
| **Date:** |

**AIM:**  
To develop a lexical analyzer to identify identifiers, constants, comments, operators etc using C program  
**ALGORITHM:**  
**Step1:** Start the program.  
**Step2:** Declare all the variables and file pointers.  
 **Step3:** Display the input program.  
**Step4:** Separate the keyword in the program and display it.  
**Step5:** Display the header files of the input program  
**Step6:** Separate the operators of the input program and display it.  
 **Step7:** Print the punctuation marks.  
**Step8:**Print the constant that are present in input program.  
**Step9:** Print the identifiers of the input program.

Thus the program for developing a lexical analyzer to recognize a few patterns in C has been executed successfully.

|  |  |
| --- | --- |
| **Ex. No. 3** | Implementation of Lexical Analyzer using Lex Tool |
| **Date:** |

**AIM:**  
    To write a program for implementing a Lexical analyser using LEX tool in Linux platform.  
  
**ALGORITHM:**  
**Step1:** Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the    delimiter, %%.  The format is as follows:   definitions %% rules %% user\_subroutines

**Step2:** In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. Identifier is defined such that the first letter of an identifier is alphabet and remaining letters are alphanumeric.

**Step3:** In rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.

**Step4:** Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.

**Step5:** When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.

**Step6:**In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.

**Step7:**The lex command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.

Result:

Thus the program for implementing a Lexical analyser using LEX tool in Linux is created and executed successfully.

|  |  |
| --- | --- |
| **Ex. No. 4** | **YACC PROGRAM TO VALID ARTHIEMATIC EXPRESSION** |
| **Date:** |

**AIM:**

       To write a Yacc program to valid arithmetic expression using Yacc .

**ALGORITHM:**

**Step1:**Start the program.

**Step2:**Reading an expression .

**Step3:** Checking the validating of the given expression according to the rule using yacc.

**Step4:**Using expression rule print the result of the given values

**Step5:** Stop the program.

**Result:**

Thus the Yacc program to valid arithmetic expression using Yacc is executed and verified successfully

|  |  |
| --- | --- |
| **Ex. No:5** | Convert the BNF rules into Yacc form and write code to generate Abstract Syntax Tree. |
| **Date:** |

**AIM:**

      To write a yacc program to change yacc form into abstract syntax Tree

**ALGORITHM:**

**Step1:** Reading an expression.

**Step2:** Calculate the value of given expression

**Step3:** Display the value of the nodes based on the precedence.

**Step4:** Using expression rule print the result of the given values

Result:

  Thus the yacc program to change yacc form into abstract syntax Tree is executed and verified successfully

|  |  |
| --- | --- |
| **Ex. No.:6** | **IMPLEMENTATION OF A TYPE CHECKING** |
| **Date:** |

**AIM:**

 To write a C program to implement type checking

**ALGORITHM:**

**Step1:** Track the global scope type information (e.g. classes and their members)

**Step2:** Determine the type of expressions recursively, i.e. bottom-up, passing the resulting types upwards.

**Step3:** If type found correct, do the operation

**Step4:** Type mismatches, semantic error will be notified

RESULT:

Thus the program for implementation of type checking is executed and verified successfully

|  |  |
| --- | --- |
| **Ex. No.:7** | Control flow analysis and Data flow analysis |
| **Date:** |

**Aim:**

To write a program to implement Control flow analysis and Data flow analysis

Algorithm:

Step 1: Start the program execution

Step 2: Read the total number of expression

Step 3:Read the left and right side of each expression

Step3: Display the expression with line no

Step 4:Display the data flow movement with particular expression

Step 5: Stop the program execution

**Result:**

Thus the program for implement data flow analysis is executed and verified successfully

|  |  |
| --- | --- |
| **Ex. No.8** | Implement storage allocation strategies |
| **Date:** |

AIM:

To write a C program for Stack to use dynamic storage allocation.

ALGORITHM:

1. Start the program

2. Enter the expression for which intermediate code is to be generated

3. If the length of the string is greater than 3, than call the procedure to return the precedence

4. Among the operands.

5. Assign the operand to exp array and operators to the array.

6. Create the three address code using quadruples structure.

7. Reduce the no of temporary variables.

8. Continue this process until we get an output.

9. Stop the program.

# RESULT:-

Thus the c program for stack allocation has been implemented successfully.

|  |  |
| --- | --- |
| **Ex. No.:9** | CONSTRUCT DAG |
| **Date:** |

Aim:

Write a C program to construct DAG

Algorithm:

Step 1: start the program

Step2;Include all the header Files

STEP 3:Check the postfix expression and construct the order DAG representation

Step4:Print the output

Step 5:Stop the program

Result:

Thus the program for implementation of DAG is executed and verified successfully.

|  |  |
| --- | --- |
| **Ex. No.:10** | **Implementation of Simple Code Optimization Techniques** |
| **Date:** |

**AIM:**

To write a program for the generation of assembly language code of relational operator

**ALGORITHM :**

Step1: Generate the program for factorial program using for and do-while loop to specify optimization technique.

Step2: In for loop variable initialization is activated first and the condition is checked next. If the condition is true the corresponding statements are executed and specified increment / decrement operation is performed.

Step3: The for loop operation is activated till the condition failure.

Step4: In do-while loop the variable is initialized and the statements are executed then the condition checking and increment / decrement operation is performed.

Step5: When comparing both for and do-while loop for optimization dowhile is best because first the statement execution is done then only the condition is checked. So, during the statement execution itself we can find the inconvenience of the result and no need to wait for the specified condition result.

Step6: Finally when considering Code Optimization in loop do-while is best with respect to performance.

# RESULT:-

Thus the program for implementation of Code Optimization technique is executed and verified.