EXPERIMENT NO. 1

LEXICAL ANALYSER IN C

AIM :-

Design and implement a lexical analyser using C language to recognise all valid tokens in the input program. The lexical analyser should ignore redundant spaces, tabs and newlines. It should ignore comments.

ALGORITHM :-

1. START
2. In function isKeyword() with type int and parameter 'char buffer[]'
   2.1. Initialise array keywords[][] with the important keywords.
   2.2. Compare the buffer with keyword array.
   2.3. If 0, then set flag = 1
   2.4. Return flag.
3. In main function
   3.1. Declare ch, buffer[15] and operators[] = "+-*/%=" of type char.
   3.2. Declare file pointer fp.
   3.3. Open the file program.txt in read mode using fp.
   3.4. If fp = NULL, then print "Error" and goto step 4
   3.5. In while loop till EOF
      3.5.1. Check whether the tokens match with

the elements of the assay operator.

3.5.2. If ch is alphanumeric
then, buffer[j++] = ch

3.5.3. Else if ch = " " or ch = '\n' and j != 0
then, buffer[j] = '\0' and set j = 0.

3.5.4. If (isKeyword(buffer) == 1), then
print 'Keyword' else 'Identifier'.

4. STOP

RESULT :-

Successfully designed and implemented a lexical analyser using C language.

31/01/23

18/09/23

EXPERIMENT NO. 2
LEXICAL ANALYSER USING LEX

AIM :-

Implement a lexical analyser for a given program using LEX tool.

ALGORITHM :-

1. START
2. If the input word is if| else| while| float| int... etc then display it as 'keyword'
3. If the input word is of the form $[0-9]^+$, display it as an integer.
4. If the word is of the form $[0-9]^+.[0-9]^*$, then display it as a 'floating point number'.
5. If the input is of the form letter [letter| digits]* then display it as an identifier.
6. If the input is of the form '{', '}', '(', ')' then display it as a separator.
7. If none of the above is valid, then display 'Unrecognised / invalid rule'
8. STOP

RESULT :-

Successfully implemented lexical analyser using Lex Tool.

18/09/23

EXPERIMENT NO. 3

LEX PROGRAM TO DISPLAY COUNT OF LINES, WORDS, SPACES AND CHARACTERS

AIM:-

Write a Lex program to display the number of lines, words, spaces and characters in an input text.

ALGORITHM:-

1. START
2. Declare and initialize space count sc = 0, line count lc = 0, word count wc = 0 and character count cc = 0 as type integer.
3. If input is "\n", then increase lc and increase cc to the count of characters in that line.
4. If input is " ", then increase sc and increase cc with the count of characters.
5. If input is not \t, \n or space, then increase wc and increase cc.
6. In main function, print all the lc, sc, wc and cc
7. STOP

RESULT:-

Successfully implemented the Lex program to display the number of lines, words, spaces and characters in an input text using Lex tool.

EXPERIMENT NO. 4

LEX PROGRAM TO PERFORM THE CONVERTION OF SUBSTRING

AIM :-

Write a Lex program to convert the substring 'abc' to 'ABC' from the given input string.

ALGORITHM :-

1. START
2. Declare integer variable i
3. Define transition rules as following
   - 3.1. Check if substring 'abc' is present, if true convert if to 'ABC' using yytext.
     - 3.1.1. Print the converted text.
   - 3.2. If any other characters are present in the string, print the same using echo.
   - 3.3. For a new line, print the text.
4. Call yylex.
5. In yywrap() function,
   - 5.1. Return 1
6. STOP

RESULT :-

Succesfully implemented the lex program to convert substring 'abc' to 'ABC'

EXPERIMENT NO. 5

LEX PROGRAM TO DISPLAY THE COUNT OF VOWELS AND CONSONANTS

AIM :-

Write a Lex program to find out the total number of vowels and consonants from given input string.

ALGORITHM :-

1. START
2. Declare integer variable i
3. Declare transition rule for a-z and A-Z and perform action cc++
4. Declare transition rule for vowel and perform action vc++
5. Inside main function
   5.1. Read the string from user
   5.2. Print number of consonants
   5.3. Print number of vowels
6. Inside function yywrap()
   6.1 Return 1
7. STOP

RESULT :-

Successfully implemented the Lex program to find total number of vowels and consonants from input string.

EXPERIMENT NO. 6

YACC SPECIFICATION TO RECOGNISE A VALID ARITHMETIC EXPRESSION

AIM :-

Generate a YACC specification to recognise a valid arithmetic expression that ~~uses~~ operators +, -, *, / and parentheses.

ALGORITHM :-

Lex program :

1. START
2. In header file, include header file y.tab.h and extern yylval
3. Declare transition rules for
   (i) number [0-9], set yylval = atoi (yytext) return the number
   (ii) If \t, then set nothing
   (iii) If \n, then return 0
4. If there are any other characters, print the string, which is present in yytext [0]
5. Call yywrap()
7. STOP

YACC program :

1. START
2. Define header files
3. Declare the tokens number and assign left precedence to '(', ')', '*', '/', '+', '-'.

4. Declare transition rules
   4.1. For E, print the result
   4.2. For T, if
      (i) T+T, assign T = $T_1 + T_3$
      (ii) T-T, assign T = $T_1 - T_3$
      (iii) T*T, assign T = $T_1 - T_3$
      (iv) T/T, assign T = T_1 / T_3
      (v) '-' NUMBER, assign T = uminus(T)
      (vi) '(' T ')', assign T = T_1
5. In main function, print the expression and accept it.
6. In yyerror(), print the expression is invalid.
7. STOP.

RESULT :-
Successfully implemented YACC specification to recognise a valid arithmetic expression.

EXPERIMENT NO. 7

YACC SPECIFICATION TO RECOGNISE A VALID IDENTIFIER

AIM :-

Generate a YACC specification to recognise a valid identifier which starts with a letter followed by any number of letters and digits.

ALGORITHM :-

Lex program :-

1. START
2. Include the header file y.tab.h
3. Declare the transition rules for
   - 3.1. [a-z A-Z][a-z A-Z _ 0-9]*
     - 3.1.1. return letter
   - 3.2. [0-9]
     - 3.2.1. return digit
   - 3.3. [\t]+
     - 3.3.1. skip tab space
   - 3.4. .
     - 3.4.1. return yytext[0]
   - 3.5. \n
     - 3.5.1. return 0
4. Call yywrap() function
   - 4.1. return 1
5. STOP

YACC program :

1. START
2. Include header file stdio.h
3. Declare and initialize variable 'valid' = 1 of type int.
4. ~~Delete~~ Declare the tokens digit, letter
5. Declare the transition rules
   - 5.1. start : letter s
   - s : letter s
     | digit s
     |
     ;
6. In yyerror() function
   - 6.1. Print 'Not an identifier'
   - 6.2. Set valid to 0
   - 6.3. Return 0
7. In main function
   - 7.1. Print 'Enter a name to be tested'
   - 7.2. Call yyparse()
   - 7.3. If valid == 1
     - 7.3.1. Print 'An identifier'
8. STOP

RESULT :-

Successfully implemented YACC specification to recognise a valid identifier.

EXPERIMENT NO.8

CALCULATOR USING LEX AND YACC

AIM:-

Implement a calculator using lex and YACC.

ALGORITHM:-

Lex program:

1. START
2. Include the header files stdio.h and y.tab.h
3. Declare extern in yylval
4. In transition rules section,
   - 4.1. [0-9]+
     - 4.1.1. Set yylval = atoi(yytext)
     - 4.1.2. Return NUMBER
   - 4.2. [\t]
     - 4.2.1. Return nothing
   - 4.3. [\n]
     - 4.3.1. Return 0
   - 4.4. .
     - 4.4.1. Return yytext[0].
5. Call yywrap() function
   - 5.1. Return 1
6. STOP

YACC program:

1. START
2. Include header file STDIO.H
3. Declare and initialise flag = 0 of type int.

4. Declare a token NUMBER and assign left precedence for '+' '-', '*' '/' '%' & '(' ')'
5. In the transition rules section
   5.1. For Arithmetic Expression : E
      5.1.1. Print the result
   5.2. For E
      5.2.1. If E+E, then add
      5.2.2. If E-E, then subtract
      5.2.3. If E*E, then multiply
      5.2.4. If E/E, then divide
      5.2.5. If E%E, then find reminder
      5.2.6. If (E), then E=E
      5.2.7. If NUMBER, then NUMBER
6. In the main() function.
   6.1. Read the expression from user
   6.2. Call yyparse()
   6.3. If flag == 0, the expression is Valid.
7. In yyerror() function,
   7.1. Print 'expression is invalid'
   7.2. Set flag = 1.
8. STOP

RESULT :-

Successfully implemented a calculator using lex and yacc.

EXPERIMENT NO. 9

ε-CLOSURE OF ALL STATES OF GIVEN NFA WITH ε TRANSITION

AIM:-

Write a program to find ε-closure of all states of any given NFA with ε transition.

ALGORITHM:-

1. START
2. Include the necessary header files and declare global arrays for storing results, copies of states and states.
3. Define a function 'add_state' to add a state to the result array.
4. In the main function,
   - 4.1. Open the input.dat file for reading
   - 4.2. Prompt the user to input the number of states (n) and the state names.
   - 4.3. For each state:
     - (i) Initialize counters and copy the same name.
     - (ii) Add the initial state to the result array.
     - (iii) Read transitions from the input file:
       - (a) State 1 is the current state
       - (b) Input is the input symbol
       - (c) State 2 is the next state
     - (iv) Check if the current state matches 'state 1' and if the input symbol is 'e' (epsilon), then add the next state to the result

array and update the current state.

(v) Display the epsilon closure for the current state.

(vi) Remind the input file to the beginning for the next state.

4.4. Close the input file and return 0.

5. The input.dat contains transition rules in the format : 'current_state input_symbol next_state'. The program reads these rules and constructs the epsilon closure for each state.

RESULT :-

Successfully implemented the C program to find ε-closure of all states of any given NFA with ε transition.

EXPERIMENT NO. 10

CONVERT ε-NFA TO NFA

AIM :-

Write a program to convert NFA with ε transitions to NFA without ε transition.

ALGORITHM :-

1. START
2. Initialize variables
   - 2.1. Read the no. of alphabets, states, start state, no. of final states, final states.
   - 2.2. Read the no. of transitions and transitions in the form 'qn alphabet qn'
3. Initialize epsilon_closure_array, buffer_array.
4. In for loop (i=1 to no. of states), reset buffer, set c=0.
   - 4.1. Call findClosure (x, i)
   - 4.2. If buffer [x] is in set, return 0
   - 4.3. Mark buffer [x] as 1 and add x to ε-closure [state][i]
5. Print equivalent NFA without epsilon
   - 5.1. Print start state and alphabets
   - 5.2. In for loop [i=1 to no. of states]
   - 5.3. Print ε-closure [i] and corresponding alphabets.
6. Construct NFA without epsilon.
   - 6.1. In for loop, i=1 to no. of states
   - 6.2. Initialize set array, union ε-closure of the states and print the transition.
7. For each final states, in for loop (j=1, no. of state)

7.1. For each ε-closure [j][k], if it reaches the final state, then print ε-closure [j]

8. STOP

RESULT :-

Successfully implemented the C program to convert NFA with ε transitions to NFA without ε transition.

EXPERIMENT NO. 11
CONVERT NFA TO DFA

AIM :-

Write a program to convert NFA to DFA.

ALGORITHM :-

1. START
2. Initialise structures for NFA state & DFA state
3. Initialise function to compute epsilon closure for set of NFA states.
4. Initialise function to convert NFA to DFA.
   - 4.1. Initialize DFA with epsilon closure of initial state of NFA
   - 4.2. Copy necessary fields from epsilon closure set to DFA state
   - 4.3. Implement queue data structure for processing states.
   - 4.4. Enqueue states into queue
   - 4.5. While queue is not empty?
     - 4.5.1. Dequeue state from queue
     - 4.5.2. For each input symbol in alphabet, compute the move of current state on input symbol
     - 4.5.3. Compute epsilon closure of move
     - 4.5.4. Check if resulting set is already in DFA; if not present,
       (a) Create a new DFA state, copy necessary fields to DFA state.

(b) Enqueue new DFA state.

(c) Create a transition from current state to that state.

5. In main function.

5.1. Initialise and define NFA representation.

5.2. Initialise and define DFA representation.

5.3. Convert NFA to DFA using function.

6. STOP

RESULT :-

Successfully implemented the C program to convert NFA to DFA.

EXPERIMENT NO. 12

FIRST AND FOLLOW

AIM :-

Write a program to find the first and follow of any given grammar.

ALGORITHM :-

I First Set :

1. START
2. Initialize empty first sets for each non terminal symbol.
3. For each production
   - 3.1. If the production starts with a terminal, add it to the first set of corresponding non-terminal
   - 3.2. If the production starts with a non terminal,
     - 3.2.1. Find the first set
     - 3.2.2. If the non-terminal can produce 'e', continues on to the next symbol.
     - 3.2.3. Otherwise add the first set of symbols.
4. Repeat the process until no new symbols are added
5. STOP

II Follow Set :

1. START
2. Initialise empty follow sets for each non-terminal symbols.
3. For each production,
   - 3.1. If its non-terminal, determine the symbol that follows it.

3.2. Update the follow set of the current non-termina based on the symbol.

4. Repeat the process until no new symbols are added to any follow set.

5. STOP

RESULT :-

Successfully implemented a program to find the first and follow of any given grammar.

8/12/22

EXPERIMENT NO. 13

RECURSIVE DESCENT PARSER

AIM :-

Design and implement a recursive descent parser for a given grammar.

ALGORITHM :-

1. START
2. Define the grammar with production rules.
3. Define the functions for non-terminal symbols, create functions for each non-terminal.
4. Create a function to handle terminal symbol and matching
   - 4.1. getNextToken() to get the next token from input.
   - 4.2. match the current token with expected token.
5. Write functions for each non-terminal symbol based on production rules.
   - 5.1. Use recursive calls to handle nested non-terminals
   - 5.2. Match terminal according to grammar rule.
6. Implement error handling for syntax or unexpected tokens
7. In main function,
   - 7.1. Get the first token from input.
   - 7.2. Call the top-level function the corresponds to the start symbol.
   - 7.3. Verify parsing success by checking if the last token is eof.
8. STOP

RESULT :-

Successfully implemented a recursive descent parser for a given grammar.

8/12/23

EXPERIMENT NO. 14

OPERATOR PRECEDENCE PARSER

AIM :-

Implement an operator precedence parser.

ALGORITHM :-

1. START
2. Initialise operand and operator stack, setting top indices to -1
3. Read the arithmetic expression
4. Iterate through each character in expression
5. If the character is a digit
   5.1. Extract the operand and push it onto operand stack.
6. If characters is an operator
   6.1. Compare it's precedence
   6.2. Perform the operation based on the precedence rules.
7. If it encounters an opening parenthesis, then push into stack.
8. If it encounters a closing parenthesis, pop operator and perform operation.
9. After parsing the expression, perform remaining operations by popping operators and operands and performing operations.
10. The final result is the value remaining at the top of the operand stack after all operations are performed.

RESULT :-

Successfully implemented the operator precedence parser

8/12/23

EXPERIMENT NO. 15

SHIFT REDUCE PARSER

AIM :-

Construct a shift reduce parser for a given language.

ALGORITHM :-

1. START
2. Initialise variables k, z, i, j and c
3. Declare arrays a[16], ac[20], stk[15] and act[10].
4. Display the grammar rules : E → E+E, E → E*E, E → (E), E → id
5. Define the check() function for reducing the input.
6. Read the input string into array 'a'.
7. Calculate the length of the input string 'c'.
8. Copy 'SHIFT →' to the act array.
9. Display headers for the stack, input and action columns.
10. Loop through the input string:
    10.1. Check if the current and next characters are 'id'. If true, reduce to 'E'.
    10.2. Otherwise, consider the current character as a symbol and reduce accordingly.
11. Implement the check() function
    11.1. If 'id' is found in the stack, replace it with 'E'
    11.2. If 'E+E' is found, replace it with 'E'
    11.3. If 'E*E' is found, replace it with 'E'
    11.4. If '(E)' is found, replace it with 'E'.

12. Repeat the loop until the end of the input string.
13. STOP

RESULT :-

Successfully implemented a shift reduce parser for a given language.

8/12/23

EXPERIMENT NO. 16

CONSTANT PROPAGATION

AIM :-

Write a program to perform constant propagation

ALGORITHM :-

1. START
2. Declare a structure 'expr' to store expression details with fields op, op1, op2, res and flag.
3. Declare an array 'arr' of 'expr' type with a maximum size of 10.
4. Define functions: input(), output(), change() and constant().
5. In the main() function:
   - 5.1. Call the input() function to read expression details
   - 5.2. Call the constant() function to optimize constant expressions.
   - 5.3. Call the output() function to display the optimised code.
6. Define the input() function:
   - 6.1. Read the maximum number of expressions 'n'
   - 6.2. Read the expression details (op, op1, op2, res) into the 'arr' array.
7. Define the constant() function:
   - 7.1. Loop through each expression in 'arr'
   - 7.2. Check if both operands are digits or the operation is '='
   - 7.3 If true, perform constant folding and store the

result in 'res1'.

7.4. Mark the expression as optimized by setting the 'flag' to 1.

7.5. Call the change() function to update dependent expressions.

8. Define the output() function:

8.1. Display the optimised code by printing expressions where 'flag' is not set.

9 Define the change() function:

9.1. Update dependent expressions by replacing occurences of the result with the optimised value.

9.2. Iterate through expressions following the optimized expression (starting from index p+1)

9.3. If the result is used as an operand, update the operand with the optimized value.

10. The main() function calls input(), constant() and output() to execute the optimization process.

11. STOP

RESULT :-

Successfully implemented a program to perform constant propagation.

EXPERIMENT NO. 17

INTERMEDIATE CODE GENERATION

AIM :-

Implement intermediate code generation for simple expressions.

ALGORITHM :-

1. START
2. Declare global variables op[2], arg1[5], arg2[5] and result[5].
3. Open input file 'input.txt' in read mode (fp1) and output file 'output.txt' in write mode (fp2).
4. While not reaching the end of the input file (feof(fp1)):
   - 4.1. Read operation ~~the end of the~~ (op), arguments (arg1, arg2) and result from the input file.
   - 4.2. Check the operation type using if statements:
     - (i) If op is '+', generate assembly code for addition.
     - (ii) If op is '*', generate assembly code for multiplication.
     - (iii) If op is '-', generate assembly code for subtraction.
     - (iv) If op is '/', generate assembly code for division.
     - (v) If op is '=', generate assembly code for assignment.
   - 4.3. Write the generated assembly code to the output

file (fp2)

5. Close both input and output files.

6 STOP

RESULT :-

Successfully implemented intermediate code generation for simple expressions.

8/12/23