LAB MANUAL

Subject : Compiler Design Lab

Code : LC-CSE-324

Department of Computer Science Engineering

RPS College of Engineering and Technology

Mahendergarh , Haryana - 123029

INSTRUCTIONS

LIST OF PROGRAMS

PROGRAM NO.	NAME OF PROGRAM
1	Write a Program for Token separation with a given expression
2	Write a Program for Token separation with a given file.
3	Write a Program for Lexical analysis using LEX tools.
4	Write a Program to identify whether a given line is a comment or not
5	Write a Program to check whether a given identifier is valid or not.
6	Write a Program to recognize strings under ‘a’, ‘a*b+’, ‘abb’.
7	Write a Program to simulate lexical analyser for validating operators
8	Write a program to check whether a grammar is Left Recursive and remove left recursion
9	Write a program to remove Left Factoring
10	Study of LEX and YACC tools

LAB REQUIREMENTS

HARDWARE REQUIREMENT :

- Processor : Intel(R) corei3 – 8100CPU 3.60GHz
- RAM : 8GB
- HDD : 250GB SSD
- Operating System : Windows10

SOFTWARE REQUIREMENT :

- C++
- VS Code

PROGRAM 1 : Write a Program for Token separation with a given expression

ALGORITHM :

1. Include Necessary Headers: Include the necessary C++ standard library headers for input/output operations (iostream), string manipulation (string), handling streams (sstream), and vectors (vector).
2. Define the Tokenization Function: Create a function named tokenize that takes a const std::string& as input (the expression) and returns a std::vector<std::string> containing the tokens.
3. Initialize Variables: Inside the tokenize function, initialize a std::vector<std::string> to store the tokens and a std::stringstream to iterate through the expression.
4. Tokenize the Expression: Use a while loop with ss >> token to extract each token from the stringstream ss.
5. Handle Special Characters: Iterate through each character of the token using a for loop. If the character is a special character like (,), +, -, *, or /, push it into the tokens vector as a separate token. If the character is not a digit or a special character, print an error message and return an empty vector.
6. Push Token into Vector: After processing each token, push it into the tokens vector.
7. Return the Tokens Vector: After tokenizing all the expressions, return the vector containing the tokens.
8. Main Function: In the main function, prompt the user to enter an expression using std::cout.
9. Read Expression: Read the expression entered by the user using std::getline into a std::string variable.
10. Tokenize Expression: Call the tokenize function with the expression as an argument to tokenize it.
11. Check Tokenization Success: Check if the tokenization process was successful by verifying if the returned tokens vector is empty. If it is empty, print an error message and exit the program.
12. Print Tokens: If tokenization was successful, print the tokens separated by spaces using a loop.
13. Return from Main: Return 0 to indicate successful execution of the program.

CODE :

```
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
#include <cctype>
std::vector<std::string> tokenize(const std::string& expression)
{
   std::vector<std::string> tokens;
   std::stringstream ss(expression);
   std::string token;
  while (ss >> token)
  {
      // Handle special cases like parentheses and operators
      for (char& c : token) {
         if (c == '(' || c == ')' || c == '+' || c == '-' || c == '*' || c == '/') {
            tokens.push_back(std::string(1, c));
         } else if (!std::isdigit(c)) {
            std::cerr << "Invalid character: " << c << std::endl;
            return {};
         }
      }

      tokens.push_back(token);
   }
   return tokens;
}
int main() {
```

```
    std::string expression;
    std::cout << "Enter an expression: ";
    std::getline(std::cin, expression);
    std::vector<std::string> tokens = tokenize(expression);
    if (tokens.empty())
    {
        std::cerr << "Failed to tokenize the expression." << std::endl;
        return 1;
    }
    std::cout << "Tokens: ";
    for (const std::string& token : tokens) {
        std::cout << token << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

PROGRAM NO. 2: Write a Program for Token separation with a given file.

ALGORITHM :

1. Include Necessary Headers: Include the necessary C++ standard library headers for input/output operations (iostream, fstream), string manipulation (string), handling streams (sstream), and vectors (vector).
2. Define the Tokenization Function: Create a function named tokenize that takes a const std::string& as input (the expression) and returns a std::vector<std::string> containing the tokens.
3. Initialize Variables: Inside the tokenize function, initialize a std::vector<std::string> to store the tokens and a std::stringstream to iterate through the expression.
4. Tokenize the Expression: Use a while loop with ss >> token to extract each token from the stringstream ss.
5. Handle Special Characters: Iterate through each character of the token using a for loop. If the character is a special character like (,), +, -, *, or /, push it into the tokens vector as a separate token. If the character is not a digit or a special character, print an error message and return an empty vector.
6. Push Token into Vector: After processing each token, push it into the tokens vector.
7. Return the Tokens Vector: After tokenizing all the expressions, return the vector containing the tokens.
8. Main Function: In the main function, open the input file using std::ifstream.
9. Check File Opening: Check if the file is opened successfully. If not, print an error message and exit the program.
10. Read Expression from File: Read the expression from the file using std::getline into a std::string variable.
11. Close File: Close the input file after reading the expression.
12. Tokenize Expression: Call the tokenize function with the expression as an argument to tokenize it.
13. Check Tokenization Success: Check if the tokenization process was successful by verifying if the returned tokens vector is empty. If it is empty, print an error message and exit the program.
14. Print Tokens: If tokenization was successful, print the tokens separated by spaces using a loop.
15. Return from Main: Return 0 to indicate successful execution of the program.

CODE :

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <cctype>
std::vector<std::string> tokenize(const std::string& expression) {
    std::vector<std::string> tokens;
    std::stringstream ss(expression);
    std::string token;
    while (ss >> token) {
        // Handle special cases like parentheses and operators
        for (char& c : token) {
            if (c == '(' || c == ')' || c == '+' || c == '-' || c == '*' || c == '/') {
                tokens.push_back(std::string(1, c));
            } else if (!std::isdigit(c)) {
                std::cerr << "Invalid character: " << c << std::endl;
                return {};
            }
        }
        tokens.push_back(token);
    }
    return tokens;
```

```
}
int main()
{
    std::ifstream inputFile("input.txt");
    if (!inputFile.is_open())
    {
        std::cerr << "Failed to open file." << std::endl;
        return 1;
    }
    std::string expression;
    std::getline(inputFile, expression);
    inputFile.close();
    std::vector<std::string> tokens = tokenize(expression);
    if (tokens.empty())
    {
        std::cerr << "Failed to tokenize the expression." << std::endl;
        return 1;
    }
    std::cout << "Tokens: ";
    for (const std::string& token : tokens)
   {       std::cout << token << " ";
    }
    std::cout << std::endl;
        return 0;
}
```

PROGRAM 3 : Write a Program for Lexical analysis using LEX tools

ALGORITHM

1. Define Regular Expressions: Define regular expressions that describe the lexical structure of the language. Each regular expression corresponds to a token type.
2. Write Lex Specifications: Write a Lex specification file (usually with a .l extension) where you define rules to match regular expressions and specify corresponding actions.
3. Generate Lexical Analyzer: Use the Lex tool to generate a C or C++ source file from the Lex specification.
4. Write Additional Code: Write additional code in C or C++ to handle the actions specified in the Lex file, such as printing tokens, performing semantic actions, or building a syntax tree.
5. Compile and Run: Compile the generated source file along with the additional code and run the resulting executable to perform lexical analysis on input text.

CODE :

```
%{
#include <iostream>
%}
DIGIT   [0-9]
LETTER  [a-zA-Z]
WHITESPACE [ \t\n]
%%
{WHITESPACE}    {/* Ignore whitespace */}
{DIGIT}+        { std::cout << "NUMBER: " << yytext << std::endl; }
{LETTER}+       { std::cout << "IDENTIFIER: " << yytext << std::endl; }

.               { std::cout << "UNKNOWN: " << yytext << std::endl; }

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

Compilation:

```
lex lexer.l
g++ lex.yy.c -o lexer –ll
```

Input Text File (input.txt):

```
123 abc xyz 456
```

Run the Lexical Analyzer:

```
./lexer < input.txt
```

PROGRAM 4 : Write a Program to identify whether a given line is a comment or not

ALGORITHM

1. Include Necessary Headers: Include the necessary C++ standard library headers for input/output operations (iostream), string manipulation (string), and regular expressions (regex).
2. Define the isComment Function: Create a function named isComment that takes a const std::string& as input (the line) and returns a boolean indicating whether the line is a comment or not.
3. Define Regular Expression: Define a regular expression pattern to match C++ style comments (single-line // and multi-line /* */).
4. Use Regex to Match: Use std::regex_match to check if the given line matches the comment regex pattern.
5. Main Function: In the main function, prompt the user to enter a line of text using std::cout.
6. Read Line: Read the line entered by the user using std::getline into a std::string variable.
7. Check if Comment: Call the isComment function with the line as an argument to determine if it's a comment or not.
8. Print Result: Print whether the given line is a comment or not based on the boolean returned by isComment.
9. Return from Main: Return 0 to indicate successful execution of the program.

CODE

```
#include <iostream>
#include <string>
#include <regex>
bool isComment(const std::string& line)
{
   // Regular expression to match C++ style comments (//) and multi-line comments (/* */)
   std::regex commentRegex(R"(^\s*//|^\s*/\*.+\*/)");
      // Use std::regex_match to check if the line matches the comment regex
   return std::regex_match(line, commentRegex);
}
int main()
{
   std::string line;
   std::cout << "Enter a line: ";
   std::getline(std::cin, line);
      if (isComment(line)) {
      std::cout << "The given line is a comment." << std::endl;
   } else {
      std::cout << "The given line is not a comment." << std::endl;
   }
   return 0;
}
```

PROGRAM 5 : Write a Program to check whether a given identifier is valid or not.

ALGORITHM

1. Define the isValidIdentifier Function: Create a function named isValidIdentifier that takes a const std::string& as input (the identifier) and returns a boolean indicating whether the identifier is valid or not.
2. Check Non-Empty: Check if the identifier is non-empty. If it is empty, return false (invalid identifier).
3. Check First Character: Check if the first character of the identifier is a letter (a-z or A-Z) or an underscore (_). If it is not, return false (invalid identifier).
4. Check Subsequent Characters: Iterate through each subsequent character of the identifier (starting from the second character). For each character, check if it is a letter (a-z or A-Z), a digit (0-9), or an underscore (_). If any character does not meet these criteria, return false (invalid identifier).
5. If All Checks Pass: If all checks pass, return true (valid identifier).
6. Main Function: In the main function, prompt the user to enter an identifier using std::cout.
7. Read Identifier: Read the identifier entered by the user using std::cin into a std::string variable.
8. Check Validity: Call the isValidIdentifier function with the entered identifier as an argument to determine if it's valid or not.
9. Print Result: Print whether the given identifier is valid or not based on the boolean returned by isValidIdentifier.
10. Return from Main: Return 0 to indicate successful execution of the program.

CODE

```
#include <iostream>
#include <string>
#include <cctype>
bool isValidIdentifier(const std::string& identifier) {
    // Check if the identifier is non-empty
    if (identifier.empty()) {
        return false;
    }
      if (!std::isalpha(identifier[0]) && identifier[0] != '_') {
        return false;
    }
    // Check if each subsequent character is a letter, digit, or underscore
    for (size_t i = 1; i < identifier.length(); ++i) {
        if (!std::isalnum(identifier[i]) && identifier[i] != '_') {
            return false;
        }
    }
    return true;
}
int main() {
    std::string identifier;
    std::cout << "Enter an identifier: ";
    std::cin >> identifier;
    if (isValidIdentifier(identifier)) {
        std::cout << "The given identifier is valid." << std::endl;
    } else {
        std::cout << "The given identifier is not valid." << std::endl;
    }
    return 0;
}
```

PROGRAM 6 : Write a Program to recognize strings under ‘a’, ‘a*b+’, ‘abb’

ALGORITHM

1. Define the recognizeString Function: Create a function named recognizeString that takes a const std::string& as input (the string) and returns a boolean indicating whether the string is recognized or not.
2. Define Regular Expressions: Define regular expressions for each of the given patterns:
   - Pattern 'a': Matches the string "a".
   - Pattern 'a*b+': Matches strings starting with zero or more 'a's followed by one or more 'b's.
   - Pattern 'abb': Matches the string "abb".
3. Check Matching: Use std::regex_match to check if the given string matches any of the defined regular expressions.
4. Return Result: If the string matches any of the patterns, return true (recognized), otherwise return false (not recognized).
5. Main Function: In the main function, prompt the user to enter a string using std::cout.
6. Read String: Read the string entered by the user using std::cin into a std::string variable.
7. Check Recognition: Call the recognizeString function with the entered string as an argument to determine if it's recognized or not.
8. Print Result: Print whether the given string is recognized or not based on the boolean returned by recognizeString.
9. Return from Main: Return 0 to indicate successful execution of the program.

CODE

```
#include <iostream>
#include <string>
#include <regex>

bool recognizeString(const std::string& str) {
    // Define regular expressions for each pattern
    std::regex patternA("a");
    std::regex patternABPlus("a*b+");
    std::regex patternABB("abb");

    // Check if the string matches any of the patterns
    if (std::regex_match(str, patternA) ||
        std::regex_match(str, patternABPlus) ||
        std::regex_match(str, patternABB)) {
        return true;
    } else {
        return false;
    }
}

int main() {
    std::string str;
    std::cout << "Enter a string: ";
    std::cin >> str;

    if (recognizeString(str)) {
        std::cout << "The given string is recognized." << std::endl;
    } else {
        std::cout << "The given string is not recognized." << std::endl;
    }
     return 0;
}
```

PROGRAM 7 : Write a Program to simulate lexical analyser for validating operators

ALGORITHM

- Define the isValidOperator Function: Create a function named isValidOperator that takes a const std::string& as input (the operator) and returns a boolean indicating whether the operator is valid or not.
- Define Set of Valid Operators: Define a set (e.g., std::unordered_set<std::string>) containing all valid operators.
- Check Operator Validity: Check if the given operator is present in the set of valid operators.
- Return Result: If the operator is present in the set, return true (valid operator); otherwise, return false (not valid operator).
- Main Function: In the main function, prompt the user to enter an operator using std::cout.
- Read Operator: Read the operator entered by the user using std::cin into a std::string variable.
- Check Operator Validity: Call the isValidOperator function with the entered operator as an argument to determine if it's valid or not.
- Print Result: Print whether the given operator is valid or not based on the boolean returned by isValidOperator.
- Return from Main: Return 0 to indicate successful execution of the program.

CODE

```
#include <iostream>
#include <string>
#include <unordered_set>
bool isValidOperator(const std::string& op) {
    // Set of valid operators
    std::unordered_set<std::string> validOperators = {"+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">=", "&&", "||", "!", "&", "|", "^", "<<", ">>", "~", "++", "--", "+=", "-=", "*=", "/=", "%=", "<<=", ">>=", "&=", "|=", "^="};
    // Check if the operator is in the set of valid operators
    return validOperators.count(op) > 0;
}
int main() {
    std::string op;
    std::cout << "Enter an operator: ";
    std::cin >> op;

    if (isValidOperator(op)) {
        std::cout << "The given operator is valid." << std::endl;
    } else {
        std::cout << "The given operator is not valid." << std::endl;
    }

    return 0;
}
```

PROGRAM 8 : Write a program to check whether a grammar is Left Recursive and remove left recursion

CODE

```
#include<stdio.h>
#include<iostream.h>
#include<conio.h>
void main()
{
char str[10],alpha[10],beta[10], x=238;
int i=0, j=0;
clrscr();
cout<<"Enter the grammar";
cin>>str;
if(str[0]==str[3])
{
for(i=4;str[i]!='/';i++)
{
alpha[j]=str[i];
j++;
}
alpha[j]='\0';
j=0;
cout<<"Left Recursion removed "<<endl;
for(++i;str[i]!='\0';i++)
{
beta[j]=str[i];
j++;
}
beta[j]='\0';
for(i=0;i<=2;i++)
cout<<str[i];
for(j=0;beta[j]!='\0';j++)
{
cout<<beta[j];
}
cout<<"S'"<<endl;
cout<<"S'->";
for(j=0;alpha[j]!='\0';j++)
{
cout<<alpha[j];
}
cout<<"S'/"<<x;
}
else
cout<<"Not recursive";
getch();
}
```

PROGRAM 9 : Write a program to remove Left Factoring
CODE

```
#include<iostream.h>
#include<conio.h>
void main()
{
char a[16],b[4],c[6],d[6],x=238;
int i,j=0,aa,bb,l=0,k[2];
clrscr();
cout<<"Enter the production\n\n";
cin>>a;
for(i=0;i<16;i++)
{
  if(a[i]=='/')
  {
    k[j]=i;
    j++;
  } }
aa=k[0];
bb=k[1];
for(i=0;i<aa-3;i++)
{
 b[i]=a[i+3];
}
for(i=aa+1,j=0;i<bb;j++,i++)
{   c[j]=a[i];
}
for(i=0;i<6;i++)
{ if(c[i]!=b[i])
{
  d[l]=c[i];
  l++;
} }
cout<<"\n\nThe result is: \n";
cout<<"S->";
for(i=0;i<4;i++)
{ cout<<b[i];
}
cout<<"Z"<<endl;
cout<<"Z->";
for(i=0;i<l;i++)
{ cout<<d[i];
}
cout<<"/"<<x;
cout<<"\nS->";
cout<<a[15];
getch();
}
```

PROGRAM 10 : Study of LEX and YACC tools

LEX:

Lex is a lexical analyzer generator, commonly used for creating lexical analyzers or scanners. It takes regular expressions as input and generates C code for a finite automaton that recognizes those expressions in the input text. Here are some key points about Lex:

- Input Specification: Lex takes as input a file containing regular expressions and corresponding actions (usually written in C code).
- Output: It generates a C program that scans the input and performs the specified actions when a pattern is recognized.
- Use Cases: Lex is used in the first phase of compilation, known as lexical analysis, to convert source code into tokens or lexemes.
- Workflow: Lex reads the input file, converts regular expressions into deterministic finite automata (DFAs), and produces C code for the DFA-based lexer.
- Flex: Flex is an alternative to Lex that provides similar functionality but with additional features and improvements.

YACC:

Yacc (Yet Another Compiler Compiler) is a tool used for generating parsers or syntax analyzers. It takes as input a formal grammar in the form of context-free grammar rules and generates C code for a parser that recognizes sentences in that grammar. Here are some key points about Yacc:

- Input Specification: Yacc takes as input a file containing context-free grammar rules and corresponding actions (usually written in C code).
- Output: It generates a C program that parses the input according to the grammar rules and performs the specified actions when a rule is recognized.
- Use Cases: Yacc is used in the second phase of compilation, known as syntax analysis, to check whether the structure of the source code conforms to the grammar rules.
- Workflow: Yacc reads the input file, constructs a parser using LR parsing techniques, and produces C code for the parser.
- Bison: Bison is an alternative to Yacc that provides similar functionality but with additional features and improvements.

WORKFLOW:

The typical workflow involving Lex and Yacc in compiler construction is as follows:

1. Lexical Analysis: Use Lex to generate a lexer or scanner that converts source code into tokens or lexemes.
2. Syntax Analysis: Use Yacc to generate a parser that checks the structure of the source code according to the grammar rules.
3. Semantic Analysis: After syntax analysis, perform semantic analysis to check for semantic errors and generate an intermediate representation of the program.
4. Code Generation: Finally, generate target code (e.g., machine code or intermediate code) based on the analyzed and validated program.

BENEFITS:

- Efficiency: Lex and Yacc-generated parsers are efficient and fast, making them suitable for use in production-level compilers.

- Flexibility: Lex and Yacc allow developers to define complex grammars and processing logic for various programming languages and domain-specific languages.
- Standardization: Lex and Yacc are well-established tools with a large user base and extensive documentation, making them a standard choice for compiler construction projects.