VISHWAKARMA UNIVERSITY
Maximising Human Potential

T. Y. B. Tech Computer Engineering

Student Name	Pranav Dambe (Nikam)
SRN No	202201704
Roll No	68
PRN	2280030506
Division	D(D3)
Subject	System Programming
Year	Third Year

Assignment - 6

QUE 1 :

Design Lexical analyzer for the subset of "C" Language. Accept input from file.

Output : Line No, Lexeme , Token , Token_Value.

Also implement any one error checking. Submit single .pdf file with input C program , Token listing and source code in sequence.

Output 1 :

Input.c :

input.c > main()

```c
int main()
{
    int p = 15;
    int n = 22;
    // ignore this line.
    int sum = p + n; // Add p & n
    int sum_ = 0;
    for(int i=1; i< 10; i= i+ 1){
        sum_ = sum_ + i;
    }
    return 0;
}
```

Tokens table :

Tokens Table:

Line_No	Lexeme	Token	Token Value
1	int	Keyword	17
1	main	Identifier	1
1	(	Delimiter	1
1	)	Delimiter	2
2	{	Delimiter	3
3	int	Keyword	17
3	p	Identifier	2
3	=	Operator	23
3	15	Constant	15
3	;	Delimiter	8
4	int	Keyword	17
4	n	Identifier	3
4	=	Operator	23
4	22	Constant	22
4	;	Delimiter	8
5	int	Keyword	17
5	sum	Identifier	4
5	=	Operator	23
5	p	Identifier	2
5	+	Operator	1
5	n	Identifier	3
5	;	Delimiter	8
6	int	Keyword	17
6	sum_	Identifier	5
6	=	Operator	23
6	0	Constant	0
6	;	Delimiter	8

```
7           for             Keyword         14
7           (               Delimiter       1
7           int             Keyword         17
7           i               Identifier      6
7           =               Operator        23
7           1               Constant        1
7           ;               Delimiter       8
7           i               Identifier      6
7           <               Operator        11
7           10              Constant        10
7           ;               Delimiter       8
7           i               Identifier      6
7           =               Operator        23
7           i               Identifier      6
7           +               Operator        1
7           1               Constant        1
7           )               Delimiter       2
7           {               Delimiter       3
8           sum_            Identifier      5
8           =               Operator        23
8           sum_            Identifier      5
8           +               Operator        1
8           i               Identifier      6
8           ;               Delimiter       8
9           }               Delimiter       4
10          return          Keyword         20
10          0               Constant        0
10          ;               Delimiter       8
11          }               Delimiter       4
PS E:\TY-LAB\SP - Lab\Assignment-6>
```

Symbol Table :

```
Symbol Table:
Index  Symbol
1      main
2      p
3      n
4      sum
5      sum_
6      i
```

Output 2 : Error handling :

Input.c :

input.c 1 ×

input.c > main()

```c
int main()
{
    int p = 15;
    int n = 22;
    // ignore this line.
    int sum = p + n; // Add p & n
    int sum_ = 0; [
    for(int i=1; i< 10; i= i+ 1){
        sum_ = sum_ + i;
    }
    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE PORTS

TERMINAL

```
PS E:\TY-LAB\SP - Lab\Assignment-6> cd "e:\TY-LAB\SP
t_6 }


Error : Missing parentheses : ]

PS E:\TY-LAB\SP - Lab\Assignment-6>
```

Output 3 : Error handling :

input.c 2 ×

input.c > main()

```c
int main()
{
    int p = 15;
    int n = 22;
    // ignore this line.
    int 10sum = p + n;
    int sum_ = 0;
    for(int i=1; i< 10; i= i+ 1){
        sum_ = sum_ + i;
    }
    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE PORTS

TERMINAL

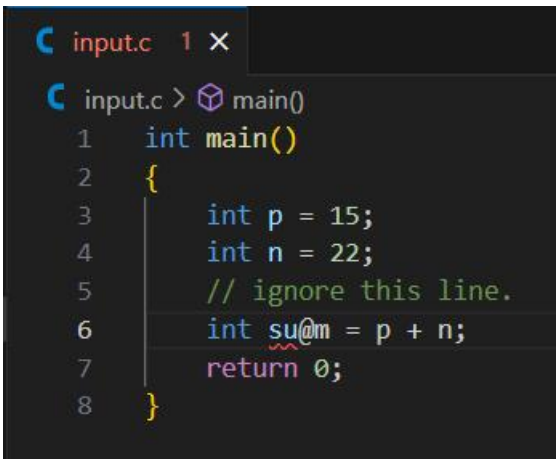
```
PS E:\TY-LAB\SP - Lab\Assignment-6> cd "e:\TY-LAB\SP - Lab
t_6 }


Error :Wrong declaration of variable: start with number

PS E:\TY-LAB\SP - Lab\Assignment-6>
```

Output : 4 Error handling :

Input.c :



CODE :

```
import java.util.*;
import java.io.*;

class CustomException extends Exception {
    String exception = null;
    public CustomException(String str) {
        exception = str;
    }
    @Override
    public String toString() {
        return exception;
    }
}

class LexicalAnalyzer {

    ArrayList<String> keywords = new ArrayList<>(List.of(
            "auto", "break", "case", "char", "const", "continue", "default", "do",
            "double", "else", "enum", "extern", "float", "for", "goto", "if", "int",
            "long", "register", "return", "short", "signed", "sizeof", "static",
            "struct", "switch", "typedef", "union", "unsigned", "void", "volatile", "while"));
```

```
    ArrayList<String> delimiters = new ArrayList<>(List.of(
            "(", ")", "{", "}", "[", "]", ",", ";", ":", "\"", "'", ".", "#"));

    ArrayList<String> operators = new ArrayList<>(List.of(
            "+", "-", "*", "/", "%", ">", "<", "!",
            "&", "|", "^", "~", "=", "?", ":", ".", "&", "*"));

    private Stack<String> stack = new Stack<>();
    private ArrayList<String> symbolTable = new ArrayList<>();
    private List<Object[]> tokensTable = new ArrayList<>();

    private int lineNumber = 0;

    private Integer getIndex_addSymbol(String token) {
        if (!symbolTable.contains(token)) {
            symbolTable.add(token);
        }
        return (symbolTable.indexOf(token) + 1);
    }

    private boolean isIdentifier(String preLexeme, String nextLexeme) {
        return (!preLexeme.equals("\"") && !nextLexeme.equals("\""));
    }

    private boolean isConstantString(String preLexeme, String curToken, String nextLexeme) {
        return preLexeme.equals("\"") && nextLexeme.equals("\"") && curToken.matches("[a-zA-Z0-
9]+");
    }

    private boolean checkParentheses(String closed) {
        if (closed.equals(")") && stack.peek().trim().equals("(")) {
            return true;
        } else if (closed.equals("}") && stack.peek().trim().equals("{")) {
            return true;
        } else if (closed.equals("]") && stack.peek().trim().equals("[")) {
            return true;
        }
        return false;
    }

    private void isValidVariable(String var) throws CustomException {
        char ch = var.charAt(0);
        if (Character.isDigit(ch)) {
            throw new CustomException("Wrong declaration of variable: start with number");
        }

        for (int i = 0; i < var.length(); i++) {
            ch = var.charAt(i);
            if (Character.isWhitespace(ch)) {
                throw new CustomException("Wrong declaration of variable: it contains a whitespace
character.");
            }
            if (!Character.isLetterOrDigit(ch) && ch != '_') {
```

```
                throw new CustomException("Wrong declaration of variable: it contains a special
character.");
            }
        }
    }

    private void handleDelimiter(String lexeme) throws CustomException {
        String token = "Delimiter";
        Integer tokenValue = delimiters.indexOf(lexeme) + 1;

        if (lexeme.trim().equals("(") || lexeme.trim().equals("{") || lexeme.trim().equals("[")) {
            stack.push(lexeme);
        }

        if (lexeme.trim().equals(")") || lexeme.trim().equals("}") || lexeme.trim().equals("]")) {
            if (!stack.isEmpty() && checkParentheses(lexeme.trim())) {
                stack.pop();
            } else if (stack.isEmpty()) {
                throw new CustomException("Extra closing parentheses: " + lexeme);
            } else {
                String expectedParen = stack.peek().equals("(") ? ")"
                        : stack.peek().trim().equals("{") ? "}" : "]";
                throw new CustomException("Invalid use of parentheses : " + expectedParen);
            }
        }
        tokensTable.add(new Object[] { lineNumber, lexeme, token, tokenValue });
    }

    private void handleToken(String preLexeme, String lexeme, String nextLexeme) throws
CustomException {
        Integer tokenValue = 0;
        String token = "";
        if (keywords.contains(lexeme)) {
            token = "Keyword";
            tokenValue = keywords.indexOf(lexeme) + 1;
        } else if (operators.contains(lexeme)) {
            token = "Operator";
            tokenValue = operators.indexOf(lexeme) + 1;
        } else if (delimiters.contains(lexeme)) {
            handleDelimiter(lexeme);
            return;
        } else if (lexeme.matches("\\d+\\.\\d+|\\d+")) {
            token = "Constant";
            tokenValue = Integer.parseInt(lexeme);
        } else if (isConstantString(preLexeme, lexeme, nextLexeme)) {
            token = "Constant";
            tokenValue = -1;
            tokensTable.add(new Object[] { lineNumber, lexeme, token, lexeme });
            return;
        } else if (isIdentifier(preLexeme, nextLexeme)) {
            isValidVariable(lexeme);
            token = "Identifier";
            tokenValue = getIndex_addSymbol(lexeme);
```

```
        }
        tokensTable.add(new Object[] { lineNumber, lexeme, token, tokenValue });
    }

    private void processTokens(String[] tokens) throws CustomException {
        for (int i = 0; i < tokens.length; i++) {
            int preIndex = (i != 0) ? (i - 1) : 0;
            int nextIndex = (i < tokens.length - 1) ? (i + 1) : i;

            if (tokens.length > 1 && tokens[0].trim().equals("/") && tokens[1].trim().equals("/")) {
                lineNumber--;
                break;
            } else if (i < tokens.length - 1 && tokens[i].trim().equals("/") &&
tokens[nextIndex].trim().equals("/")) {
                break;
            }
            handleToken(tokens[preIndex].trim(), tokens[i].trim(), tokens[nextIndex].trim());
        }
    }

    private void scanProgram() throws CustomException {
        String regex = "(?<=\\W)(?<![+\\-*/%])|(?=\\W)";
        try (BufferedReader fileReader = new BufferedReader(new FileReader("input.c"))) {
            String line;
            while ((line = fileReader.readLine()) != null) {
                if (!line.isEmpty()) {
                    lineNumber++;
                    String[] tokenArr = line.split(regex);
                    String tokenList = "";
                    for (int k = 0; k < tokenArr.length; k++) {
                        if (tokenArr[k] != null && !tokenArr[k].trim().isEmpty()) {
                            tokenList += tokenArr[k].trim() + "\t";
                        }
                    }
                    String[] tokens = tokenList.split("\t");
                    processTokens(tokens);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void printTables() {
        System.out.println("\nSymbol Table:");
        System.out.println(String.format("%-6s %-10s", "Index", "Symbol"));
        for (int i = 0; i < symbolTable.size(); i++) {
            System.out.println(String.format("%-6d %-10s", (i + 1), symbolTable.get(i)));
        }

        System.out.println("\nTokens Table:");
        System.out.println(String.format("%-10s %-15s %-15s %-10s", "Line_No", "Lexeme", "Token",
"Token Value"));
```

```
        for (Object[] entry : tokensTable) {
            System.out.println(String.format("%-10d %-15s %-15s %-10s", entry[0], entry[1], entry[2],
entry[3]));
        }
    }

    public void lexicalAnalysis() throws CustomException{
        scanProgram();
        printTables();
    }
}

public class Assignment_6 extends LexicalAnalyzer {
    public static void main(String[] args) {
        Assignment_6 analyzer = new Assignment_6();
        try {
            analyzer.lexicalAnalysis();
        } catch (CustomException e) {
            System.out.println("Error encountered: " + e);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```