SAKHI PATEL

SCANNER

```
/*
 *
 *
 Reads a specified input file,
 Stores individual line from file in a buffer
Passes individual lines in lexical analyzer
 lexical analyzer takes char input and returns token nums
 *
 *
*/
package Scanner;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import Scanner.LexAnalyzer;
import Scanner.Token;

public class Scanner
{

        private static BufferedReader br;
        private static String[] str;
        static LexAnalyzer lexer;


        public static void main(String[] args) throws IOException,FileNotFoundException
         {


                 String inFile = "C:\\Users\\sakhi\\workspace\\Lex\\src\\Test"; //Input File

                ArrayList<String> list = new ArrayList<String>();               //Buffer of lines from
                                                                                //input file
                String line;

                br = new BufferedReader ( new FileReader(inFile));

                while((line = br.readLine()) != null)
                        {


                                list.add(line);                         //Storing individual lines
                                                                        // as string and
                                                                        //adding to list


                        }

                str = list.toArray(new String[0]);                      //convert list to string array
```

```
for(int i=0; i<str.length;i++)

        {
                String s=str[i];
                  Lex(s);                                                   //call to line 1


                Token t;                                                    //object of Token class

                while ((t = lexer.nextToken()) != null)
                {

                        System.out.println(t.toString());
                        System.out.println("\n");;

                }
                }
        }


        private static void Lex(String s) throws IOException
                {
                        String str = s;
                        lexer = new LexAnalyzer(str);            //send individual lines to Lexical
                                                                 //Analyzer
                }
 }
```

//LEXICAL ANALYZER CLASS

```
package Scanner;
 import java.io.BufferedReader;
import java.io.ByteArrayInputStream;
import java.io.IOException; import
java.io.InputStream; import
java.io.InputStreamReader; import
java.io.Reader;
        /*
* Class LexAnalyzer
* reads the input for a file and appropriates it
* to the proper token
         */
        public class LexAnalyzer
        {

                private BufferedReader reader;
                private char curr;

                private static final char EOF = (char) (-1);

                // End of file

                public LexAnalyzer(String s) throws IOException
                {
                        String str=s;

                        //convert string to InputStream
                        ByteArrayInputStream is = new ByteArrayInputStream(str.getBytes());

                        reader = new BufferedReader(new InputStreamReader(is));          //reads line 1
```

```
                //reads first character
               curr = getchar();
        }


    private char getchar() throws IOException
        {
                char c = (char) reader.read();                          //returns one character
                return c;
        }
        // Checks if a character is a digit


  private boolean isNumeric(char c)
         {
                if (c >= '0' && c <= '9')
                        return true;

                return false;
        }



        // Checks if a character is a alphabet


  public boolean isAlpha(char c)
        {
                if (c >= 'a' && c <= 'z')
                        return true;
                if (c >= 'A' && c <= 'Z')
                        return true;

                return false;

        }



public Token nextToken() throws IOException
        {

                int state = 1;                          // Initial state
                int numBuffer = 0;                      // A buffer for number literals
        String alphaBuffer = "";                       boolean skipped = false;
        while (true) {
                        if (curr == EOF && !skipped)
                         {
                                skipped = true;

                         }

                        else if (skipped)
                         {

                                try
                                {
                                        reader.close();

                                        }  catch
                                (IOException e)  {
                                        e.printStackTrace();
                                }
```

```
                        return null;
                }

                switch (state)
                 {
                // Controller
case 1:
          switch (curr)
                {
                 case ' ': // White space
                 case '\n':
                 case '\b':
                case '\f':
                case '\r':
                case '\t':
                        curr = getchar();
                        continue;


                        //TOKEN 4
                case '+':
                  curr = getchar();
                  return new Token(" 4 for addition operators ", "+");


                 case '-':
                 curr = getchar();
                 return new Token(" 4 for addition operators ", "-");



        //TOKEN 5


                        case '*':
                         curr = getchar();
                        return new Token(" 5 for multiplication operators ","*");


                        case '%':
                         curr = getchar();
                        return new Token(" 5 for multiplication operators ","%");


                        case '/':
                         curr = getchar();
                         return new Token(" 5 for multiplication operators ","/");



        //TOKEN 6
                        case '=':
                          curr = getchar();
                          state = 7 ;  //goes to case 7 to check for = and ==
                         continue;



                        case '!':
                        curr = getchar(); //goes to case 8 to check for != and !
```

```
state = 8;
continue;




case ',':
 curr = getchar();
return new Token(" 6 for relational operators ", ",");



case '>':
 curr = getchar();
 return new Token(" 6 for relational operators ", ">");



case '(':
 curr = getchar();
 return new Token(" 14 for open parenthesis ", "(");


 case ')':
 curr = getchar();
 return new Token(" 15 for close parenthesis ", ")");


 case ';':
  curr = getchar();
 return new Token(" 16 for semicolon ", ";");


 case '.':
  curr = getchar();
        return new Token(" 18 for dot ", ".");



        //INVALID
        case ':':
                curr = getchar();
                state = 12;
                continue;

        case '&':

               curr=getchar();
                state = 9;
                continue;

        case '|':
               curr = getchar();
                state = 10;
                continue;


        //STRING
        case '"':
               curr = getchar();
                state = 11; //goes to case 11 for string
                alphaBuffer = "";
        continue;
```

```
        default:
                state = 2; // Check the next possibility
                continue;
        }



        // Integer - Start
case 2:
        if (isNumeric(curr))
         {
                numBuffer = 0; // Reset the buffer.
                numBuffer += (curr - '0');
                state = 3;
               curr = getchar();
        }


else
        {
                 state = 4; // does not start with number or symbol go to
                                    // case 4
        }
        continue;

        // Integer - Body


case 3:
        if (isNumeric(curr))
        {
                numBuffer *= 10;
                numBuffer += (curr - '0');
                curr = getchar();
        }
        else
        {
                return new Token("2 for literals", "" + numBuffer);
        }

        continue;







        //identifier -start
case 4:
               if (isAlpha(curr))
        {
                alphaBuffer = "";
                alphaBuffer += curr;
                state = 5;
              curr = getchar();
        }
        else {

                alphaBuffer = "";
                alphaBuffer += curr;
               curr = getchar();
                return new Token("ERROR", "Invalid input:" +alphaBuffer);
        }
        continue;
```

```
// identifier - Body
case 5:
        if ((isAlpha(curr) || isNumeric(curr) || curr == '_'))
        {

                alphaBuffer += curr;
                curr = getchar();

        }
         else {
                if (alphaBuffer.equals("STRING") ||
                alphaBuffer.equals("LOGICAL") ||
                alphaBuffer.equals("INTEGER"))
                 {
                return new Token(" 3 for Keywords" , "" + alphaBuffer);
                 }


                else if (alphaBuffer.equals("OR"))
                {
                return new Token(" 4 for addition operators", "" +
                alphaBuffer);
                 }


                else if (alphaBuffer.equals("AND") ||
                alphaBuffer.equals("DIV") || alphaBuffer.equals("REM"))
                {
                return new Token(" 5 for multiplication operators", "" +
                alphaBuffer);
                 }


                else if (alphaBuffer.equals("BEGIN"))
                {
                return new Token("7 for begin ", "" + alphaBuffer);
                 }


                else if (alphaBuffer.equals("END"))
                {
                return new Token("8 for end ", "" + alphaBuffer);
                 }


                else if (alphaBuffer.equals("IF"))
                {
                return new Token("9 for if statement", "" + alphaBuffer);
                 }


                else if (alphaBuffer.equals("THEN"))
                {
                return new Token("10 for then statement ", "" +
                alphaBuffer);
                 }


                else if (alphaBuffer.equals("WHILE"))
                {
                return new Token("11 for while loop ", "" + alphaBuffer);
                 }


                else if (alphaBuffer.equals("DO"))
```

```
                {
                return new Token("7 for do ", "" + alphaBuffer);
                 }


                else if (alphaBuffer.equals("READ") ||
                alphaBuffer.equals("WRITE") ||
                alphaBuffer.equals("WRITELN"))
                {
                return new Token("13 for read and write ", "" +
                alphaBuffer);
                 }


                else if (alphaBuffer.equals("TRUE") ||
                alphaBuffer.equals("FALSE"))
                {
                return new Token("2 for Boolean ", "" + alphaBuffer);
                }


                 return new Token("1 for identifier ", "" + alphaBuffer);
            }
            continue;


        case 7:
            if (curr == '=')
            {
                curr = getchar();
                return new Token("ERROR", "Invalid input: ==");
            }
             else
             {

                return new Token("6 for relational operator ", "=");
            }



        case 8:
            if (curr == '=')
             {
                curr = getchar();
                return new Token("6 for relational operator ", "!=");
            }
             else
            {
                return new Token("17 for Not equal ", "!");
            }



        case 9:
            if (curr == '&')
            {
                curr = getchar();
                return new Token("ERROR", "Invalid input: &&");
            }
            else
            {
                return new Token("ERROR", "Invalid input: &");
            }
```

```
                                case 10:
                                        if (curr == '|')
                                        {
                                               curr = getchar();
                                                return new Token("ERROR", "Invalid input: |");
                                        }  else
                                        { return new Token("ERROR", "Invalid input: ||");
                                        }



                                case 12:
                                        if (curr == '=')
                                        {
                                               curr = getchar();
                                                return new Token("19 for := ", ":=");




                                        }
                                        else
                                        {
                                                return new Token("ERROR ", "Invaid input =  :");
                                        }



                                case 11:
                                        if (curr == '"')
                                        {
                                               curr = getchar();
                                                return new Token("3 for STRING ", "\"" + alphaBuffer +
                                                "\"");
                                        }
                                        else if (curr == '\n' || curr == EOF)  {
                                               curr = getchar();
                                                return new Token("ERROR", "Invalid string literal");




                                        }
                                        else
                                        {
                                                alphaBuffer += curr;
                                               curr = getchar();
                                        }
                                        continue;
                                        }
                        default:
                                return new Token("ERROR", "Invalid string literal");
                }
        }
}

//TOKEN CLASS

package Scanner;

public class Token
{
```

```
        private String token;
        private String lexeme;

        public Token(String token, String lexeme) {
                this.token = token;
                this.lexeme = lexeme;
        }

        @Override
        public String toString() {
                return "Token [ token = " + token + ", lexeme = " + lexeme + " ] ";
        }
}
```

INPUT:test

BEGIN;

INTEGER n;

REAL M, p, S;

n := 10;

S := 12;

M := 0;

READ(p);

WHILE p > 0

DO

BEGIN

N := N + 1;

S := S + p;

WRITE("SUM IS ", S);

IF p > M

THEN M := p;

READ(p);

END;

WRITE(?n,:S,M)

END.

Output:

```
Token [ token = 7 for begin , lexeme = BEGIN ]

Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token =  3 for Keywords, lexeme = INTEGER ]

Token [ token = 1 for identifier , lexeme = n ]

Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 1 for identifier , lexeme = REAL ]

Token [ token = 1 for identifier , lexeme = M ]

Token [ token =  6 for relational operators , lexeme = , ]

Token [ token = 1 for identifier , lexeme = p ]

Token [ token =  6 for relational operators , lexeme = , ]

Token [ token = 1 for identifier , lexeme = S ]

Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 1 for identifier , lexeme = n ]

Token [ token = 19 for := , lexeme = := ]

Token [ token = 2 for literals, lexeme = 10 ]

Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 1 for identifier , lexeme = S ]
```

```
Token [ token = 19 for := , lexeme = := ]

Token [ token = 2 for literals, lexeme = 12 ]

Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 1 for identifier , lexeme = M ]

Token [ token = 19 for := , lexeme = := ]

Token [ token = 2 for literals, lexeme = 0 ]

Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 13 for read and write , lexeme = READ ]

Token [ token =  14 for open parenthesis , lexeme = ( ]

Token [ token = 1 for identifier , lexeme = p ]

Token [ token =  15 for close parenthesis , lexeme = ) ]

Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 11 for while loop , lexeme = WHILE ]

Token [ token = 1 for identifier , lexeme = p ]

Token [ token =  6 for relational operators , lexeme = > ]

Token [ token = 2 for literals, lexeme = 0 ]

Token [ token = 7 for do , lexeme = DO ]

Token [ token = 7 for begin , lexeme = BEGIN ]

Token [ token = 1 for identifier , lexeme = N ]

Token [ token = 19 for := , lexeme = := ]

Token [ token = 1 for identifier , lexeme = N ]

Token [ token =  4 for addition operators , lexeme = + ]

Token [ token = 2 for literals, lexeme = 1 ]
```

```
Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 1 for identifier , lexeme = S ]

Token [ token = 19 for := , lexeme = := ]

Token [ token = 1 for identifier , lexeme = S ]

Token [ token =  4 for addition operators , lexeme = + ]

Token [ token = 1 for identifier , lexeme = p ]

Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 13 for read and write , lexeme = WRITE ]

Token [ token =  14 for open parenthesis , lexeme = ( ]

Token [ token = 3 for STRING , lexeme = "SUM IS " ]

Token [ token =  6 for relational operators , lexeme = , ]

Token [ token = 1 for identifier , lexeme = S ]

Token [ token =  15 for close parenthesis , lexeme = ) ]

Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 9 for if statement, lexeme = IF ]

Token [ token = 1 for identifier , lexeme = p ]

Token [ token = 6 for relational operator , lexeme = != ]

Token [ token = 1 for identifier , lexeme = M ]

Token [ token = 10 for then statement , lexeme = THEN ]

Token [ token = 1 for identifier , lexeme = M ]

Token [ token = 19 for := , lexeme = := ]

Token [ token = 1 for identifier , lexeme = p ]
```

```
Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 13 for read and write , lexeme = READ ]

Token [ token =  14 for open parenthesis , lexeme = ( ]

Token [ token = 1 for identifier , lexeme = p ]

Token [ token =  15 for close parenthesis , lexeme = ) ]

Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 8 for end , lexeme = END ]

Token [ token =  16 for semicolon , lexeme = ; ]

Token [ token = 13 for read and write , lexeme = WRITE ]

Token [ token =  14 for open parenthesis , lexeme = ( ]

Token [ token = ERROR, lexeme = Invalid input:? ]

Token [ token = 1 for identifier , lexeme = n ]

Token [ token =  6 for relational operators , lexeme = , ]

Token [ token = ERROR , lexeme = Invaid input =  : ]

Token [ token = 1 for identifier , lexeme = S ]

Token [ token =  6 for relational operators , lexeme = , ]

Token [ token = 1 for identifier , lexeme = M ]

Token [ token =  15 for close parenthesis , lexeme = ) ]

Token [ token = 8 for end , lexeme = END ]

Token [ token = 18, lexeme = . ]
```