```
1 (sum + 47) / total
```

```java
import java.io.PrintWriter;
import java.util.Scanner;


/**
 * File: Lex.java
 *Lexical analyzer that
 * Created by JacobHertl
 */
public class Lex extends Syntax{
    //Variables
    private int charClass;//represents type of character
    String lexeme = "";//Holds the current lexeme
    private char nextChar;//hold the next char to be evaluated
    int nextToken;//represents the type of lexeme
    Parser pars;//the parser
    private Scanner reader;//reader that reads from front.in
    private PrintWriter writer;//writer that writes to front.out

    public Lex(Parser parser, Scanner scanner, PrintWriter fWriter)
    {
        pars = parser;
        reader = scanner;
        writer = fWriter;
    }

    /**
     * lookup operators and parentheses and return the token
     * @param ch is the character that needs to be looked up
     * @return nextToken is the type of the next token
     */
    public int lookup(char ch)
    {
        lexeme += nextChar;
        switch (ch)
        {
            case '(':
                nextToken = lParen;
                break;
            case ')':
                nextToken = rParen;
                break;
            case '+':
                nextToken = add;
                break;
            case '-':
                nextToken = sub;
                break;
            case '*':
                nextToken = mult;
                break;
            case '/':
                nextToken = div;
                break;
            default:
                nextToken = EOF;
```

```
57                  break;
58          }//end of switch
59          return nextToken;
60      }//end of fx lookup
61
62      /**
63       * gets the next character of input and determine its character class
64       */
65      public void getChar()
66      {
67          if(reader.hasNext() )
68          {
69              nextChar = reader.next().charAt(0);//gets next character
70              if(Character.isLetter(nextChar))
71              {
72                  charClass = alpha;
73              }
74              else if (Character.isDigit(nextChar))
75              {
76                  charClass = digit;
77              }
78              else
79              {
80                  charClass = unknown;
81              }
82          }
83          else//if reader does not have next it must be end of file
84          {
85              charClass = EOF;
86          }
87      }//end of fx getChar
88
89      /**
90       * lexical analysis
91       * @return nextToken is the type of the next lexeme
92       */
93      public int lex()
94      {
95          lexeme = "";
96          while (Character.isWhitespace(nextChar))//continues to pass through extra
    space
97          {
98              getChar();
99          }
100         switch (charClass)
101         {
102             case alpha://continues to get all character of lexeme
103                 lexeme += nextChar;
104                 getChar();
105                 while(charClass == alpha ||charClass == digit)
106                 {
107                     lexeme += nextChar;
108                     getChar();
109                 }
110                 nextToken = id;
111                 break;
```

```
112             case digit://continues to get all digits of a number
113                 lexeme += nextChar;
114                 getChar();
115                 while (charClass == digit)
116                 {
117                     lexeme += nextChar;
118                     getChar();
119                 }
120                 nextToken = num;
121                 break;
122             case unknown: //parentheses and operators
123                 lookup(nextChar);
124                 getChar();
125                 break;
126             case EOF:
127                 nextToken = EOF;
128                 lexeme = "EOF";
129                 break;
130         }//End of switch
131         System.out.println("Next token is: " + nextToken + " next lexeme is " +
    lexeme );
132         writer.println("Next token is: " + nextToken + " next lexeme is " +
    lexeme );
133         return nextToken;
134     }//End of function lex
135
136
137 }
138
```

```
Next token is: 9 next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 4 next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 5 next lexeme is +
Exit <factor>
Exit <term>
Next token is: 3 next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 10 next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 8 next lexeme is /
Exit <factor>
Next token is: 4 next lexeme is total
Enter <factor>
Next token is: -1 next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>

```

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.PrintWriter;
import java.util.Scanner;

/**
 * File:Parser.java
 * Created by JacobHertl
 */
public class Parser extends Syntax{
    private File inF;//the file to be read
    private File outF;//file to be written to
    private Scanner reader;
    private PrintWriter writer;
    Lex lexA;//the lexical analyzer

    /**
     * creates the paser and assigns necessary files
     * @param args there are no arguments needed for this program
     */
    public static void main(String[] args)
    {
        Parser pars = new Parser();
        //OPen the input file and process its contents
        pars.inF = new File("front.in");
        pars.outF = new File("front.out");
        System.out.println("files opened");
        pars.startParsing();
    }

    /**
     * opens both files to be read and written to
     * starts the parsing and closes the files when the parsing is finished
     */
    public void startParsing()
    {
        try
        {
            reader = new Scanner(new BufferedReader(new FileReader(inF)));
            reader.useDelimiter("");
            writer = new PrintWriter(outF, "UTF-8");
        }
        catch (Exception e){
            System.out.println("Error reading file " + e);
        }
        lexA = new Lex(this, reader, writer);
        lexA.getChar();
        while(lexA.nextToken != EOF)
        {
            lexA.lex();
            expr();
        }
        reader.close();
        writer.close();
    }
```

```
57
58      /**
59       * parses expressions by following the rule:
60       * <expr> -> <term> {(+|-)<term>}
61       */
62      public void expr()
63      {
64          System.out.println("Enter <expr>");
65          writer.println("Enter <expr>");
66          term();//parses the first term
67          while ((lexA.nextToken == add)||(lexA.nextToken == sub))
68          {
69              lexA.lex();
70              term();
71          }
72          System.out.println("Exit <expr>");
73          writer.println("Exit <expr>");
74      }// end of fx expr
75
76      /**
77       * parses terms by following the rule:
78       * <term> -> <factor> {(*|/)<factor>}
79       */
80      public void term()
81      {
82          System.out.println("Enter <term>");
83          writer.println("Enter <term>");
84          factor();//parses the first factor
85          while ((lexA.nextToken == mult)|| (lexA.nextToken == div))
86          {
87              lexA.lex();
88              factor();
89          }
90          System.out.println("Exit <term>");
91          writer.println("Exit <term>");
92      }//end of fx term
93
94      /**
95       * parses factors by following the rule:
96       * <factor> -> id | int_constant | (<expr>)
97      */
98      public void factor()
99      {
100         System.out.println("Enter <factor>");
101         writer.println("Enter <factor>");
102         if((lexA.nextToken == id) || (lexA.nextToken == num))
103         {
104             lexA.lex();
105         }
106         //if the RHS is (<expr>), call lex to pass over the left parenthesis,
    call expr,
107         //and check for the right parenthesis
108         else
109         {
110             try
111             {
```

```
112                 if(lexA.nextToken == lParen)
113                 {
114                     lexA.lex();
115                     expr();
116                     if(lexA.nextToken == rParen)
117                     {
118                         lexA.lex();
119                     }
120                     else//
121                     {
122                         throw new Error("Parenthesis expected before " + lexA.
    lexeme);
123                     }
124                 }
125                 //it was not an id, an integer literal, or a left parenthesis
126                 else//it was not a parenthesis
127                 {
128                     throw new Error("Expression surrounded by parentheses
    expected before " + lexA.lexeme);
129                 }
130             }
131             catch (Error e)
132             {
133                 System.out.println(e);
134             }
135         }//end of else right_paren
136         System.out.println("Exit <factor>");
137         writer.println("Exit <factor>");
138     }//end of fx factor
139
140
141
142
143 }
144
```

```java

/**
 * File: Syntax.java
 * This contains the lexeme and character codes used by the parser as well as the error subprogram
 * Syntax is inherited by the parser and lexical analyzer so they can use these important codes and the error class
 * Created by JacobHertl
 */
public abstract class Syntax {
    //character codes
    static final int unknown = 0;//represents characters that need to be looked up or are unknown
    static final int alpha = 1;//represents characters of the alphabet
    static final int digit = 2;//represents digits of a number
    //Lexeme codes
    static final int num = 3;//represents numbers
    static final int id = 4;//represents identifiers
    static final int add = 5;//represents addition operators
    static final int sub = 6;//represents subtraction operators
    static final int mult = 7;//represents multiplication operators
    static final int div = 8;//represents division operators
    static final int lParen = 9;//represents left parentheses
    static final int rParen = 10;//represents right parentheses
    static final int EOF = -1;//represents the end of the file

    /**
     * Class: Error
     * Error handles errors and identifies what may have caused it
     */
    class Error extends Exception
    {
        Error(){}

        Error (String message)
        {
            super (message);
        }


    }
}

```