```java
import java.util.Map;

/**
 * Classname: Cos
 * An Unary expression composed of an operator which provides the cosine of the expression.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */

public class Cos extends UnaryExpression implements Expression {

    //members
    private Expression exp;

    /**
     * main constructor.
     *
     * @param exp Expression
     */
    public Cos(Expression exp) {
        this.exp = exp;
    }

    /**
     * shortcut constructor.
     *
     * @param num - double
     */
    public Cos(double num) {
        this(new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param var - String
     */
    public Cos(String var) {
        this(new Var(var));
    }

    /**
     * getter for Expression.
     *
     * @return Expression
     */
    public Expression getExp() {
        return this.exp;
    }


    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate(Map<String, Double> assignment) throws Exception {
        double value = exp.evaluate(assignment);
        //fix incorrect values returned by Math.cos for these angles
        if (value % 360 == 90 || value % 360 == 270) {
            return 0;
        }
        return Math.cos(Math.toRadians(value));
    }

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    public String toString() {
        return "cos(" + this.exp.toString() + ")";
    }

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param var        to replace
     * @param expression to assign instead of the var
     * @return new Expression after assigning
     */
    public Expression assign(String var, Expression expression) {
        return new Cos(this.exp.assign(var, expression));
```

```
    }

    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
     *
     * @param var differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    public Expression differentiate(String var) {
        return new Mult(new Neg(new Sin(exp)), exp.differentiate(var));
    }

    /**
     * Returns a simplified version of the current expression.
     *
     * @return simplified version of the current expression
     */
    public Expression simplify() {
        Cos cos = new Cos(this.exp.simplify());
        Expression newExp = cos.getExp();
        //Expression doesn't have any Variables
        if (newExp.getVariables().isEmpty()) {
            double value = 0;
            try {
                value = cos.evaluate();
            } catch (Exception evalFailed) {
                throw new RuntimeException("Evaluation failed!");
            }
            return new Num(value);
        }
        //Expression has a least one Var
        return new Cos(newExp.simplify());
    }

    /**
     * Returns a simplified version of the current expression for the bonus part.
     *
     * @return simplified version of the current expression
     */
    public Expression simplifyBonus() {
        return this.simplify();
    }
}
```

```java
import java.util.Map;

/**
 * Classname: Div
 * A binary expression composed of an operator which Divides the left Expression by the right Expression.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public class Div extends BinaryExpression implements Expression {

    //members
    private Expression left; //numerator
    private Expression right; //denominator

    /**
     * main constructor.
     *
     * @param left  left Expression
     * @param right right Expression
     */
    public Div(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param num right Expression - double
     */
    public Div(Expression exp, double num) {
        this(exp, new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param var right Expression - String
     */
    public Div(Expression exp, String var) {
        this(exp, new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param num1 left Expression - double
     * @param num2 right Expression - double
     */
    public Div(double num1, double num2) {
        this(new Num(num1), new Num(num2));
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param exp right Expression - Expression
     */
    public Div(double num, Expression exp) {
        this(new Num(num), exp);
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param var right Expression - String
     */
    public Div(double num, String var) {
        this(new Num(num), new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param var1 left Expression - String
     * @param var2 right Expression - String
     */
    public Div(String var1, String var2) {
        this(new Var(var1), new Var(var2));
    }

    /**
     * shortcut constructor.
     *
     * @param var left Expression - String
```

```java
     * @param num right Expression - double
     */
    public Div(String var, double num) {
        this(new Var(var), new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param var left Expression - String
     * @param exp right Expression - Expression
     */
    public Div(String var, Expression exp) {
        this(new Var(var), exp);
    }

    /**
     * getter for left Expression.
     *
     * @return left Expression
     */
    public Expression getLeft() {
        return left;
    }

    /**
     * getter for right Expression.
     *
     * @return right Expression
     */
    public Expression getRight() {
        return right;
    }


    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate(Map<String, Double> assignment) throws Exception {

        if (right.evaluate(assignment) == 0) {
            throw new RuntimeException("Cannot divide by 0!");
        }
        return left.evaluate(assignment) / right.evaluate(assignment);
    }

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    public String toString() {
        return "(" + this.left.toString() + " / " + this.right.toString() + ")";
    }

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param var        to replace
     * @param expression to assign instead of the var
     * @return new Expression after assigning
     */
    public Expression assign(String var, Expression expression) {
        return new Div(this.left.assign(var, expression), this.right.assign(var, expression));
    }

    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
     *
     * @param var differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    public Expression differentiate(String var) {
        Expression leftDif = this.left.differentiate(var);
        Expression rightDif = this.right.differentiate(var);
        return new Div(new Minus(new Mult(leftDif, this.right), new Mult(this.left, rightDif)), new Pow(this.right, 2));
    }

    /**
     * Returns a simplified version of the current expression.
     *
```

```
     * @return simplified version of the current expression
     */
    public Expression simplify() {
        //Expression doesn't have any Variables
        Div div = new Div(this.left.simplify(), this.right.simplify());
        Expression newLeft = div.getLeft();
        Expression newRight = div.getRight();
        if (newLeft.getVariables().isEmpty() && newRight.getVariables().isEmpty()) {
            double value = 0;
            try {
                value = div.evaluate();
            } catch (Exception evalFailed) {
                throw new RuntimeException("Evaluation failed!");
            }
            return new Num(value);
        } else { //Expression has a least one Var
            //simplifying X / X --> 1
            if (newRight.toString().equals(newLeft.toString())) {
                return new Num(1);
            }
            //simplifying X / 1 --> X
            if (newRight.toString().equals("1") || newRight.toString().equals("1.0")) {
                return newLeft.simplify();
            }
        }
        return new Div(newLeft.simplify(), newRight.simplify());
    }


    /**
     * Returns a simplified version of the current expression for the bonus part.
     *
     * @return simplified version of the current expression
     */
    public Expression simplifyBonus() {
        return this.simplify();
    }
}
```

```java
import java.util.Map;

/**
 * Classname: Log
 * A binary expression composed of an operator which provides the logarithm the right Expression in the base of the
 * left Expression.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public class Log extends BinaryExpression implements Expression {

    //members
    private Expression left; //base
    private Expression right; //log of

    /**
     * main constructor.
     *
     * @param left  left Expression
     * @param right right Expression
     */
    public Log(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param num right Expression - double
     */
    public Log(Expression exp, double num) {
        this(exp, new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param var right Expression - String
     */
    public Log(Expression exp, String var) {
        this(exp, new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param num1 left Expression - double
     * @param num2 right Expression - double
     */
    public Log(double num1, double num2) {
        this(new Num(num1), new Num(num2));
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param exp right Expression - Expression
     */
    public Log(double num, Expression exp) {
        this(new Num(num), exp);
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param var right Expression - String
     */
    public Log(double num, String var) {
        this(new Num(num), new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param var1 left Expression - String
     * @param var2 right Expression - String
     */
    public Log(String var1, String var2) {
        this(new Var(var1), new Var(var2));
    }

    /**
     * shortcut constructor.
     *
```

```java
     * @param var left Expression - String
     * @param num right Expression - double
     */
    public Log(String var, double num) {
        this(new Var(var), new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param var left Expression - String
     * @param exp right Expression - Expression
     */
    public Log(String var, Expression exp) {
        this(new Var(var), exp);
    }

    /**
     * getter for left Expression.
     *
     * @return left Expression
     */
    public Expression getLeft() {
        return left;
    }

    /**
     * getter for right Expression.
     *
     * @return right Expression
     */
    public Expression getRight() {
        return right;
    }


    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate(Map<String, Double> assignment) throws Exception {
        double evalbase = left.evaluate(assignment);
        double evalLogOf = right.evaluate(assignment);
        if (evalbase <= 0 || evalbase == 1 || evalLogOf <= 0) {
            throw new Exception("Invalid Log values!");
        } else {
            return Math.log(evalLogOf) / Math.log(evalbase);
        }
    }

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    public String toString() {
        return "log(" + this.left.toString() + ", " + this.right.toString() + ")";
    }

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param var        to replace
     * @param expression to assign instead of the var
     * @return new Expression after assigning
     */
    public Expression assign(String var, Expression expression) {
        return new Log(this.left.assign(var, expression), this.right.assign(var, expression));
    }

    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
     *
     * @param var differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    public Expression differentiate(String var) {
        //log b(x)= 1/(x*ln(b))
        return new Div(1, new Mult(this.right, new Log("e", this.left)));
    }

    /**
```

```java
    /**
     * Returns a simplified version of the current expression.
     *
     * @return simplified version of the current expression
     */
    public Expression simplify() {
        Log log = new Log(this.left.simplify(), this.right.simplify());
        Expression newLeft = log.getLeft();
        Expression newRight = log.getRight();
        //Expression doesn't have any Variables
        if (newLeft.getVariables().isEmpty() && newRight.getVariables().isEmpty()) {
            double value = 0;
            try {
                value = log.evaluate();
            } catch (Exception evalFailed) {
                throw new RuntimeException("Evaluation failed!");
            }
            return new Num(value);
        } else { //Expression has a least one Var
            //simplifying log(X, X) --> 1
            if (newRight.toString().equals(newLeft.toString())) {
                return new Num(1);
            }
        }
        return new Log(newLeft.simplify(), newRight.simplify());
    }

    /**
     * Returns a simplified version of the current expression for the bonus part.
     *
     * @return simplified version of the current expression
     */
    public Expression simplifyBonus() {
        return this.simplify();
    }
}
```

```java
import java.util.Map;

/**
 * Classname: Neg
 * An Unary operator which provides the negation of the expression.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */

public class Neg extends UnaryExpression implements Expression {

    //members
    private Expression exp;

    /**
     * main constructor.
     *
     * @param exp Expression
     */
    public Neg(Expression exp) {
        this.exp = exp;
    }

    /**
     * shortcut constructor.
     *
     * @param num - double
     */
    public Neg(double num) {
        this(new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param var - String
     */
    public Neg(String var) {
        this(new Var(var));
    }

    /**
     * getter for Expression.
     *
     * @return Expression
     */
    public Expression getExp() {
        return this.exp;
    }


    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate(Map<String, Double> assignment) throws Exception {
        return -this.exp.evaluate(assignment);
    }

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    public String toString() {
        return "(-" + this.exp.toString() + ")";
    }

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param var        to replace
     * @param expression to assign instead of the var
     * @return new Expression after assigning
     */
    public Expression assign(String var, Expression expression) {
        return new Neg(this.exp.assign(var, expression));
    }

    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
```

```java
     *
     * @param var differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    public Expression differentiate(String var) {
        return new Neg(this.exp.differentiate(var));
    }

    /**
     * Returns a simplified version of the current expression.
     *
     * @return simplified version of the current expression
     */
    public Expression simplify() {
        return this;
    }

    /**
     * activates the additional simplification if exists for the bonus part.
     *
     * @return new bonus-simplified Expression.
     */
    public Expression simplifyBonus() {
        //-0 --> 0
        if (this.exp.simplifyBonus().simplify().toString().equals("0.0")) {
            return new Num(0.0);
        }
        //(-(-x)) --> x
        if (this.exp instanceof Neg) {
            return ((Neg) this.exp).exp;
        }
        return new Neg(this.getExp().simplify().simplifyBonus());
    }

}
```

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

/**
 * Classname: Num
 * A class representing a number.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public class Num implements Expression {
    private double num;

    /**
     * double constructor.
     *
     * @param num number
     */
    public Num(double num) {
        this.num = num;
    }

    /**
     * int constructor.
     *
     * @param num number
     */
    public Num(int num) {
        this.num = (double) num;
    }


    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate(Map<String, Double> assignment) throws Exception {
        return this.num;
    }

    /**
     * A convenience method. Like the `evaluate(assignment)` method above,
     * but uses an empty assignment.
     *
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate() throws Exception {
        return this.num;
    }

    /**
     * Returns a list of the variables in the expression.
     *
     * @return list of Strings
     */
    public List<String> getVariables() {
        return new ArrayList<String>();
    }

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    public String toString() {
        return String.valueOf(this.num);
    }

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param var        to replace
     * @param expression to assign instead of the var
     * @return new Expression after assigning
     */
    public Expression assign(String var, Expression expression) {
        return this;
    }


    /**
```

```
    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
     *
     * @param var differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    public Expression differentiate(String var) {
        return new Num(0);
    }

    /**
     * Returns a simplified version of the current expression.
     *
     * @return simplified version of the current expression
     */
    public Expression simplify() {
        return this;
    }

    /**
     * Returns a simplified version of the current expression for the bonus part.
     *
     * @return simplified version of the current expression
     */
    public Expression simplifyBonus() {
        return this;
    }

    /**
     * swaps the sides of this expression- left to right, right to left.
     *
     * @return new Expression of this expression with swapped sides
     */
    public Expression swapSides() {
        return null;
    }
}
```

```java
import java.util.Map;

/**
 * Classname: Pow
 * A binary expression composed of an operator which provides the power with the left Expression as a base, and the
 * right expression as the exponent.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public class Pow extends BinaryExpression implements Expression {
    //members
    private Expression left; //base
    private Expression right; //exponent

    /**
     * main constructor.
     *
     * @param left  left Expression
     * @param right right Expression
     */
    public Pow(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param num right Expression - double
     */
    public Pow(Expression exp, double num) {
        this(exp, new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param var right Expression - String
     */
    public Pow(Expression exp, String var) {
        this(exp, new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param num1 left Expression - double
     * @param num2 right Expression - double
     */
    public Pow(double num1, double num2) {
        this(new Num(num1), new Num(num2));
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param exp right Expression - Expression
     */
    public Pow(double num, Expression exp) {
        this(new Num(num), exp);
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param var right Expression - String
     */
    public Pow(double num, String var) {
        this(new Num(num), new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param var1 left Expression - String
     * @param var2 right Expression - String
     */
    public Pow(String var1, String var2) {
        this(new Var(var1), new Var(var2));
    }

    /**
     * shortcut constructor.
     *
     * @param var left Expression - String
```

```java
     * @param num right Expression - double
     */
    public Pow(String var, double num) {
        this(new Var(var), new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param var left Expression - String
     * @param exp right Expression - Expression
     */
    public Pow(String var, Expression exp) {
        this(new Var(var), exp);
    }

    /**
     * getter for left Expression.
     *
     * @return left Expression
     */
    public Expression getLeft() {
        return left;
    }

    /**
     * getter for right Expression.
     *
     * @return right Expression
     */
    public Expression getRight() {
        return right;
    }


    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate(Map<String, Double> assignment) throws Exception {
        double eval = Math.pow(left.evaluate(assignment), right.evaluate(assignment));
        if (Double.isNaN(eval)) {
            throw new Exception("Cannot evaluate the power of negative base by fractional exponent!");
        }
        return eval;
    }

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    public String toString() {
        return "(" + this.left.toString() + "^" + this.right.toString() + ")";
    }

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param var        to replace
     * @param expression to assign instead of the var
     * @return new Expression after assigning
     */
    public Expression assign(String var, Expression expression) {
        return new Pow(this.left.assign(var, expression), this.right.assign(var, expression));
    }


    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
     *
     * @param var differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    public Expression differentiate(String var) {
        Expression baseDif = this.left.differentiate(var);
        Expression exponentDif = this.right.differentiate(var);
        //Generalized power rule: (f^g)'=(f^g)*(f'*(g/f)+g'*ln(f)
        Expression powerFbyG = new Pow(this.left, this.right);
        Expression divGbyF = new Div(this.right, this.left);
        Expression multDifGbyLnF = new Mult(exponentDif, new Log("e", this.left));
        return new Mult(powerFbyG, new Plus(new Mult(baseDif, divGbyF), multDifGbyLnF));
```

```
    }


    /**
     * Returns a simplified version of the current expression.
     *
     * @return simplified version of the current expression
     */
    public Expression simplify() {
        Pow pow = new Pow(this.left.simplify(), this.right.simplify());
        Expression newLeft = pow.getLeft();
        Expression newRight = pow.getRight();
        //Expression doesn't have any Variables
        if (newLeft.getVariables().isEmpty() && newRight.getVariables().isEmpty()) {
            double value = 0;
            try {
                value = pow.evaluate();
            } catch (Exception evalFailed) {
                throw new RuntimeException("Evaluation failed!");
            }
            return new Num(value);
        } else { //Expression has a least one Var
            return new Pow(newLeft.simplify(), newRight.simplify());
        }
    }

    /**
     * Returns a simplified version of the current expression for the bonus part.
     *
     * @return simplified version of the current expression
     */
    public Expression simplifyBonus() {
        return this.simplify();
    }
}
```

```java
import java.util.Map;

/**
 * Classname: Sin
 * An Unary expression composed of an operator which provides the Sinine of the expression.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */

public class Sin extends UnaryExpression implements Expression {

    //members
    private Expression exp;

    /**
     * main constructor.
     *
     * @param exp Expression
     */
    public Sin(Expression exp) {
        this.exp = exp;
    }

    /**
     * shortcut constructor.
     *
     * @param num - double
     */
    public Sin(double num) {
        this(new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param var - String
     */
    public Sin(String var) {
        this(new Var(var));
    }

    /**
     * getter for Expression.
     *
     * @return Expression
     */
    public Expression getExp() {
        return this.exp;
    }


    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate(Map<String, Double> assignment) throws Exception {
        double value = exp.evaluate(assignment);
        //fix incorrect values returned by Math.sin for these angles
        if (value % 360 == 0 || value % 360 == 180) {
            return 0;
        }
        return Math.sin(Math.toRadians(value));
    }

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    public String toString() {
        return "sin(" + this.exp.toString() + ")";
    }

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param var        to replace
     * @param expression to assign instead of the var
     * @return new Expression after assigning
     */
    public Expression assign(String var, Expression expression) {
        return new Sin(this.exp.assign(var, expression));
```

```
    }


    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
     *
     * @param var differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    public Expression differentiate(String var) {
        return new Mult(new Cos(exp), exp.differentiate(var));
    }

    /**
     * Returns a simplified version of the current expression.
     *
     * @return simplified version of the current expression
     */
    public Expression simplify() {
        Sin sin = new Sin(this.exp.simplify());
        Expression newExp = sin.getExp();
        //Expression doesn't have any Variables
        if (newExp.getVariables().isEmpty()) {
            double value = 0;
            try {
                value = sin.evaluate();
            } catch (Exception evalFailed) {
                throw new RuntimeException("Evaluation failed!");
            }
            return new Num(value);
        }
        //Expression has a least one Var
        return new Sin(newExp.simplify());
    }

    /**
     * Returns a simplified version of the current expression for the bonus part.
     *
     * @return simplified version of the current expression
     */
    public Expression simplifyBonus() {
        return this.simplify();
    }
}
```

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

/**
 * Classname: Var
 * A class representing a Var.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public class Var implements Expression {

    private String var;

    /**
     * constructor.
     *
     * @param var variable
     */
    public Var(String var) {
        this.var = var;
    }

    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate(Map<String, Double> assignment) throws Exception {
        if (assignment.containsKey(this.var)) {
            return assignment.get(this.var);
        } else {
            if (this.var.equals("e") && !assignment.containsKey(this.var)) {
                return Math.E;
            }
            throw new Exception("The expression contains a variable which is not in the assignment!");
        }
    }

    /**
     * A convenience method. Like the `evaluate(assignment)` method above,
     * but uses an empty assignment.
     *
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate() throws Exception {
        throw new Exception("The expression contains a variable which is not in the assignment!");
    }

    /**
     * Returns a list of the variables in the expression.
     *
     * @return list of Strings
     */
    public List<String> getVariables() {
        List<String> vars = new ArrayList<String>();
        vars.add(this.var);
        return vars;
    }

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    public String toString() {
        return var;
    }

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param varToReplace variable to replace
     * @param expression   to assign instead of the var
     * @return new Expression after assigning
     */
    public Expression assign(String varToReplace, Expression expression) {
        if (varToReplace.equals(this.var)) {
            return expression;
        }
        //assignment of another variable - returns the Expression as is.
        return this;
```

```
    }

    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
     *
     * @param varToDif differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    public Expression differentiate(String varToDif) {
        if (varToDif.equals(this.var)) {
            return new Num(1);
        } else {
            return new Num(0);
        }
    }

    /**
     * Returns a simplified version of the current expression.
     *
     * @return simplified version of the current expression
     */
    public Expression simplify() {
        return this;
    }

    /**
     * swaps the sides of this expression- left to right, right to left.
     *
     * @return new Expression of this expression with swapped sides
     */
    public Expression swapSides() {
        return null;
    }

    /**
     * Returns a simplified version of the current expression for the bonus part.
     *
     * @return simplified version of the current expression
     */
    public Expression simplifyBonus() {
        return this;
    }

}
```

```java
import java.util.Map;

/**
 * Classname: Mult
 * A binary expression composed of an operator which multiplies the expressions from both of its sides.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public class Mult extends BinaryExpression implements Expression {

    //members
    private Expression left;
    private Expression right;

    /**
     * main constructor.
     *
     * @param left  left Expression
     * @param right right Expression
     */
    public Mult(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param num right Expression - double
     */
    public Mult(Expression exp, double num) {
        this(exp, new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param var right Expression - String
     */
    public Mult(Expression exp, String var) {
        this(exp, new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param num1 left Expression - double
     * @param num2 right Expression - double
     */
    public Mult(double num1, double num2) {
        this(new Num(num1), new Num(num2));
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param exp right Expression - Expression
     */
    public Mult(double num, Expression exp) {
        this(new Num(num), exp);
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param var right Expression - String
     */
    public Mult(double num, String var) {
        this(new Num(num), new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param var1 left Expression - String
     * @param var2 right Expression - String
     */
    public Mult(String var1, String var2) {
        this(new Var(var1), new Var(var2));
    }

    /**
     * shortcut constructor.
     *
     * @param var left Expression - String
```

```
     * @param num right Expression - double
     */
    public Mult(String var, double num) {
        this(new Var(var), new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param var left Expression - String
     * @param exp right Expression - Expression
     */
    public Mult(String var, Expression exp) {
        this(new Var(var), exp);
    }

    /**
     * getter for left Expression.
     *
     * @return left Expression
     */
    public Expression getLeft() {
        return left;
    }

    /**
     * getter for right Expression.
     *
     * @return right Expression
     */
    public Expression getRight() {
        return right;
    }


    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate(Map<String, Double> assignment) throws Exception {
        return left.evaluate(assignment) * right.evaluate(assignment);
    }

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    public String toString() {
        return "(" + this.left.toString() + " * " + this.right.toString() + ")";
    }

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param var        to replace
     * @param expression to assign instead of the var
     * @return new Expression after assigning
     */
    public Expression assign(String var, Expression expression) {
        return new Mult(this.left.assign(var, expression), this.right.assign(var, expression));
    }


    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
     *
     * @param var differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    public Expression differentiate(String var) {
        return new Plus(new Mult(this.left.differentiate(var), this.right),
                new Mult(this.left, this.right.differentiate(var)));
    }

    /**
     * Returns a simplified version of the current expression.
     *
     * @return simplified version of the current expression
     */
    public Expression simplify() {
        Mult mult = new Mult(this.left.simplify(), this.right.simplify());
```

```
    Expression newLeft = mult.getLeft();
    Expression newRight = mult.getRight();
    //Expression doesn't have any Variables
    if (newLeft.getVariables().isEmpty() && newRight.getVariables().isEmpty()) {
        double value = 0;
        try {
            value = mult.evaluate();
        } catch (Exception evalFailed) {
            throw new RuntimeException("Evaluation failed!");
        }
        return new Num(value);
    } else { //Expression has a least one Var
        //simplifying X * 0 --> 0
        if (newRight.toString().equals("0") || newRight.toString().equals("0.0")) {
            return new Num(0);
        }
        //simplifying 0 * X --> X
        if (newLeft.toString().equals("0") || newLeft.toString().equals("0.0")) {
            return new Num(0);
        }
        //simplifying X * 1 --> X
        if (newRight.toString().equals("1") || newRight.toString().equals("1.0")) {
            return newLeft.simplify();
        }
        //simplifying 1 * X --> X
        if (newLeft.toString().equals("1") || newLeft.toString().equals("1.0")) {
            return newRight.simplify();
        }
    }
    return new Mult(newLeft.simplify(), newRight.simplify());
}

/**
 * Returns a simplified version of the current expression for the bonus part.
 *
 * @return simplified version of the current expression
 */
public Expression simplifyBonus() {
    //((x + y) * (y + x)) --> (x + y)^2, (X * X) --> (2 ^ X)
    if (leftEqualsRight()) {
        return new Pow(this.left.simplifyBonus(), 2);
    }
    // (-x) * (-y) --> (x * y)
    if (this.left instanceof Neg && this.right instanceof Neg) {
        Neg negLeft = (Neg) this.left;
        Neg negRight = (Neg) this.right;
        return new Mult(negLeft.getExp(), negRight.getExp());
    }
    // x * (-y) --> -(x * y)
    if (this.right instanceof Neg) {
        Neg negRight = (Neg) this.right;
        return new Neg(new Mult(this.left, negRight.getExp()));
    }
    // (-x) * y --> -(x * y)
    if (this.left instanceof Neg) {
        Neg negLeft = (Neg) this.left;
        return new Neg(new Mult(negLeft.getExp(), this.right));
    }
    return new Mult(this.getLeft().simplify().simplifyBonus(), this.getRight().simplify().simplifyBonus());
}

/**
 * swaps the sides of this expression- left to right, right to left.
 *
 * @return new Expression of this expression with swapped sides
 */
@Override
public Expression swapSides() {
    return new Plus(this.right, this.left);
}

/**
 * checks whether the left side of the Expression equals(mathematically!) the right side.
 *
 * @return true if equals, false otherwise.
 */
public Boolean leftEqualsRight() {
    if (this.left.toString().equals(this.right.toString())) {
        return true;
    }
    if ((this.left instanceof Plus || this.left instanceof Mult)
            && (this.right instanceof Plus || this.right instanceof Mult)) {
        if (this.left.swapSides().toString().equals(this.right.toString())) {
            return true;
        }
        if (this.left.toString().equals(this.right.swapSides().toString())) {
            return true;
        }
    }
    return false;
```

```java
import java.util.Map;

/**
 * Classname: Plus
 * A binary expression composed of an operator which sums up the expressions from both of its sides.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public class Plus extends BinaryExpression implements Expression {

    //members
    private Expression left;
    private Expression right;

    /**
     * main constructor.
     *
     * @param left  left Expression
     * @param right right Expression
     */
    public Plus(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param num right Expression - double
     */
    public Plus(Expression exp, double num) {
        this(exp, new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param var right Expression - String
     */
    public Plus(Expression exp, String var) {
        this(exp, new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param num1 left Expression - double
     * @param num2 right Expression - double
     */
    public Plus(double num1, double num2) {
        this(new Num(num1), new Num(num2));
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param exp right Expression - Expression
     */
    public Plus(double num, Expression exp) {
        this(new Num(num), exp);
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param var right Expression - String
     */
    public Plus(double num, String var) {
        this(new Num(num), new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param var1 left Expression - String
     * @param var2 right Expression - String
     */
    public Plus(String var1, String var2) {
        this(new Var(var1), new Var(var2));
    }

    /**
     * shortcut constructor.
     *
     * @param var left Expression - String
```

```java
     * @param num right Expression - double
     */
    public Plus(String var, double num) {
        this(new Var(var), new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param var left Expression - String
     * @param exp right Expression - Expression
     */
    public Plus(String var, Expression exp) {
        this(new Var(var), exp);
    }

    /**
     * getter for left Expression.
     *
     * @return left Expression
     */
    public Expression getLeft() {
        return left;
    }

    /**
     * getter for right Expression.
     *
     * @return right Expression
     */
    public Expression getRight() {
        return right;
    }


    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate(Map<String, Double> assignment) throws Exception {
        return this.left.evaluate(assignment) + this.right.evaluate(assignment);
    }

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    public String toString() {
        return "(" + this.left.toString() + " + " + this.right.toString() + ")";
    }

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param var        to replace
     * @param expression to assign instead of the var
     * @return new Expression after assigning
     */
    public Expression assign(String var, Expression expression) {
        return new Plus(this.left.assign(var, expression), this.right.assign(var, expression));
    }


    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
     *
     * @param var differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    public Expression differentiate(String var) {
        return new Plus(this.left.differentiate(var), this.right.differentiate(var));
    }


    /**
     * Returns a simplified version of the current expression.
     *
     * @return simplified version of the current expression
     */
    public Expression simplify() {
        Plus plus = new Plus(this.left.simplify(), this.right.simplify());
```

```
    Expression newLeft = plus.getLeft();
    Expression newRight = plus.getRight();
    //Expression doesn't have any Variables
    if (newLeft.getVariables().isEmpty() && newRight.getVariables().isEmpty()) {
        double value = 0;
        try {
            value = plus.evaluate();
        } catch (Exception evalFailed) {
            throw new RuntimeException("Evaluation failed!");
        }
        return new Num(value);
    } else { //Expression has a least one Var
        //simplifying X + 0 --> X
        if (newRight.toString().equals("0") || newRight.toString().equals("0.0")) {
            return newLeft.simplify();
        }
        //simplifying 0 + X --> X
        if (newLeft.toString().equals("0") || newLeft.toString().equals("0.0")) {
            return newRight.simplify();
        }
        return new Plus(newLeft.simplify(), newRight.simplify());
    }
}

/**
 * Returns a simplified version of the current expression for the bonus part.
 *
 * @return simplified version of the current expression
 */
public Expression simplifyBonus() {
    ///((x + y) + (y + x)) --> (2.0 * (x + y)), (X + X) --> (2 * X)
    if (leftEqualsRight()) {
        return new Mult(2, this.left.simplify().simplifyBonus());
    }
    if (this.left instanceof Mult && this.right instanceof Mult) {
        Mult multLeft = (Mult) this.left;
        Mult multRight = (Mult) this.right;
        // ((x * 2) + (x * 5)) --> (7 * x)
        if (multLeft.getLeft().toString().equals(multRight.getLeft().toString()) && multLeft.getRight()
                instanceof Num && multRight.getRight() instanceof Num) {
            try {
                double value = multLeft.getRight().evaluate() + multRight.getRight().evaluate();
                return new Mult(value, multLeft.getLeft()).simplify().simplifyBonus();
            } catch (Exception evalExc) {
                String exc = evalExc.toString();
            }
        }
        // ((x * 2) + (5 * x)) --> (7 * x)
        if (multLeft.getLeft().toString().equals(multRight.getRight().toString()) && multLeft.getRight()
                instanceof Num && multRight.getLeft() instanceof Num) {
            try {
                double value = multLeft.getRight().evaluate() + multRight.getLeft().evaluate();
                return new Mult(value, multLeft.getLeft()).simplify().simplifyBonus();
            } catch (Exception evalExc) {
                String exc = evalExc.toString();
            }
        }
        // ((2 * x) + (5 * x)) --> (7 * x)
        if (multLeft.getRight().toString().equals(multRight.getRight().toString()) && multLeft.getLeft()
                instanceof Num && multRight.getLeft() instanceof Num) {
            try {
                double value = multLeft.getLeft().evaluate() + multRight.getLeft().evaluate();
                return new Mult(value, multLeft.getRight()).simplify().simplifyBonus();
            } catch (Exception evalExc) {
                String exc = evalExc.toString();
            }
        }
        // ((2 * x) + (x * 5)) --> (7 * x)
        if (multLeft.getRight().toString().equals(multRight.getLeft().toString()) && multLeft.getLeft()
                instanceof Num && multRight.getRight() instanceof Num) {
            try {
                double value = multLeft.getLeft().evaluate() + multRight.getRight().evaluate();
                return new Mult(value, multLeft.getRight()).simplify().simplifyBonus();
            } catch (Exception evalExc) {
                String exc = evalExc.toString();
            }
        }
    }
    if (this.left instanceof Neg || this.right instanceof Neg) {
        ///((-x) + (-y)) --> (-(x + y))
        if (this.left instanceof Neg && this.right instanceof Neg) {
            Neg negLeft = (Neg) this.left;
            Neg negRight = (Neg) this.right;
            return new Neg(new Plus(negLeft.getExp(), negRight.getExp())).simplify().simplifyBonus();
        }
        if (this.left instanceof Neg) {
            Neg negLeft = (Neg) this.left;
            ///((-x) + x) --> 0
            if (negLeft.getExp().toString().equals(this.right.toString())) {
                return new Num(0);
```

```
            } else { ///((-x) + y) --> (y - x)
                return new Minus(this.right, negLeft.getExp());
            }
        }
        ///(x + (-x)) --> 0
        if (this.right instanceof Neg) {
            Neg negRight = (Neg) this.right;
            if (negRight.getExp().toString().equals(this.left.toString())) {
                return new Num(0);
            } else { ///(x + (-y)) --> (x - y)

                return new Minus(this.left, negRight.getExp());
            }
        }
    }
    return new Plus(this.getLeft().simplify().simplifyBonus(), this.getRight().simplify().simplifyBonus());
}

/**
 * swaps the sides of this expression- left to right, right to left.
 *
 * @return new Expression of this expression with swapped sides
 */
@Override
public Expression swapSides() {
    return new Plus(this.right, this.left);
}

/**
 * checks whether the left side of the Expression equals(mathematically!) the right side.
 *
 * @return true if equals, false otherwise.
 */
public Boolean leftEqualsRight() {
    if (this.left.toString().equals(this.right.toString())) {
        return true;
    }
    if ((this.left instanceof Plus || this.left instanceof Mult)
            && (this.right instanceof Plus || this.right instanceof Mult)) {
        if (this.left.swapSides().toString().equals(this.right.toString())) {
            return true;
        }
        if (this.left.toString().equals(this.right.swapSides().toString())) {
            return true;
        }
    }
    return false;
}
}
```

```java
import java.util.Map;

/**
 * Classname: Minus
 * A binary expression composed of an operator which subtracts the expressions from both of its sides.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public class Minus extends BinaryExpression implements Expression {

    //members
    private Expression left;
    private Expression right;

    /**
     * main constructor.
     *
     * @param left  left Expression
     * @param right right Expression
     */
    public Minus(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param num right Expression - double
     */
    public Minus(Expression exp, double num) {
        this(exp, new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param exp left Expression - Expression
     * @param var right Expression - String
     */
    public Minus(Expression exp, String var) {
        this(exp, new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param num1 left Expression - double
     * @param num2 right Expression - double
     */
    public Minus(double num1, double num2) {
        this(new Num(num1), new Num(num2));
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param exp right Expression - Expression
     */
    public Minus(double num, Expression exp) {
        this(new Num(num), exp);
    }

    /**
     * shortcut constructor.
     *
     * @param num left Expression - double
     * @param var right Expression - String
     */
    public Minus(double num, String var) {
        this(new Num(num), new Var(var));
    }

    /**
     * shortcut constructor.
     *
     * @param var1 left Expression - String
     * @param var2 right Expression - String
     */
    public Minus(String var1, String var2) {
        this(new Var(var1), new Var(var2));
    }

    /**
     * shortcut constructor.
     *
     * @param var left Expression - String
```

```
     * @param num right Expression - double
     */
    public Minus(String var, double num) {
        this(new Var(var), new Num(num));
    }

    /**
     * shortcut constructor.
     *
     * @param var left Expression - String
     * @param exp right Expression - Expression
     */
    public Minus(String var, Expression exp) {
        this(new Var(var), exp);
    }

    /**
     * getter for left Expression.
     *
     * @return left Expression
     */
    public Expression getLeft() {
        return left;
    }

    /**
     * getter for right Expression.
     *
     * @return right Expression
     */
    public Expression getRight() {
        return right;
    }


    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate(Map<String, Double> assignment) throws Exception {
        return left.evaluate(assignment) - right.evaluate(assignment);
    }

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    public String toString() {
        return "(" + this.left.toString() + " - " + this.right.toString() + ")";
    }

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param var        to replace
     * @param expression to assign instead of the var
     * @return new Expression after assigning
     */
    public Expression assign(String var, Expression expression) {
        return new Minus(this.left.assign(var, expression), this.right.assign(var, expression));
    }

    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
     *
     * @param var differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    public Expression differentiate(String var) {
        return new Minus(this.left.differentiate(var), this.right.differentiate(var));
    }

    /**
     * Returns a simplified version of the current expression.
     *
     * @return simplified version of the current expression
     */
    public Expression simplify() {
        Minus minus = new Minus(this.left.simplify(), this.right.simplify());
        Expression newLeft = minus.getLeft();
        Expression newRight = minus.getRight();
```

```
    //Expression doesn't have any Variables
    if (newLeft.getVariables().isEmpty() && newRight.getVariables().isEmpty()) {
        double value = 0;
        try {
            value = minus.evaluate();
        } catch (Exception evalFailed) {
            throw new RuntimeException("Evaluation failed!");
        }
        return new Num(value);
    } else { //Expression has a least one Var
        //simplifying X - 0 --> X
        if (newRight.toString().equals("0") || newRight.toString().equals("0.0")) {
            return newLeft.simplify();
        }
        //simplifying 0 - X --> -X
        if (newLeft.toString().equals("0") || newLeft.toString().equals("0.0")) {
            return new Neg(newRight.simplify());
        }
        //simplifying X - X --> 0
        if (newLeft.toString().equals(newRight.toString())) {
            return new Num(0);
        }
    }
    return new Minus(newLeft.simplify(), newRight.simplify());
}

/**
 * Returns a simplified version of the current expression for the bonus part.
 *
 * @return simplified version of the current expression
 */
public Expression simplifyBonus() {
    if (this.left instanceof Mult && this.right instanceof Mult) {
        Mult multLeft = (Mult) this.left;
        Mult multRight = (Mult) this.right;
        // ((x * 5) - (x * 2)) => (3 * x)
        if (multLeft.getLeft().toString().equals(multRight.getLeft().toString()) && multLeft.getRight()
                instanceof Num && multRight.getRight() instanceof Num) {
            try {
                double value = multLeft.getRight().evaluate() - multRight.getRight().evaluate();
                return new Mult(value, multLeft.getLeft()).simplify().simplifyBonus();
            } catch (Exception evalExc) {
                String exc = evalExc.toString();
            }
        }
        // ((x * 5) - (2 * x)) --> (3 * x)
        if (multLeft.getLeft().toString().equals(multRight.getRight().toString()) && multLeft.getRight()
                instanceof Num && multRight.getLeft() instanceof Num) {
            try {
                double value = multLeft.getRight().evaluate() - multRight.getLeft().evaluate();
                return new Mult(value, multLeft.getLeft()).simplify().simplifyBonus();
            } catch (Exception evalExc) {
                String exc = evalExc.toString();
            }
        }
        // ((5 * x) - (2 * x)) --> (3 * x)
        if (multLeft.getRight().toString().equals(multRight.getRight().toString()) && multLeft.getLeft()
                instanceof Num && multRight.getLeft() instanceof Num) {
            try {
                double value = multLeft.getLeft().evaluate() - multRight.getLeft().evaluate();
                return new Mult(value, multLeft.getRight()).simplify().simplifyBonus();
            } catch (Exception evalExc) {
                String exc = evalExc.toString();
            }
        }
        // ((5 * x) - (x * 2)) --> (3 * x)
        if (multLeft.getRight().toString().equals(multRight.getLeft().toString()) && multLeft.getLeft()
                instanceof Num && multRight.getRight() instanceof Num) {
            try {
                double value = multLeft.getLeft().evaluate() - multRight.getRight().evaluate();
                return new Mult(value, multLeft.getRight()).simplify().simplifyBonus();
            } catch (Exception evalExc) {
                String exc = evalExc.toString();
            }
        }
    }
    if (this.left instanceof Neg || this.right instanceof Neg) {
        ///((-x) - (-y)) --> (y - x)
        if (this.left instanceof Neg && this.right instanceof Neg) {
            Neg negLeft = (Neg) this.left;
            Neg negRight = (Neg) this.right;
            return new Minus(negRight.getExp(), negLeft.getExp()).simplify().simplifyBonus();
        }

        if (this.left instanceof Neg) {
            Neg negLeft = (Neg) this.left;
            ///((-x) - x) --> (-2) * x
            if (negLeft.getExp().toString().equals(this.right.toString())) {
                return new Mult(new Neg(2), negLeft.getExp()).simplify().simplifyBonus();
            } else { ///((-x) - y) --> -(x + y)
```

```
                    return new Neg(new Plus(negLeft.getExp(), this.right)).simplify();
                }
            }
            if (this.right instanceof Neg) {
                Neg negRight = (Neg) this.right;
                //(x - (-x)) --> 2 * x
                if (negRight.getExp().toString().equals(this.left.toString())) {
                    return new Mult(2, this.left).simplify().simplifyBonus();
                } else { //(x - (-y)) --> (x + y)
                    return new Plus(this.left, negRight.getExp()).simplify().simplifyBonus();
                }
            }
        }
        return new Minus(this.getLeft().simplify().simplifyBonus(), this.getRight().simplify().simplifyBonus());
    }
}
```

```java
import jdk.nashorn.internal.runtime.ECMAException;

import java.util.Map;
import java.util.TreeMap;

public class TestPart1 {
    public static void main(String[] args) {


        Expression e66 = new Plus(new Plus("x","x"),"x");
        System.out.println(e66);
        System.out.println(e66.toString());
        System.out.println(e66.simplify());

        Expression e5 = new Neg(new Plus(new Var("x"), new Num(2)));
        System.out.println(e5);

        Expression e6 = new Cos(450);
        try {
            System.out.println(e6.evaluate());
        }catch(Exception rr){
            System.out.println(rr.toString());
        }
        //Expression e = new Sin(new Pow(new Mul(new Plus(new Mul(new Num(2), new Var("x")),
        //        new Var("y")),new Num(4)),new Var("x")));

        Expression eee=new Plus(5,new Plus(new Plus(4,new Num(0)),new Plus(new Plus(0.0,4),7)));
        System.out.println(eee);
        System.out.println(eee.toString());
        System.out.println(eee.simplify());
        /*
        Expression e2 = new Pow(new Plus(new Var("x"), new Var("y")), new Num(2));
        String s = e2.toString();
        System.out.println(s);
        */

        //Expression e2 = new Plus(new Plus(new Var("x"), new Var("y")), new Num(2));

        //Expression e2=new Plus("x", 5);
        Expression e2=new Cos("x");

        Map<String, Double> assignment = new TreeMap<String, Double>();
        assignment.put("x", 2.0);
        assignment.put("y", 4.0);
        double value=0;
        try {
            value = e2.evaluate(assignment);
        }catch (Exception e)
        {
            System.out.println("exception");
        }



        System.out.println("The result is: " + value);

        System.out.println(e2);


        Expression e3=new Sin(90);
        try {
            System.out.println("The result is: " + e3.evaluate());
        }catch(Exception e144){
            System.out.println("exception");
        }
        System.out.println(e3);
    }
}
```

```java
import java.util.List;
import java.util.Map;

/**
 * interface defining an expression.
 * an Expression can be evaluated, printed, differentiated, simplified.
 * you can also get the variables in the expression and assign a value to them.
 */
public interface Expression {
    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    double evaluate(Map<String, Double> assignment) throws Exception;

    /**
     * A convenience method. Like the `evaluate(assignment)` method above,
     * but uses an empty assignment.
     *
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    double evaluate() throws Exception;

    /**
     * Returns a list of the variables in the expression.
     *
     * @return list of Strings
     */
    List<String> getVariables();

    /**
     * Returns a nice string representation of the expression.
     *
     * @return String representation
     */
    String toString();

    /**
     * Returns a new expression in which all occurrences of the variable
     * var are replaced with the provided expression (Does not modify the current expression).
     *
     * @param var        to replace
     * @param expression to assign instead of the var
     * @return new Expression after assigning
     */
    Expression assign(String var, Expression expression);

    /**
     * Returns the expression tree resulting from differentiating
     * the current expression relative to variable `var`.
     *
     * @param var differentiating according to this variable
     * @return new Expression containing the differentiation of current expression.
     */
    Expression differentiate(String var);

    /**
     * Returns a simplified version of the current expression.
     *
     * @return simplified version of the current expression
     */
    Expression simplify();

    /**
     * swaps the sides of this expression- left to right, right to left.
     *
     * @return new Expression of this expression with swapped sides
     */
    Expression swapSides();

    /**
     * activates the additional simplification if exists for the bonus part.
     *
     * @return new bonus-simplified Expression.
     */
    Expression simplifyBonus();
}
```

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;


/**
 * Classname: BaseExpression
 * An abstract class made to share common code between both BinaryExpression and UnaryExpression.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public abstract class BaseExpression {

    /**
     * A convenience method. Like the `evaluate(assignment)` method above,
     * but uses an empty assignment.
     *
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public double evaluate() throws Exception {
        Map<String, Double> assignment = new TreeMap<String, Double>();
        return this.evaluate(assignment);
    }

    /**
     * Removes duplicated values in the List of String received.
     *
     * @param listWithDup list with duplicates(possibly).
     * @return new List without duplicates.
     */
    public List<String> removeDuplicates(List<String> listWithDup) {
        if (listWithDup == null || listWithDup.isEmpty()) {
            return listWithDup;
        }
        List<String> listWithoutDup = new ArrayList<>();
        while (!listWithDup.isEmpty()) {
            if (!listWithoutDup.contains(listWithDup.get(0))) {
                listWithoutDup.add(listWithDup.remove(0));
            } else {
                listWithDup.remove(0);
            }
        }
        return listWithoutDup;
    }

    /**
     * Evaluate the expression using the variable values provided
     * in the assignment, and return the result.  If the expression
     * contains a variable which is not in the assignment, an exception
     * is thrown.
     *
     * @param assignment a Map contains Vars as keys and their values to assign
     * @return result after assignment
     * @throws Exception If the expression contains a variable which is not in the assignment
     */
    public abstract double evaluate(Map<String, Double> assignment) throws Exception;

    /**
     * swaps the sides of this expression- left to right, right to left.
     *
     * @return new Expression of this expression with swapped sides
     */
    public Expression swapSides() {
        return null;
    }


}
```

```java
import java.util.Map;
import java.util.TreeMap;

/**
 * Classname: ExpressionsTest
 * Test class for ass4.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public class ExpressionsTest {
    /**
     * main method- runs the test.
     *
     * @param args unused.
     */
    public static void main(String[] args) {
        try {
            //Create the expression (2x) + (sin(4y)) + (e^x).
            Expression exp = new Plus(new Plus(new Mult(2, "x"), new Sin(new Mult(4, "y"))), new Pow("e", "x"));
            //Print the expression.
            System.out.println(exp);
            //Print the value of the expression with (x=2,y=0.25,e=2.71).
            Map<String, Double> assignment = new TreeMap<String, Double>();
            assignment.put("x", 2.0);
            assignment.put("y", 0.25);
            assignment.put("e", 2.71);
            System.out.println(exp.evaluate(assignment));
            //Print the differentiated expression according to x.
            System.out.println(exp.differentiate("x"));
            //Print the value of the differentiated expression according to x with the assignment above.
            System.out.println(exp.differentiate("x").evaluate(assignment));
            //Print the simplified differentiated expression.
            System.out.println(exp.differentiate("x").simplify());
        } catch (Exception exception) {
            System.out.println(exception.toString());
        }
    }
}
```

```java
import java.util.ArrayList;
import java.util.List;

/**
 * Classname: UnaryExpression
 * An abstract class made to share common code between all unary expressions(for operator on one operands).
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public abstract class UnaryExpression extends BaseExpression {

    /**
     * Returns a list of the variables in the expression.
     *
     * @return list of Strings
     */
    public List<String> getVariables() {
        List<String> vars = new ArrayList<String>();
        if (getExp().getVariables() != null) {
            //joins all lists returning from the recursion(which adds all vars)
            vars.addAll(getExp().getVariables());
        }
        //remove vars that appear more the once in the expression
        vars = removeDuplicates(vars);
        return vars;
    }


    /**
     * getter for Expression.
     *
     * @return Expression
     */
    public abstract Expression getExp();
}
```

```java
import java.util.ArrayList;
import java.util.List;

/**
 * Classname: BinaryExpression
 * An abstract class made to share common code between all binary expressions(for operator with two operands).
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public abstract class BinaryExpression extends BaseExpression {


    /**
     * Returns a list of the variables in the expression.
     *
     * @return list of Strings
     */
    public List<String> getVariables() {
        List<String> vars = new ArrayList<String>();
        if (getLeft().getVariables() != null) {
            //joins all lists returning from the left side of the recursion(which adds all left vars)
            vars.addAll(getLeft().getVariables());
        }
        if (getRight().getVariables() != null) {
            //joins all lists returning from the right side of the recursion(which adds all left vars)
            vars.addAll(getRight().getVariables());
        }
        //remove vars that appear more the once in the expression
        vars = removeDuplicates(vars);
        return vars;
    }

    /**
     * getter for left Expression.
     *
     * @return left Expression
     */
    public abstract Expression getLeft();

    /**
     * getter for right Expression.
     *
     * @return right Expression
     */
    public abstract Expression getRight();


}
```

```java
/**
 * Classname: SimplificationDemo
 * Test class for bonus part of ass4.
 *
 * @author Elad Israel
 * @version 1.0 01/05/2018
 */
public class SimplificationDemo {
    /**
     * main method- runs the test.
     *
     * @param args unused.
     */
    public static void main(String[] args) {
        SimplificationDemo demoSimp = new SimplificationDemo();
        demoSimp.minusSimplifications();
        demoSimp.multSimplifications();
        demoSimp.negSimplifications();
        demoSimp.plusSimplifications();
    }

    /**
     * demonstrates simplification that can be done to Minus.
     */
    public void minusSimplifications() {
        // ((x * 5) - (x * 2)) => (3 * x)
        Expression minusSimplify1 = new Minus(new Mult("x", 5), new Mult("x", 2));
        System.out.println("Non-Simplified:");
        System.out.println(minusSimplify1);
        System.out.println("Bonus Simplified:");
        System.out.println(minusSimplify1.simplifyBonus());
        System.out.println("--------------");

        // ((x * 5) - (2 * x)) --> (3 * x)
        Expression minusSimplify2 = new Minus(new Mult("x", 5), new Mult(2, "x"));
        System.out.println("Non-Simplified:");
        System.out.println(minusSimplify2);
        System.out.println("Bonus Simplified:");
        System.out.println(minusSimplify2.simplifyBonus());
        System.out.println("--------------");

        // ((5 * x) - (2 * x)) --> (3 * x)
        Expression minusSimplify3 = new Minus(new Mult(5, "x"), new Mult(2, "x"));
        System.out.println("Non-Simplified:");
        System.out.println(minusSimplify3);
        System.out.println("Bonus Simplified:");
        System.out.println(minusSimplify3.simplifyBonus());
        System.out.println("--------------");

        // ((5 * x) - (x * 2)) --> (3 * x)
        Expression minusSimplify4 = new Minus(new Mult(5, "x"), new Mult("x", 2));
        System.out.println("Non-Simplified:");
        System.out.println(minusSimplify4);
        System.out.println("Bonus Simplified:");
        System.out.println(minusSimplify4.simplifyBonus());
        System.out.println("--------------");

        //((-x) - (-y)) --> (y - x)
        Expression minusSimplify5 = new Minus(new Neg("x"), new Neg("y"));
        System.out.println("Non-Simplified:");
        System.out.println(minusSimplify5);
        System.out.println("Bonus Simplified:");
        System.out.println(minusSimplify5.simplifyBonus());
        System.out.println("--------------");

        //((-x) - x) --> ((-2) * x)
        Expression minusSimplify6 = new Minus(new Neg("x"), "x");
        System.out.println("Non-Simplified:");
        System.out.println(minusSimplify6);
        System.out.println("Bonus Simplified:");
        System.out.println(minusSimplify6.simplifyBonus());
        System.out.println("--------------");

        //(x - (-x)) --> (2 * x)
        Expression minusSimplify7 = new Minus("x", new Neg("x"));
        System.out.println("Non-Simplified:");
        System.out.println(minusSimplify7);
        System.out.println("Bonus Simplified:");
        System.out.println(minusSimplify7.simplifyBonus());
        System.out.println("--------------");

        //((-x) - y) --> (-(x + y))
        Expression minusSimplify8 = new Minus(new Neg("x"), "y");
        System.out.println("Non-Simplified:");
        System.out.println(minusSimplify8);
        System.out.println("Bonus Simplified:");
        System.out.println(minusSimplify8.simplifyBonus());
        System.out.println("--------------");

        //(x - (-y)) --> (x + y)
```

```
        Expression minusSimplify9 = new Minus("x", new Neg("y"));
        System.out.println("Non-Simplified:");
        System.out.println(minusSimplify9);
        System.out.println("Bonus Simplified:");
        System.out.println(minusSimplify9.simplifyBonus());
        System.out.println("--------------");
    }

    /**
     * demonstrates simplification that can be done to Mult.
     */
    public void multSimplifications() {
        //(X * X) --> (2 ^ X)
        Expression multSimplify1 = new Mult("x", "x");
        System.out.println("Non-Simplified:");
        System.out.println(multSimplify1);
        System.out.println("Bonus Simplified:");
        System.out.println(multSimplify1.simplifyBonus());
        System.out.println("--------------");

        // ((x + y) * (y + x)) --> (x + y)^2
        Expression multSimplify2 = new Mult(new Plus("x", "y"), new Plus("y", "x"));
        System.out.println("Non-Simplified:");
        System.out.println(multSimplify2);
        System.out.println("Bonus Simplified:");
        System.out.println(multSimplify2.simplifyBonus());
        System.out.println("--------------");

        //(-x) * (-y) --> (x * y)
        Expression multSimplify3 = new Mult(new Neg("x"), new Neg("y"));
        System.out.println("Non-Simplified:");
        System.out.println(multSimplify3);
        System.out.println("Bonus Simplified:");
        System.out.println(multSimplify3.simplifyBonus());
        System.out.println("--------------");

        // x * (-y) --> -(x * y)
        Expression multSimplify4 = new Mult("x", new Neg("y"));
        System.out.println("Non-Simplified:");
        System.out.println(multSimplify4);
        System.out.println("Bonus Simplified:");
        System.out.println(multSimplify4.simplifyBonus());
        System.out.println("--------------");

        //(-x) * y --> -(x * y)
        Expression multSimplify5 = new Mult(new Neg("x"), "y");
        System.out.println("Non-Simplified:");
        System.out.println(multSimplify5);
        System.out.println("Bonus Simplified:");
        System.out.println(multSimplify5.simplifyBonus());
        System.out.println("--------------");

    }

    /**
     * demonstrates simplification that can be done to Neg.
     */
    public void negSimplifications() {
        //(-(-x)) --> x
        Expression negSimplify1 = new Neg(new Neg("x"));
        System.out.println("Non-Simplified:");
        System.out.println(negSimplify1);
        System.out.println("Bonus Simplified:");
        System.out.println(negSimplify1.simplifyBonus());
        System.out.println("--------------");

        // (-0.0) --> 0.0
        Expression negSimplify2 = new Neg(new Num(0));
        System.out.println("Non-Simplified:");
        System.out.println(negSimplify2);
        System.out.println("Bonus Simplified:");
        System.out.println(negSimplify2.simplifyBonus());
        System.out.println("--------------");
    }

    /**
     * demonstrates simplification that can be done to Plus.
     */
    public void plusSimplifications() {
        // (x + x) --> (2.0 * x))
        Expression plusSimplify1 = new Plus("x", "x");
        System.out.println("Non-Simplified:");
        System.out.println(plusSimplify1);
        System.out.println("Bonus Simplified:");
        System.out.println(plusSimplify1.simplifyBonus());
        System.out.println("--------------");

        // ((x + y) + (y + x)) --> (2.0 * (x + y))
        Expression plusSimplify2 = new Plus(new Plus("x", "y"), new Plus("y", "x"));
        System.out.println("Non-Simplified:");
```

```
        System.out.println(plusSimplify2);
        System.out.println("Bonus Simplified:");
        System.out.println(plusSimplify2.simplifyBonus());
        System.out.println("--------------");

        // ((x * 2) + (x * 5)) => (7 * x)
        Expression plusSimplify3 = new Plus(new Mult("x", 2), new Mult("x", 5));
        System.out.println("Non-Simplified:");
        System.out.println(plusSimplify3);
        System.out.println("Bonus Simplified:");
        System.out.println(plusSimplify3.simplifyBonus());
        System.out.println("--------------");

        // ((x * 2) + (5 * x)) --> (7 * x)
        Expression plusSimplify4 = new Plus(new Mult("x", 2), new Mult(5, "x"));
        System.out.println("Non-Simplified:");
        System.out.println(plusSimplify4);
        System.out.println("Bonus Simplified:");
        System.out.println(plusSimplify4.simplifyBonus());
        System.out.println("--------------");

        // ((2 * x) + (5 * x)) --> (7 * x)
        Expression plusSimplify5 = new Plus(new Mult(2, "x"), new Mult(5, "x"));
        System.out.println("Non-Simplified:");
        System.out.println(plusSimplify5);
        System.out.println("Bonus Simplified:");
        System.out.println(plusSimplify5.simplifyBonus());
        System.out.println("--------------");

        // ((2 * x) + (x * 5)) --> (7 * x)
        Expression plusSimplify6 = new Plus(new Mult(2, "x"), new Mult("x", 5));
        System.out.println("Non-Simplified:");
        System.out.println(plusSimplify6);
        System.out.println("Bonus Simplified:");
        System.out.println(plusSimplify6.simplifyBonus());
        System.out.println("--------------");

        //((-x) + (-y)) --> (-(x + y))
        Expression plusSimplify7 = new Plus(new Neg("x"), new Neg("y"));
        System.out.println("Non-Simplified:");
        System.out.println(plusSimplify7);
        System.out.println("Bonus Simplified:");
        System.out.println(plusSimplify7.simplifyBonus());
        System.out.println("--------------");

        //(-x) + x --> 0
        Expression plusSimplify8 = new Plus(new Neg("x"), "x");
        System.out.println("Non-Simplified:");
        System.out.println(plusSimplify8);
        System.out.println("Bonus Simplified:");
        System.out.println(plusSimplify8.simplifyBonus());
        System.out.println("--------------");

        //x + (-x) --> 0
        Expression plusSimplify9 = new Plus("x", new Neg("x"));
        System.out.println("Non-Simplified:");
        System.out.println(plusSimplify9);
        System.out.println("Bonus Simplified:");
        System.out.println(plusSimplify9.simplifyBonus());
        System.out.println("--------------");

        //((-x) + y) --> (y - x)
        Expression plusSimplify10 = new Plus(new Neg("x"), "y");
        System.out.println("Non-Simplified:");
        System.out.println(plusSimplify10);
        System.out.println("Bonus Simplified:");
        System.out.println(plusSimplify10.simplifyBonus());
        System.out.println("--------------");

        //(x + (-y)) --> (x - y)
        Expression plusSimplify11 = new Plus("x", new Neg("y"));
        System.out.println("Non-Simplified:");
        System.out.println(plusSimplify11);
        System.out.println("Bonus Simplified:");
        System.out.println(plusSimplify11.simplifyBonus());
        System.out.println("--------------");
    }
}
```