```
package hw04;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.EmptyStackException;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.Stack;
import java.util.StringTokenizer;
import java.util.logging.Level;
import java.util.logging.Logger;



/**
 * @author said
 * @version 1.0
 * @created 31-Mar-2014 10:09:39 PM
 */
public class GITCompiler implements Compilable {

    private BufferedReader reader;
    private FileReader sourceFile;
    private List<Variable> varList;
    private List<Object> infixCode;
    private List<Object> postfixCode;
    private int curLineNum;

    /** The operators */
    private static final String OPERATORS = "+-*/()=";

    /** The precedence of the operators, matches order of OPERATORS. */
    private static final int[] PRECEDENCE = {2, 2, 3, 3, 1, 1, 0};

    /** The operator stack */
    private Stack < Character > operatorStack;
    private Stack < Operand > operandStack;

    public GITCompiler(String filePath){
        try {
            this.sourceFile = new FileReader(filePath);
            this.reader = new BufferedReader( sourceFile );
            varList = new LinkedList<>();
            infixCode = new LinkedList<>();
            curLineNum = 0;

        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
            System.err.println("Invalid path to source code");
            System.exit(1);
        }

    }

    @Override
    public void compileAndRun() {
        String line;
        double result;
```

```
        try {
            while ((line = reader.readLine()) != null) {
                curLineNum++;

                /* Burada infix hali var */
                infixCode = checkLine(line);

                // Burada initialization yapılacak
                initializeValues(infixCode);

                //Burada postfixe çevrilip evoluate
                toPostfix(infixCode);

                //Burada hesaplama yapılıyor
                result = eval(postfixCode);
            }
        } catch (IOException ex) {
            System.err.println("Input Error");
            ex.printStackTrace();
            System.exit(1);
        } catch (UnknownOperatorException | UndefinedVariableException ex) {
            ex.printStackTrace();
            System.exit(-1);
        } catch (Exception ex) {
            Logger.getLogger(GITCompiler.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }

    private double eval(List<Object> postfixCode) throws Exception {
        operandStack = new Stack<>();
        ListIterator iter = postfixCode.listIterator();

        while(iter.hasNext()){
            Object obj = iter.next();

            if(obj instanceof String){
                String op = (String)obj;
                if(IsOperator(op)){
                    double res = evalOp(op.charAt(0));
                    operandStack.push(new MyDouble(res));
                }

            }else if(obj instanceof Operand){
                Operand op = (Operand)obj;
                operandStack.push(op);
            }

        }
        MyDouble answer;

        Object obj  = operandStack.pop();
        if((obj instanceof Var) || (obj instanceof Print) || (obj instanceof
Input)){
            Operand curOp = (Operand)obj;
            return curOp.returnValue();
        }else if(obj instanceof MyDouble){
            answer = (MyDouble)obj;
            return answer.returnValue();
        }else{
            throw new Exception("Syntax Error: Unknown token");
        }
```

```
    }

    private double evalOp(char op) {
        Operand rhs = operandStack.pop();
        Operand lhs = operandStack.pop();
        double result = 0.0;

        switch(op){
            case '+':
                result = lhs.returnValue() + rhs.returnValue();
                break;
            case '-':
                result = lhs.returnValue() - rhs.returnValue();
                break;
            case '/':
                result = lhs.returnValue() / rhs.returnValue();
                break;
            case '*':
                result = lhs.returnValue() * rhs.returnValue();
                break;
            case '=':
                Variable v = (Variable)lhs;
                v.setValue(rhs.returnValue());
                result = v.returnValue();
        }

        return result;
    }

    private void toPostfix(List<Object> infixCode) throws
UnknownOperatorException {
        try {
            postfixCode = convert(infixCode);
        } catch (Exception ex) {
            ex.printStackTrace();
        }

    }

    private List<Object> convert(List<Object> infixCode) throws
UnknownOperatorException, Exception {
        this.operatorStack = new Stack<>();
        LinkedList<Object> postfix = new LinkedList<>();
        ListIterator<Object> iter = infixCode.listIterator();

        try {
            // infix kodunun tüm tokenlarını gez
            while(iter.hasNext()){
                Object nextToken = iter.next();

                // Eğer operatorse
                if(nextToken instanceof String){
                    String Op = (String)nextToken;
                    if(IsOperator(Op))
                        processOperator(Op.charAt(0), postfix);
                    else
                        throw new UnknownOperatorException("Unknown Operator at"
+ getCurLineNum() +". line");
                }else{  // Eğer Operand ise
                    postfix.add(nextToken);
                }
            }
```

```
            // Pop any remaining operators
            // and append them to postfix.
            while (!operatorStack.empty()) {
                char op = operatorStack.pop();
                // Any '(' on the stack is not matched.
                if (op == '(')
                    throw new Exception(
                        "Unmatched parenthesis at " + getCurLineNum() + ".
line");
                // Burada stackte son kalan operatorleri ekliyoruz.
                postfix.add(String.valueOf(op));

            }
        }
        catch(EmptyStackException ex){
            throw new Exception("Missing operand at "+ getCurLineNum() +".
line");
        }

        return postfix;
    }

    private void processOperator(char op, LinkedList<Object> postfix){
        if(operatorStack.empty() || op == '('){
            operatorStack.push(op);
        }else {
            // Peek the operator stack and
            // let topOp be the top operator.
            char topOp = operatorStack.peek();
            if (precedence(op) > precedence(topOp)) {
                operatorStack.push(op);
            }
            else {
                // Pop all stacked operators with equal
                // or higher precedence than op.
                while (!operatorStack.empty()
                        && precedence(op) <= precedence(topOp)) {
                    operatorStack.pop();

                    if (topOp == '(') {
                        // Matching '(' popped - exit loop.
                        break;
                    }
                    // Stackin üstteki elemanı postfixe ekliyoruz
                    postfix.add(String.valueOf(topOp));

                    if (!operatorStack.empty()) {
                    // Reset topOp.
                        topOp = operatorStack.peek();
                    }
                }

                // assert: Operator stack is empty or
                //         current operator precedence >
                //         top of stack operator precedence.
                if (op != ')')
                  operatorStack.push(op);
            }
        }
    }
```

```
    /** Determine the precedence of an operator.
    @param op The operator
    @return the precedence
    */
    private int precedence(char op) {
        return PRECEDENCE[OPERATORS.indexOf(op)];
    }

    private void initializeValues(List<Object> Code) throws
UnknownOperatorException, UndefinedVariableException {
        ListIterator<Object> iter = Code.listIterator();
        Object currentObj;
        Operand Curoper;

        while(iter.hasNext()){
            currentObj = iter.next();


            if(currentObj instanceof Executable){
                try{
                    Operand nextOp = (Operand) iter.next();
                    if(nextOp instanceof Variable){
                        if(!(checkVariable(((Variable)nextOp).getVarName()))){
                            throw new UndefinedVariableException("There is no
such a variable " + ((Variable)nextOp).getVarName());
                        }
                    }
                    Curoper = (Operand)currentObj;
                    Curoper.setValue(nextOp);
                    iter.remove();
                }catch (ClassCastException ex){
                    throw new UnknownOperatorException("Unknown Operator in "+
getCurLineNum() + ". line");
                }
            }
            else if(currentObj instanceof Var){
                Operand nextOp2 = (Operand) iter.next();
                Curoper = (Operand)currentObj;
                varList.add((Variable) nextOp2);
                Curoper.setValue((Variable) nextOp2);
                iter.remove();
            }

        }
    }

    private LinkedList<Object> checkLine(String line) throws
UnknownOperatorException, UndefinedVariableException{

        StringTokenizer tokenizer = new StringTokenizer(line);
        LinkedList<Object> gitFormat = new LinkedList<>();

        while(tokenizer.hasMoreTokens()){
            String nextExp = tokenizer.nextToken();


            if(IsOperator(nextExp)){
                gitFormat.add(nextExp);
            }else if(IsDouble(nextExp)){
                gitFormat.add(new MyDouble(Double.valueOf(nextExp)));
            }else if (IsInput(nextExp)) {
                gitFormat.add(new Input(null));
```

```
            }else if (IsPrint(nextExp)) {
                gitFormat.add(new Print(null));
            }else if (IsVar(nextExp)) {
                gitFormat.add(new Var(null));
            }else if(IsFunction(nextExp)){
                gitFormat.add(FuncToDouble(nextExp));
            }else if(Is_a_Variable(nextExp)){
                if(checkVariable(nextExp)){
                    Variable var = findVar(nextExp);
                    gitFormat.add(var);
                }else if(!checkVariable(nextExp))
                    gitFormat.add(new Variable(nextExp, 0));
                else
                    throw new UndefinedVariableException("There is no such a
variable defined before");
            }else
                throw new UnknownOperatorException("Unknown Operator Detected at
" + getCurLineNum() + ". line");
        }
            return gitFormat;
    }
    public Variable findVar(String varName){
        ListIterator<Variable> iter = varList.listIterator();

        while(iter.hasNext()){
            Variable curVar = iter.next();
            if(curVar.getVarName().equals(varName))
                return curVar;
        }
        return null;
    }
    /**
     * @return the curLineNum
     */
    public int getCurLineNum() {
        return curLineNum;
    }

    private Operand FuncToDouble(String nextExp) {
        switch (nextExp) {
            case "sin":
                return new Sin(null);
            case "cos":
                return new Cos(null);
            case "sqrt":
                return new Sqrt(null);
            case "log" :
                return new Log(null);
            case "abs" :
                return new Abs(null);
            case "tan" :
                return new Tan(null);
            case "exp" :
                return new Exp(null);
        }
        return null;
    }

    private boolean checkVariable(String nextExp) {
        ListIterator<Variable> iter = varList.listIterator();

        while(iter.hasNext()){
```

```
            if(iter.next().getVarName().equals(nextExp))
                return true;
        }
        return false;
    }



    public static class UnknownOperatorException extends Exception {

        /** Construct a SyntaxErrorException with the specified
            message.
            @param message The message
         */
        UnknownOperatorException(String message) {
          super(message);
        }
    }

    public static class UndefinedVariableException extends Exception {

        /** Construct a SyntaxErrorException with the specified
            message.
            @param message The message
         */
        UndefinedVariableException(String message) {
          super(message);
        }
    }

    public boolean IsVar(String expr){
        return expr.equals("var");
    }

    public boolean IsPrint(String expr){
        return expr.equals("print");
    }

    public boolean IsInput(String expr){
        return expr.equals("input");
    }

    public boolean IsDouble(String expr){

        try {
            Double.parseDouble(expr);
        } catch (NumberFormatException e) {
            return false;
        }

        return true;
    }

    public boolean Is_a_Variable(String expr) throws UnknownOperatorException{

        if( (!IsFunction(expr) && !(expr.charAt(0) > 47 && expr.charAt(0) <
58 )) )
            return true;

        throw new UnknownOperatorException(expr);
    }
```

```
    public boolean IsOperator(String expr){
        char ch = expr.charAt(0);

        return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '=');
    }

    public boolean IsFunction(String expr){
        return (expr.equals("sin") || expr.equals("cos") || expr.equals("sqrt")
                || expr.equals("log") || expr.equals("abs") ||
expr.equals("cos")
                || expr.equals("tan") || expr.equals("exp"));
    }

}
```