

```
// src/com/github/sahasatvik/math/ExpressionParser.java

package com.github.sahasatvik.math;

/**
 * ExpressionParser provides methods for evaluating mathematical expressions, specifically
 * tailored for parsing with this library. ExpressionParser supports basic arithmetic operators,
 * parenthesized expressions, variable substitution as well as basic functions.
 *
 * @author Satvik Saha
 * @version 1.0, 16/10/2016
 * @see com.github.sahasatvik.math.MathParser
 * @since 1.0
 */

public class ExpressionParser extends MathParser {

    /**
     * Regex which matches a number. It may be signed or use scientific notation.
     */
    protected static final String numberRegex = "([+-]?\\d+(\\.\\d+)?([eE](-?)\\d+)?)";

    /**
     * Regex which matches a signed number. It may use scientific notation.
     */
    protected static final String signedNumberRegex = "([+-]\\d+(\\.\\d+)?([eE](-?)\\d+)?)";

    /**
     * Regex which matches an assignment statement. It is simply a word, followed by an
     * equals sign (=) and an expression.
     */
    protected static final String assignmentRegex = "(\\s+)?(\\w+)(\\s+)(=)(.*)";

    /**
     * Array of supported operators. The operators are arranged in their order of precedence.
     * Thus, operators to the left will be evaluated before those to the right.
     */
    protected static final String[] operators = {"^", "%", "/", "*", "+", "-"};

    /**
     * Array of variables maintained by an ExpressionParser object. The first String in each
     * line stores the variable name, while the second stores the value.
     */
}
```

```

public String[][] variables;

/**
 * Index of the last variable in the 'variables' array. Elements in the 'variables' array
 * after this index are all blank, so they are not parsed during expression evaluation.
 */
public int numberOfVars;

/**
 * Constructor of ExpressionParser. This constructor initializes the variable cache with
 * the specified maximum size.
 *
 * @param maxVars the maximum number of variables to be stored
 * @since 1.0
 */
public ExpressionParser (int maxVars) {
    variables = new String[maxVars][2];
    numberOfVars = 0;
}

/**
 * Adds a variable to the variable cache. This method accepts the variable
 * name, as well as the String each occurrence is to be substituted with.
 *
 * @param name the name of the variable
 * @param value the value the variable holds
 * @since 1.0
 */
public void addVariable (String name, String value) {
    /* Loop through the stored variables */
    for (int i = 0; i < numberOfVars; i++) {
        /* If the variable already exists, simply update the value */
        if (variables[i][0].equals(name)) {
            variables[i][1] = value;
            return;
        }
    }
    /*
     * Create a new variable by storing the name and value in the variables array,
     * then update numberOfVars
     */
    variables[numberOfVars][0] = name;
    variables[numberOfVars][1] = value;
}

```

```

        numberOfVars++;
    }

    /**
     * Evaluates a String representation of a mathematical expression into a
     * number (stored as a String).
     *
     * @param exp the expression to be evaluated
     * @return the result after evaluating the expression
     * @throws com.github.sahasatvik.math.NullExpressionException
     *         thrown when the expression is empty
     * @throws com.github.sahasatvik.math.ExpressionParserException
     *         thrown when the expression cannot be parsed
     *
     * @see #addVariable(String, String)
     * @see #parseVariables(String)
     * @see #parseParenthesis(String)
     * @see #parseFunctions(String)
     * @see #parseOperators(String)
     * @since 1.0
     */

    public String evaluate (String exp) throws ExpressionParserException {
        String result = exp;
        if (exp.trim().length() == 0) {
            /* Throw an Exception if the expression is blank */
            throw new NullExpressionException();
        } else if (isNumber(exp)) {
            /* If the expression is already a number, there is nothing to evaluate */
            return "" + Double.parseDouble(exp);
        } else if (exp.matches(assignmentRegex)) {
            /*
             * If the expression is an assignment statement, interpret everything before
             * the equals sign (=) as the variable name. The rest is simply another
             * expression, which is also the value of the variable.
             */
            String varName = exp.substring(0, exp.indexOf("=")).trim();
            String varValue = evaluate(exp.substring(exp.indexOf("=") + 1));

            /*
             * Add the variable in the cache, then use the value of the variable
             * as the evaluated result
             */
            addVariable(varName, varValue);
            exp = varValue;
        } else {
            /*

```

```

        * Replace all variables with their values,
        * solve everything within parenthesis,
        * then Solve all functions
        */
        exp = parseVariables(exp);
        exp = parseParenthesis(exp);
        exp = parseFunctions(exp);

        /*
        * The expression is now simply a collection of numbers and arithmetic operators.
        * Finish off the process by solving each operation, following the BODMAS rule.
        */
        exp = parseOperators(exp);
    }

    try {
        /* Check if the result is a valid number */
        result = "" + Double.parseDouble(exp);
    } catch (Exception e) {
        /* Throw an Exception if the result is not a number */
        throw new ExpressionParserException(exp);
    }
    return result;
}

/**
 * Substitutes all instances of the variables in the cache with their values.
 * A variable name present in the expression must be enclosed within angled brackets
 * (<code>#{code}</code>, <code>#{code}</code>) in order to be recognized.
 * For example, if <code>x = 10.0</code>, then all instances of <code>#{code}</code>
 * will be replaced with <code>10.0</code>
 *
 * @param exp the expression to be parsed
 * @return the expression after substituting known values
 *         of variables stored in the cache
 * @throws com.github.sahasatvik.math.VariableNotFoundException
 *         thrown when an unrecognized variable name is
 *         found in the expression
 *
 * @since 1.0
 */

```

```

protected String parseVariables (String exp) throws VariableNotFoundException {
    /*
    * Loop through the variable cache, checking for occurrences of the variables
    * (enclosed within angled brackets)(<var_name>)
    */

```

```

        for (int i = 0; i < numberOfVars; i++) {
            /* Replace all instances of the variable with its value directly */
            exp = exp.replaceAll("<(\s+)?" + variables[i][0] + "(\s+)?>"
                               , variables[i][1]);
        }

        /*
         * Check if any unrecognized variables are present. This can be done very simply as
         * the presence of angled brackets (<>) indicates an unreplaced variable.
         */
        int start = exp.indexOf("<");
        int end = exp.indexOf(">");
        if (start != -1 && end != -1 && start < end) {
            /*
             * Extract the unreplaced variable name, which is clearly in between the angled
             * brackets, then throw an Exception.
             */
            throw new VariableNotFoundException(exp, exp.substring(start, end + 1));
        }

        /* Adjust the number spacing before passing the expression back to the evaluator */
        exp = adjustNumberSpacing(exp);
        return exp.trim();
    }

    /**
     * Substitutes expressions within parenthesis (<code></code>, <code></code>) with their results.
     * This ensures that while evaluating an expression containing parenthesized parts, those
     * parenthesized parts are evaluated first. This is done so that ExpressionParser follows the
     * BODMAS rule.
     *
     * @param exp the expression to be parsed
     * @return the expression such that all parenthesized parts
     *         have been evaluated
     * @throws com.github.sahasatvik.math.UnmatchedBracketsException
     *         thrown when brackets in the expression are not
     *         closed
     * @throws com.github.sahasatvik.math.ExpressionParserException
     *         thrown if the parenthesized sections cannot be
     *         parsed
     * @see #indexOfMatchingBracket(String, int, char, char)
     * @since 1.0
     */
    protected String parseParenthesis (String exp) throws ExpressionParserException {
        String result = "";
    
```

```

/*
 * Buffer the extreme ends with spaces, to make sure no Exceptions are thrown
 * while extracting portions of the expression.
 */
exp = " " + exp + " ";

/* Continue replacing parenthesized sections as long as a parenthesis is present */
while (exp.indexOf("(") != -1) {
    /* Store the indices of the opening and closing parenthesis */
    int start = exp.indexOf("(");
    int end = indexOfMatchingBracket(exp, start, '(', ')');
    /* The enclosed section is simply another expression. Pass it to the evaluator */
    result = evaluate(exp.substring(start + 1, end));

    /*
     * This is a special case. Make sure that ' -(some_expression) ' is interpreted
     * as the negative of that expression.
     */
    if (exp.charAt(start - 1) == '-') {
        /* Multiply the enclosed section by -1, then evaluate the result */
        result = " ( -1 * ( " + result + " ) ) ";
        start--;
    }

    /* Graft the evaluated parenthesized portion back into the original expression */
    exp = exp.substring(0, start) + " "           // before the opening bracket
          + result + " "                         // evaluated part
          + exp.substring(end + 1);              // after the closing bracket
    }
    /* Adjust the number spacing before passing the expression back to the evaluator */
    exp = adjustNumberSpacing(exp);
    return exp.trim();
}

/**
 * Substitutes all occurrences of supported mathematical functions with their result.
 * A function must be present in the expression in the following format :
 * <code>function_name[function_argument]</code>, where the function argument can also
 * be an expression. The function name must be exactly 3 characters long, and be
 * immediately followed by a square bracket (<code>[</code>).
 * See {@link com.github.sahasatvik.math.MathParser#solveUnaryFunction(String, double)} for a
 * list of supported function names.
 *
 * @param exp the expression to be parsed
 * @return the expression such that all instances of
 * functions are evaluated

```

```

*      @throws com.github.sahasatvik.math.MissingOperandException
*              thrown if there is no function argument
*      @throws com.github.sahasatvik.math.FunctionNotFoundException
*              thrown when an unrecognized function name
*              is found in the expression
*      @throws com.github.sahasatvik.math.UnmatchedBracketsException
*              thrown when a square bracket is not closed
*      @throws com.github.sahasatvik.math.ExpressionParserException
*              thrown if the function argument cannot be
*              parsed
*      @see      com.github.sahasatvik.math.MathParser#solveUnaryFunction(String, double)
*      @since    1.0
*/

```

```

protected String parseFunctions (String exp) throws ExpressionParserException {
    String result = "";
    String func = "";
    double x = 0.0;
    /*
     * Buffer the extreme ends with spaces, to make sure no Exceptions are thrown
     * while extracting portions of the expression.
     */
    exp = " " + exp + " ";
    try {
        /*
         * This is another special case. Make sure that expressions of the form
         * 'number!' are interpreted as the factorial of that number. This can
         * be done simply by replacing all such cases with the expression 'fct[number]',
         * as 'fct[]' is a valid function name and can be calculated later.
         */
        exp = exp.replaceAll(numberRegex + "\\s+!", " fct[$1] ");

        /*
         * Continue evaluating functions as long as square brackets ([]) are present.
         * Here, a function is represented in the format 'fnc[expression]'. Thus, the
         * presence of square brackets ([]) implies that a function is present.
         */
        while (exp.indexOf("[") != -1) {
            /* Store the indices of the opening and closing square brackets */
            int start = exp.indexOf("[");
            int end = indexOfMatchingBracket(exp, start, '[', ']');

            /*
             * Here, all function names are exactly 3 characters long. Thus, the
             * function name is simply the 3 characters preceding the opening bracket.
             */

```

```

func = exp.substring(start - 3, start);

/*
 * The section enclosed within the brackets is also an expression.
 * Evaluate it, and check whether it is a number. This will be the
 * function argument.
 */
x = Double.parseDouble(evaluate(exp.substring(start + 1, end)));

/* Pass the function name and argument to a function solver */
result = "" + solveUnaryFunction(func, x);

/*
 * This is a special case similar to that in parseParenthesis(String).
 * Make sure that ' -fnc[some_expression] ' is interpreted as the negative
 * of the result of that function.
 */
if (exp.charAt(start - 4) == '-') {
    /* Multiply the enclosed section by -1, then evaluate the result */
    result = evaluate(" ( -1 * ( " + result + " ) ) ");
    start--;
}
/* Graft the evaluated portion back into the original expression */
exp = exp.substring(0, start-3) + " "           // before the function
      + result + " "                             // evaluated part
      + exp.substring(end+1);                     // after the function
}
} catch (NullExpressionException e) {
    /* Throw an Exception if the function is missing its argument */
    throw new MissingOperandException(exp, func + "[]");
} catch (FunctionNotFoundException e) {
    /* Throw an Exception if an extracted function name is unsupported */
    throw new FunctionNotFoundException(exp, func);
} catch (ExpressionParserException e) {
    /* Pass on any Exceptions encountered while evaluating the argument */
    throw e;
} catch (Exception e) {
    /* Pass on any other Exceptions as ExpressionParserExceptions */
    throw new ExpressionParserException(exp);
}
/* Adjust the number spacing before passing the expression back to the evaluator */
exp = adjustNumberSpacing(exp);
return exp.trim();
}

/**

```



```

* Substitutes all binary expressions involving arithmetic operators with their result.
* Operations are performed following the BODMAS rule. The resultant parsed String
* is free of all operators, thus containing only numbers.
* See {@link com.github.sahasatvik.math.MathParser#solveBinaryOperation(double, String, double)}
* for a list of supported operators. See {@link #operators}, which defines the order of
* operations.
*
* @param exp the expression to be parsed
* @return the expression such that all arithmetic operations
*         have been carried out
* @throws com.github.sahasatvik.math.MissingOperandException
*         thrown if a binary operator is missing an operand
* @see com.github.sahasatvik.math.MathParser#solveBinaryOperation(double, String, double)
* @since 1.0
*/

```

```

protected String parseOperators (String exp) throws MissingOperandException {
    int leftIndex, rightIndex;

    /* Split the expression into a stack of operators and operands */
    String[] stack = exp.split("\\s+");

    /* Loop through all supported operators (in order) */
    for (String op : operators) {
        /* Loop through the stack, searching for a match with the operator */
        for (int i = 0; i < stack.length; i++) {
            if (stack[i].equals(op)) {
                leftIndex = rightIndex = i;
                /* Keep on searching before the operator until a valid operand is found */
                while (leftIndex >= 0 && !isNumber(stack[leftIndex]))
                    leftIndex--;
                /* Keep on searching after the operator until a valid operand is found */
                while (rightIndex < stack.length && !isNumber(stack[rightIndex]))
                    rightIndex++;
                try {
                    /* Get the operands */
                    double left = Double.parseDouble(stack[leftIndex]);
                    double right = Double.parseDouble(stack[rightIndex]);
                    /*
                     * Pass the operands and the operator to an operator solver,
                     * then replace the operator with the result. Also remove the
                     * operands.
                     */
                    stack[i] = "" + solveBinaryOperation(left, op, right);
                    stack[leftIndex] = stack[rightIndex] = "";
                } catch (Exception e) {

```

```

        /* Throw an Exception if there is a missing operand */
        throw new MissingOperandException(exp, op);
    }
}

}
exp = "";

/* Recombine the stack into the solved expression */
for (String s : stack) {
    exp += s;
}
return exp.trim();
}

/**
 * Adjusts the spacings between numbers, variables, functions, operators, etc in an expression.
 * Each number will be enclosed withhin a 'buffer' of spaces. Instances of signed numbers
 * immediately following anotheoer number will be interpreted as their sum.
 * (<code>    1  -1</code> is simply <code>1 + -1</code>)
 *
 * @param exp        the expression to be parsed
 * @return           the expression with adjusted spacing
 * @since 1.0
 */

protected static String adjustNumberSpacing (String exp) {
    /* Make sure numbers are all spaced out from other symbols */
    exp = exp.replaceAll(numberRegex, " $0 ");
    /* Make sure the sign signed numbers is also considered during addition/subtraction */
    exp = exp.replaceAll(numberRegex + "\\s+" + signedNumberRegex, " $1 + $6 ");
    return exp;
}

/**
 * Finds the index of a matching closing bracket in a String, given the index of the
 * opening one. This method can also be given any characters as opening and closing brackets.
 * Nesting of brackets has also been dealt with.
 *
 * @param str        the String containing the brackets
 * @param pos        the index of the opening bracket
 * @param open       the character to be recognized as an opening bracket
 * @param close      the character to be recognized as a closing bracket
 * @return           the index of the matching closing bracket
 * @throws com.github.sahasatvik.math.UnmatchedBracketsException
 *                  thrown if the specified opening bracket is unclosed

```

```

*      @since 1.0
*/

protected static int indexOfMatchingBracket (String str, int pos, char open, char close)
                                         throws UnmatchedBracketsException {
    int tmp = pos;
    /* Loop through the String, forward from the position of the opening bracket */
    while (++pos < str.length()) {
        /* Exit the loop as soon as a closing bracket is found */
        if (str.charAt(pos) == close)
            return pos;
        /*
         * If another opening bracket is found, it becomes clear that bracketed expressions
         * have been nested. Thus, the next closing bracket will not match the bracket
         * we have targeted. In order to return the correct bracket, simply skip everything
         * within the nested portion. This is done by calling
         * indexOfMatchingBracket(String, int, char, char) recursively.
         */
        if (str.charAt(pos) == open)
            pos = indexOfMatchingBracket(str, pos, open, close);
    }
    if (pos >= str.length()) {
        /* Throw an Exception if a matching bracket is not present */
        throw new UnmatchedBracketsException(str, tmp);
    }
    return pos;
}
}

```