CSC 151 Assignment 5

Objectives:

- Implementing Stacks with java.util.Stack
- Using Generics
- Using Interfaces
- Evaluating parenthesized prefix Lisp expressions

Definition

Please refer to Project 7 of Chapter 5 in the textbook. In the language Lisp, each of the four basic arithmetic operators appears before an arbitrary number of operands, which are separated by spaces. The resulting expression is enclosed in parentheses. The operators behave as follows:

- (+ a b c ...) returns the sum of all the operands, and (+) returns 0.
- (− a b c ...) returns a − b − c − ..., and (− a) returns −a. The minus operator must have at least one operand.
- (* a b c ...) returns the product of all the operands, and (*) returns 1.
- (/ a b c ...) returns a / b / c /..., and (/ a) returns 1 / a. The divide operator must have at least one operand.

You can form larger arithmetic expressions by combining these basic expressions using a fully parenthesized prefix notation. For example, the following is a valid Lisp expression:

(+ (− 6) (* 2 3 4) (/ (+ 3) (*) (− 2 3 1)))

This expression is evaluated successively as follows:

(+ (− 6) (* 2 3 4) (/ 3 1 −2))

(+ −6 24 −1.5)

16.5

Design and implement an algorithm that uses a stack to evaluate a legal Lisp expression composed of the four basic operators and integer values. Write a program that reads such expressions and demonstrates your algorithm.

The interfaces and the classes in this assignment are partially or completely provided in your textbook. Please complete the partial implementations and submit all files listed in the submission instructions.

Interface StackInterface

- Use the same code provided at the end of the previous assignment.
- It will be in **package assignment** (it is required otherwise **Gradescope** cannot run the tests)

Class OurStack

- Use the partial code provided at the end of the document.
- It will be in **package assignment** (it is required otherwise **Gradescope** cannot run the tests)
- See the details below written as comments and complete the partial code as instructed.
- Test your class in the main method.
- Test your class by Junit

Class LispToken

- Use the partial code provided at the end of the document.
- It will be in **package assignment** (it is required otherwise **Gradescope** cannot run the tests)
- See the details below written as comments and complete the partial code as instructed.
- Test your class in the main method.
- Test your class by JUnit

Class LispExpressionEvaluator

- Use the partial code provided at the end of the document.
- It will be in **package assignment** (it is required otherwise **Gradescope** cannot run the tests)
- See the details below written as comments and complete the partial code as instructed.
- Test your class in the main method.
- Test your class by JUnit

General Submission Instructions

- You may have groups for the assignments **if it is allowed.** If you have a group state it **explicitly in the honor code** by writing all group members' names. Please name the pdf files accordingly. **However, all group members should submit individually.**
- All submissions require a package called **assignment** and your source codes (.java files) will be under it.
- You should submit
    - Your java files to **Gradescope** as required and
    - Your java files and sample outputs for various scenarios as a **single pdf** file named as **YourFullName(s)_AssignmentN.pdf** with the **honor code** at the top as a comment to Nexus under the appropriate link. Replace N in AssignmentN with the appropriate value. (e.g., if I submit Assignment1 I will submit a pdf file whose name is **ZeynepOrhan_Assignment1.pdf.** If I have a group whose members are me and Jane Doe, then the file name will be **JaneDoeZeynepOrhanAssignment1.pdf** and the honor code will be modified). Use the template .docx file attached and modify it accordingly.

- **If you use any resource to get help, please state it in the honor code. Otherwise, it will be an honor code violation!!!**
- **Gradescope provides a feature to check code similarity by comparing all submissions. If the percentage of your code's similarity is above a threshold, then this will be an honor code violation!!!**
- Make sure that you have no compiler errors.
- When you have compiler error free codes and submit appropriately to Gradescope, your code will be tested automatically and graded. You do not need to know the test cases, but you will see the test results. Informative messages will be provided for success and failure cases.
- You can resubmit your files any number of times if you have failing tests until the due date.
- Your final submission or the one you selected will be graded after the due date of the assignment.
- Please start as early as possible so that you may have time to come and ask if you have any problems.
- Read the late submission policy in the syllabus.
- Please be aware that the correctness, obeying the OOP design and submission rules are equally important
- Gradescope testing will generally be 100% and sometimes manual grading will be done. Please be aware that **your grade can be reduced if you do not follow the instructions and apply good programming practices**.
- Use the starter code if provided.
- Do not use any predefined Collection classes of Java unless stated otherwise.

Assignment specific instructions

- General submission instructions apply.
- **Group work: ALLOWED!!!**
- **Due date:** February 6, 2021, until 11:50 PM
- **Nexus:** Submit the code files (**StackInterface.java, OurStack.java, LispToken.java, LispExpressionEvaluator.java**) as a single pdf file named as **YourFullName_Assignment4.pdf** with the honor code at the top as a comment under Assignment 5 link.
- **Gradescope:** Submit the java files (**StackInterface.java, OurStack.java, LispToken.java, LispExpressionEvaluator.java**) under the Assignment 5
- **Grading:**
    - Gradescope: Yes
    - Manual: No
- **Starter code:** Use the code given below

```java
/*
 * Honor code
 */

package assignment;

/**
 * An interface for the ADT stack.
 *
 * @author Frank M. Carrano
 * @author Timothy M. Henry
 * @version 5.0
 */
public interface StackInterface<T> {
        /**
         * Adds a new entry to the top of this stack.
         *
         * @param newEntry An object to be added to the stack.
         */
        public void push(T newEntry);

        /**
         * Removes and returns this stack's top entry.
         *
         * @return The object at the top of the stack.
         * @throws EmptyStackException if the stack is empty before the operation.
         */
        public T pop();

        /**
         * Retrieves this stack's top entry.
         *
         * @return The object at the top of the stack.
         * @throws EmptyStackException if the stack is empty.
         */
        public T peek();

        /**
         * Detects whether this stack is empty.
         *
         * @return True if the stack is empty.
         */
        public boolean isEmpty();

        /** Removes all entries from this stack. */
        public void clear();
}
```

```java
package assignment;

import java.util.Stack;

/**
 * A class of stacks.
 *
 * @author Frank M. Carrano
 * @version 5.0
 */
public class OurStack<T> implements StackInterface<T> {
        private Stack<T> theStack;
        // Implement the constructor with no parameter, push, peek, pop, isEmpty and clear
        // by using the private Stack theStack of type java.util.Stack









} // end OurStack
```

```java
package assignment;


/**
 * This class represents either an operand or an operator for an arithmetic
 * expressions in Lisp.
 *
 * @author Charles Hoot
 * @version 5.0
 */
public class LispToken {

        private Character operator;
        private Double operand;
        private boolean isOperator;


        /**
         * Constructor for objects of class LispToken for operators.
         * isOperator is true and operand is 0.0, operator is anOperator
         *
         * @param anOperator of type Character
         */




        /**
         * Constructor for objects of class LispToken for operands.
         * isOperator is false and operand is the value, operator is ' '
         *
         * @param value of type Double
         */




        /**
         * applyOperator: Applies this operator to two given operand values.
         *
         * @param value1 The value of the first operand.
         * @param value2 The value of the second operand.
         * @return The real result of the operation.
         */




        /**
         * getIdentity: Gets the identity value of this operator. For example, x + 0 = x, so 0 is the
         * identity for + and will be the value associated with the expression (+).
         *
         * @return The identity value of the operator as Double.
         */




        /**
         * takesZeroOperands: Detects whether this operator returns a value when it has no operands.
         *
         * @return True if the operator returns a value when it has no operands, or
         *         false if not.
         */




        /**
         * getValue: Gets the value of this operand.
         *
         * @return The real value of the operand.
         */
```

```
        /**
         * isOperator: Returns true if the object is an operator.
         *
         * @return True is this object is an operator.
         */




        /**
         * toString: Returns a string representation of the operator or operand
         *
         * @return String
         */



}
```

```
package assignment;

import java.util.ArrayList;
import java.util.Scanner;

/**
 * This class evaluates a simple arithmetic Lisp expression of numeric values.
 *
 * @author Charles Hoot
 * @author Jesse Grabowski
 * @author Joseph Erickson
 * @author Zeynep Orhan modified
 * @version 5.0
 */
public class LispExpressionEvaluator {

        /**
         * Evaluates a Lisp expression.
         *
         * The algorithm: Scan the tokens in the string.
         * If you see "(", push the next operator onto the stack.
         * If you see an operand, push it onto the stack.
         * If you see ")", Pop operands and push them onto a second stack until you find an
         * operator. Apply the operator to the operands on the second stack. Push the
         * result on the stack.
         *
         * What may occur? (Samples only)

         * If you run out of tokens, the value on the top
         * of the stack is the value of the expression.
         * OR
         * How to detect illegal expressions:
         * If you read numeric values and the
         * expression stack is empty
         * Error message: Bad Expression: needs an operator for the data
         *
         * If there is a ) and the expression stack is empty
         * Error message: mismatched )
         *
         * If there is a ) and operands needed but the expression stack is empty
         * Error message: mismatched )
         *
         * If the top operator requires at least one operand but it is not in the expression stack
         * Error message:operator nameOfTheOperand + " requires at least one operand"
         *
         * If the string does not have any more characters but the expression stack is not empty
         * Error message:incomplete expression / multiple expressions
         *
         * If the operator is not one of the +, -, *, ?
         * Error message: found an operator when we should not
         *
         * If the expression is legal
         * Message:" and evaluates to " + whateverTheResultIs
         *
         *
         *
         *
```

```
         * Format of sample messages:
         * Message for a legal expression
         *
         * The expression '(+ (- 1) (* 3 3 4) (/ 3 2 3) (* 4 4))' is legal in Lisp:
         * and evaluates to 51.5
         *
         * Message for an illegal expression
         *
         * The expression '(+ (-) (* 3 3 4) (/ 3 2 3) (* 4 4))' is not legal in Lisp:
         * operator - requires at least one operand
         *
         * @param lispExp A string that is a valid lisp expression.
         * @param mes     An ArrayList of strings that stores the messages generated.
         * @return A double that is the value of the expression.
         */
        @SuppressWarnings("resource")
        public static double evaluate(String lispExp, ArrayList<String> mes) {
                StackInterface<LispToken> expressionStack = new OurStack<>();
                StackInterface<LispToken> secondStack = new OurStack<>();

                boolean nextIsOperator = false;
                Scanner lispExpScanner = new Scanner(lispExp);

                // Use zero or more white spaces as delimiter
                // that breaks the string into single characters
                lispExpScanner = lispExpScanner.useDelimiter("\\s*");

                // Hint: use
                // lispExpScanner.hasNext() to test if there are more tokens
                // lispExpScanner.hasNextInt() to test if there is an integer
                // lispExpScanner.next() to get the next String









        }

        public static void main(String args[]) {

                String tests[] = {
                                "(+ 1 3)",
                                "(- 1)",
                                "(-)",
                                "(+)",
                                "(*)",
                                "(/)",
                                "(- 1 2)",
                                "(+ (- 1) (* 3 3 4) (/ 3 2 3) (* 4 4))",
                                "(+ (-) (* 3 3 4) (/ 3 2 3) (* 4 4))",
                                "(+ (- 1) (* 3 3 4) ) 5 (* (/ 3 2 3) (* 4 4))",
                                "(+ (- 1) (* 3 3 4) (/ 3 2 3)) (* 4 4))",
                                "+ (- 1) (* 3 3 4) (/ 3 2 3)) (* 4 4))"
                                };

                ArrayList<String> mes = new ArrayList<>();
                for (int i = 0; i < tests.length; i++) {
                        evaluate(tests[i], mes);
                        System.out.println(mes.get(i));
                }

                System.out.println("Done.");
        }
}
```

Output

```
The expression '(+ 1 3)'
is legal in Lisp:
and evaluates to 4.0


The expression '(- 1)'
```

```
is legal in Lisp:
and evaluates to -1.0


The expression '(-)'
is not  legal in Lisp:
operator - requires at least one operand

The expression '(+)'
is legal in Lisp:
and evaluates to 0.0


The expression '(*)'
is legal in Lisp:
and evaluates to 1.0


The expression '(/)'
is not  legal in Lisp:
operator / requires at least one operand

The expression '(- 1 2)'
is legal in Lisp:
and evaluates to -1.0


The expression '(+ (- 1) (* 3 3 4) (/ 3 2 3) (* 4 4))'
is legal in Lisp:
and evaluates to 51.5


The expression '(+ (-) (* 3 3 4) (/ 3 2 3) (* 4 4))'
is not  legal in Lisp:
operator - requires at least one operand

The expression '(+ (- 1) (* 3 3 4) ) 5 (* (/ 3 2 3) (* 4 4))'
is not  legal in Lisp:
incomplete expression / multiple expressions

The expression '(+ (- 1) (* 3 3 4) (/ 3 2 3)) (* 4 4))'
is not  legal in Lisp:
mismatched )

The expression '+ (- 1) (* 3 3 4) (/ 3 2 3)) (* 4 4))'
is not  legal in Lisp:
found an operator when we should not
Done.
```