מבני נתונים – תרגיל 2 (מעשי)

ייצוג ביטויים אריטמתטיים - Postfix,Infix, Prefix

תאריך פרסום: 17.04.2022

תאריך הגשה: 8.5.2022 ,23:59

מתרגל אחראי: ענבל רושנסקי

הנחיות:

- יש לקרוא הנחיות לגבי הגשת העבודה באתר הקורס.
- שימו לב כי העבודה תיבדק אוטומטית, אך תיבדק גם צורת כתיבה של הקוד שלכם (כפי שיפורט בהמשך) באופן ידני. ועל כן, הציון המתקבל בזמן ההגשה עשוי לרדת לאחר הבדיקה הידנית, במידה והעבודה לא כתובה על פי הסטנדרט שמפורט במסמך זה.
- שאלות לגבי העבודה יש לשאול בפורום באתר הקורס או בשעות קבלה של ענבל רושנסקי.
- תיאור המחלקות הנתונות והדרושות למימוש מופיע בסוף המסמך.
- יש לקרוא את כל המסמך טרם תחילת העבודה.

נושאי העבודה: מחסנית, עץ בינארי והכרת ג'אווה.

מבוא

אחד היישומים החשובים של מדעי המחשב הינו ייצוגם של ביטויים מתמטיים.

כל אחת מהפעולות המתמטיות - חיבור, חיסור, כפל ,וחילוק הינה **פעולה בינארית**, כלומר פעולה שבה יש **אופרטור** (operator) אחד - הפעולה החישובית, הפועל על שני **אופרנדים** (operand) - המשתנים או המספרים עליהם מבוצעת הפעולה החישובית.

ניתן לייצג פעולה חישובים באחד משלושת הייצוגים:

- ייצוג **תוכי infix** - האופרטור נמצא בין שני האופרנדים: A + B
- ייצוג **תחילי prefix** - האופרטור נמצא לפני האופרנדים: + A B
- ייצוג **סופי postfix** - האופרטור נמצא אחרי שני האופרנדים: A B +

כדי לחשב את הביטוי A + B * C כפי שהוא כתוב בייצוג תוכי, יש להתחשב ב**כללי הקדימויות** (precedence) של הפעולות החישוביות.

<u>כללי הקדימות (מהגבוה לנמוך):</u>

- העלאה בחזקה (נסמן אותה בסימן ^)
- כפל, חילוק
- חיבור, חיסור
- כאשר מופיעים כמה אופרטורים מאותה דרגת קדימות, יהיה סדר הקדימויות משמאל לימין
- כאשר מופיעות כמה פעולות חזקה A ^ B ^ C, יהיה סדר הקדימויות מימין לשמאל: A ^ (B ^ C)
- שימוש בסוגריים מאפשר לכפות סדר קדימויות שונה.

במתמטיקה אנו מבצעים חישובים בייצוג תוכי (האופרטור נמצא בין האופרנדים עליהם הוא מתבצע (לדוגמה: 5 + 3). ייצוג תוכי מחייב שימוש בסוגריים כדי לשמור על סדר החישוב הרצוי: (5 + 3) * 2.

לעומת זאת, **ביטויים בייצוג סופי ותחילי אינם זקוקים לסוגריים.** סדר הופעת האופרטורים בביטוי בייצוגים אלו קובע את סדר הפעולות בחישוב הביטוי.

ביטויים בצורת infix יותר אינטואיטיביים לבני אדם, ואנו משתמשים בהם ביום יום. אך למחשב יותר קל לחשב את הביטויים הנתונים בצורה prefix או postfix. מכיוון שאנחנו לא יכולים לצפות מהמשתמש להקליד ביטוי בצורה prefix או postfix, אנחנו חייבים להמיר את הביטוי infix (שצוין על ידי המשתמש) לביטוי prefix/postfix לפני שמחשב יחשב את התוצאה.

בתרגיל זה תכתבו תוכנית הממשת מחשבון פשוט לחישוב ביטויים אריתמטיים.

התוכנית תכלול את החלקים הבאים:

1) המרה בין ייצוגים שונים.
2) חישוב תוצאת הביטוי.

<u>המרה בין ייצוג תוכי לייצוג סופי - Infix-to-postfix converter</u>

קיימות מספר שיטות להמרת הביטוי מייצוג אחד לאחר. בתרגיל זה נממש שתי שיטות שונות להמרה.

א. בעזרת מחסנית
ב. בעזרת עצים בינאריים

חלק א' - המרת ביטוי תוכי לביטוי סופי בעזרת מחסנית וחישוב ערך הביטוי שנתון בייצוג סופי

בחלק זה של העבודה, תקבלו ביטוי בייצוג תוכי. יש להמיר את הביטוי לייצוג סופי, ולאחר מכן לחשב את ערך הביטוי בייצוג סופי.

המרה מייצוג תוכי לייצוג סופי

1. הקלט:
   - ביטוי infix הנתון כמחרוזת **כאשר מופיעים רווחים לפני ואחרי כל אופרנד, לפני ואחרי כל אופרטור, ולפני ואחרי כל סוגר, מלבד בהתחלה ובסוף של הביטוי**.
     לדוגמה, "(2 - 18) * (3 + 7)".
   - ביטוי הקלט מכיל סוגריים עגולים בלבד (סוגריים מקוננים).
   - ביטוי הקלט מכיל אך ורק את הפעולות '^', '/', '*', '-', '+'.
   - **ניתן להניח כי הקלט תקין.**
2. הפלט: התוצאה של ההמרה תוחזר כמחרוזת. **כאשר מופיעים רווחים לפני ואחרי כל אופרנד, ולפני ואחרי כל אופרטור, מלבד בהתחלה ובסוף הביטוי.** לדוגמה, עבור הקלט "(2 - 18) * (3 + 7)", התוכנית תחזיר מחרוזת "* - 2 18 + 3 7".

האלגוריתם: infix to postfix converter

נעבור על מחרוזת הקלט משמאל לימין. כל איבר בקלט נקרא token.
לדוגמה, עבור מחזורת הקלט "5 * (4 + 3)", ה-tokens הם:), 3 +, 4, (, * ו-5.

1. input - an infix expression
2. *str* ← "" // Create an empty string
3. *stack* ←∅ // Create an empty stack
4. for each token *token* in the input (from left to right) do
   - 4.1 if (*token* is an opening bracket), push *token* to the *stack*
   - 4.2 else if (*token* is a closing bracket)
     - pop elements from the *stack* until (not including) opening bracket
     - concatenate each popped element to the end of the *str*
     - at the end, pop opening bracket from the *stack*
   - 4.3 else if (*token* is an operator)
     - pop elements from the *stack* until (not including) operator with lower precedence or non-operator (e.g. opening bracket)

- concatenate each popped element to the end of the *str*
- at the end, push *token* to the *stack*

4.4 else if (*token* is a number), concatenate *token* to the end of the *str*

5. while *stack* is not empty

5.1 pop and concatenate each element to the end of the *str*

6. return *str*

הערה: על מנת לבדוק מה ה-class של משתנה מסויים, נשתמש באופרטור של ג'אווה instanceof (דוגמה כאן).

להלן הדגמה של פעולת האלגוריתם על הביטוי "(2 - 18) * (3 + 7)"

token	str	stack	הפעולה
(	""	(	דחיפת (
7	"7 "	(	שרשור למחרוזת
+	"7 "	+ , (	דחיפת +
3	"7 3 "	+ , (	שרשור למחרוזת
)	"7 3 + "	The stack is empty	שליפה ושרשור, שליפת)
*	"7 3 + "	*	דחיפת *
(	"7 3 + "	* , (	דחיפת)
18	"7 3 + 18 "	* , (	שרשור למחרוזת
-	"7 3 + 18 "	* , (, -	דחיפת -
2	"7 3 + 18 2 "	* , (, -	שרשור למחרוזת
)	"7 3 + 18 2 - "	*	שליפה ושרשור, שליפת)
	"7 3 + 18 2 - *"	The stack is empty	הסתיים הקלט, שליפה ושרשור למחרוזת

תוצאת ההמרה מביטוי תוכי (infix) לביטוי סופי (postfix): "7 3 + 18 2 - *" .

חישוב ערך הביטוי בייצוג סופי

להלן האלגוריתם לחישוב ערך הביטוי הנתון בייצוג סופי. האלגוריתם משתמש במחסנית.

שימו לב: (1) כל אופרטור במחרוזת מתייחס לשני האופרנדים הקודמים לו במחרוזת, (2) כל אחד מהאופרנדים יכול להיות תוצאה של הפעלת אופרטור קודם.

האלגוריתם עובר על הביטוי שנתון בייצוג postfix משמאל לימין ומבצע חישובים.

1. for each token *token* in the input (from left to right) do
    1.1. if the token *token* is an operator
        1.1.1. pop the top two elements from the stack
        1.1.2. perform the operation on the elements
        1.1.3. push the result of the operation to the stack
    1.2. else // the token is a number
        1.2.1. push *token* to the stack
2. pop from the stack the only element left and return it // return the result

כאשר לא נשארו יותר איברים בקלט, המחסנית חייבת להכיל רק איבר אחד - הערך הסופי של הביטוי.
מידע נוסף על תהליך החישוב תוכלו למצוא בעמוד הוויקיפדיה שמתאר postfix notation.

להלן הדגמת פעולת האלגוריתם על הביטוי הבא בייצוג סופי: "+ 3 ^ 2 * + / 2 8 - 3 + 2 3 6".

token	op1	op2	result	stack
6				6
2				2, 6
3				3, 2, 6
+	2	3	5	5, 6
-	6	5	1	1
3	6	5	1	3, 1
8	6	5	1	8, 3, 1
2	6	5	1	2, 8, 3, 1
/	8	2	4	4, 3, 1
+	3	4	7	7, 1

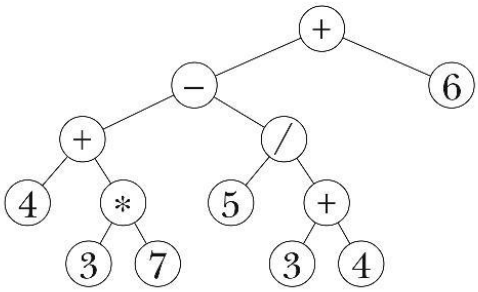
*	1	7	7	7
2	1	7	7	2, 7
^	7	2	49	49
3	7	2	49	3, 49
+	49	3	52	52

ותוצאת החישוב היא 52.

חלק ב' – שימוש בעץ בינארי להמרה בין ייצוגים שונים של הביטוי וחישוב ערך הביטוי

ניתן לתאר ביטוי אריתמטי בעזרת עץ בינארי, כאשר קדקודים פנימיים שלו הם פעולות חשבון ועלים הם מספרים.
לדוגמה, העץ שבציור הבא מתאר את הביטוי: 6 + (((4 + (3 * 7)) + 4) – (5 / (3 + 4))) .

עץ בינארי שמתאר ביטוי אריתמטי נקרא expression tree.



שימו לב: אם נסרוק את עץ הביטויים בסריקת inorder, נקבל ביטוי אריתמטי בצורת infix, אם נסרוק את העץ בסריקת preorder נקבל ביטוי אריתמטי בצורת prefix, ואם נסרוק את העץ בסריקת postorder, נקבל ביטוי אריתמטי בצורת postfix.

בחלק זה של התרגיל, בהינתן ביטוי בייצוג postfix, יש לבנות ממנו expression tree.
בנוסף, תשתמשו בעץ הביטויים כדי לחשב את ערך הביטוי.

האלגוריתם לבניית expression tree מביטוי בייצוג postfix

1. הקלט:
   - ביטוי postfix הנתון כמחרוזת **כאשר מופיעים רווחים לפני ואחרי כל אופרנד, לפני ואחרי כל אופרטור, מלבד בהתחלה ובסוף הביטוי.** לדוגמה, "+ 3 6" .
   - ביטוי הקלט מכיל אך ורק את הפעולות '^' , '/' , '*' , '-' , '+'.
   - **ניתן להניח שהקלט תקין.**

2. הפלט : עץ בינארי (expression tree) המתאר את הביטוי.

האלגוריתם לבניית עץ הביטויים משתמש במחסנית עזר.

1. *stack* ←∅ // Create an empty stack
2. for each token *token* in the input (from left to right) do
   - 1.1 if *token* is an operand
     - i *node* ← new leaf node with *token*
     - ii *stack*.push (*node*)
   - 1.2 else if *token* is an operator
     - i *right* ← *stack*.pop ()
     - ii *left* ← *stack*.pop ()
     - iii *node* ← new node with *token*, and *left* and *right* as its left and right children
     - iv *stack*.push (*node*)
3. *root* ← *stack*.pop ()

<u>הערכת (חישוב) ביטוי מתוך expression tree</u>

תהליך ההערכה הוא החלק הקל ביותר. בניגוד לתהליך ההמרה, אלגוריתמים להערכת הביטוי הם קצרים וקלים להבנה. הסיבה לכך היא שעכשיו אנחנו לא צריכים לדאוג לסדר העדיפות של פעולות.

את השיטה שמחשבת את ערך הביטוי יש לממש כשיטה **רקורסיבית**, שמקבלת הפנייה לצומת בעץ ומחשבת ערך הביטוי המיוצג על ידי תת עץ המושרש בצומת זה.

<u>Algorithm</u> evaluateExpression (Node *node*)

1. if (*node* is an operator)
   - 1.1 *operand1* ← evaluateExpression (*node*.left)
   - 1.2 *operand2* ← evaluateExpression (*node*.right)
   - 1.2 *result* ← *operand1* <*operator*> *operand2*
2. else
   - 2.1 *result* ← *node*.data // node contains a number
3. return *result*

חלק ג' - בדיקת הקוד

כחלק מהתרגיל, אתם תידרשו לכתוב בדיקות (tests) עבור הקוד שלכם, מעבר לבדיקות שיינתנו לכם במערכת ההגשה.
פירוט של חלק זה נמצא תחת המחלקה Tester בחלק המפרט את המחלקות בתרגיל.

חשוב:

יש להקפיד על קוד ברור וקריא, עם שמות משתנים משמעותיים. יש לכתוב הערה לפני כל בלוק קוד לא טריוויאלי ולפני כל פונקציה, שמסבירה מה הבלוק/פונקציה עושה. **אין לייבא ספריות חיצוניות ובכללי אסור להשתמש בפקודה import באף אחד מהמחלקות שאתם מממשים.** שימו לב ש-eclipse או כל סביבה אחרת, לא מוסיפים לכם בטעות import מיותר ותדאגו למחוק את כל ה-imports לפני הגשת העבודה, כיוון שיש כלי אוטומטי שבודק זאת.
במידה והקוד לא יעמוד בסטנדרט שנכתב לעיל, יורדו על כך נקודות מכל אחד מהחלקים בו הקוד לא יעמוד בדרישות.

פירוט המחלקות בתרגיל:

המחלקות שאנו מספקים לכם- אין לשנות מחלקות אלה:

Interface *Stack* - The `Stack` interface (in the Stack.java file) declares the basic functionality of a stack (i.e. push, pop and isEmpty).

Class *StackAsArray* - The `StackAsArray` class (in the StackAsArray.java file) implements the `Stack` interface using a dynamic array with an increasing capacity that can be expanded as needed. You are encouraged to look at the code and try to understand it.

Class *CalcToken* - The abstract class `CalcToken` is provided with basic common capabilities that will assist you in parsing the tokens (elements) of the string. **All types of token must extend** this class.

Class *BinaryOp* - The abstract class `BinaryOp` (in the BinaryOp.java file) is provided and is used to represent a binary operation, that is an operation that is executed on two operands. You are encouraged to look at the code and try to understand it. You should also think why this class is abstract.

Classes *OpenBracket and CloseBracket* - The classes `OpenBracket` (in the OpenBracket.java file) and `CloseBracket` (in the CloseBracket.java file) extend `CalcToken` and represent the open and close brackets respectively. Really nothing exciting.

Class *TreeNode -* This class represents a node in a binary tree and will assist you in building an expression tree. And again, you are encouraged to look at the code and try to understand it.

המחלקות שאתם צריכים לממש או שניתנו באופן חלקי:

Token Classes:

You should **create** all the token classes, which are all subclasses of the abstract class `CalcToken`. There are two types of tokens:

1. Tokens which represent a numerical value. You must create a class `ValueToken` with the following methods/constructors:
   a. `ValueToken(double val)` where `val` is the number the token should represent.
   b. `double getValue()` returns the value of the token.
   c. `String toString()` which is inherited from the abstract parent class `CalcToken`.
2. Tokens which represent operators. These are described by the abstract class `BinaryOp`. You must implement classes `AddOp, SubtractOp, MultiplyOp, DivideOp`, and `PowOp`, which represent addition, subtraction, multiplication, division, and power respectively. All subclasses of `BinaryOp` must have the following methods:
   a. `double operate(double left, double right)` - returns the result of the operation using its left and right operands (note that operand order can matter, depending on operator).
   b. `double getPrecedence()` - returns the precedence of the operation.
   c. `String toString()` - inherited from `CalcToken` and represents the mathematical symbol of the operation.

Class *ExpTokenizer:*

In order to convert a string into a series of tokens, you will need to have the class `ExpTokenizer`. For this purpose, we created most of the class for you.

Note: you may assume that all the tokens are separated by the space character like in the examples we showed.

You will need to modify the method `nextElement()`, as exemplified below.

```java
public Object nextElement() {
	CalcToken resultToken = null;
	String token =  nextToken();
	if (token.equals("+"))
		resultToken =  new AddOp();
	else if (token.equals("*"))
		resultToken =  new MultiplyOp();

	// Fill the rest of the token cases by yourself

	else
		resultToken = new ValueToken(Double.parseDouble(token));

	return resultToken;
}
```

Note: you may assume that all the tokens in the input expression are either real numbers, open and closed brackets, or one of the five operators: "+", "-", "*", " ^" and "/".

You will need to support both real positive and negative numbers (e.g. `-4` or `4.2`).

Class *Calculator*

This will be an abstract class that you will need to create in a file **Calculator.java**. This class will be used to define a calculator's features, and both the StackCalculator and TreeCalculator will extend it. The class must have the following public abstract method:

- `double evaluate(String expr)`, where `expr` is a String representing some expression. This method evaluates (i.e. computes the numerical value of) `expr`.

Class *StackCalculator*

One of the primary classes in the code, which you will implement for Part A of the assignment. This class will extend the `Calculator` class. Note that the method `evaluate(String expr)`, receives `expr`, which is a string representing a valid **postfix** expression. This method evaluates (i.e. computes the numerical value of) `expr`, and returns its value. This method should use the algorithm described above, and should be implemented by using a `StackAsArray` object.

You should also write the function `infixToPostfix(String expr)` which receives a valid **infix** expression, and returns a valid postfix expression.

Both methods should be public. In order to compute an infix expression, first run `infixToPostfix` and then run `evaluate` on its result.

Class *TreeCalculator*

The second primary class in the code, which you will implement for Part B. This class will extend the `Calculator` class. Note that the method `evaluate (String expr)`, receives `expr`, which is a string representing a valid postfix expression. The `evaluate` method builds a corresponding expression tree, and evaluates (i.e. computes the numerical value of) `expr`, and returns its value. The `evaluate` method should use the algorithms described above and should be implemented using an `ExpressionTree`.

In addition, the `TreeCalculator` class should implement the following functions:

`String getInfix()`	השיטה מחזירה מחרוזת בצורת infix שמתארת את הביטוי. יש לשים רווח לפני ואחרי כל מספר, פעולה וסוגר, מלבד בהתחלה ובסוף הביטוי. **כמו כן, לפני ואחרי כל פעולה בין שני אופרנדים יש לשים סוגר בהתאם**, לדוגמה, ביטוי סופי "+ 3.0 2.0" יש להמיר לביטוי תוכי "(3.0 + 2.0)". מכאן, ייתכן שביטוי תוכי שמתקבל מכיל סוגריים מיותרים.
`String getPostfix()`	השיטה מחזירה מחרוזת בצורת postfix שמתארת את הביטוי. יש לשים רווח לפני ואחרי כל מספר, פעולה וסוגר, מלבד בהתחלה ובסוף הביטוי.
`String getPrefix()`	השיטה מחזירה מחרוזת בצורת prefix שמתארת את הביטוי. יש לשים רווח לפני ואחרי כל מספר, פעולה וסוגר, מלבד בהתחלה ובסוף הביטוי.

על שלושת השיטות לעבוד בזמן ריצה של $O(n)$, כאשר n הוא מספר הצמתים בעץ.

Class *ExpressionTree*

The class that should hold most of the code for your calculations with `TreeCalculator`. The ExpressionTree class should implement the following function:

`void BuildExpressionTree(String postfixExp)`	השיטה מקבלת ביטוי postfix תקין, ובונה את העץ לפי האלגוריתם שתואר בתרגיל.

Class *Tester*

The class `Tester` will help you test your code. You should add the tests for every public method. Your tests should include extreme ("boundary") cases, as well as the obvious "middle of the road" cases. Remember to choose **diverse** test cases. For example, testing `StackCalculator.evaluate (String expr)` with strings "1 2 +", "3 7 +", and "21 4 +" is redundant, since they all test the exact same thing - in this case, the addition of two numbers.

We suggest the following approach:

- Test basic methods before complicated ones.
- Test simple objects before objects that include or contain the simple objects.
- Do not test too many things in a single test case.

We also provide the file Tester.java. This file includes a main function that will be in-charge of running all the tests for your code. A helper function that you will use is the following:

```
private static void test (boolean exp, String msg)
```

This function receives a Boolean expression, and an error message. If the Boolean expression is evaluated to false, the function will print the error message to the screen. The function will also "count" for you the number of successful tests that you executed.

Please read carefully the code in this file. As you can see, we already provided a few tests to your code. You are required to add your own tests to check both Part A and Part B of the assignment, so the output of running the `Tester`, will be:

```
All <i> tests passed!
```

Where `i` (the total number of tests) should be at least **50**.

This part will be checked manually and automatically. Therefore, if you did not perform the tester correctly, the score on this part may decrease after the submission time has expired.

<u>הנחיות הגשה:</u>

עליכם להגיש את כל המחלקות שצוינו לעיל (מלבד המחלקות שניתנו לכם באופן מלא) למערכת ההגשה, לפי הוראות ההגשה במודל. **יש להגיש את הקבצים הבאים:**

ValueToken.java, AddOp.java, SubtractOp.java, MultiplyOp.java, DivideOp.java, PowOp.java, ExpTokenizer.java, Calculator.java, StackCalculator.java, TreeCalculator.java, ExpressionTree.java, Tester.java

הנחיות נוספות בנוגע להגשת עבודות יצורפו במסמך נפרד במודל.

בהצלחה ושבועה מהנה!