Assignment 03: Abstract Classes, Interfaces, and Dynamic Binding

Deadline: 26.03.2009, 8:15

Name: ______________________________ **Student ID**: ____________________

Informatik: ☐ G1 (Prähofer) ☐ G2 (Prähofer) ☐ G3 (Würthinger) ☐ G4 (Prähofer)

WIN: ☐ G1 (Khalil) ☐ G2 (Khalil) ☐ G3 (Schwinger)

Task	Points	Paper submission	Electronic submission	corr.	Points
Assignment 3	24	Prose description, Java source code, testing output	Java source code	☐	

Expression Tree (20 Points)

Abstract syntax trees can be used to represent arithmetic expressions. Numbers are converted to leaf nodes and operators to nodes with two children. Your task is to create classes for the elements of this tree that can visualize an abstract syntax tree as shown in Figure 1.

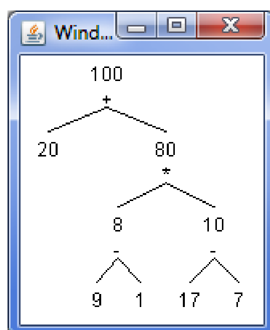

Figure 1: Expression tree generated from the input "20+(9-1)*(17-7)".

We provide Java source code such that you can focus on the essential parts of the implementation. In the package `expression` you can find three predefined public types that you should use:

- `Expression` is an interface that every node in the tree should implement. The interface defines the following methods:
    - `int evaluate()` – evaluates the expression and returns its value.
    - `void draw(int x, int y)` – draws the expression subtree at position x/y.
    - `int getWidth()` – computes the total width of a node. Note that the width of a node must take child nodes into account (see Figure 2).
    - `int getCenter()` – determines the center coordinate of the expression. For a nice symmetric drawing, a node should always be placed centered above its child nodes (see Figure 2).
- `ExpressionFactory` is an interface, whose implementer is able to create instances of the concrete expression classes. It defines the following methods:
    - `Expression createConstant(int value)` – creates a new expression instance that represents a constant value.

- `Expression createBinary(char op, Expression left, Expression right)` – creates a new expression instance that represents the specified operation and has `left` and `right` as child expressions.

- `ExpressionParser` allows you to convert a text to an abstract syntax tree using an `ExpressionFactory` object. Look at the source code documentation for an example.

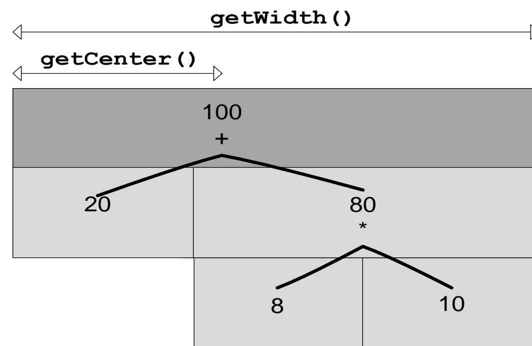

Figure 2: Geometric structure of the tree.

Based on those classes you should define a class hierarchy for the nodes of the abstract syntax tree. You should define concrete classes `ConstantExpression`, `AddExpression`, `SubExpression`, and `MulExpression`. For code reuse you should create an abstract base class for all expressions that take two arguments called `BinaryExpression`. Implement the methods of the interface `Expression` appropriately. Furthermore, define a class named `BaseExpressionFactory` that implements the interface `ExpressionFactory` and creates the instances of the expression classes.

Graphical Output:

For displaying the graph on the screen you should use the following methods of the class `Window` (for more information on those methods read the comments for them):

- `Window.getTextHeight()` – returns the text height on the screen (see Figure 3).
- `Window.getTextWidth(String)` - returns the width that a certain string needs on the screen (see Figure 3).

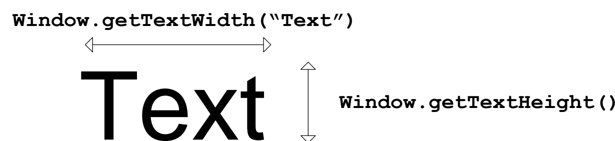

Figure 3: Obtaining the size of a string on the screen.

- `Window.drawLine(int startX, int startY, int endX, int endY)` - draws a black line connecting the points `(startX/startY)` and `(endX/endY)`.
- `Window.drawTextCentered(String text, int x, int y, int width)` – draws a string horizontally centered (see Figure 4).

`Window.drawTextCentered("Text", x, y)`

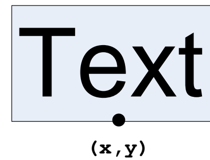

Figure 4: Drawing text horizontally centered.

Testing:

Write several test cases that check the evaluated result of the expression and display the expression tree. Include at least one screen shot in your submission.

Extended Factories (4 Points)

Compilers transform abstract syntax trees in order to optimize the execution speed. Your task is to create a modified factory that transforms the input expression according to certain rules.

No Substractions:

Imagine that the executor of the expression does not know how to do subtractions. Write a class `NoSubExpressionFactory` that extends the expression factory of the previous task. Make sure that you reuse the base factory as much as possible. This factory should convert every subtraction to an addition with the second expression multiplied by -1. Figure 5 shows an example expression and how it is transformed when this factory is used. The following formula describes the transformation:

```
a − b = a + (−1) * b
```

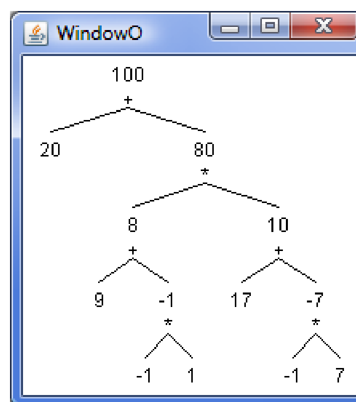

Figure 5: Expression tree generated from the input "20+(9-1)*(17-7)" without a `SubExpression`

Testing:

Verify your transformations by comparing the results of evaluating an expression between the two different expression factories. Write a method `check(String)` that takes an expression as the parameter and parses it two times using a different expression factory. Compare the results of evaluating the retrieved expressions and check whether they are equal.

BONUS (4 points)

No Multiplications:

Imagine that the executor of the expression does not know how to do multiplications. Write a class `NoMulExpressionFactory` that extends the expression factory of the previous task. This factory should convert every multiplication to a series of additions. Whenever the right operator of a multiplication is a complicated expression itself, it should be evaluated immediately. The following formula describes the transformation:

$$a * b = \underbrace{a + a + \ldots + a + a}_{b \text{ times}}$$

Here are two examples that show how the transformation should work; Figure 6 represents another example tree.

```
4 * (1 + 2) = 4 * 3 = 4 + 4 + 4
(1 + 2) * 4 = (1 + 2) + (1 + 2) + (1 + 2) + (1 + 2)
```

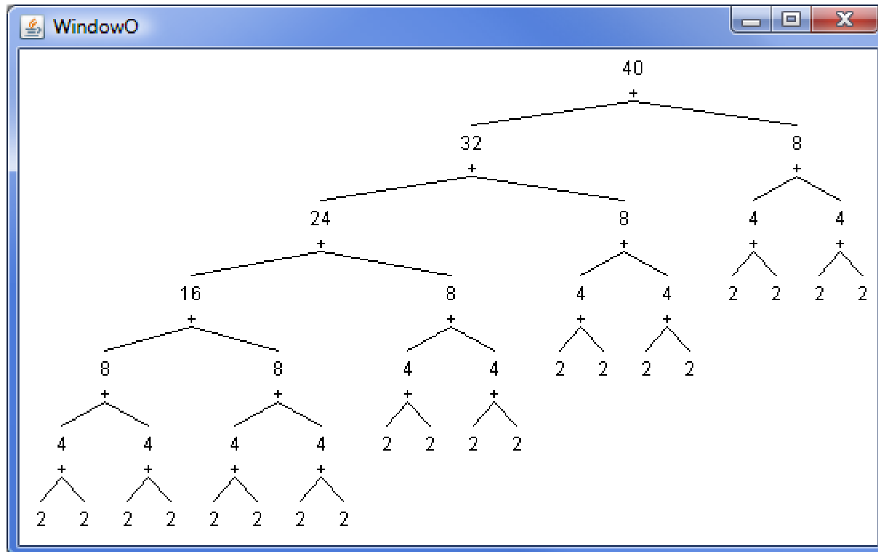

Figure 6: Expression tree generated from the input "2*2*2*5" without a `MulExpression`