Assignment 5: Composites

Related files:
Interval.java Intervals.java Expression.java

Goals: Practice designing, implementing and testing tree structures.

1 Instructions

Due Date: Monday October 28th, 9:00 pm.

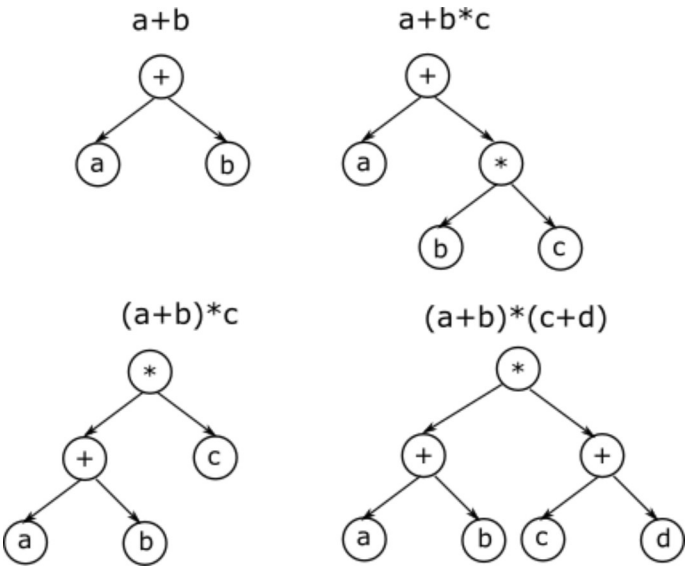
Trees

Many examples of data can be represented in a hierarchical fashion. In this description we will encounter two such examples.

1 Expression Trees

1.1 Introduction

An expression tree is a hierarchical representation for algebraic and logical expressions. Consider the algebraic expression $a + b$. This expression uses the binary operation + (binary means it requires two operands). This expression can be represented using a tree as follows: a + node with two children, a and b. The figure below shows some other examples:



If all the operators in an expression are binary, the expression tree is a binary tree.

Such a representation has many advantages. One can store the algebraic expression using variables as a tree and then repeatedly evaluate it for different values of the variables. It is also easy to change from one expression form to another (e.g. postfix, infix, etc.)

1.2 Expression forms

The postfix form of $a + b$ is $ab+$, while its prefix form is $+ab$. Various forms of the same expression have distinct advantages.

The infix form is "human-readable", because it corresponds to how we write expressions. However its mathematical evaluation is complicated. For example $a + b * c$ can be evaluated in two possible ways, depending on which operator is used first. To disambiguate this, we follow the PEMDAS rule (also known as BODMAS) in Math, which sets the precedence of operators (e.g. $a + b * c$ computes the multiplication before the addition). However we must resort to using parentheses to "override" the rules of precedence. For example we specify the expression as $(a + b) * (c + d)$ to perform the additions before the multiplications.

The postfix form is free of this complication. No operator precedence must be specified, and therefore parentheses are not required to override precedence. The order of the operations changes the expression itself ($a + b * c + d$ becomes $abc * +d+$, where $(a + b) * (c + d)$ becomes $ab + cd + *$). Postfix expressions are simpler to write and easier to algorithmically evaluate, but are not as human-readable.

The prefix form shares many of the features of the postfix form. However the prefix form is closer to both algebraic functions and programming syntax. For example adding two numbers a and b is specified in prefix form as $+ab$. If we treat addition as a function *add* with two arguments, then the prefix form above reads as `add a b` or $add(a, b)$. This is syntactically similar to how we call functions/methods in many programming languages.

1.2.1 Evaluating a postfix expression

A postfix expression can be evaluated using a stack-based algorithm as follows:

1. Consider the postfix expression as a sequence of tokens. A token is either an operand or an operator.
2. Scan the expression from left to right, one token at a time.
3. Read the next token.
4. If the token is an operand, push it into a stack.
5. If the token is an operator, pop the top two operands from the stack, apply this operator to them and push the result back into the stack. If there aren't two operands on the stack, report the expression as an error and exit.
6. If there are more tokens to read, go to step 3 else go to step 7.
7. The evaluation is done, the answer is the only thing in the stack.

1.2.2 Constructing an expression tree

The algorithm to construct an expression tree from an expression depends on the form of the expression provided to us. There are many similarities between constructing an expression tree and evaluating that expression (without its tree form). Therefore the algorithm to construct an expression tree is somewhat easier if the expression is provided to us in postfix notation.

Examine the above algorithm to evaluate a postfix expression to see how it can be modified to instead construct an expression tree from it.

1.3 What to do

All your code for the expression tree should be in the `expression` package. Your tests should still be in the default package.

You are provided an interface named `Expression` that defines the following operations:

- A method `infix` that takes no arguments and returns the expression as a string in infix form. The details of the format of this string is dependent on the implementation.
- A method `evaluate` that takes no arguments and returns the result of evaluating this expression as a `double`.

- **A method `textTree` that returns the tree structure as a formatted string. Please look at the documentation of this method in the provided interface for details.**
- A method `schemeExpression` that takes no arguments and returns the expression as a string as it would be typed in a popular functional programming language, Scheme. Please see details below.

Now implement this interface in a class `ExpressionTree`. This implementation should have the following features/obey these constraints:

- This class should represent the expression as a tree. You must implement the tree yourself.
- This class should have a constructor that takes an expression in postfix form, parses it and creates the expression tree accordingly. The string contains the expression with each term (operator and operand) separated by spaces. There may be leading and trailing spaces. Each valid operand is a numeral. The following examples should work with your constructor:
  - `"1 2 +"`
  - `"1.2 5.4 *  -4.5 + "`
  - `"3 2 + 65.12 -"`
  - `"3 5 + 4 -"`

  The constructor should throw an `IllegalArgumentException`-type exception if the expression provided to it is an invalid expression (i.e. violates one of the above rules).
- The `infix` method should return a string that surrounds each sub-expression with parentheses. In the four example trees above, the strings returned should be `( a + b )`, `( a + ( b * c ) )`, `( ( a + b ) * c )` and `( ( a + b ) * ( c + d ) )` respectively. Note that there is no leading or trailing whitespace.
- While you are free to write helper methods, you are not allowed to write any other public methods other than those in the interface and the above constructor.

1.3.1 Scheme Expressions

The Scheme programming language has a specific syntax for expression. In Scheme an expression starts with an operator and then the operands in that order, and is surrounded by parentheses. Here are a few examples:

- $1 - 3 * 2$ becomes `(- 1 (* 3 2))`
- $(a + b) * (c + d)$ becomes `(* (+ a b) (+ c d))`
- $1 - 2 + 3 - 4$ becomes `(- (+ (- 1 2) 3) 4)`

Note that there are no spaces after `(` and before `)`, but there are spaces between the operator and the operands.

You can verify your syntax by using this Scheme evaluator. As an added advantage, this evaluator can also evaluate the Scheme expression.

1.4 Tests

Write tests that thoroughly test this implementation. As always, it is recommended to write the test *before* completing the `ExpressionTree` implementation.

2 Interval operation trees

2.1 Introduction

A simple interval is represented as (start,end) where “start” is not greater than “end”. For example, an interval

may represent an appointment on a calendar, etc.

Similar to algebraic expressions, intervals can be operated with each other. Two simple operations on intervals are intersection and union. The intersection of two intervals is defined as the largest interval that overlaps with both intervals (if it exists). A union of two intervals is defined as the smallest interval that both intervals overlap with. Most of the operations that apply to arithmetic expressions above can also apply to these intervals.

2.2 What to do

All your code for the intervals should be in the `intervals` package. Your tests should still be in the default package.

An `Interval` class has been provided to you. You have also been provided an interface named `Intervals` that defines the following operations:

- A method `evaluate` that takes no arguments and returns an `Interval` object that is the result of evaluating an expression on intervals.
- **A method `textTree` that returns the tree structure as a formatted string. Please look at the documentation of this method in the provided interface for details.**

Now implement this interface in a class `IntervalTree`. This implementation should have the following features/obey these constraints:

- This class should represent an expression on intervals as a tree. You must implement the tree yourself.
- This class should have a constructor that takes an intervals expression in postfix form, parses it and creates the expression tree accordingly. The string contains the expression with each term (operator and operand) separated by spaces. There may be leading and trailing spaces. Each valid operand is a string that contains two integral numeric values separated by a comma. An operator can be "I" or "U" representing intersection and union respectively. The following examples should work with your constructor:
  - `"1,4 2,5 U"`
  - `"-4,4 2,5 U  -1,4 I "`
  - `"3,7 2,6 4,10 I U"`
  - `"3,10 5,12 U 4,4 I"`

  The constructor should throw an `IllegalArgumentException`-type exception if the expression provided to it is an invalid expression (i.e. violates one of the above rules).
- The `evaluate` method should return the result as an `Interval` object. Note that according to the provided `Interval` class, all operations on intervals always produce an object.
- While you are free to write helper methods, you are not allowed to write any other public methods other than those in the interface and the above constructor.

3 Abstraction

While the above two trees are very different and support different operations, there are common aspects between them. Can you detect them, and suitably abstract to minimize code duplication? Will the abstraction be useful in other contexts?

Try to find opportunities and abstract accordingly. Ensure that each class and interface you create has a legitimate purpose (as evidenced by meaningful documentation). It is recommended that you first implement both trees fully and then retrospectively abstract (although proactively abstracting is also OK if you can design it that way).

Criteria for grading

You will be graded on:

1. The design of your interfaces.
2. The design of your trees (nodes, their contents, how each operation is implemented, etc.)
3. Whether your implementations obeys all the above constraints.
4. The correctness of your methods.
5. How well you have tested your methods.
6. Whether you have written enough comments for your classes and methods, and whether they are in proper Javadoc style.
7. Whether you have used access modifiers correctly: `public`, `protected` or `private`.
8. Whether your code is formatted correctly (according to the style grader).

How to submit

1. Ensure that all your source files are in *src*, your tests are in *test* and any additional files for submission are in *res* folders.
2. Create a zip file that contains directly your *src*, *test* and *res* folders. When you unzip the file, you should see only these three folders.
3. Log on to the Bottlenose submission server.
4. Navigate to *Assignment 5* and submit the zip file.
5. Wait for a few minutes for the style grader to appear, and take action if needed.