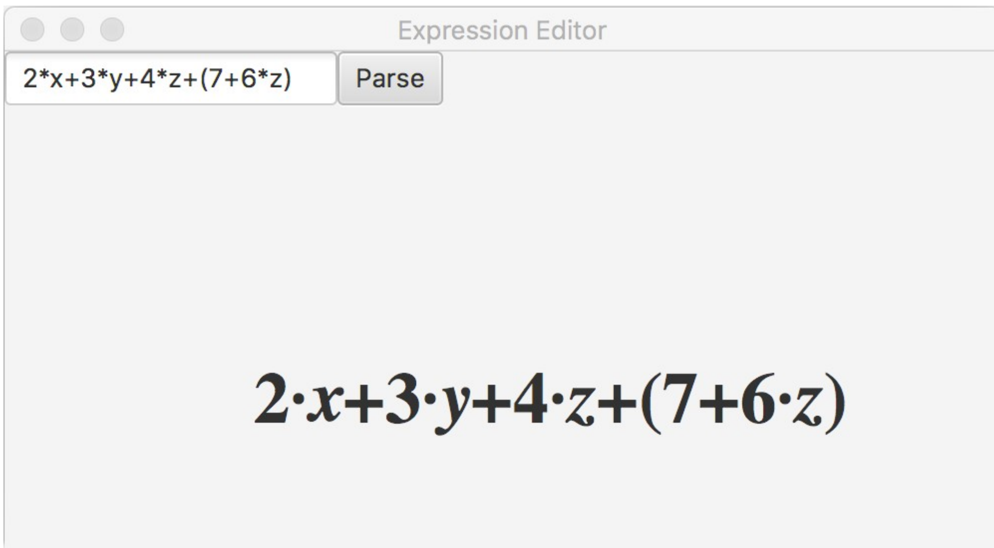
CS2103 2018 B-Term -- Project 5 -- Mathematical Expression Editor

Prof. Jacob Whitehill

Introduction



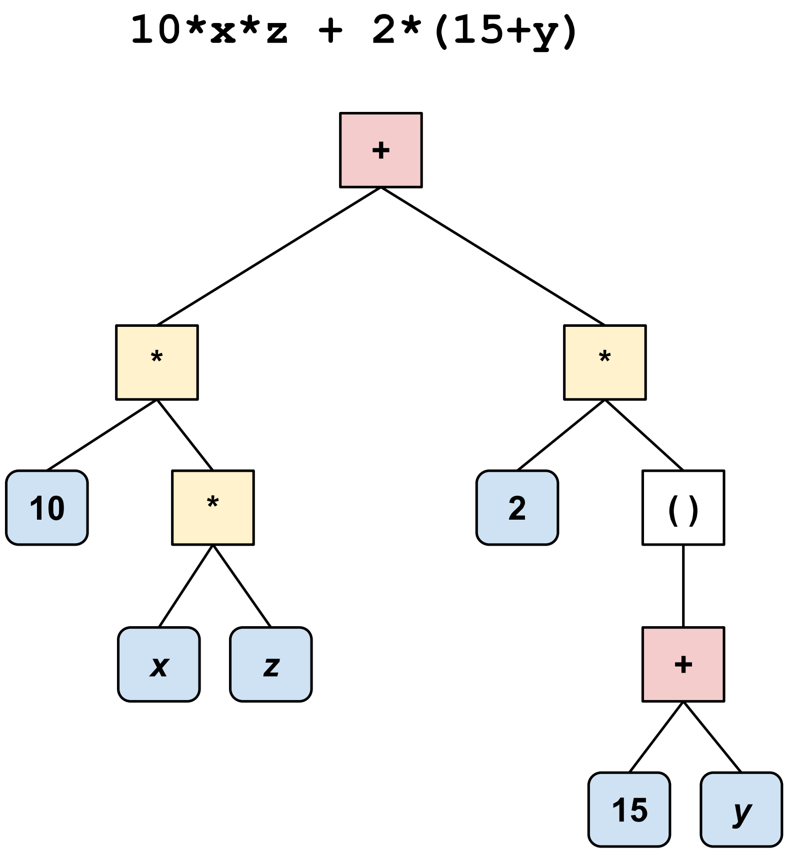
In this project you will build an interactive (event-driven) mathematical expression editor with a graphical user interface (GUI). In particular, the tool you build will allow the user to type in a mathematical expression, which will then be parsed into an "expression tree" and then displayed graphically. Then, the user will be able to drag-and-drop different subexpressions -- at **arbitrary levels of granularity** -- so as to rearrange the expression **while preserving the same mathematical semantics**.

Example

Suppose the user is editing the expression `2*x + 3*y + 4*z + (7+6*z)`. Because of commutativity of addition, this expression can be rearranged into `3*y + 2*x + 4*z + (7+6*z)` *without changing its meaning*. Similarly, because of commutativity of multiplication, we can also change it into `y*3 + 2*x + z*4 + (7+z*6)`. In contrast, if we were to (nonsensically) reorder the substring `y + 2` (in `3*y + 2*x`) to be `2 + y`, then this would yield `3*2 + y*x + 4*z + (7+6*z)`, which clearly has a different mathematical meaning from the original expression.

R1: Parsing Mathematical Expressions

In the first part of the assignment, you will need to build a **recursive descent parser**, based on a **context-free grammar** (CFG), to convert a string -- e.g., `10*x*z + 2*(15+y)` -- into a **parse tree** that captures the expression's mathematical meaning, e.g.:



In the figure above, each blue node is a `LiteralExpression`; each yellow node is a `MultiplicativeExpression`, each red node is an `AdditiveExpression`, and the clear node is a `ParentheticalExpression`. Obviously, these nodes are arranged into a *tree*. Every kind of expression *except* a `LiteralExpression` can have children. Moreover, the different "expression" classes belong to a *class hierarchy* to maximize code re-use.

CFG for Mathematical Expressions

We will discuss context-free grammars (CFGs) in class. There are many possible grammars you could use to complete this project. One suggestion (which I used in my own implementation) is the following:

- `E → A | X`
- `A → A+M | M`
- `M → M*M | X`
- `X → (E) | L`
- `L → [a-z] | [0-9]+`

The `[0-9]+` means *one or more characters from the set [0-9]* -- e.g., `1`, `51`, `5132762351`, etc.
(Note: the grammar above does not lend itself to a particularly efficient parser, but it is arguably easier to understand than other grammars that could also work.)

Based on this CFG, you can implement a recursive descent parser, in a similar manner as described in class. However, in contrast to the example in class in which each "parse" method returned a `boolean`, your parse methods should return an object of type `Expression` (an interface type described below). Each parse method should either return an `Expression` object representing the sub-tree for the string that you are parsing, or `null` if the string passed to the parse method cannot be parsed.

In this assignment, you should create a class called `SimpleExpressionParser` that implements the `ExpressionParser` interface.

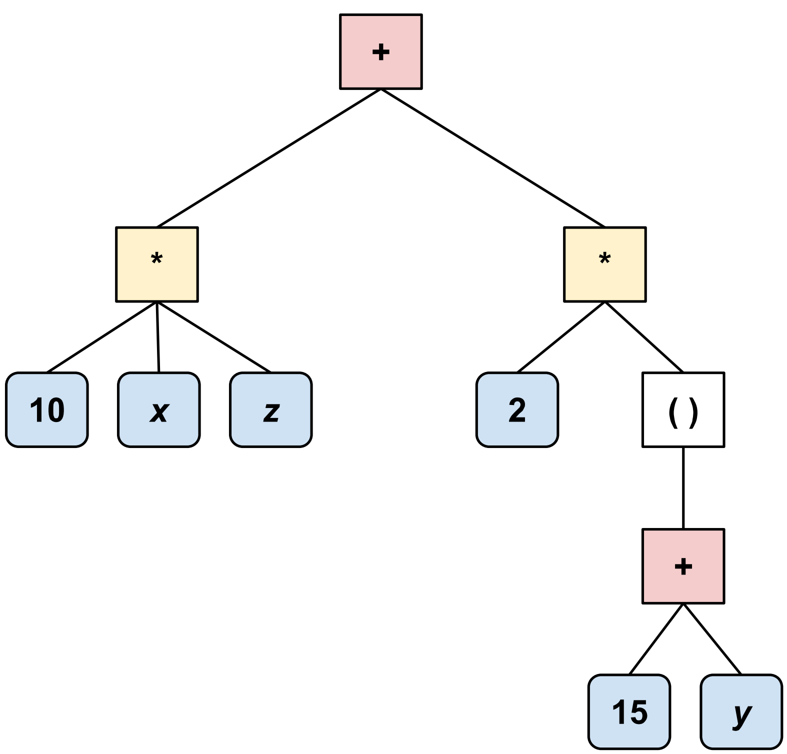
`Expression` and `CompoundExpression` interfaces

This assignment will require multiple classes that represent different kinds of mathematical expressions. Every expression has some methods that must be supported, however. Accordingly, we have defined an `Expression` interface. For non-terminal expression nodes, we have also created a `CompoundExpression` interface, which extends `Expression` and includes one extra method `addSubexpression(subexpression)`. See the comments in the `Expression.java` and `CompoundExpression.java` files for more details. **Important note**: these interfaces will be expanded upon during R2 of Project 5.

Equivalent Parse Trees

Multiple parse trees can be generated that have the same mathematical meaning. Consider the following example:

`10*x*z + 2*(15+y)`



This is a different parse tree than the one shown above, but it encodes the same sequence of mathematical operations. The only differences are that (1) the `MultiplicationExpression` on the left now has three children, whereas before it had only two; and (2) the children and grandchildren have been "merged" into one layer. This equivalency is reminiscent of the fact that `10*x*z` is completely equivalent to `10*(x*z)`: In the first parse tree, the multiplication of `x` by `z` is computed first, and then its product is multiplied by `10`. Due to the commutativity of multiplication, however, both parse trees are equivalent.

Important note: While you could apply the same logic to claim (correctly) that the expression `10*x*z` can be rearranged into `z*x*10`, you should not do so in this project. In particular, **your parser should preserve the**

left-to-right ordering of the sub-expressions in the parse tree. The reason is that, in the GUI of R2, when the user types in `10*x*z`, we want them to *see* `10*x*z` on the screen -- not some arbitrary rearrangement thereof (which would be counterintuitive for them).

`withJavaFXControls`

The `parse` method of every `ExpressionParser` class takes a parameter called `withJavaFXControls`. For R1, the value of this parameter should always be `false`, and you can thus ignore it altogether. This parameter will become more important for R2, whose GUI requires that every `Expression` object also be associated with a JavaFX "node". We'll talk more about this in class.

Converting an expression to a string

In order to verify that your parser is working correctly -- and to enable grading of your R1 submission -- you need to implement a `convertToString` method (see the `Expression` interface). This method should print out the contents of the entire expression tree, using one line per node of the tree, such that each child node is indented (using `\t`) one more time than its parent. For example, if we parse the string "10*x*z + 2*(15+y)":

```
final ExpressionParser parser = new SimpleExpressionParser();
final Expression expression = parser.parse("10*x*z + 2*(15+y)", false);
```

and then we call `expression.convertToString(0)`, then the result should be:

```
+
	*
		10
		x
		z
	*
		2
		()
			+
				15
				y
```

In the output above, the `+` in the first line signifies that the root expression is an `AdditiveExpression` (that's what I call an expression that performs addition in my own implementation). Its two children are both `MultiplicativeExpression` objects. The first such `MultiplicativeExpression` itself has *three* children, namely `10`, `x`, and `z`, and so on.

Flattening the Parse Tree

It turns out that, for the purpose of implementing a GUI-based mathematical expression editor, the second parse tree shown above is more useful. The reason is that it will facilitate intuitive drag-and-drop behavior whereby the user can "move" (using drag-and-drop) each child expression to be anywhere among the list of its siblings. (We will discuss this in more detail in class.) Therefore, we require that every class that implements the `Expression` interface must have a method called `flatten` that modifies the target `Expression` in the following way: whenever a child *c* has a type (`AdditiveExpression`, `MultiplicativeExpression`, etc.) that is the same as the type of its parent *p*, you should replace *c* with its *own* children, which thereby become children of *p*. For example, by flattening the first parse tree shown above, we obtain the second parse tree. The `flatten` method should recursively flatten the entire `Expression` tree as much as possible. Note that you are only required to flatten the `AdditiveExpression` and `MultiplicativeExpression` objects.

Requirements

1. R1 (50 points): Build a parser to convert a `String` into an `Expression`. Your parser must be able to handle the operations of **addition** (`+`) and **multiplication** (`*`). It must also be able to handle **arbitrarily deeply nested balanced parentheses** (e.g., `((2+(((z)))+3)))`). Note, however, that you do **not** have to handle subtraction or division.

Design and Style

Your code must adhere to reasonable Java style. In particular, please adhere to the following guidelines:

- **Factor out** the logic that is common to the various `Expression` classes.
- Each class name should be a singular noun that can be easily pluralized.
- Class names should be in `CamelCase`; variables should be in `mixedCase`.
- Avoid "magic numbers" in your code (e.g., `for (int i = 0; i < 999 /*magic number*/; i++)`). Instead, use **constants**, e.g., `private static final int NUM_ELEPHANTS_IN_THE_ROOM = 999;`, defined at the top of your class file.
- Use whitespace consistently.
- No method should exceed 50 lines of code (for a "reasonable" maximum line length, e.g., 100 characters). If your method is larger than that, it's probably a sign it should be decomposed into a few helper methods.
- Use comments to explain non-trivial aspects of code.
- Use a Javadoc comment to explain what each method does, what parameters it takes, and what it returns. Use the `/**...*/` syntax along with `@param` and `@return` tags, as appropriate.
- Use the `final` keyword whenever possible.
- Use the **most restrictive** access modifiers (e.g., `private`, default, `protected`>, `public`), for both variables and methods, that you can. Note that this does not mean you can never use non-`private` access; it just means you should have a good reason for doing so.
- Declare variables using the **weakest type** (e.g., an interface rather than a specific class implementation) you can; ithen instantiate new objects according to the actual class you need. This will help to ensure **maximum flexibility** of your code. For example, instead of
  ```
  final ArrayList<String> list = new ArrayList();
  ```
  use
  ```
  final List<String> list = new ArrayList<String>();
  ```
  If, on the other hand, you have a good reason for using the actual type of the object you instantiate (e.g., you need to access specific methods of `ArrayList` that are not part of the `List` interface), then it's fine to declare the variable with a stronger type.

Teamwork

You may work as a team on this project; the maximum team size is 2.

Getting started

R1

1. Please download the R1 starter file.
2. Have a look at the `ExpressionParserPartialTester.java` file, which includes some -- but not all -- of the test cases with which we will test your expression parser.
3. Write the parse methods necessary to implement the `SimpleExpressionParser` that we will test as part of R1.

How, What, and When to Submit

R1

- Create a Zip file containing only and all those files necessary to test your parser using `ExpressionParserPartialTester.java`.
- **Submission deadline for R1**: Wednesday, Dec 5, at 11:59pm EDT.