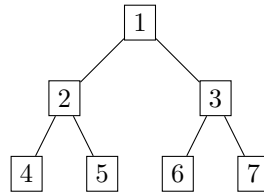


CSCI 235 Parse Trees

The first part of tomorrow’s lab will be spent completing the expression-tree program that most of you did not get to on Friday. This handout describes what you will need to do; so you can think through the problems and have something like pseudocode ready when you come to lab.

Review

We have worked some with lists, and in class we introduced another kind of linked structure: the *tree*. In a tree, a node can have multiple *children*; the case of at most two children is called a *binary* tree. If we draw a picture with each node above its children, we get something like this:



We call the node at the top, which has no parent, the *root*, and the nodes at the bottom, that have no children, are called *leaves*. (Yes, computer scientists grow trees upside down.)

One way to make trees is for each node to have two child references, and to use a `null` reference to indicate that a particular child is not present. The sample code that we looked at in class used this approach.

Parsing

One use for trees is in the grammatical (or syntactical) analysis (or “parsing”) of languages—both human languages and programming languages. (You might recall that we have encountered “parse” in the name of the library method `Integer.parseInt()`, which converts a string representing an integer into an `int` value.)

We can write a description of a language as a *grammar*; in the notation below, the \rightarrow means “consists of” and the $|$ separates alternatives. So we can specify a language of fully parenthesized arithmetic expressions with this grammar:

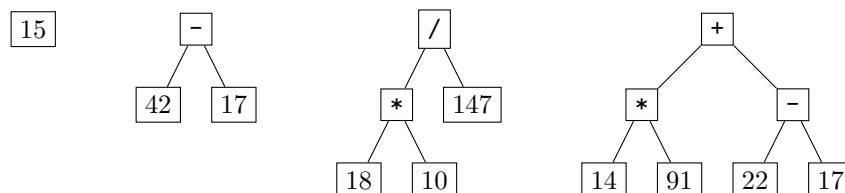
$$\begin{aligned} \text{expression} &\rightarrow \text{integer} \\ &\quad | (\text{expression} \text{ op } \text{expression}) \\ \text{op} &\rightarrow + \mid - \mid / \mid * \end{aligned}$$

The first line tells us that an expression can be a single integer; the second says that an expression can also consist of a left parenthesis, an expression, an operator, another expression, and a right parenthesis. The final line says that an operator can be one of the four symbols listed.

According to this grammar, the following are expressions:

- 15
- (42 - 17)
- ((18 * 10) / 147)
- ((14 * 91) + (22 - 17))

These can be represented as trees. (We can ignore the parentheses now, since the information they give about grouping is contained in the structure of the tree itself.)



We can represent an expression tree in Java using an interface to represent *any* expression, and two classes to represent the two alternatives—the two lines in our grammar rule:

$$\begin{array}{l} \textit{expression} \rightarrow \textit{integer} \\ \quad \quad \quad | (\textit{expression} \textit{op} \textit{expression}) \end{array}$$

So we will have the interface `ExprNode` corresponding to the left side (*expression*). The class for the first line corresponds to a leaf of the tree; it needs to store the integer value. The second line is more complicated; its class needs to store the operator *op* plus two child expressions, its left operand and its right operand. Note that either child can be any expression, but they will always be present. This interior node does not need to store the parentheses, because they are implied by the structure.

Our `ExprNode` interface specifies whatever operations we want to be able to do on expressions. The obvious thing to do is to *evaluate* them. That gets a definition something like:

```
public interface ExprNode {
    public int evaluate();
}
```

The leaves of our trees represent constants; so the skeleton for that could be

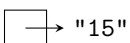
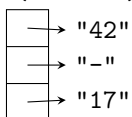
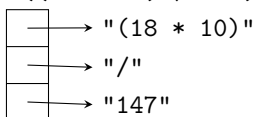
```
public class ConstantNode implements ExprNode {
    private int value;
    public ConstantNode(String text) {
        ...
    }
    public int evaluate() { return value; }
}
```

Each interior node represents an operator and its operands; which might look like

```
public class OperatorNode implements ExprNode {
    ...
    public OperatorNode(String text, ExprNode leftOperand, ExprNode rightOperand) {
        ...
    }
    public int evaluate() {
        ...
    }
}
```

The static method `parse()` in class `Interpreter` has the job of taking a string and converting it into the corresponding tree. The first problem is figuring out which kind of node the string should turn into.

You have been provided with a method `ExprStringSlicer.slice()` that turns a single string (the input) into an array of strings—and does so in a helpful way. If the input string represents a constant, the result is an array of length one, and the string in the array is the value (as a string). A parenthesized expression gets turned into an array with three elements, corresponding to the left operand, the operator, and the right operand. So the length of the result array tells us which line of the grammar applies. Here are some examples:

input	"15"	"(42 - 17)"	"((18 * 10) / 147)"
result			

So `parse()` needs to call `slice()` on its input string. Based on length of the array returned, it either constructs a `ConstantNode` or an `OperatorNode`. In the latter case, it will need to recurse to parse the operand expressions.

It is a precondition of `parse()` that the string passed to it be a legitimate expression. That means that your implementation of this is permitted to throw an exception if the expression is malformed. The provided `slice()` may throw an exception (most likely to be a `StringIndexOutOfBoundsException`) if the string is illegal; you don't need to catch that. It is also reasonable for one of the constructors you write to throw an exception (such as `NumberFormatException`) if the string given to it is bad.

Evaluating

The driver program is very simple. It first calls `parse()` to make a tree, then it calls `evaluate()` on the tree's root to get the value of the expression so that it can print it.

You will need to fill in the `evaluate()` method on each of your classes. If your `ConstantNode` stores the `int` value, that method is trivial. The `OperatorNode` is more complicated: it needs to evaluate both of the operand expressions, then it needs to combine those results appropriately for the operator string it was given.

The Driver

Here is a stripped down listing of the driver class `Interpreter`:

```
public class Interpreter {

    public static void main(String[] args) {

        ExprNode expr = parse(args[0]);

        System.out.println(expr.evaluate());
    }

    /**
     * Parse an expression.
     * @param expr The string to parse.
     * @return The root of a tree corresponding to the expression.
     * PRECONDITION: expr is a properly formed expression
     */
    public static ExprNode parse(String expr) {

        //... you need to write this
    }
}
```