Programming assignment #5: CSE 12 Spring 2013

Here's a description of the fifth programming assignment for CSE 12, Spring 2013. In this assignment, you will create linked trees with dynamically created nodes, and write recursive methods that will parse and evaluate expressions defined in terms of a BNF grammar. The result will be useful as a key component in a command-line calculator application.

>>>Due Fri Jun 07 5:00 PM

>>> Required solution files: Expr.java, Assmt.java, Oprn.java, Term.java, Factor.java, Const.java, Ident.java

This means that the files that contain your solution to the assignment must be turned in by the due deadline by using the bundleP5 script while logged into your cs12 account on ieng6..

It's a good idea to plan to do the assignment well before the deadlines; terminals and tutors can get very busy at the last minute. As always, keep in mind that in doing this assignment, it's your responsibility to understand and follow the course rules for integrity of scholarship.

Getting started

Review Chapter 8 and Chapter 10 in the text book, and corresponding lecture notes. This assignment uses material introduced and discussed there. Create a directory named "P5" under your cs12 home directory. Your solution to this assignment should be turned in from that directory by the due date and time. Use the bundleP5 script to do that.

This assignment involves defining classes with methods that parse and evaluate expressions written in a simple expression language. Parsing means checking that the expression conforms to the syntax rules for the language, and, if it does, constructing an abstract syntax tree (AST) representing the syntactic structure of the expression. (For a little more information about these concepts, see the Free Online Dictionary of Computing entries on parsing and abstract syntax.) After the AST is constructed, it will be recursively evaluated according to the semantics of the expression language, in order to compute the value of the expression, and to perform any required 'side effects'.

The syntax rules for the expression language are given below as a Backus-Naur Form (BNF) grammar. These are basically a subset of the rules for Java and C expression syntax. The semantic rules for the language are not given in a formal notation, but they follow closely the form of the BNF grammar. The BNF definitions are recursive, and the definitions of your parsing and evaluation methods will reflect this by being recursive themselves.

Copy the file `ASTNode.java` from the `~/../public/P5/` directory to your P5 directory. This defines the abstract class that you will be extending in this assignment.

The expression language: syntax and semantics

Here is the BNF grammar defining the syntax of the expression language. The start symbol for the grammar is `<expr>`. The symbol "`:=`" means "is defined as"; the symbol "`|`" separates alternative definitions; nonterminal symbols of the language are the ones that appear in corner brackets `<>`; the terminal symbols are the literal characters "`=`", "`+`", "`-`", "`*`", "`/`", "`(`", and "`)`". The nonterminal symbols `<const>` and `<ident>` are defined outside the grammar.

```
<expr> := <assmt> | <oprn>
<oprn> := <term> | <oprn> "+" <term> | <oprn> "-" <term>
<term> := <factor> | <term> "*" <factor> | <term> "/" <factor>
<factor> := <const> | <ident> | "(" <expr> ")"
<assmt> := <ident> "=" <expr>
```

In English:

- An expression is an assignment or an operation.
- An operation is a term; or an operation followed by "+" followed by a term; or an operation followed by "-" followed by a term.
- A term is a factor; or a term followed by "*" followed by a factor; or a term followed by "/" followed by a factor.
- A factor is a constant; or an identifier; or "(" followed by an expression followed by ")".
- An assignment is an identifier followed by "=" followed by an expression.

Definitions of the nonterminal symbols `<const>` and `<ident>` are as follows:

- A constant is any String `s` that can be parsed by `java.lang.Double.parseDouble(s)` without throwing a NumberFormatException
- An identifier is any String `s` that starts with a JavaIdentifierStart character, followed by 0 or more JavaIdentifierPart characters. (Note: Java's `Character` class static methods `isJavaIdentifierStart()` and `isJavaIdentifierPart()` provide a way to check that a character meets these conditions.)

Here are examples of some strings that are in the language defined by that grammar:

```
3
2.0 + 2
1+2*3-4/5
(1+2)*(3-4)/5
1 / 10 - -3.0
(((((-1e3)))))
x = y = z = 0
2 + (a = 2) + y
area = pi * r * r
```

Here are some strings that are not in the language defined by that grammar:

```
3+
1+2*3-4//5
1 / 10 - - -3.0
(((((-1e3))))
3 = 3
2 + a = 2
```

The semantic rules for determining the value of an expression are also familiar from C and Java. The semantic rules show a close correspondence with the syntactic rules:

- Depending on the form of an expression, the value of the expression is the value of the assignment, or the value of the operation.
- Depending on the form of an operation, the value of the operation is the value of the term; or the value of the operation plus the value of the term; or the value of the operation minus the value of the term.
- Depending on the form of a term, the value of the term is the value of the factor; or the value of the term times the value of the factor; or the value of the term divided by the value of the factor.
- Depending on the form of a factor, the value of the factor is the value of the constant, or the value of the identifier, or the value of the expression in parentheses.
- The value of a constant is the value of the constant as a Java double.
- The value of an identifier is the current (most recently assigned) value of the variable named by the identifier. If that variable has not yet been assigned a value, evaluating it must cause a RuntimeException to be thrown, containing a message indicating the identifier. All variables are of type `double` (there are no declaration statements in the language).
- The value of an assignment is the value of the expression on the right-hand-side of the = sign. Evaluating an assignment also has the side effect of assigning that value to the variable named by the identifier on the left-hand-side. This side effect persists over subsequent expression evaluations, until it is changed by another assignment to that variable.

These syntactic and semantic rules imply the usual associativity and precedence rules for the operators: The assignment operator "=" associates right to left; the other operators associate left to right. "*" and "/" have equal precedence, as do "+" and "-", but "*" and "/" have higher precedence than "+" and "-", which is in turn higher than "=". Parentheses can be used to change the order of applicaton of operators in an expression, in the usual way. Here are the values of some expressions when evaluated in this order:

```
3 has value 3.0
2.0 + 2 has value 4.0
1+2*3-4/5 has value 6.2
(1+2)*(3-4)/5 has value -0.6
1 / 10 --3.0 has value 3.1
(((((-1e3))))) has value -1000.0
x = y = z = 0 has value 0.0
2 + (a = 2) + y has value 4.0
area = pi * r * r throws RuntimeException with message: UNINITIALIZED VARIABLE: pi
```

Parsing the expression language

Take an object-oriented approach to solving the problem of parsing Strings. Consider the abstract class `ASTNode`, defined in the ASTNode.java file, and its Javadoc documentation. The result of parsing a String will be an abstract syntax tree (or null, if the String cannot be parsed) consisting of `ASTNode` objects. Each node in the AST corresponds to a nonterminal symbol in the grammar; since each of these nonterminals is defined differently, there should be different kind of nodes -- different subclasses of `ASTNode` -- for the different nonterminal symbols. In fact, a class that extends `ASTNode` will be defined for each of the nonterminal symbols in the expression language grammar. These classes will be named `Expr`, `Oprn`, `Assmt`, `Term`, `Factor`, `Ident`, and `Const`; see the Javadoc documentation for more information.

As stated in the documentation for `ASTNode`, every class `X` extending `ASTNode` interface must provide a static factory method with this signature:

```
public static X parse(java.lang.String s)
```

This is a factory method for the class `X`. (For more information on the Factory Design Pattern, see this Wikipedia entry.) It is supposed to 'know how' to parse the argument String according to the definintion of the nonterminal symbol corresponding to the `ASTNode` subclass `X`. If the String can be parsed, this method creates and returns an instance of the class `X` as the root of an AST (sub)tree that represents the parse. If the String can't be parsed, this method returns `null`.

So, for example, the class `Oprn` must define a method with signature:

```
public static Oprn parse(java.lang.String s)
```

Refer to the BNF definition of <oprn> above. Following that definition, the parse() method in the `Oprn` class can be defined along these lines:

1. If `s` can be parsed as a <term>, create and return an `Oprn` object that has an appropriate `Term` object as its only child
2. Otherwise, if `s` can be parsed as an <oprn> followed by the character "+", follwed by a <term>, create and return an `Oprn` object that has an appropriate `Oprn` object as its first child, an appropriate `Term` object as its second child, and an instance variable indicating that the operator is the addition operator.
3. Otherwise, if `s` can be parsed as an <oprn> followed by the character "-", follwed by a <term>, create and return an `Oprn` object that has an appropriate `Oprn` object as its first child, an appropriate `Term` object as its second child, and an instance variable indicating that the operator is the subtraction operator.
4. Otherwise, return `null`.

How can this method determine that a substring of the String `s` can be parsed as a <term>, and get an appropriate instance of the `Term` class? By calling `Term.parse(s)`. Perhaps more interestingly, how can `Oprn`'s parse() method determine that a substring of the String `s` can be parsed as a <oprn>, and get an appropriate instance of the `Oprn` class? The answer is, of course, by recursively calling itself. See the "Hints and other remarks" section below for more thoughts on defining your `parse` methods.

Evaluating the AST

Since the start symbol for the grammar is `<expr>`, the root of the overall AST constructed by parsing an expression will be an instance of the class `Expr`. The existence of the AST shows that the String is a legal expression according to the grammar, and it also represents the syntactic structure of the expression. However, we want to do more than that: we want to evaluate the expression.

Expression evaluation is implemented in terms of the `eval()` instance methods of the concrete ASTNode classes. Calling the `eval()` instance method on a node in the AST should enforce the semantic rule for that kind of node. This is a recursive process in general; the `eval()` method for a node will call the `eval()` methods of its children, perform any required operation on the values it gets back, and return the result. The situation is simpler for a node with just one child and no operator; it just returns the value it gets from evaluating its child. And the situation is simpler yet for leaf nodes (in this assignment, these are instances of `Const` or `Ident`); they don't need to call `eval()` on any other nodes at all.

Note that calling the `eval()` instance method on the root of an AST essentially performs a postorder traversal of the tree, in the sense that child nodes are first recursively traversed in postorder, from left to right (to evaluate them), and then the root is visited (to apply its operator to the result of evaluating its children). The `eval()` method is declared to take as argument an object that implements the `Map<String,Double>` interface (for example a `java.util.HashMap<String,Double>`), with String keys and Double values. This object is a symbol table that keeps track of the values that have been assigned to variables. Initially, the symbol table is empty. Evaluating an `Assmt` will put a String key (a variable name) associated with a Double value (the variable's value) in the table; evaluating an `Ident` will get a Double value associated with a String (the identifier name) from the table, or throw a RuntimeException if none exists. No other `ASTNode` types actually use the symbol table themselves, but it must be passed to `eval()` in recursive calls so it can be used when it is needed.

Hints and other remarks

1. For this assignment, you are not required to turn in a JUnit test suite for your classes. However, it would be a good idea to create one for your own use, to thoroughly test your classes as you are developing them.
2. Work on a piece of your solution at a time, test that piece, and get it working before moving on to other parts. You could start by getting `Const.parse()` right, and then work on `Const`'s `eval()` method. The other classes' `parse()` and `eval()` methods can be defined simply to start with, and refined later. For example, in your `Expr` class, you initially might have something like

   ```
   private Expr(ASTNode n) { super(n); }

   public static Expr parse(String s) {
     ASTNode result = Oprn.parse(s);
     if(result != null) return new Expr(result);

     return null;
   }

   public double eval(Map<String,Double> symtab) {
     return getChild(0).eval(symtab);
   }
   ```

   With similarly simple implementations of `Oprn`, `Term`, and `Factor`, all your classes will compile, and a now a simple `<const>` can be parsed and evaluated as an `<expr>`. Then go on and implement more details of your classes' `parse` and `eval` methods, breaking them down into smaller pieces, and working on them incrementally. Test each piece as you refine it.
3. The easiest way to do this assignment is to pay close attention to the structure of the BNF definitions given, and have the definitions of your methods closely follow that structure. Think recursively! If there is a method that solves a subproblem, call it to solve that subproblem. The result will be a recursive descent parser for the language.
4. Since any amount of whitespace is allowed in an expression anywhere except within a `<const>` or an `<ident>`, you may find the instance method `trim()` in the String class useful.

   You will certainly find String's `substring` methods useful, perhaps in conjunction with `charAt`, `indexOf` or `lastIndexOf`. For example, the following string is a `<oprn>`:

   ```
   "8 + 376 - 0.0 + 12 - 3.14159"
   ```

because `"3.14159"` is a `<term>` and `"8 + 376 - 0.0 + 12"` is a `<oprn>`. However, `Oprn.parse()` may have to try all the possibilities to find the location of the operator `+` or `-` that "splits" the string into substrings corresponding to these parts. Since the grammar we have defined is an unambiguous one, there will be at most one occurrence of an operator that is the "right" one to split the string on; but it is not always the last occurrence or the first one:

```
"(8 + 376) - 0.0 + (12 - 3.14159)"
```

is a `<oprn>`, but `"3.14159)"` is not a `<term>`, nor is `"376) - 0.0 + (12 - 3.14159)"`.

5. `~/../public/P5/Calculator.java` defines an application that is a useful command-line calculator, once it has your `ASTNode` classes to use. `~/../public/P5/Calculator.jar` is a correctly functioning reference solution. Don't use these programs for all of your testing; they don't test anything about the structure of the AST you build, just the result of evaluating the root of the AST you construct. But they do demonstrate how it is possible to write a useful program quite elegantly, using the principles of object-oriented design and recursive problem decomposition.

Grading

There are 25 possible points on this assignment, not including early-turnin points. We will test your solution as follows:

- We will copy your turned-in files into an otherwise empty directory.
- Then we will copy in the `ASTNode.java` file from the `public/P5` directory.
- We will then compile your solution with the commands:

```
javac Const.java
javac Ident.java
javac Factor.java
javac Term.java
javac Oprn.java
javac Assmt.java
javac Expr.java
```

  (If your solution files do not compile, you will receive 0 points for this assignment.)
- We will test your `Const`,`Factor`,`Term`, and `Expr` classes to make sure that they implement correctly the requirements of this assignment.

If you have any questions about what constitutes good commenting or coding style, see a tutor or TA. In particular: Make sure that you include javadoc comments for each public method, and that these document any relevant preconditions, postconditions, arguments, return values, and side effects. Make sure that you include javadoc comments for each public class.

In addition, each function body should, in general, contain some comments explaining the basic flow of control and the logic of the code. Don't overcomment; put them where they are useful for understanding.

Your code should be indented in a consistent and readable way. Indent 2 to 4 spaces at each level. Use whitespace in your code to increase readability. Try to keep each line of code less than 80 characters long.

Good luck!