Visitors with Expression Trees

In the last lab you implemented code to construct expression trees as well as visit all the nodes in such a tree as a form of *unparsing*. There are many reason why we might want to visit all the nodes in an expression tree. For example, we might be interested in evaluating the expression, converting the expression to prefix notation, or analyzing the structure of the tree-based expression. In this lab we will implement expression tree traversals using a design pattern known as a *visitor*.

Visitor Design Pattern

In the last lab you implemented your traversal technique by adding an `unparse` method to almost all of the classes in the AST hierarchy. In that case, adding a method to each class was not necessarily onerous. However, if we were to implement many other traversal algorithms over an expression tree, our AST classes and the associated ASTnode file would become cluttered (to say the least). Our goal is to have one file contain all code related to one traversal technique. For example, if we wish to evaluate an expression tree, we would want to place all code related to that idea in a dedicated `Evaluator` class.

To avoid clutter and (more importantly) to maintain separation of functionality for each type of traversal, we will use a visitor. The *visitor design pattern* is a well-established paradigm to accomplish the goal of separating functionality while also providing a clear, consistent interface to traverse the tree. We will describe our implementation below, but it is beneficial to read a more general description of this double dispatching technique here.

Visitor with Expression Trees

In order to understand the implementation of the visitor pattern for expression trees, first observe the changes we need to make in the `ASTnode` class. The following code should be familiar from the last lab.

```
public abstract class ASTnode
{
	ASTnode() { }

	boolean isNull(){ return false; }

	public abstract void unparse(StringBuilder sb);
}
```

Previously, we implemented an `unparsing` method in each of our classes in order to acquire an understanding of the representation of an expression tree. However, with the *visitor design pattern* the top-level AST node requires only an `accept` method (as the first part of double dispatch).

```
public abstract class ASTnode
{
    ASTnode() { }

    // Visitor design pattern: will be defined in sub-classes
    abstract void accept(Visitor v);
}

class IntLitNode extends LiteralNode
{
    IntLitNode(int value) { super(value); }

    void accept(Visitor v) { v.visit(this); }
}
```

The role of the `accept` method is to facilitate traversal of the tree by accepting an object as input, a `Visitor`. Each node inheriting from ASTnode also then implements an accept method as shown above with the `IntLitNode` class. The specific `accept` method simply calls the visit method with `this` object as input. The effect is that the `visit` method in a class that inherits from `Visitor` is called (the second half of the double dispatch).

It is our responsibility to then define a `Visitor` class; our abstract `Visitor` class is shown below. In the `Visitor` class observe that each AST class is represented with a `visit` method. This allows us as programmers the ability to define, in one file, all methods related to expression tree traversal (e.g., evaluation, infix traversal, etc.).

```
abstract class Visitor
{
    void visit(ASTnode n) { n.accept(this); }

    // Literals
    abstract void visit(LiteralNode n);

    abstract void visit(IntLitNode n);
    abstract void visit(RealLitNode n);

    // Operations
    abstract void visit(OperationNode n);

    // Unary Operations
    abstract void visit(UnaryExprNode n);

    abstract void visit(UnaryPlusNode n);
    abstract void visit(UnaryMinusNode n);
    abstract void visit(UnaryAbsoluteValueNode n);
    abstract void visit(UnarySquareRootNode n);


    // Binary Operations
    abstract void visit(BinaryExprNode n);

    abstract void visit(BinaryPlusNode n);
    abstract void visit(BinaryMinusNode n);
    abstract void visit(BinaryTimesNode n);
    abstract void visit(BinaryDivideNode n);
    abstract void visit(BinaryPowerNode n);
}
```

In order to understand the importance of maintaining traversal code in one file, consider the alternative: defining all traversal methods inside the `ASTnode` file. That means one method in each class would be devoted to each traversal operation. Thus, if we have many traversal operations, one file is polluted with many methods that are seemingly unrelated. This contradicts our goal of separating concerns in software development since we would be mixing many unrelated algorithms / implementations in one file. For example, *unparsing* an expression tree has nothing to do with *evaluating* an expression tree; hence, we wish to keep those computations separated.

As an example of a 'visitor' class inheriting from `Visitor`, consider (part of) class `PostfixUnparser` shown below. This code listing depicts a partial implementation of unparsing functionality we had from the last lab in the form of a `Visitor`.

```
public class PostfixUnparser extends Visitor
{
    protected ASTnode       _root;       // root of the Expression tree
    protected StringBuilder _sb;         // output stream we will write to

    /** Invokes unparsing of this tree using postfix traversal. */
    public String unparse()
    {
        this.visit(_root);
        return _sb.toString();
    }

    @Override
    public void visit(UnaryPlusNode n)
    {
        _sb.append(Constants.PLUS);
        this.visit(n._expr);
    }

    @Override
    void visit(BinaryPlusNode n)
    {
        this.visit(n._left);
        _sb.append(Constants.DELIMITER);
        this.visit(n._right);
        _sb.append(Constants.DELIMITER);
        _sb.append(Constants.PLUS);
    }
```

In this code listing, we can visit the children of nodes (`_expr` in `UnaryPlusNode`; `_left` and `_right` in `BinaryPlusNode`). Otherwise, we can continue to build the string that is returned from the unparsed method.

Use the code in `PostfixUnparser` as a guide for other visitors implemented for expression trees.

What You Need to Do: Implementing Other Visitors

You are to implement and junit test three different visitors discussed below.

Visitor 1: Prefix Unparsing

We can convert from a reverse Polish notation (postfix) to a Polish notation (prefix). Prefix notation also does not require parentheses to encode order of operations. Some examples are shown in Table 1.

Table 1: Sample Postfix and Corresponding Prefix expressions

Postfix Notation	Prefix Notation
3 4 +	+ 3 4
3 2 1 − *	* 3 − 2 1
3 A 2 S +	+ A 3 S 2
1 2 3 4 5 + − * /	/ 1 * 2 − 3 + 4 5

Traversing an expression tree to generate prefix expressions is simply an issue of traversing the tree using a pre-order traversal technique.

Visitor 2: Full Parenthesization Infix Unparsing

We can also convert from postfix notation to an infix format; you are familiar with infix notation since it is how we write mathematical expressions. As you know, with infix notation, parentheses are required in order to properly evaluate according to precedence and associativity rules of operators (e.g., +, −, etc.). Some examples are shown in Table 2. Please pay careful attention to the expected spacing between parentheses, binary operators, and values (e.g., `"5 3 +"` → `"( 5 + 3 )"`. Also note that prefix operators do not require spaces for separation (e.g., `"+-3.0"` → `"+-3.0"` and `"3.0 A"` → `"A( 3.0 )"`).

Table 2: Sample Postfix and Corresponding Fully Parenthesized Infix Expressions

Postfix Notation	Infix Notation
3 4 +	(3 + 4)
3 2 1 − *	(3 * (2 - 1))
3 A 2 S +	(A(3) + S (2))
1 2 3 4 5 + − * /	(1 / (2 * (3 - (4 + 5))))
1 2 3 − 4 5 + * /	(1 / ((2 − 3) * (4 + 5)))

Traversing an expression tree to generate infix notation is simply an issue of traversing the tree using an in-order technique.

Visitor 3: Evaluation

Our last visitor will evaluate a mathematical expression and return the result as a `double`. Some examples are shown in Table 2.

Table 3: Sample Postfix Expression Evaluations

Postfix Notation	Evaluation
3 4 +	7
3 2 1 − *	3
3 A 2 S +	$3+\sqrt{2}$
1 2 3 − 4 5 + * /	$-\frac{1}{9}$

Evaluation of an expression requires a post-order traversal: evaluating an expression requires that all sub-expressions must be evaluated first.

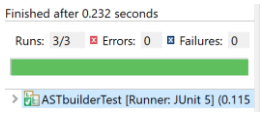
What You Need To Do and What is Provided

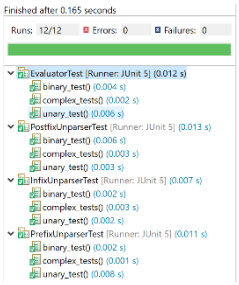
You will implement three visitor classes (`PrefixUnparser`, `InfixUnparser`, and `Evaluator`) and one junit test class (`EvaluatorTest`). You must provide a working `ASTbuilder` class from the previous lab.

Junit testing code has been provided in other corresponding test files.

JUnit Testing

No output should be produced by your tests; we are seeking only a 'green' output indication in Eclipse. Make sure your screenshot shows success of all junit testing methods and not just the overall summary. For example, the image on the left is bad (and on the right is good).





Submitting: Source Code

Your code should be well documented, including docstring comments of methods, blocks of code, and header comments in each file. Junit tests should have reasonable String messages output, if failure occurs.

Header Comments

Your program must use the following standard comment at the top of *each source code file*. Copy and paste this comment and modify it accordingly into these files.

```
/**
 * Write a succinct, meaningful description of the class here. You should avoid wordiness
 * and redundancy. If necessary, additional paragraphs should be preceded by <p>,
 * the html tag for a new paragraph.
 *
 * <p>Bugs: (a list of bugs and / or other problems)
 *
 * @author <your name>
 * @date   <date of completion>
 */
```

Inline Comments

Comment your code with a *reasonable amount of comments* throughout the program. Each method should have a comment that includes information about input, output, overall operation of the function, as well as any limitations that might raise exceptions; Javadoc comments are ideal. Each *block* of code (3-4 or more lines in sequence) in a function should be commented.

It is ***prohibited*** to use *long* comments to the right of lines of source code; attempt 80 character-wide text in source code files.

Submitting: Proof of Program Execution

Execute your code and take a screenshot of the associated junit window and output console (with no output). Place these screenshots into a word processing document (Word, OpenOffice, GoogleDocs, etc.). If multiple screenshots are necessary, label each clearly. Please make sure to crop and enlarge the screenshots so that the picture and / or text is clear (and doesn't strain my old eyes).

Create a PDF of this document and call it `evidence.pdf`.

Source Code Files

Please submit only the source code files for your visitors and one test file: `PrefixUnparser`, `InfixUnparser`, `Evaluator`, and `EvaluatorTest`.

Final Submission File

In a folder named `lab`, place (1) the source code files and (2) `evidence.pdf`. Zip folder `lab` and label that zip file as `lab.zip`. This zip file is to be submitted via Moodle.

Please be reminded that following instructions explicitly and submitting well-formatted documents (with consistent fonts and typeset equations) is an important part of professionalism.

Only one person in your pair needs to submit the final product.