CI346
Lab Session 5
Design Patterns

Download the file `Visitor.zip` from studentcentral. It contains a Java project relating to the *Visitor pattern*. There are unit tests in the package `CI346.week5` which you can run to test your progress.

1. Read the code for the classes in the package `CI346.tree` and `CI346.tree.visitor`, which are used to construct *binary trees* and *visitors* for them. Note that the base class of nodes, `BinaryTree`, implements the `VisitableTree` interface. That means that the two subclasses of `BinaryTree` (`Branch` and `Leaf`) have to implement the `accept` method.

2. Add the missing `visit` methods to the `TreeVisitor` interface.

3. The `ListTreeVisitor` class is a visitor for binary trees that produces a list of the labels in a tree. Update `ListTreeVisitor` so that it implements the `visit` methods from the `TreeVisitor` interface. The `visit` methods should add the label of the current node to `result`, which is a list of `String`s. It should do this in a **in-order** traversal. That is, `Branch` nodes should pass the visitor to the left hand child first, then do something with the label of the current node, then pass the visitor to the right hand child.

4. The `SumTreeVisitor` class is a visitor for binary trees with numeric labels that calculates the total of all labels in the tree. Update `SumTreeVisitor` so that it implements the `visit` methods from the `TreeVisitor` interface. The `visit` methods should add the label of the current node to the `sum` field. It should carry out a **pre-order** traversal: do something with the label of the current node, then pass the visitor to the left hand child, then pass the visitor to the right hand side.

5. Read the code for the classes in the package `CI346.ast` and `CI346.ast.visitor`, which are used to construct *abstract syntax trees* and visitors for them. An abstract syntax tree is a representation of an expression that is used by parsers and compilers. In this case, the expression is arithmetic with numbers, variables, addition and subtraction. An expression such as $5 + y - 2$ would produce the following AST:

```
1  new Plus(new Val(5), new Minus(new Id("y"), new Val(2)))
```

Evaluating this expression successfully would require an *environment* in which y is defined; i.e., a lookup table that allows us to look up the value assigned to y. `Exp` is the base class of all AST nodes, and implements the `VisitableAST` interface. Note that the tests work with the type `Exp<Integer>`, or expressions that produce integers when evaluated, but the code is polymorphic so we could also create, say, logical expressions with the type `Exp<Boolean>`.

6. Add the missing `visit` method signatures to the `ASTVisitor` class. Note that, unlike the previous example, each `visit` method should return a value of type `T`.

7. Update the `EvalVisitor` class so that it implements the `visit` methods from the `ASTVisitor` class. Unlike `ASTVisitor`, the `visit` methods of `EvalVisitor` should be specialised to return values of `Integer`, rather than the type variable `T`. The `visit` methods for binary operators (`Plus` and `Minus`) should evaluate the left and right hand sides of the expression then combine them appropriately. The `visit` method for `Val` objects should return the inner `int` value. The `visit` method for `Id` objects should attempt to look up the identifier in the `env` lookup table, returning a `RunTimeException` with an appropriate message if the identifier is undefined.

8. Extend the AST code to include multiplication and exponentiation. Begin by creating two new subclasses of `BinaryOp` called `Mul` and `Pow`. Both classes will need a constructor that takes two `Exp` parameters. Extend the `ASTVisitor` and `EvalVisitor` classes to accommodate the new types. Evaluating a `Mul` node means evaluating the left and right hand sides then multiplying the results. Evaluating a `Pow` node means raising the result of evaluating the first parameter to the value produced by evaluating the second parameter.

   Create new tests in the class `CI346.week5.ASTVisitorTest` to test your work. You should now be able to evaluate expressions such as $x \times 2^{y-1}$, given an environment that contains declarations of x and y. This expression is represented by the following AST:

```
1 new Mul(new Id("x"), new Pow(new Val(2), new Minus(new Id("y")
      , new Val(1))))
```