Homework 11

Due Wednesday, 12/3/08

Please turn in your homework solutions by the beginning of class to the dropoff folder on `vpn.cs.pomona.edu` in directory `/common/cs/cs131/dropbox`. Make sure that all of your files include your name and the problem number in a comment.

1. (15 points) **Smalltalk Reading**

   Read "Design Principles Behind Smalltalk" by Dan Ingalls (available on the "links" page).

   Think about the Principles set forth. Which are new? Which have we seen before? What principles are remniscent of Lisp? What is the scope of the language (ie, what is included in its definition)? A few sentences on each item is sufficient— clarity is more important than length of answer.

2. (15 points) **C++**

   From "An Overview of C++," by Bjarne Stroustrup, SIGPLAN Notices, 1986-10:

   "Section 6. What is Missing?

   C++ was designed under severe constraints of compatibility, internal consistency, and efficiency: no feature was included that

   (a) would cause a serious incompatibility with C at the source or linker levels.

   (b) would cause run-time or space overheads for a program that did not use it.

   (c) would increase run-time or space requirements for a C program.

   (d) would significantly increase the compile time compared with C.

   (e) could only be implemented by making requirements of the programming environment (linker, loader, etc.) that could not be simply and efficiently implemented in a traditional C programming environment.

   Features that might have been provided but weren't because of these criteria include garbage collection, parameterized classes, exceptions, multiple inheritance, support for concurrency, and integration of the language with a programming environment. Not all of these possible extensions would actually be appropriate for C++, and unless great constraint is exercised when selecting and designing features for a language, a large, unwieldy, and inefficient mess will result. The severe constraints on the design of C++ have probably been beneficial and will continue to guide the evolution of C++."

   Contrast the design goals to those of Smalltalk? What is different? What is the same? What is the intended use of C++, and how does that influence the design?

3. (10 points) **Subtyping and Binary Methods**

   Please do problem 11.8 from Mitchell, page 334.

4. (15 points) **Like Current in Eiffel**

   Please do problem 12.8 from Mitchell, page 376.

5. (15 points) **Expression Objects**

We now look at an object-oriented way of representing arithmetic expressions given by the grammar

$$e \quad ::= \quad num \mid e + e$$

We begin with an "abstract class" called `SimpleExpr`. While this class has no instances, it lists the operations common to all instances of this class or subclasses. In this case, it is just a single method to return the value of the expression.

```
abstract class SimpleExpr {
    int eval();
}
```

Since the grammar gives two cases, we have two subclasses of `SimpleExpr`, one for numbers and one for sums.

```
class Number extends SimpleExpr {
    private int n;
    public Number(int n) { this.n = n; }
    public int eval() { return n; }
}

class Sum extends SimpleExpr {
    private SimplExpr left, right;
    public Sum(SimpleExpr left, SimpleExpr right) {
        this.left = left;
        this.right = right;
    }
    public int eval() { left.eval() + right.eval(); }
}
```

(a) *Product Class*

Extend this class hierarchy by writing a `Times` class to represent product expressions of the form

$$e \quad ::= \quad \ldots \mid e * e$$

(b) *Method Calls*

Suppose we construct a compound expression by

```
SimpleExpr a = new Number(3);
SimpleExpr b = new Number(5);
SimpleExpr c = new Number(7);
SimpleExpr d = new Sum(a,b);
SimpleExpr e = new Times(d,c);
```

and send the message `eval` to `e`. Explain the sequence of calls that are used to compute the value of this expression: `e.eval()`. What value is returned?

(c) *Comparison to "Type Case" constructs*

Let's compare this programming technique to the expression representation used in ML, in which we declared a datatype and defined functions on that datatype by pattern matching. The following `eval` function is one form of a "type case" operation, in which the program inspects the actual tag (or type) of a value being manipulated and executes different code for the different cases:

```
datatype MLExpr =
    Number   of int
  | Sum      of MLExpr * MLExpr;

fun eval (Number(x)) = x
  | eval (Sum(e1,e2)) = eval(e1) + eval(e2);
```

This idiom also comes up in class hierarchies or collections of structures where the programmer has included a *Tag* field in each definition that encodes the actual type of an object.

i. Discuss, from the point of view of someone maintaining and modifying code, the differences between adding the `Times` class to the object-oriented version and adding a `Times` constructor to the `MLExpr` datatype. In particular, what do you need to add/change in each of the programs. Generalize your observation to programs containing several operations over the arithmetic expressions, and not just `eval`.

ii. Discuss the differences between adding a new operation, such as `toString`, to each way of representing expressions. Assume you have already added the product representation so that there is more than one class with a nontrivial `eval` method.

6. (25 points) **Visitor Design Pattern**

The extension and maintenance of an object hierarchy can be greatly simplified (or greatly complicated) by design decisions made early in the life of the hierarchy. This question explores various design possibilities for an object hierarchy representing arithmetic expressions.

The designers of the hierarchy have already decided to structure it as shown below, with a base class `Expr` and derived classes `Number, Sum, Times`, and so on. They are now contemplating how to implement various operations on Expressions, such as printing the expression in parenthesized form or evaluating the expression. They are asking you, a freshly-minted language expert, to help.

The obvious way of implementing such operations is by adding a method to each class for each operation. The Expression hierarchy would then look like:

```
public interface Expr {
    public String toString();
    public int eval();
}

public class Number implements Expr {
    private int n;

    public Number(int n) { this.n = n; }
    public String toString() { ... }
    public int eval() { ... }
}
```

```
public class Sum implements Expr {
    private Expr left, right;

    public Sum(Expr left, Expr right) {
        this.left = left;
        this.right = right;
    }
    public String toString() { ... }
    public int eval() { ... }
}
```

Suppose there are n subclasses of `Expr` altogether, each similar to `Number` and `Sum` shown here. How many classes would have to be added or changed to add each of the following things?

(a) A new class to represent product expressions.

(b) A new operation to graphically draw the expression parse tree.

Another way of implementing expression classes and operations uses a pattern called the Visitor Design Pattern. In this pattern, each operation is represented by a `Visitor` class. Each Visitor class has a `visit`*`Class`* method for each expression class *Class* in the hierarchy. Each expression class *Class* is set up to call the `visit`*`Class`* method to perform the operation for that particular class. In particular, each class in the expression hierarchy has an `accept` method which accepts a `Visitor` as an argument and "allows the Visitor to visit the class and perform its operation." The expression class does not need to know what operation the visitor is performing.

If you write a Visitor class `ToString` to construct a string representation of an expression tree, it would be used as follows:

```
Expr expTree = ...some code that builds the expression tree...;
ToString printer = new ToString();
String stringRep = expTree.accept(printer);
System.out.println(stringRep);
```

The first line defines an expression, the second defines an instance of your `ToString` class, and the third passes your visitor object to the `accept` method of the expression object.

The expression class hierarchy using the Visitor Design Pattern has the following form, with an `accept` method in each class and possibly other methods. Since different kinds of visitors return different types of values, the accept method is parameterized by the type that the visitor computes for each expression tree:

```
public interface Expression {
        <T> T accept(Visitor<T> v);
}

public class Number implements Expression {
        private int n;

        public Number(int n) {
                this.n = n;
```

```
        }

        public <T> T accept(Visitor<T> v) {
                return v.visitNumber(this.n);
        }
}


public class Sum implements Expression {
        private Expression left, right;

        public Sum(Expression left, Expression right) {
                this.left = left;
                this.right = right;
        }

        public <T> T accept(Visitor<T> v) {
                T leftVal = left.accept(v);
                T rightVal = right.accept(v);
                return v.visitSum(leftVal, rightVal);
        }
```

The associated `Visitor` abstract class, naming the methods that must be included in each visitor, and some example subclasses, have this form:

```
public interface Visitor<T> {
        T visitNumber(int n);

        T visitSum(T left, T right);
}

public class Eval implements Visitor<Integer> {
        public Integer visitNumber(int n) {
                return new Integer(n);
        }

        public Integer visitSum(Integer left, Integer right) {
                return new Integer(left.intValue() + right.intValue());
        }
}


public class ToString implements Visitor<String> {
        public String visitNumber(int n) {
                return "" + n;
        }

        public String visitSum(String left, String right) {
                return "(" + left + " + " + right + ")";
        }
}
```

Here is an example of using the visitor to evaluate and print the value of an expression.

```
public class ExpressionVisitor {
        public static void main(String s[]) {
                Expression e = new Sum(new Number(3), new Number(2));
                Eval eval = new Eval();
                int n = e.accept(eval);
                System.out.print(e.accept(new ToString()) + " = " + n);
        }
}
```

(c) Starting with the call to `e.accept(eval)`, what is the sequence of method calls that will occur while evaluating the expression tree `e`? (Do not record calls to constructors and `intValue`– just consider the `visit` and `accept` methods).

Suppose there are n subclasses of `Expression`, and m subclasses of `Visitor`. How many classes would have to be added or changed to add each of the following things using the Visitor Design Pattern?

(d) A new class to represent product expressions.

(e) A new operation to graphically draw the expression parse tree.

The designers want your advice.

(f) Under what circumstances would you recommend using the standard design?

(g) Under what circumstances would you recommend using the Visitor Design Pattern?