Homework 10

Due **Tuesday**, 12/3/13

Please turn in your homework solutions online at `http://www.dci.pomona.edu/tools-bin/cs131upload.php` by midnight on the due date.

1. (15 points) **C++**

   The following is an excerpt 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++."

   Given this description, please 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?

   You do not need to read Stroustrup's paper to answer this question, though you might find it interesting. If you want to read it, you can search for it through the ACM digital library, `http://portal.acm.org/`.

2. (10 points) **Subtyping and Binary Methods**

   Please do problem 11.8 from Mitchell, page 334.

3. (15 points) **Like Current in Eiffel**

   Please do problem 12.8 from Mitchell, page 376.

4. (15 points) **Expression Objects**

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

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

public class Sum extends SimpleExpr {
    private SimplExpr left, right;
    public Sum(SimpleExpr left, SimpleExpr right) {
        this.left = left;
        this.right = right;
    }
    public int eval() { return 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 Haskell, in which we declared a data type and defined functions on that type by pattern matching. The following `eval` and `toString` functions illustrate 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:

```
data HaskellExp = Number Integer | Sum (HaskellExp, HaskellExp)

eval:: HaskellExp -> Integer
eval (Number(x)) = x
eval (Sum(e1,e2)) = eval(e1) + eval(e2)

toString:: HaskellExp -> String
toString(Number(x)) = show x
toString(Sum(e1,e2)) = toString(e1) ++ " + " ++ toString(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 `HaskellExpr` data type. 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` and `toString`.

ii. Discuss the differences between adding a new operation, such as `prettyPrint` (`toString` is pretty lame, not introducing any parenthesization), to each way of representing expressions. Assume you have already added the product representation so that there is more than one class with nontrivial `eval` methods.

You might also want to look back at problem 1 in the last homework to look at a similar problem in writing Haskell code.

5. (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 builds on the previous question in exploring 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. Discuss (put don't write code) 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 on all expressions 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.

An `Eval` visitor might work similarly, but evaluating an expression rather than converting it to a string.

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;
        }

        // Do visitor operation to number being represented
        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;
        }

        // Do visitor operation on left and right and then do
        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 classes implementing it, have this form:

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

        T visitSum(T left, T right);
}
```

Notice that there is a `visit`*Class* method for each class that the visitor must handle. (Traditionally, these are all just named `visit` without the class name, relying on static overloading. Needless to say, I find this more confusing than helpful!)

```
// A Visitor class to evaluate expressions, returning an Integer
public class Eval implements Visitor<Integer> {
        public Integer visitNumber(int n) {
            // evaluating an integer is easy
            return new Integer(n);
        }
```

```
        public Integer visitSum(Integer left, Integer right) {
            // evaluate sum by adding values of given left and right sides
            return new Integer(left.intValue() + right.intValue());
        }
}


// A Visitor class to convert an expression ot a String
public class ToString implements Visitor<String> {
        public String visitNumber(int n) {
            // convert the integer to a string
            return "" + n;
        }

        public String visitSum(String left, String right) {
            // Given the String representations of left and right
            // create the String representations of the sum
            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);
        }
}
```

In the above, once the expression is created, it "accepts" an evalution visitor whose job is to evaluate the expression. In the "println" statement a `ToString` visitor is accepted, resulting in a string representation of the expression.

(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 and how many would have to be 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?