
Reading

1. Mitchell, Chapters 9–10

Problems

1. (15 points) Equivalence of Abstract Data Types

Mitchell, Problem 9.2

2. (40 points) Generic Programming in Java

The goal of this question is to become become familiar with parameterized classes in Java, in the context of some of the ideas behind the C++ STL library.

We will be using the 1.5 release of Java, which includes generic types. For example, the following is a generic Stack class:

```
class Stack<T> {
    Vector<T> elems = new Vector<T>();

    public boolean isEmpty() {
        return elems.size() == 0;
    }

    public void push(T t) {
        elems.add(t);
    }

    public T pop() {
        return elems.remove(elems.size() - 1);
    }
}

...
Stack<String> st = new Stack<String>();
st.push("cow");
st.push("moo");
...
Stack<Integer> si = new Stack<Integer>();
si.push(3);
int x = si.pop();
```

(Only objects may be stored in a Stack, but the Java 1.5 compiler also supports automatic boxing and unboxing between primitive ints and Integer objects, so you do not need to perform such conversions yourself.)

This question explores how to implement and use a number of concepts from the STL, with an emphasis on how those concepts can be realized in Java. We start with Function Objects and Generic Containers of several different forms and then examine Iterators and Mapping.

Before starting, download the starter files from the cs334 handouts page.

A Function Object in Java will be an extension of the following class, which is found in `Function.java`:

```
abstract class Function<Domain,Range> {
    abstract public Range apply(Domain d);
}
```

Subclasses override the `apply` method to perform the appropriate operation on a value from the domain to produce a value in the function's range. Here are two simple examples:

```
/**
 * Function object to convert a string like ``36`` into
 * the number 36.
 */
class StringToIntFunction extends Function<String,Integer> {
    public Integer apply(String s) {
        return Integer.parseInt(s);
    }
}

/**
 * Function object to negate a double value.
 */
class NegateFunction extends Function<Double,Double> {
    public Double apply(Double d) {
        return -d;
    }
}
```

Another example would be a Function that converts an Integer value into the long-hand written form. For example, this function would convert the number 123 to “one-hundred twenty-three”:

```
class IntToWordsFunction extends Function<Integer,String> {
    public String apply(Integer intObject) { ... }
}
```

- (a) Your first job is to implement a `ComposeFunction` object that composes the behavior of two Function Objects. Consider the following example:

```
StringToIntFunction stringToInt = new StringToIntFunction();
IntToWordsFunction intToWords = new IntToWordsFunction();
Function<String,String> stringToWords =
    new ComposeFunction<...>(stringToInt, intToWords);
System.out.println(stringToWords.apply("63"));
```

This code fragment would print out “Sixty-three”. The application in the fifth line performs first a conversion of “63” into an integer, and then converts that integer into the words, using two previously defined Function objects. I have left out the type arguments to `ComposeFunction`. What should they be in this case?

Complete the class definition in `ComposeFunction.java`, and demonstrate that your code works by changing the `Numbers.java` program to convert the numbers passed as strings to the program from the command line into words. For example:

```
> java Numbers 12 123
12 --> twelve
123 --> one-hundred twenty-three
```

- (b) It is often useful or necessary to cache the results of performing a function, so that we don't have to compute the result again. In the `Fib.java` file, I define a `FibFunction` that computes the Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, This series is defined by

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } x = 1 \\ f(x-1) + f(x-2) & \text{otherwise} \end{cases}$$

The implementation I have given you is the naïve implementation that takes exponential time to compute f . However, it is representative of costly operations whose result we may wish to cache, such as web searches (see the on-line Google search API), solutions to graph traversal problems, etc.

Implement the `CachedFunction` class in `CachedFunction.java`. This class takes another `Function` object g as a parameter to the constructor and its `apply` method returns the same results that g would return. However, it should cache the results, so that subsequent calls to `apply` will not query g unless the argument to the call is an argument that has not been seen before. See `CachedFunction.java` for more details.

Demonstrate this class by changing the `Fib.java` program to use a cached function to repeatedly compute `fib` for a reasonably large number (25–30).

You may wish to look at the documentation for `java.util.Hashtable` as one way to implement this class.

- (c) We will next implement a mapping function. The program `Translate.java` takes input from the terminal and should convert it into “Pig Latin” and “Pirate”. Here is an example:

```
> java Translate < test2.txt
```

```
ORIGINAL:
```

```
Pardon, officer, where is the pub?
```

```
PIG LATIN:
```

```
ardonPay, officerway, erewhay isway ethay ubpay?
```

```
PIRATE:
```

```
avast, foul braggart, whar be th' Skull & Scuppers?
```

I have given you the pig latin translator and the pirate translator as `Function` objects from `String` to `String`, as well as some supporting code to convert the standard input text into a `Vector` of words. You must write the `map` function in `Translate.java`, that takes an `Iterator` of elements and a function f , and constructs a new `Iterator` containing the results of applying f to each element of the original `Iterator`. The `java.util.Iterator` interface is defined below:

```
public interface Iterator<T> {
    /**
     * Returns true if there are more elements, false if not.
     */
    boolean hasNext();

    /**
     * Returns the next element from the iterator.
     */
    T next();
}
```

See the online javadoc for more details.

One straightforward way to implement this is to just construct a temporary `Vector` for the results of applying the function to each element the `Iterator` returns and then construct a new `Iterator` for that vector.

Implement `map` in this way, and verify that it works by changing `main` in `Translate` to convert the input into Pig Latin and Pirate.

- (d) Add at least one additional translation function— Surfer Dude Translation, Swedish Chef Translation for Muppets Fans (Bork Bork Bork), Pirate-Pig-Latin (think composition), etc. It can be as simple or as complicated as you like, and can even map `Strings` to something other than `Strings`. Bonus points for creativity.
- (e) If the mapping function takes a long time to compute, or there are many elements in the iterator, then the `map` function above may cause undesirable delays in your program as it empties the iterator and builds the new one. Moreover, if the initial `Iterator` is infinite (ie, it is just counting the integers), then `map` will never terminate.

However, it is possible to create a “lazy map” operation that will work in this case. To do this, you need to write a new `MappingIterator` class that takes an iterator `e` and a `Function` Object `f` as parameters to the constructor. A call to `next()` on the `MappingIterator` object will cause it to get an element from its underlying iterator `e`, apply `f` to it, and then return that value. In this way, we never compute the mapped values for more than one element at a time. Add a generic `MappingIterator` class to `Translate.java` and demonstrate that it works in the `main` function.

The `MappingIterator` class should be parameterized by two types— `S`, the type of element stored in the initial `Iterator`, and `T`, the result type of the `Function` Object. Your class should also implement the `java.util.Iterator<T>` interface for the type `T`, so that you can use it in place of any normal `Iterator`.

- (f) Turn in all Java files with the `turnin -c 334` script (individually or, preferable, as a single tar file). Include a printout of the files you changed with your problem set.
- (g) **(Hand in this part with your written answers)** Compare this experience to using the related programming patterns in Lisp and ML. In what ways was this easier in Java? In what ways was it more difficult? Given your experience, do you think that the Java designers should add higher-order functions and mapping primitives to the language?

3. (15 points) Expression Objects

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

$$e ::= \text{num} \mid e + e$$

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

```

class Sum extends SimpleExpr {
  SimpleExpr left, right;
  public Number(SimpleExpr left, SimpleExpr right) {
    this.left = left;
    this.right = right;
  }
  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

$$e ::= \dots \mid e * e$$

(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 MLEExpr =
  Number of int
  | Sum of MLEExpr * MLEExpr;

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 `MLEExpr` 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.

4. (20 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 (like the one above) that represents 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:

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

class Number extends Expr {
    int n;

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

class Sum extends Expr {
    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 division 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 `visitClass` method for each expression class `Class` in the hierarchy. Each expression class `Class` is set up to call the `visitClass` 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:

```
abstract class Expr {
    abstract <T> T accept(Visitor<T> v);
}

class Number extends Expr {
    int n;

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

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

class Sum extends Expr {
    Expr left, right;

    public Sum(Expr left, Expr 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 the ToString visitor, have this form:

```
abstract class Visitor<T> {
    abstract T visitNumber(int n);
    abstract T visitSum(T left, T right);
}

class ToString extends 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 an expression.

```
class ExprVisitor {
    public static void main(String s[]) {
        Expr e = new Sum(new Number(3), new Number(2));
        ToString printer = new ToString();
        String stringRep = e.accept(printer);
        System.out.print(stringRep);
    }
}
```

- (c) Starting with the call to `e.accept(printer)`, what is the sequence of method calls that will occur while building the string for the expression tree `e`?
- (d) The code for these classes is located on the handouts webpage in the `ExprVisitor.java` file. Download that code and compile it with `javac`. Add the following classes to that file. You will need to change some of the existing classes to accommodate them.
 - i. An `Eval` visitor class that computes the value of an expression tree. The visit methods should return an `Integer`. Recall that Java 1.5 has auto-boxing, so it can convert `int` values to `Integer` objects and vice-versa, as needed.
 - ii. `Subtract` and `Times` classes to represent subtraction and product expressions.
 - iii. A `Compile` visitor class that returns a sequence of stack-based instructions to evaluate the expression. You may use the following stack instructions (Refer back to HW 3 if you need a refresher on how these instructions operate):

```
PUSH(n)
ADD
MULT
SUB
DIV
SWAP
```

The visit methods can simply return a `String` containing the sequence of instructions. For example, compiling $3 * (1 - 2)$ should return the string

```
PUSH(3) PUSH(1) PUSH(2) SUB MULT
```

The instruction sequence should just leave the result of computing the expression on the top of the stack. Hint: the order of instructions you need to generate is exactly a post-order traversal of the expression tree.

Aside: Most modern compilers (including the Sun Java compiler) are implemented using the Visitor Pattern. The compilation process is really just a sequence of visit operations over the abstract syntax tree. Common steps include visitors 1) to resolve the declaration to which each variable access refers; 2) to perform type checking; 3) to optimize the program; 4) to generate code as above; and so on.

Use turnin to submit your modified source file, and include a printout with your problem set.

Suppose there are n subclasses of `Expr`, 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?

- (e) A new class to represent division expressions.
- (f) A new operation to graphically draw the expression parse tree.

The designers want your advice.

- (g) Under what circumstances would you recommend using the standard design?
- (h) Under what circumstances would you recommend using the Visitor Design Pattern?

5. (10 points) Alternative Expressions

It is sometimes useful to design an expression class hierarchy in a way that allows code defined outside of the hierarchy to access the subtrees of a node. Consider the following code and specification for the expression grammar:

```
abstract class SimpleExpr {
    // returns true if and only if no subexpressions
    public boolean isAtomic();
}
```

```

// returns the left subexpression if not atomic,
// or null if it is
public SimpleExpr lsub();

// returns the right subexpression if not atomic,
// or null if it is
public SimpleExpr rsub();

// returns the value of the represented expression
public int value();
}

class Number extends SimpleExpr {
    private int n;
    public Number(int n) { this.n = n; }
    public boolean isAtomic() { return true; }
    public SimpleExpr lsub() { return null; }
    public SimpleExpr rsub() { return null; }
    public int value() { return n; }
}

class Sum extends SimpleExpr {
    private SimpleExpr left, right;
    public Number(SimpleExpr left, SimpleExpr right) {
        this.left = left;
        this.right = right;
    }
    public boolean isAtomic() { return false; }
    public SimpleExpr lsub() { return left; }
    public SimpleExpr rsub() { return right; }
    public int value() { left.value() + right.value(); }
}

```

This makes it easy to, for example, write a method over SimpleExpr trees that determines the number of leaf nodes:

```

//count # of leafs
int countLeaves(SimpleExpr e) {
    if (e.isAtomic()) {
        return 1;
    } else {
        return countLeaves(e.lsub()) + countLeaves(e.rsub());
    }
}

```

- (a) Extend the class hierarchy to include a Square class to represent squaring expressions of the form

$$e ::= \dots | e^2$$

In order to support this class, what changes will be required to:

- i. the interface and specification of the SimpleExpr class?
- ii. subclasses of SimpleExpr?
- iii. functions like countLeaves that use lsub, rsub, and isAtomic to manipulate expression trees?

Try to make as few changes as possible to the program. (You need not write out all of the code—just `Square` and a description of what else needs to change.)

- (b) Similar issues arise if we wish to extend the class hierarchy to have a `Cond` class that represents ternary (3-argument) conditional expressions of the form

$$e ::= \dots \mid e?e:e$$

As in C or Java, a conditional expression `a?b:c` evaluates `a`, and then returns the value of `b` if `a` is true, or the value of `c` if `a` is false.

Suppose we anticipate adding `Cond` and other expressions that have even more subexpressions to our program. What kind of interface to `SimpleExpr` would let us easily support atomic, unary, binary, ternary and n -ary operators without making further changes to the interface or client code each time a new expression form is added?

In other words, redefine the `SimpleExpr` abstract class so that we can minimize changes that may be needed in the future as we add more expression forms. Just writing out the abstract class definition is sufficient—do not bother writing out the code for the entire class hierarchy.