Homework 5

Due 7 November

Handout 6
CS242: Autumn 2012
31 October

Reading

1. Chapter 10, except section 10.4
2. Chapter 11, sections 11.1, 11.2, 11.3.1 and 11.4., 11.5, 11.6 only
3. Chapter 12, sections 12.1, 12.2, and 12.3 only
4. Chapter 13, sections 13.1 and 13.2 only

Problems

1. Existential Halting Problem

Suppose you are given a function $\mathsf{Halt}_\exists$ that, given a program P as input, will determine whether or not there exists an input n for which $P(n)$ halts. To make this concrete, assume that you are writing a C or Pascal program that reads in another program as a string. Your program is allowed to call $\mathsf{Halt}_\exists$ with a string input. Assume that the call to $\mathsf{Halt}_\exists$ returns true if the argument is a program that will halt on some unspecified input, and returns false if the argument is a program that runs forever on all inputs. If $\mathsf{Halt}_\exists$ returns true, all you know is that there exists an input for which P halts; it does not return the specific *value* of that input. You should not make any assumptions about the behavior of $\mathsf{Halt}_\exists$ on an argument that is not a syntactically correct program.

Can you solve the halting problem using $\mathsf{Halt}_\exists$? More specifically, can you write a program that reads a program text P as input, reads an integer n as input, and then decides whether $P(n)$ halts? You may assume that any program P you are given begins with a read statement that reads a single integer from standard input. This problem does not ask you to write the program to solve the halting problem. It just asks whether it is possible to do so.

If you believe that the halting problem can be solved if you are given $\mathsf{Halt}_\exists$, then explain your answer by describing how a program solving the halting problem would work. If you believe that the halting problem cannot be solved using $\mathsf{Halt}_\exists$, then explain briefly why you think not.

2. 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 `Expression` and derived classes `IntegerExp, AddExp, MultExp` 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:

```
class Expression
{
    virtual void parenPrint();
    virtual void evaluate();
```

```
    //...
}
class IntegerExp : public Expression
{
    virtual void parenPrint();
    virtual void evaluate();
    //...
}
class AddExp : public Expression
{
    virtual void parenPrint();
    virtual void evaluate();
    //...
}
```

Suppose there are n subclasses of `Expression` altogether, each similar to `IntegerExp` and `AddExp` 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 `visitCLS` method for each expression class `CLS` in the hierarchy. The expression class `CLS` is set up to call the `visitCLS` method to perform the operation for that particular class. 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 `ParenPrintVisitor` to print an expression tree, it would be used as follows:

```
    Expression *expTree = ...some code that builds the expression tree...;
    Visitor *printer = new ParenPrintVisitor();
    expTree->accept(printer);
```

The first line defines an expression, the second defines an instance of your `ParenPrintVisitor` 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 this form, with an `accept` method in each class and possibly other methods.

```
class Expression
{
    virtual void accept(Visitor *vis) = 0; //Abstract class
    //...
}
class IntegerExp : public Expression
{
    virtual void accept(Visitor *vis) {vis->visitIntExp(this);};
    //...
}
class AddExp : public Expression
{
    virtual void accept(Visitor *vis)
    { lhs->accept(vis); vis->visitAddExp(this); rhs->accept(vis); }
    //...
}
```

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

```
class Visitor
{
    virtual void visitIntExp(IntegerExp *exp) = 0;
    virtual void visitAddExp(AddExp *exp) = 0;   // Abstract class
}

class ParenPrintVisitor : public Visitor
{
    virtual void visitIntExp(IntegerExp *exp) {// IntExp print code};
    virtual void visitAddExp(AddExp *exp) {// AddExp print code};
}

class EvaluateVisitor : public Visitor
{
    virtual void visitIntExp(IntegerExp *exp) {// IntExp eval code};
    virtual void visitAddExp(IntegerExp *exp) {// AddExp eval code};
}
```

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?

(c) A new class to represent product expressions.

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

The designers want your advice.

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

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

3. ..... Simula Inheritance and Access Links

In Simula, a class is a procedure that returns a pointer to its activation record. Simula prefixed classes are a precursor to C++ derived classes, providing a form of inheritance. This question asks about how inheritance might work in an early version Simula, assuming that the standard static scoping mechanism associated with activation records is used to link the derived class part of an object with the base class part of the object.

Sample `Point` and `ColorPt` classes are given in the text (Section 11.2). For the purpose of this problem, assume that if `cp` is a `ColorPt` object, consisting of a `Point` activation record followed by a `ColorPt` activation record, the access link of the parent class (`Point`) activation record points to the activation record of the scope in which the class declaration occurs, and the access link of the child class (`ColorPt`) activation record points to the activation record of the parent class.

(a) Fill in the missing information in the following activation records, created by executing the following code:

```
ref(Point) r;
ref(ColorPt) cp;
r :- new Point(2.7, 4.2);
cp :- new ColorPt(3.6, 4.9, red);
cp.distance(r);
```

Remember that function values are represented by closures, and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code.

In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure, or compiled code. Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link." The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.

Activation Records *Closures* *Compiled Code*

(0)	r	(1)
	cp	(3)
(1) Point(. . .)	access link	(0)
	x	
	y	
	equals	•
	distance	•
(2) Point part of cp	access link	(0)
	x	
	y	
	equals	•
	distance	•
(3) ColorPt(. . .)	access link	( )
	c	
	equals	•
(4) cp.distance(r)	access link	( )
	q	(*r*)

⟨ ( ), • ⟩ code for equals

⟨ ( ), • ⟩

⟨ ( ), • ⟩ code for distance

⟨ ( ), • ⟩

⟨ ( ), • ⟩ code for cpt equals

(b) The body of `distance` contains the expression

$$\texttt{sqrt}((\texttt{x} - \texttt{q.x}) * *2 + (\texttt{y} - \texttt{q.y}) * *2)$$

which compares the coordinates of the point containing this `distance` procedure to the coordinate of the point `q` passed as an argument. Explain how the value of `x` is found when `cp.distance(r)` is executed. Mention specific links in your diagram. What value of `x` is used?

(c) This illustration shows that a reference `cp` to a colored point object points to the `ColorPt` part of the object. Assuming this implementation, explain how the expression `cp.x` can be evaluated. Explain the steps used to find the right `x` value on the stack, starting by following the pointer `cp` to activation record (3).

(d) Explain why the call `cp.distance(r)` only needs access to the `Point` part of `cp` and not the `ColorPt` part of `cp`.

(e) If you were implementing Simula, would you place the activation records representing objects `r` and `cp` on the stack, as shown here? Explain briefly why you might consider allocating memory for them elsewhere.

4. ..... Delegation-Based OO Languages

In this problem, we explore a delegation-based object-oriented language SELF in which objects can be defined directly from other objects. Classes are not needed and not supported. SELF has run-time type-checking and garbage collection.

The SELF language description says:

In Smalltalk, ... everything is an object and every object contains a pointer to its class, an object that describes its format and holds its behavior. In SELF too, everything is an object. But, instead of a class pointer, a SELF object contains named slots [which] may store either state or behavior. If an object receives a message and it has no matching slot, the search continues via a *parent* pointer. This is how SELF implements inheritance. **Inheritance in SELF allows objects to share behavior, which in turn allows the programmer to alter the behavior of many objects with a single change. For instance, a [Cartesian] point object would have slots for its non-shared characteristics: x and y. Its parent would be an object that held the behavior shared among all points: +, etc.**

In SELF, there is no direct way to access a variable: instead, objects send messages to access data residing in named slots. So to access its "x" value, a point sends itself the "x" message. The message finds the "x" slot, and evaluates the object found therein.... In order to change contents of the "x" slot to, say, 17, instead of performing an assignment like "x←17," the point must send itself the "x:" message with 17 as the argument. The point object must contain a slot named "x:" containing the assignment [function].

(a) Using this description, draw the run-time data structure of the SELF version of a (3,4) Point object, as described in the last two sentences of the first paragraph, and its parent object. Assume assignments to the x and y characteristics are permitted.

(b) SELF 's lack of classes and instance variables make inheritance more powerful. For example, to create two points sharing the same "x" coordinate, the "x" and "x:" slots can be put in a separate object that is a parent of each of the two points. Draw a picture of the run-time data structures for two points sharing the same "x" coordinate.

(c) To create a new Point object, the `clone` message is sent to an existing point. The language description continues:

> Creating new objects ... is accomplished by a simple operation, copying.... [In Smalltalk,] creating new objects from classes is accomplished by instantiation, which includes the interpretation of format information in a class. Instantiation is similar to building a house from a plan.

However, cloning an object does not clone its parent.

If a *point* object contains fields `x` and `y`, and methods `x:`, `y:`, `move`, then cloning the object will create another object with two fields and three methods. Each point will have a parent pointer, two fields, and three methods – six entries in all. The SELF point will be twice the size of the corresponding Smalltalk point.

Explain how you would structure a SELF program so that each point can be cloned without cloning its methods?

(d) SELF also allows a *change parent* operation on an object. The parent pointer of an object can be set to point to any other object. (An exception is that the first object must not be an ancestor of the second – we don't want a cycle.)

The change parent operation is useful for objects that change behavior in different states. For example, a window can be in the *visible* or *iconified* (minimized) state. When iconified, mouse clicks and window display work differently than when the window is visible. A window's parent pointer can be set to `VisibleWindow` initially, then changed to `IconifiedWindow` when the "minimized" button is pressed. Another example is a file object that can be in the *open* or *closed* state. The `open` method changes the parent pointer from `ClosedFile` to `OpenFile`.

The change parent operation adds a lot of flexibility to SELF. Can you think of disadvantages of this feature? Describe a possible disadvantage in 2-3 sentences.

5. Multiple Inheritance and Thunks

Most contemporary implementations of C++ use a small function, commonly called a "thunk", in the implementation of multiple inheritance, instead of the offset δ's described in the textbook. The purpose of the thunk is to adjust the `this` pointer. Although some of the mechanics may seem a little more complicated, the thunk implementation technique has some definite advantages.

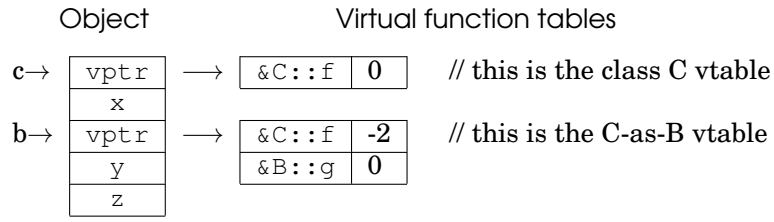
Consider class `C` defined by inheriting from classes `A` and `B`, and the following call to a virtual function:

```
class A { int x; virtual void f(); }
class B { int y; virtual void f();
                 virtual void g(); }
class C : A, B { int z; void f(); }

C *c = new C;
B *b = c;
b->f(); // calls C::f()
```

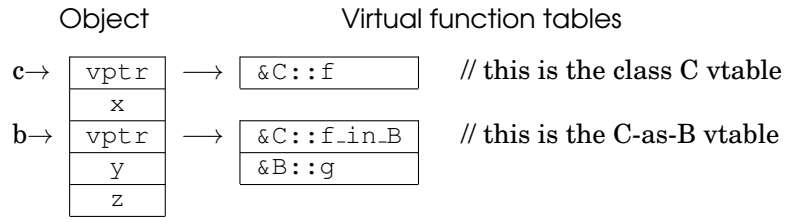
Below are `vtable` illustrations. In understanding those, section 12.5.1 of the textbook will help.

Using the older implementation of multiple inheritance with offsets stored in the `vtable`, the run-time structures associated with this code have the following form:



In this illustration, the first entry in each row of the `vtable` is the address of the function that will be called, and the second entry of the row is an offset that needs to be added to the `this` pointer as part of the calling sequence.

In the alternate thunk-based implementation, the run-time structures look like this:



In this implementation, `C::f_in_B` is a thunk that subtracts 2 from the `this` pointer and then calls `C::f`. Here is pseudo-code for `C::f_in_B`, using a `goto` to avoid the usual calling sequence and creation of an activation record:

```
C::f_in_B(void *this){
    this = this - 2;
    goto C::f;
}
```

This problem asks you to compare the two implementations.

(a) If a C++ program uses only single inheritance, and no multiple inheritance, are offsets (in implementation 1) or thunks (in implementation 2) needed? Explain?

(b) Suppose you built a compiler for C++ programs using only single inheritance, then modified it, using implementation 1, for multiple inheritance. How does the size of each `vtable` change? More specifically, suppose that you compile code defining and using a class. If the `vtable` for this class has n entries under single inheritance, how many entries (pointers or offsets) will it have when compiled using your modified (multiple inheritance) compiler? Is there a difference between objects that use multiple inheritance and objects that don't?

(c) Under the same conditions (modifying a single-inheritance compiler to support multiple inheritance, using the offset-based first implementation), how does the execution time required to call a virtual function change?

(d) Now consider the questions posed in parts (b) and (c) for the second, thunk-based, implementation. How does the number of entries in the `vtable` change for a base class (a class that is not derived from any class)? How does the execution time change for a call to a virtual function of a base-class object? Is there a difference between objects that use multiple inheritance and objects that don't?

(e) Suppose you compile a program defining class `A` using a C++ compiler that does not support multiple inheritance and does not use offsets or thunks. Suppose that you then compile a derived class `B` using another compiler. Class `B` may inherit from `A` and also from other classes in your program. If the second compiler uses offsets, would it be possible to link and run the program sections compiled by two different compilers? What if the second compiler uses thunks?

6. Java Interfaces and Multiple Inheritance

In C++, a derived class may have multiple base classes. In contrast, a Java derived class may only have one base class but may implement more than one interface. This question asks you to compare these two language designs.

(a) Draw a C++ class hierarchy with multiple inheritance using the following classes:

Pizza, for a class containing all kinds of pizza,
Sausage, for pizza that has sausage topping,
Ham, for pizza that has ham topping,
Pineapple, for pizza that has pineapple topping,
Mushroom, for pizza that has mushroom topping,
Hawaiian, for pizza that has ham and pineapple topping,
MeatLover, for pizza that has ham and Sausage topping,
Supreme, for pizza that has everything

(b) If you were to implement these classes in C++, for some kind of pizza manufacturing robot, what kind of potential conflicts associated with multiple inheritance might you have to resolve?

(c) If you were to represent this hierarchy in Java, which would you define as interfaces and which as classes? Write your answer by carefully redrawing your picture, identifying which are classes and which are interfaces. If your program creates objects of each type, you may need to add some additional classes. Include these in your drawing.

(d) Give an advantage of C++ multiple inheritance over Java classes and interfaces and one advantage of the Java design over C++.