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. Finding Shortest Programs

Some optimizers try to find the fastest possible code, and some try to find the shortest sequence of instructions. Code size is often important in practice, and critical for devices like smart cards and embedded processors that have limited memory.

Suppose you are given a Haskell program `shortest` that, given any syntactically correct function as an argument, produces the shortest equivalent function. When we say two Haskell functions are equivalent, we mean that they compute the same partial function. Among several equally short representations, `shortest` chooses the one which is first in lexicographic order.

For example, consider the function that adds one to its argument, twice:

\[
f(x) = x + 1 + 1
\]

We might find that applying `shortest` to \( f \) results in a new function:

\[
f'(x) = x + 2
\]

Can you solve the halting problem using `shortest`? More specifically, can you write a program that takes a Haskell function \( P \) and an integer \( n \) as arguments, and then decides whether \( (P \, n) \) halts?

If you believe that the halting problem can be solved if you are given `shortest`, 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 `shortest`, then explain briefly why you think not.

2. Objects vs. Type Case

With object oriented programming, classes and objects can be used to avoid “type case” statements. Here is a program using a form of case statement that inspects a user-defined type tag to distinguish between different classes of shape objects. This program would not statically type-check in most typed languages since the correspondence between the tag field of an object and the class of the object is not statically guaranteed and visible to the type checker. However, in an untyped language like Smalltalk, a program like this could behave in a computationally reasonable way.

```plaintext
enum shape_tag { s_point, s_circle, s_rectangle };

class point {
    shape_tag tag;
    int x;
```
int y;

point (int xval, int yval)
  { x = xval; y = yval; tag = s_point; }
int x_coord () { return x; }
int y_coord () { return y; }
void move (int dx, int dy) { x += dx; y += dy; }
);

class circle {
  shape_tag tag;
  point c;
  int r;

circle (point center, int radius)
  { c = center; r = radius; tag = s_circle }
point center () { return c; }
int radius () { return radius; }
void move (int dx, int dy) { c.move (dx, dy); }
void stretch (int dr) { r += dr; }
};

class rectangle {
  shape_tag tag;
  point tl;
  point br;

rectangle (point topleft, point botright)
  { tl = topleft; br = botright; tag = s_rectangle; }
point top_left () { return tl; }
point bot_right () { return br; }
void move (int dx, int dy) { tl.move (dx, dy); br.move (dx, dy); }
void stretch (int dx, int dy) { br.move (dx, dy); }
};

/* Rotate shape 90 degrees. */
void rotate (void *shape) {
  switch ((shape_tag *) shape) {
  case s_point:
    case s_circle:
      break;
  case s_rectangle:
    {
      rectangle *rect = (rectangle *) shape;
      int d = ((rect->bot_right ().x_coord ()
        - rect->top_left ().x_coord ()) -
        (rect->top_left ().y_coord ()
        - rect->bot_right ().y_coord ()));
      rect->move (d, d);
      rect->stretch (-2.0 * d, -2.0 * d);
    }
  }
}

(Q) Rewrite this so that instead of rotate being a function, each class has a rotate method,
and the classes do not have a tag.

(b) Discuss, from the point of view of someone maintaining and modifying code, the differences between adding a triangle class to the first version (as written above) and adding a triangle class to the second (produced in part (a) of this question).

(c) Discuss the differences between changing the definition of rotate (say, from 90 degrees to the left to 90 degrees to the right) in the first and second versions. Assume you have added a triangle class so that there is more than one class with a nontrivial rotate method.

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:

```plaintext
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.
<table>
<thead>
<tr>
<th>Activation Records</th>
<th>Closures</th>
<th>Compiled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>r (1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cp (3)</td>
<td></td>
</tr>
<tr>
<td>(1) Point(…)</td>
<td>access link (0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>(( ), •)</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>(( ), •)</td>
</tr>
<tr>
<td>(2) Point part of cp</td>
<td>access link (0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>(( ), •)</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>(( ), •)</td>
</tr>
<tr>
<td>(3) ColorPt(…)</td>
<td>access link ()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>(( ), •)</td>
</tr>
<tr>
<td>(4) cp.distance(r)</td>
<td>access link ()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>q (r)</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\sqrt{(x - q.x)^2 + (y - q.y)^2}
\]

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. Smalltalk Run-time Structures

Here is a Smalltalk class whose instances represent points in the two-dimensional Cartesian plane:

<table>
<thead>
<tr>
<th>class name</th>
<th>Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>superclass</td>
<td>Object</td>
</tr>
<tr>
<td>class variables</td>
<td>comment: none</td>
</tr>
<tr>
<td>instance variables</td>
<td>x y</td>
</tr>
<tr>
<td>class messages and methods</td>
<td>comment: instance creation</td>
</tr>
<tr>
<td>newX: xValue Y: yValue</td>
<td>↑self new x: xValue y: yValue</td>
</tr>
<tr>
<td>instance messages and methods</td>
<td>comment: accessing instance vars</td>
</tr>
<tr>
<td>x: xCoordinate y: yCoordinate</td>
<td>↑</td>
</tr>
<tr>
<td>x ← xCoordinate</td>
<td></td>
</tr>
<tr>
<td>y ← yCoordinate</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>↑x</td>
</tr>
<tr>
<td>y</td>
<td>↑y</td>
</tr>
<tr>
<td>comment: arithmetic</td>
<td></td>
</tr>
<tr>
<td>+ aPoint</td>
<td>↑</td>
</tr>
<tr>
<td>↑Point newX: (x + aPoint x) Y: (y + aPoint y)</td>
<td></td>
</tr>
</tbody>
</table>

This class provides methods for accessing its instance variables and for adding two points.

(a) Complete the top half of the drawing of the Smalltalk run-time structure shown in Figure 1 for a point object with coordinates (3, 4) and its class. Label each of the parts of the top half of the figure, adding additional parts to the drawing as needed.

(b) A Smalltalk programmer has access to a library containing the Point class, but she cannot modify the Point class code. In her program, she wants to be able to create points using either Cartesian or polar coordinates, and she wants to calculate both the polar coordinates (radius and angle) and the Cartesian coordinates of points. Given a point \((x, y)\) in Cartesian coordinates, the radius is \(\sqrt{(x * x) + (y * y)}\), and the angle is \(\tan^{-1}(x/y)\). Given a point \((r, \theta)\) in polar coordinates, the \(x\) coordinate is \(r * \cos(\theta)\) and the \(y\) coordinate is \(r * \sin(\theta)\).

i. Write out the pseudocode for a subclass of Point called PolarPoint in tabular form just like the one shown above for the Point class.

ii. Which parts of Point can you re-use and which parts must you define differently for PolarPoint?

(c) Complete the drawing of the Smalltalk run-time structure by adding a PolarPoint object and its class to the bottom half of Figure 1. Label each of the parts and add additional parts to the drawing as needed.
Figure 1: Smalltalk Run-Time Structures for Point and PolarPoint
5. ........................................... C++ Multiple Inheritance and Casts

An important aspect of C++ object and virtual function table (vtbl) layout is that if class D has class B as a public base class, then the initial segment of every D object must look like a B object, and similarly for the D and B virtual function tables. The reason is that this makes it possible to access any B member data or member function of a D object in exactly the same way we would access the B member data or member function of a B object. While this works out fairly easily with only single inheritance, some effort must be put into the implementation of multiple inheritance to make access to member data and member functions uniform across publicly derived classes.

Suppose class C is defined by inheriting from classes A and B:

```cpp
class A {
    public:
        int x;
        virtual void f();
};
class B {
    public:
        int y;
        virtual void f();
        virtual void g();
};
class C : public A, public B {
    public:
        int z;
        virtual void f();
};
C *pc = new C; B *pb = pc; A *pa = pc;
```

and pa, pb and pc are pointers to the same object, but with different types. The representation of this object of class C and the values of the associated pointers are illustrated in this chapter.

(a) Explain the steps involved in finding the address of the function code in the call pc->f(). Be sure to distinguish what happens at compile time from what happens at run time. Which address is found, &A::f(), &B::f(), or &C::f()?

(b) The steps used to find the function address for pa->f() and to then call it are the same as for pc->f(). Briefly explain why.

(c) Do you think the steps used to find the function address for and to call pb->f() have to be the same as the other two, even though the offset is different? Why or why not?

(d) How could the call pc->g() be implemented?

6. ......................................... Java Interfaces and Multiple Inheritance

In C++, a derived class may have multiple base classes. The designers of Java chose not to allow multiple inheritance. Therefore, a Java derived class may only have one base class. However, Java programs may contain interfaces (roughly, classes without an implementation) and a class may be declared to implement more than one interface. This question asks you to compare these two language designs.

This question asks you to consider the following kinds of movies. For sanity’s sake, the list is non-exhaustive.

- **Movie:** all movies
- **Action:** movies containing lots of explosions
- **Romance:** movies where romantic interest drives the plot
- **Comedy:** movies with largely humorous content
**Mystery:** who-dunnit movies

**Rescue:** hybrid action-romance movies, where the main character attempts to save his or her romantic interest from almost-certain doom

**Romantic Comedy:** hybrid romance-comedy movies with large amounts of both humorous and romantic content

**Hollywood Blockbuster:** action-romance-comedy-mystery movies designed to please crowds

(a) Draw a C++ class hierarchy with multiple inheritance for the above set of classes.

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

(c) If you were to represent this hierarchy in Java, what interfaces and classes would you use? Write your answer by carefully drawing a class/interface hierarchy, identifying which nodes are classes and which are interfaces. Note that there must be a class for each of the movie genres, but you may use any interfaces you require to preserve the relationships between genres. For example, one way of doing this would be to have the Comedy and RomanticComedy genres both implement some kind of IComedy interface.

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

---

8