Homework 6

Due 19 November

Handout 18
CS242: Autumn 2002
12 November

Reading

1. Read Chapters 10 and 11.

Problems

1. .... 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.

```
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 += dy; 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);
      }
  }
}
```

(a) 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.

2. .... 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 on page 306 and 308 respectively. 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 activation record of the parent class.

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

```
real x, y;
x := -1.0;
```

```
y := -2.0;
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

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

Closures

⟨( ), • ⟩

⟨( ), • ⟩

⟨( ), • ⟩

⟨( ), • ⟩

⟨( ), • ⟩

Compiled Code

code for equals

code for distance

code for cpt equals

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

$$\mathtt{sqrt}((x - p.x) * *2 + (y - p.y) * *2)$$

which compares the coordinates of the point containing this `distance` procedure to the coordinate of the point `p` passed as an argument. Explain how the value of `x` is found when `cp.distance(r)` is executed. Mention specific pointers 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.

3. .... Loophole in Encapsulation

A priority queue (also known as a heap) is a form of "queue" that allows the minimum element to be retrieved. Assuming that Simula has a list type, a Simula pq class might look something like this:

```
class pq();
begin
   (int list) contents;
   bool procedure isempty();
       < ... return true if pq empty, else false ... >
   procedure insert(x); int x;
       < ... put x at appropriate place in list ... >
   int procedure deletemin();
       < ... delete first list elt and return it ... >

   contents := nil
end
```

where the sections `< ... >` would be filled in with appropriate executable Simula code. A priority queue object would be created by writing

```
h :- new pq();
```

and the components accessed by writing something like

```
h.contents;
h.isempty;
h.insert(x);
h.deletemin();
```

In ML we can write a function that produces a closure with essentially the same behavior as a Simula priority queue object. This is written below, together with the associated exception declaration.

```
exception Empty;

fun new_pq() =
    let val contents = (ref nil)  : int list ref
        fun insert(x,nil)    = [x]
        |   insert(x, y::l) = if x<y then x::(y::l)
                                     else y::insert(x,l)
    in
        {
         emp_meth = fn () => !contents = nil,
         ins_meth = fn x => (contents := insert(x,!contents)),
         del_meth = fn () => case !contents of
                               nil => raise Empty
                             | y::l => (contents := l; y)
         }
    end;
```

This question asks you to think about a "loophole" in the Simula object system that allows a client program to interfere with the correct behavior of a priority queue object.

(a) The main property of a priority queue is that if n elements are inserted, and k are removed (for any $k < n$), then these k elements are returned in increasing order. Explain in two or three sentences why both the Simula and ML forms of priority queue objects should exhibit this behavior in a "reasonable" program.

(b) Explain in a few sentences how a "devious" program using a Simula priority queue object `h` could cause the following behavior: After inserting `0` and `1` into the priority queue by calls `h.insert(0)` and `h.insert(1)`, the first `deletemin` operation on the priority queue returns some number other than `0`. (This will not be possible for an ML priority queue object.) Do not use any pointer arithmetic or other operations that would not be allowed in a language like Pascal.

(c) Write a short sequence of priority queue operations of the form

```
h :- new pq();
h.insert(0);
h.insert(1);
...
h.deletemin();
```

where the ellipsis (...) may contain `h` operations, but not `insert` or `deletemin`, demonstrating the behavior you described in part b.

4. .... Smalltalk Run-time Structures

Here is a Smalltalk `Point` class whose instances represents points in the two-dimensional Cartesian plane. In addition to accessing instance variables, an instance method allows point objects to be added together.

class name	Point
superclass	Object
class variables	*comment: none*
instance variables	x y
class messages and methods	*comment: instance creation* **newX: xValue Y: yValue** | | ↑ self new x: xValue y: yValue
instance messages and methods	*comment: accessing instance vars* **x: xCoordinate y: yCoordinate** | | x ← xCoordinate y ← yCoordinate **x** | | ↑ x **y** | | ↑ y *comment: arithmetic* **+ aPoint** | | ↑ Point newX: (x + aPoint x) Y: (y + aPoint y)

(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 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 `((x * x) + (y * y)) squareRoot`, and the angle is `(x/y) arctan`. Given a point (r, θ) in polar coordinates, the x coordinate is `r * (`θ` cos)` and the y coordinate is `r * (`θ` sin)`

   i. Write out a subclass, PolarPoint, of Point and explain how this solves the programming problem.

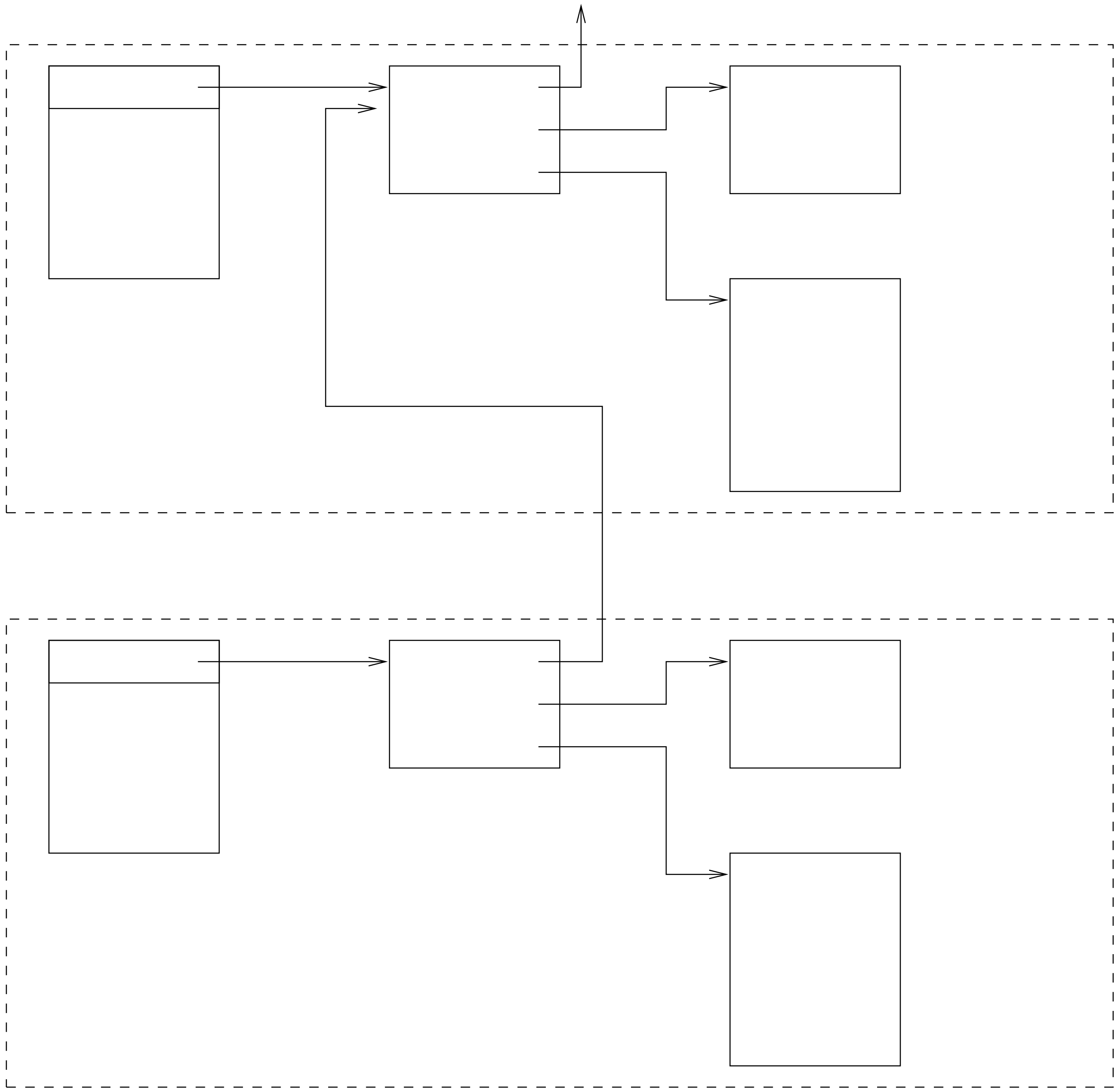

Figure 1: Smalltalk Run-Time Structures for Point and PolarPoint

ii. Which parts of Point could you reuse and which would you have to define differently for PolarPoint?

(c) Complete the drawing of the Smalltalk run-time structure by adding a `PolarPoint` object with coordinates $(3,4)$ and its class to the bottom half of the figure you already filled in with `Point` structures. Label each of the parts and add to the drawing as needed.

5. .... Protocol Conformance

We can compare Smalltalk interfaces to classes using *protocols,* which are lists of operation names (selectors). When a selector allows parameters, as in `at:     put:` , the selector name includes the colons but not the spaces. More specifically, if `dict` is an updatable collection object, such as a dictionary, then we could send `dict` a message by writing `dict at:'cross' put:'angry'.` (This makes our dictionary definition of "cross" the single word "angry.") The protocol for updatable collections will therefore contain the seven-character selector name `at:put:`. Here are some example protocols.

```
             stack : {isEmpty, push:, pop }
             queue : {isEmpty, insert:, remove }
    priority_queue : {isEmpty, insert:, remove }
           dequeue : {isEmpty, insert:,insertFront:, remove, removeLast }
 simple_collection : {isEmpty }
```

Briefly, a stack can be sent the message `isEmpty`, returning `true` if empty, `false` otherwise; `push:` requires an argument (the object to be pushed onto the stack), `pop` removes the top element from the stack and returns it. Queues work similarly, except they are first-in/first-out instead of first-in/last-out. Priority queues are first-in/minimum-out and dequeues are doubly-ended queues with the possibility of adding and removing from either end. The `simple_collection` class just collects methods that are common to all the other classes. We say that the protocol for `A` *conforms* to the protocol for `B` if the set of `A` selector names contains the set of `B` selector names.

(a) Draw a diagram of these classes, ordered by protocol conformance. You should end up with a graph that look's like William Cook's drawing shown in the text.

(b) Describe briefly, in words, a way of implementing each class so that you only make `B` a subclass of `A` if the protocol for `B` conforms to the protocol set for `A`.

(c) For some classes `A` and `B` that are unrelated in the graph, describe a strategy for implementing `A` as a subclass of `B` in a way that keeps them unrelated.

(d) Describe implementation strategies for two classes `A` and `B` (from the set of classes above) so that `B` is a subclass of `A`, but `A` conforms to `B`, not the other way around.

6. .... Subtyping and Binary Methods

This question is about the relationship between subtyping and inheritance. Recall that the main principle associated with subtyping is substitutivity: if `A` is a subtype of `B`, then wherever a `B` object is required in a program, an `A` object may be used instead without producing a type error. For the purpose of this question, we will use *Message not understood* as our Smalltalk type error. This is the most common error message resulting from a dynamic type failure in Smalltalk; it is the object-oriented analog of the error *Cannot take car of an atom* that every Lisp programmer has generated at some time. Remember that Smalltalk is a dynamically-typed language. This question asks you to show how substitutivity can fail, using the fact that a method given in a superclass can be redefined in a subclass.

If there are no restrictions on how a method (member function) may be redefined in a subclass, then it is easy to redefine a method so that it requires a different number of arguments. This will make it impossible to meaningfully substitute a subclass object for a superclass object. A more subtle fact is that subtyping may fail when a method is redefined in a way that appears natural and (unless you've seen this before) unproblematic. This is illustrated using the following `Point` class and `ColoredPoint` subclass.

class name	Point
class variables:	
instance variables:	xval yval
instance messages and methods	**xcoord** $\uparrow$ xval **ycoord** $\uparrow$ yval **origin** xval $\leftarrow$ 0 yval $\leftarrow$ 0 **movex: dx movey: dy** xval $\leftarrow$ xval + dx. yval $\leftarrow$ yval + dy **equal: pt** $\uparrow$ (xval = pt xcoord & yval = pt ycoord)
class name	ColoredPoint
class variables:	
instance variables:	color
instance messages and methods:	**color** $\uparrow$ color **changecolor: newc** color $\leftarrow$ newc **equal: cpt** $\uparrow$ (xval = cpt xcoord & yval = cpt ycoord & color = cpt color)

The important part here is the way that `equal` is redefined in the colored point class. This change would not be allowed in Simula or C++, but is allowed in Smalltalk. (The C++ compiler does not consider this an error, but it would not treat it as redefinition of a member function either.) The intuitive reason for redefining `equal` is that two colored points are equal only if they have the same coordinates and are the same color.

Problem: Consider the expression `p1 equal:p2` where p1 and p2 are either Point objects or ColoredPoint objects. It is guaranteed not to produce *Message not understood* if both p1 and p2 are either Points or ColoredPoints, but may produce an error if one is a Point and the other a ColoredPoint. Consider all four combinations of p1 and p2 as Points and ColoredPoints, and explain briefly how each message is interpreted.