Homework 6

Due 19 November

Handout 18
CS242: Autumn 2003
12 November

____

(Print your name)

Prob	# 1	# 2	# 3	# 4	# 5	# 6	# 7	Total
C+								
C								
C-								

Reading

1. Read Chapters 10 and 11.

Problems

1. .... 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. 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:

```
ref(Point) p;
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)	x			
	y			
(1) r →	access link	(0)		
	x			code for
	y		⟨( ), • ⟩	equals
	equals	•		
	distance	•	⟨( ), • ⟩	
(2) Point part of cp	access link	(0)		
	x			code for
	y		⟨( ), • ⟩	distance
	equals	•		
	distance	•	⟨( ), • ⟩	
(3) cp →	access link	( )		
	c		⟨( ), • ⟩	code for
	equals	•		cpt equals
(4) cp.distance(r)	access link	( )		
	q	(r)		

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

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

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 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.

2. .... Subtyping of Refs in Simula

In Simula, the procedure call `assignA(b)` in the following context is considered statically type correct:

```
class A  ... ;    /* A is a class  */
A class B ... ;   /* B is a subclass of A */

ref (A) a;        /* a is a variable pointing to an A object  */
ref (B) b;        /* b is a variable pointing to a  B object  */

proc assignA (ref (A) x)
   begin
      x := a
   end;

assignA(b);
```

(a) Assume that *if* $B <: A$ *then* $ref(B) <: ref(A)$. Using this principle, explain why both the procedure `assignA` and the call `assignA(b)` can be considered statically type correct.

(b) Explain why actually executing the call `assignA(b)`, and performing the assignment given in the procedure, may lead to a type error at run time.

(c) The problem is that the "principle", *if* $B <: A$ *then* $ref(B) <: ref(A)$, is not semantically sound. However, type checking using this principle can be made sound by inserting run-time tests. Explain the run-time test you think a Simula compiler should insert in the compiled code for procedure `assignA`. Can you think of reason why the designers of Simula might have decided to use run-time tests instead of disallowing ref subtyping in this situation? (You don't have to agree with them; just try to imagine what rationale might have been used at the time.)

3. .... 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.

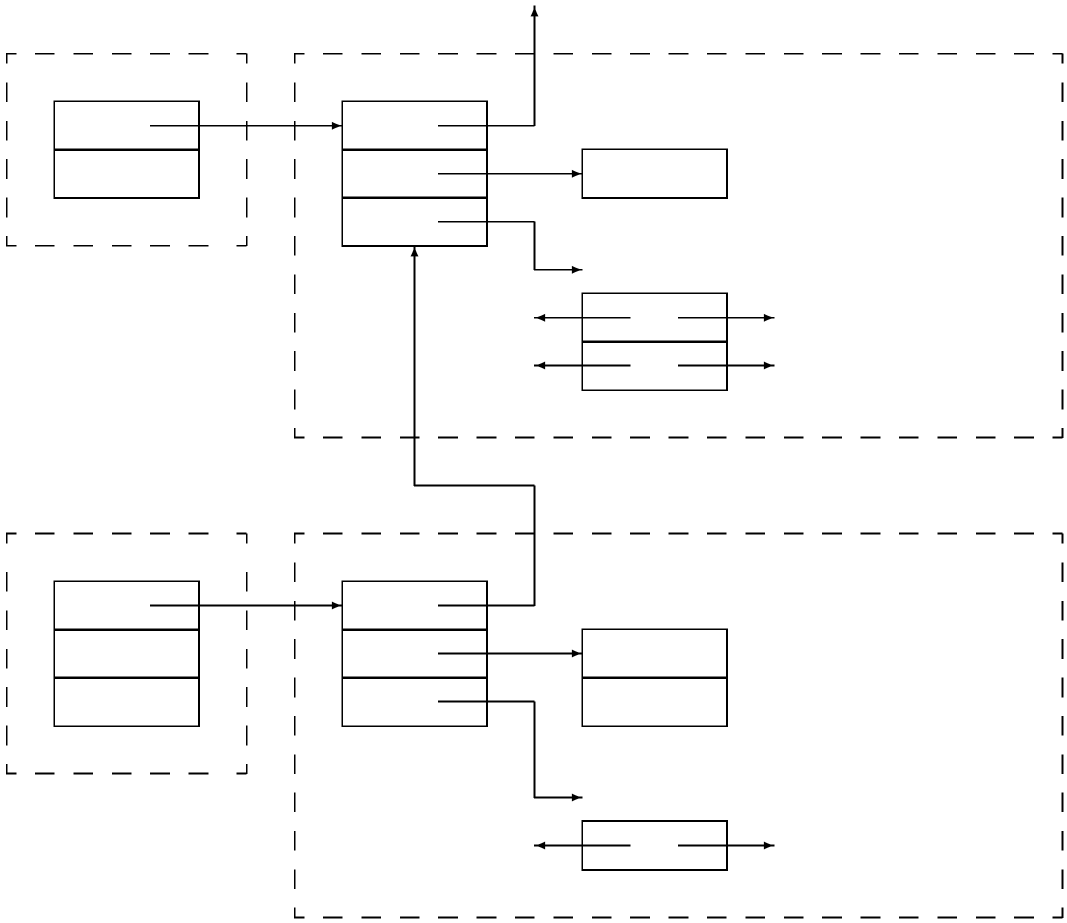

Figure 1: Smalltalk Run-Time Structures for Point and PolarPoint

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 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.

4. .... Removing a Method

Smalltalk has a mechanism for "undefining" a method. Specifically, if a class `A` has method `m` then a programmer may cancel `m` in subclass `B` by writing

```
m:
   self shouldNotImplement
```

With this declaration of `m` in subclass `B`, any invocation of `m` on a `B` object will result in a special error indicating that the method should not be used.

(a) What effect does this feature of Smalltalk have on the relationship between inheritance and subtyping?

(b) Suppose class `A` has methods `m` and `n`, and method `m` is canceled in subclass `B`. Method `n` is inherited and not changed, but method `n` sends the message `m` to `self`. What do you think happens if a `B` object `b` is sent a message `n`? There are two possible outcomes. See if you can identify both, and explain which one you think the designers of Smalltalk would have chosen and why.

5. .... Objective CAML

Objective CAML (OCAML) is an object-oriented extension of the CAML dialect of ML, designed and implemented by researchers at INRIA, the French national computer science research institute.

The 2002 ICFP programming contest challenged each team to implement a program that acts as a player in a multi-player robot game. Each entry played against the others in a tournament, with the winning program declared the winner of the programming contest. The winner was a 1500-line Objective CAML program, beating a 3000-line C program and many others.

Objective CAML has classes, objects, self, initializers (analogous to constructors), virtual methods, and private methods. Here is a simple form of "point" class:

```
class point =
   object
     val mutable x = 0
     method get_x = x
     method move d = x <- x + d
  end;;
```

When this declaration is given to the OCAML compiler, the code is compiled and the output is

```
class point :
  object val mutable x : int method get_x : int method move : int -> unit end
```

This means that class point creates objects that have a mutable (assignable) x field, and methods `get_x` and `move` with the given types given above.

A point `p` is created using `new`, as in `let p = new point`. Type of `p` is `point`, which is an abbreviation for the object interface type `<get_x : int; move : int -> unit>` that lists the methods of class `point` along with their types. The object type does not include the field `x`. Note that although each class name can be used as a type name, each class type is treated as an abbreviation for an interface type. We could eliminate the syntactic convenience of writing class names as type names and write all OCAML programs using interface types such as `<get_x : int; move : int -> unit>`.

Inheritance makes it possible to use one class in the definition of another. For example,

```
class colored_point (c:string) =
   object
     inherit point
     val c = c
     method color = c
   end;;
```

defines a class with all the fields and methods of point, plus those defined in the derived class. The effect is the same as copying the field and method definitions from point into the definition of colored_point.

Questions

(a) Which of the four basic concepts (dynamic lookup, encapsulation, inheritance, and subtyping) are implied by the short description of OCAML given so far (above)? For each concept, say "yes" or "no" and write a short phrase to explain why you think it must be or might not be part of OCAML.

(b) Assume subtyping between OCAML object types is defined along the same lines as subtyping between object interfaces in our discussion of Smalltalk. What is the relationship between subtyping and inheritance?

(c) Given what you know from the presentation of OCAML so far, what kind of method lookup algorithm would you expect? Explain by drawing a parallel between OCAML and at least one of the object-oriented languages we covered in CS242 this quarter. Include a sentence or phrase about the efficiency of this implementation, in comparison with other object-oriented languages.

(d) Since OCAML is based on ML, the types of methods are inferred using type inference and may be polymorphic. However, subtyping is never implicit. If an object x has type t and t <: s then x is coerced to type s by writing t_to_s(x). As in Standard ML, there is no user-defined overloading. Why do you think subtyping requires explicit type coercions in the source code, even if the coercion does not produce any run-time conversion of object represenations?

6. .... Function Subtyping

Assume that $A <: B$ and $B <: C$. Which of the following subtype relationships involving the function type $B \to B$ hold in principle?

i) $(B \to B) <: (B \to B)$
ii) $(B \to A) <: (B \to B)$
iii) $(B \to C) <: (B \to B)$
iv) $(C \to B) <: (B \to B)$
v) $(A \to B) <: (B \to B)$
vi) $(C \to A) <: (B \to B)$
vii) $(A \to A) <: (B \to B)$
viii) $(C \to C) <: (B \to B)$

7. .... "Like Current" in Eiffel

Eiffel is a statically-typed object-oriented programming language designed by Bertrand Meyer and his collaborators. The language designers did not intend the language to have any type loopholes. However, there are some problems surrounding an Eiffel type expression called `like current`. When the words `like current` appear as a type in a method of some class, they mean, "the class that contains this method" To give an example, the following classes were considered statically type correct in the language Eiffel.

```
Class Point
   x : int
   method equals (pt : like current) : bool
      return self.x == pt.x

class ColPoint inherits Point
   color : string
   method equals (cpt : like current) : bool
      return self.x == cpt.x and self.color == cpt.color
```

In `Point`, the expression `like current` means the type `Point`, while in `ColPoint`, `like current` means the type `ColPoint`. However, the type checker accepts the redefinition of method `equals` because the declared parameter type is `like current` in both cases. In other words, the declaration of `equals` in `Point` says that the argument of `p.equals` should be of the same type as `p`, and the declaration of `equals` in `ColPoint` says the same thing. Therefore, the types of `equals` are considered to match.

(a) Using the basic rules for subtyping objects and functions, explain why `ColPoint` should not be considered a subtype of `Point` "in principle."

(b) Give a short fragment of code that shows how a type error can occur if we consider `ColPoint` to be a subtype of `Point`.

(c) Why do you think the designers of Eiffel decided to allow subtyping in this case? In other words, why do you think they wanted `like current` in the language?

(d) When this error was pointed out (by W. Cook after the language had been in use for several years), the Eiffel designers decided not to remove `like current`, since this would "break" lots of existing code. Instead, they decided to modify the type checker to perform whole-program analysis. More specifically, the modified Eiffel type checker examined the whole Eiffel program to see if there was any statement that was likely to cause a type error.

i. What are some of the disadvantages of whole-program analysis? Don't just say, "it has to look at the whole program." Instead, think about trying to debug a program in a language where the type checker uses whole-program analysis. Are there any situations where the error messages would not be as useful as in traditional type checking where the type of an expression depends only on the types of its parts?

ii. Suppose you were trying to design a type checker that allows safe uses of `like current`. What kind of statements or expressions would your type checker look for? How would you distinguish a type error from a safe use of `like current`?