CSE341 Winter 2005 - Final Exam Review Questions

March 13, 2005

Here are a bunch of exam-like questions that are not on the final. Some are probably harder than the actual exam questions, but it's difficult to judge.

1. Consider this Scheme code:

```
(define (add x y) (+ x y))
(define (double x) (* x 2))
(define-syntax quadruple
  (syntax-rules ()
    [(quadruple x)
     (add (double x) (double x))]))
```

   (a) give a use of the quadruple macro that returns a number not divisible by 4

   (b) Assuming Scheme macros are NOT hygienic (even though they are), give a use of the quadruple macro that returns an odd answer.

2. A Scheme program with blanks is below. Assume that f is a "pure" function (always returns the same outputs given the same inputs). Fill in the blanks such that:

   - x ends up holding true if there exists a number $n >= 0$ such that (f n) evaluates to true
   - the program goes into an infinite-loop otherwise.

```
(define (f x) ...) ; some pure body, not a blank to be filled

(define (g k n)
  (if (f n)
      (k #t)
      (h k (+ n 2))))

(define (h k n)
  (if (f n)
```

```
      (g k n)
      (g ____________)))

(define x (let/cc k ___________))
```

3. Write a Smalltalk class-method instance:of: for class Foo such that Foo instance: e1 of: e2 evaluates to true if and only if e1 is an instance of the class-object e2 evaluates to. (As in Java, this includes being an instance of a subclass.) Recall every Smalltalk object accepts the class message and every class-object accepts the superclass message.

4. Suppose you have to produce a Java program but you hate static typing. You devise a plan:

   - Every time you're supposed to write down a type (local variables, fields, method arguments, etc.), you'll write down the same type: Everything
   - You'll implement a translation from your program into one that typechecks but still uses Everything for every type.

   Give an overview of such a translation. Hints:

   - Have every class implement an interface Everything.
   - You'll have to figure out what should be in Everything.
   - You may have to give methods new names.
   - You will have to add methods to classes.

   How could multiple subclassing (i.e., multiple inheritance) make your translation (particularly part 4) more convenient?

5. Which of the following programs runs faster. Explain.

   - ```
     (letrec ([even (lambda (x) (if (zero? x) #t (not (odd  (- x 1)))))]
              [odd  (lambda (x) (if (zero? x) #f (not (even (- x 1)))))])
       (let ([odd (lambda (x) (= 1 (remainder x 2)))])
         (even 10000000)))
     ```
   - ```
     Methods for instances of A:
       even: n
         n = 0 ifTrue:  [^ true]
               ifFalse: [^ (self odd: n - 1) not]
       odd: n
         n = 0 ifTrue: [^ false]
               ifFalse:  [^ (self even: n - 1) not]

     Methods for instances of B, which subclasses A:
     ```

```
  odd: n
    ^ (n rem: 2) = 1

(B new) even: 10000000
```

6. Recall Java has static overloading, but does not allow two methods with the same argument types and different return types. Here's a proposed relaxation of this restriction:

   - You can define two methods in a class with the same name and argument types, but different return types.
   - But you can only call such methods when "initializing a variable", for example: T x = m(e1,...,en);
   - The method called depends on the declared type of the variable (T in the previous step).

   Explain why this proposal does not always work out. That is, explain what is ambiguous about it and why there's not a very good way to resolve the ambiguity.

7. For each of the following questions, determine under what conditions it is sound for the first type to be a subtype of the second:

   (a) When is $\tau_1 \to \tau_2$ a subtype of $\tau_3 \to \tau_4$?

   (b) When is $\tau_a \to \tau_b \to \tau_c$ a subtype of $\tau_1 \to \tau_2 \to \tau_3$?

   (c) When is $(\tau_a \to \tau_b) \to (\tau_c \to \tau_d)$ a subtype of $(\tau_1 \to \tau_2) \to (\tau_3 \to \tau_4)$?

8. (Picking on Java) This program type-checks and runs:

```
class C {
  public static void f(Object x, Object arr[]) {
    arr[0] = x;
  }
  public static void main(String args[]) {
    Object o = new Object();
    C []   a = new C[10];
    f(o, a);
  }
}
```

   (a) For this program, where does the type-checker use subsumption? From what type to what type? What is Java's subtyping rule for arrays?

   (b) Does this program execute any downcasts when it runs? What happens when it runs?

(c) Informally, what is the semantics of array-update in Java? (Start your answer with, "Array update takes an array-object a, an index i, and an object o…". Discuss what exceptions might be thrown under what conditions and what occurs if no exceptions are thrown.)

(d) Is it possible to compile a Java program without run-time type information, even if the program has no downcasts, method overriding, or reflection? (Note that compilation must preserve the behavior you described in the previous question.)

9. Assume a class-based OO language where "subclassing *is* subtyping" and there is no static overloading. Consider a class C, a class D that extends C, and a client P that uses classes C and D. Now consider each of these potential source-code changes to class C:

   (a) We add a method f to C.

   (b) We take an existing method f of C and change f from taking one argument of type T_1 to taking one argument of type T_2, where $T_1 \leq T_2$.

   (c) We take an existing method f of C and change f from taking one argument of type T_1 to taking one argument of type T_2, where $T_2 \leq T_1$.

   (d) We take an existing method f of C and make it abstract (removing its implementation, but still requiring all objects of type C to have it).

   For each of these changes:

   - Describe the conditions under which D will no longer typecheck. That is, describe all D where the definition of class D should type-check before the change of C but should not type-check after the change.
   - Describe the conditions under which P will no longer typecheck. That is, describe all P where the code for P should type-check before the change of C but should not type-check after the change.

10. Consider an OO language with public fields. (Whether the language has classes or not is irrelevant.)

    (a) Explain how we can translate programs in this language into a similar one where all fields are private. (Hint: Add two methods per field to every object. Explain how to change method bodies to use these fields.)

    (b) Explain how we can translate programs in the language with private fields into one with *no fields at all*, but with *method update*. Method update, written `o.m := mbody`, mutates an object `o` such that its method name `m` is bound to `mbody`. Here is a silly example: This method changes `o.m` to multiply its argument by n and "optimizes" the case $n == 2$:

```
unit f(C o, int n) {
   if(n==2) then o.m := int m(int x) { x + x; }
   else o.m := int m(int x) { x * n; }
}
```