# CSE341 Winter 2005 - Final Exam Review Questions (Solutions)

March 15, 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.
   *(quadruple (begin (set! x (+ x 1)) x)) ; x some number*

   (b) Assuming Scheme macros are NOT hygienic (even though they are), give a use of the quadruple macro that returns an odd answer.
   *(let ([add (lambda (x y) 1)]) (quadruple 0))*

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 __________))
```

*(g k (- n 1))))*
*(let/cc k(g k 0))*

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.

   *Send e1 the class message; put the result in a local tmp. Then use a while-loop to compare the result with e2. On each iteration, tmp := tmp superclass. Return false when tmp is Object (and e2 isn't).*

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?

   *Make an interface or abstract class Everything that has every method in the whole program. (Details: (1) remove duplicates. (2) systematically rename methods that have the same name and argument types but different return types (i.e., one returns void and the other returns Everything. An alternate solution makes every method return a value (e.g., null)). Change every class to have every method by creating methods that throw MessageNotUnderstood.*

*Subclassing makes this a bit easier (in particular, inheriting from a class that that throws on every message it's sent works fine).*

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
     ```

   *Smalltalk is faster because of late-binding.*

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.

   *Consider the following java code:*

```
class A {
      Foo m();
```

```
    Bar m();
  }
  ...
   A a = new A();
   Object x = a.m();
```

*How would you resolve which method m to call??*

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 $\tau_1 \rightarrow \tau_2$ a subtype of $\tau_3 \rightarrow \tau_4$?
   *When $\tau_3 \leq \tau_1$ and $\tau_2 \leq \tau_4$*

   (b) When is $\tau_a \rightarrow \tau_b \rightarrow \tau_c$ a subtype of $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$?
   *When $\tau_1 \leq \tau_a$ and $\tau_2 \leq \tau_b$ and $\tau_c \leq \tau_3$*

   (c) When is $(\tau_a \rightarrow \tau_b) \rightarrow (\tau_c \rightarrow \tau_d)$ a subtype of $(\tau_1 \rightarrow \tau_2) \rightarrow (\tau_3 \rightarrow \tau_4)$?
   *When $\tau_a \leq \tau_1$ and $\tau_2 \leq \tau_b$ and $\tau_3 \leq \tau_c$ and $\tau_d \leq \tau_4$*

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?
   *In main, the second argument (a) in the call to f subsumes C[] to Object[]. Javas rule for subtyping arrays is C[]<D[] if C<D.*

   (b) Does this program execute any downcasts when it runs? What happens when it runs?
   *The program does not execute any downcasts, implicit or explict. Running it causes an uncaught ArrayStoreException to be thrown.*

(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.)

*Array update takes an array-object a, an index i, and on object o. The array-object a has a length and an element-type, both chosen when the object is constructed. For example, the expression new C[10] constructs an array with length 10 and element-type C. If i is greater than the length, an ArrayBoundsException is thrown. If o has a different run-time type than as element type, an ArrayStoreException is thrown. Else, the i th element of the array is mutated to refer to o.*

(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.)

*No. Implementing array-update requires, at run-time, the type of the arrays elements and the type of the object being assigned to an array element. (This would not be necessary if array subtyping was invariant!)*

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 $C$.

*The class D will not type-check if D defined a method named f with a type that is not a subtype of the type of f added to C. A client P will always continue to typecheck. Note that it is only partially correct to say that D will fail to typecheck if it defined a method f at a different type than that added to C. As long as Ds existing f method is at a subtype of the new f in C everything will be cool.*

(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 $T_1 \leq T_2$.

*The class D will not type-check if it overrode f with a method taking an argument of type $T_1$. A client P will always continue to typecheck.*

(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 $T_2 \leq T_1$.

*The class D or client P will not typecheck if they used method f of C for arguments that are strict supertypes of $T_2$. Note that if class D overrides method f, the overriding can still soundly typecheck though many languages do not allow subtyping on overriding.*

(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).

*The class D or client P will not typecheck if C was previously a concrete class and they used this information. Specifically, calling Cs constructor should not longer typecheck.*

*Furthermore, if D was concrete but no longer is (because D does not override f), then calling Ds constructor should no longer typecheck. Finally, an explicit resend (super call) in D to Cs f method must no longer typecheck.*

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.)

    *If class C declares a field x of type T, add methods:* $T\ \ get\_x()\{x\}$ *and* $unit\ \ set\_x(T\ y)\{x := y\}$. *Replace all field accesses e.x (or at least non-self field accesses) with e.get_x(). Replace all field assignments e.x := e' (or at least non-self field assignments) with e.set_x(e').*

    (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; }
}
```

    *Assume we have replaced all field accesses and field assignments (including self accesses and assignments) as described above. Then the only use of fields are in the get and set methods (and constructors), which we change as follows: In the constructor used to initialize field x to v, we instead have it update (i.e., initialize) get_x to* $T\ \ get\_x()\{v\}$. *We also have it define:*

```
unit set_x(T y) {
     self.get_x := T get_x() { y }
}
```

*Note that we can still do any other processing (e.g. keeping access counts) in these accessor functions that we could do before, although the set functions must be careful that the new get functions they put in place will continue to perform the same steps as those put in place by the constructor.*