CSE 341 Final — March 2003 — Answer Key

This exam is closed book and notes. 115 points total.

1. (5 points) What is the difference between final and regular methods in Java? (Circle the one correct answer.)

   (a) Final methods cannot be overloaded by other methods in the same class with different types of parameters

   (b) **Final methods cannot be overriden in subclasses**

   (c) Final methods can only be called by member functions in the same class

   (d) None of the above

2. (5 points) Consider the following definition of a `member` function in Haskell.

```
member x []    = False
member x (y:z) | x==y      = True
               | otherwise = member x z
```

   Which of the following would be legal type declarations for `member`? Circle all that are legal.

   (a) `a -> [a] -> c`

   (b) `a -> [a] -> Bool`

   (c) **Integer − > [Integer] − > Bool**

   (d) **Eq a => a − > [a] − > Bool**

3. (10 points) Discuss two reasons **inner classes** are useful for defining iterators in Java.

   An inner class is a class declared within the scope of another class. One reason they are useful for defining Java iterators is that an instance of a (non-static) inner class has direct access to the fields of the parent instance. These fields would hold the state of the collection. Another reason is that the name of the iterator is known only within the parent class, so that it doesn't clutter up the package name space.

4. (10 points) Consider the following Java class definitions. (These compile correctly.)

```
class C1 {

  public void speak() {
    System.out.println("this is the speak method in C1");
    }
  public void sayMore() {
    System.out.println("this is the sayMore method in C1");
    }
}

class C2 extends C1 {
  public void speak() {
    System.out.println("this is the speak method in C2");
    super.speak();
    this.sayMore();
    }

}

class C3 extends C2 {
  public void sayMore() {
    System.out.println("this is the sayMore method in C3");
    }

}

class Test {

  public static void main(String[] args) {
    C1 a, b, c;
    a = new C1();
    b = new C2();
    c = new C3();
    System.out.println("invoking a.speak()");
    a.speak();
    System.out.println("invoking b.speak()");
    b.speak();
    System.out.println("invoking c.speak()");
    c.speak();
    }
}
```

What gets printed when we run the `main` method in `Test`?

```
invoking a.speak()
this is the speak method in C1
invoking b.speak()
this is the speak method in C2
this is the speak method in C1
this is the sayMore method in C1
invoking c.speak()
this is the speak method in C2
this is the speak method in C1
this is the sayMore method in C3
```

5. (8 points) Suppose we have the following Java declarations:

```
float f1, f2;
int i1, i2;
short s1, s2;
```

For each of the following statements, label it “legal no coercion,” “legal with coercion,” “compile time error,” or “run time error” as appropriate.

   (a) `f1 = s1;` legal with coercion
   (b) `s1 = f1;` compile time error
   (c) `i1 = s1;` legal with coercion
   (d) `s1 = i1;` compile time error

6. (5 points) Which of the following are advantages of static typing? (Circle all that apply.)

   (a) **Errors can be caught at compile time rather than run time.**

(b) It provides more flexibility than dynamic typing.

(c) **The compiler may be able to generate more efficient code.**

(d) **Type declarations provide machine-checkable documentation.**

7. (6 points) Write a Scheme expression to pick the `squid` out of the list `s`. For example, if `s` is bound to `’(squid clam octopus)`, then you would write `(car s)`.

   (a)
   ```
   (define s ’(tuna haddock squid clam))

   (caddr s)

   or alteratively:

   (car (cdr (cdr s)))
   ```

   (b)
   ```
   (define s ’(tuna (haddock) (squid) clam))

   (caaddr s)

   or alteratively:

   (car (car (cdr (cdr s))))
   ```

   (c)
   ```
   (define s ’((tuna haddock) (squid clam)))

   (caadr s)

   or alteratively:

   (car (car (cdr s)))
   ```

8. (8 points - 2 points each for parts a and b; 4 points for part c) What is the result of evaluating each of the following Scheme expressions? If there is more than one expression, just give the result of evaluating the final expression.

   (a)
   ```
   (define x 100)
   (define y 200)
   (define z 300)
   (let ((x (+ 2 4))
         (y (+ x 1)))
      (+ x y z))

   407
   ```

   (b)
   ```
   (define x 100)
   (define y 200)
   (define z 300)
   (let* ((x (+ 2 4))
          (y (+ x 1)))
      (+ x y z))
   ```

```
313
```

(c)
```
(define x 'y)
(define y 'z)
(define z 7)
(list
   x
   y
   z
   (eval x user-initial-environment)
   (eval y user-initial-environment)
   (eval z user-initial-environment))

(y z 7 z 7 7)
```

9. (10 points) What problem is Pizza's parametric polymorphism trying to solve?

   In Haskell and related languages, one can declare types such as `[a] -> [a] -> [a]`

   (This is the type of the append function.) Here, `a` is a type variable, which can be instantiated to any type.

   The user can't declare types of this sort in Java. This leads to less precise type declarations, less machine-checkable documentation, and more runtime casts. For example, the built-in collection classes in Java (Set, ArrayList, etc) all have Object as the type of the contents — you can't declare that x is a set of strings, or a set of points.

   The parametric polymorphism capability in Pizza is designed to solve this problem, so that one can declare sets of strings, sets of points, and so forth, and have the types all checked and declared at compile time. This parametric polymorphism must also interact correctly with object-oriented inheritance.

10. (20 points) Suppose that the following Haskell code has been loaded.

```
-- a polymorphic tree data type
data Tree a = Leaf a | Node a (Tree a) (Tree a)
            deriving (Eq,Show,Read)

treemap f (Leaf x) = Leaf (f x)
treemap f (Node x left right) = Node (f x) (treemap f left) (treemap f right)

combine_trees f (Leaf x1) (Leaf x2) = Leaf (f x1 x2)
combine_trees f (Node x1 left1 right1) (Node x2 left2 right2)
    = Node (f x1 x2)
           (combine_trees f left1 left2)
           (combine_trees f right1 right2)

tree1 = Node 5 (Leaf 1) (Leaf 2)

tree2 = Node 100 (Leaf 10) (Leaf 20)

bigtree x = Node x (bigtree x) (bigtree x)

member a (Leaf x) = a==x
member a (Node x left right) = a==x || member a left || member a right
```

```
double :: Integer -> Integer
double x = 2*x

twice f x = f (f x)

my_repeat x = x : my_repeat x

my_if True x y = x
my_if False x y = y
```

What is the result of evaluting the following Haskell expressions? Remember that `:type x` asks Haskell to print the type of `x`, so in that case give just the type. Otherwise just give the value. If there is a compile-time type error, or a non-terminating computation, say so. If the result is an infinite data structure, give some initial parts of it. If Haskell would give an error of some sort rather than producing output, say so.

(a) `:type treemap`
```
treemap :: (a -> b) -> Tree a -> Tree b
```

(b) `:type combine_trees`
```
combine_trees :: (a -> b -> c) -> Tree a -> Tree b -> Tree c
```

(c) `:type bigtree`
```
bigtree :: a -> Tree a
```

(d) `treemap double tree1`
```
Node 10 (Leaf 2) (Leaf 4)
```

(e) `combine_trees (+) tree1 tree2`
```
Node 105 (Leaf 11) (Leaf 22)
```

(f) `:type my_repeat`
```
my_repeat :: a -> [a]
```

(g) `:type twice`
```
twice :: (a -> a) -> a -> a
```

(h) `my_repeat 8`
```
[8,8,8,8,8,8,8,8,8,8,8,8,8,8,8 .....
```

(i) `member 10 tree2`
```
True
```

(j) `member 10 (bigtree 0)`
```
non-terminating computation
```

11. (9 points) Suppose that Haskell used call-by-value semantics instead of lazy evaluation. What would the result be of evaluating each of the following Haskell expressions? (Use the function definitions from Question 10.)

(a) `member 0 (bigtree 0)`
```
non-terminating computation
```

(b) `member 10 (bigtree 0)`
```
non-terminating computation
```

(c) `my_if (1==0) (1/0) (1+4)`
```
run-time divide-by-zero error
```

12. (9 points) Consider the following Java code fragments. (The first 3 lines are the same for all of them; it's just the last line that is different.) In each case, does the code compile correctly? If so, does it execute without error, or is there an exception?

```
String[] strings = new String[10];
Object[] objects;
String s;
objects = strings;
strings[0] = "hello sailor";
s = (String)objects[0];

executes without errror

String[] strings = new String[10];
Object[] objects;
objects = strings;
objects[0] = new Integer(100);

gives a runtime error (array store exception)

String[] strings;
Object[] objects = new Object[10];
String s;
strings = objects;
objects[0] = "hello sailor";
s = strings[0];

gives a compile time error:

T.java:7: incompatible types
found   : java.lang.Object[]
required: java.lang.String[]
        strings = objects;
```

13. (10 points) Joe Mocha is defining an interface `Unionable` in Pizza that includes an `union` method. Joe then defines two classes, `MySet` and `MyDictionary`, which both implement `Unionable`. He wants Pizza's type system to accept an expression that finds the union of two sets, or the a union of two dictionaries, but not the union of a set and a dictionary.

Here is his definition of `Unionable` and `MySet`:

```
interface Unionable {
  Unionable union(Unionable a);
}

class MySet implements Unionable {
  public MySet union(MySet s) {....}
}
```

Unfortunately, pizza complains that `MySet` *doesn't* implement the `Unionable` interface. Why not?

The problem is that the definition of `union` in `Unionable` requires that the parameter and return type both be of type `Unionable`, but in `MySet` they are of type `MySet`. In Java, to implement a method declared in an interface (or to override a method declared in a superclass), the types of the parameters and return type must match.

So Joe tries again:

```
interface Unionable {
  Unionable union(Unionable a);
}

class MySet implements Unionable {
  public Unionable union(Unionable s) {....}
}
```

This definition compiles correctly ... but there is still a problem. What is the problem?

The problem is that this definition will let us take the union of a set and a dictionary (since either would be OK as the parameter).

What is a correct definition for both `Unionable` and `MySet`. (As in the definition above, just put ... in the body of the method — we only care about the header.)

```
interface Unionable<elem> {
  elem union(elem a);
}

class MySet implements Unionable<MySet> {
  public MySet union(MySet s) {....}
}
```