CSE 341 Final — June 13, 2002 — Answer Key

This exam is closed book and notes. 10 points per question — 130 points total.

1. Consider static vs. dynamic typing. Discuss one advantage of each.

   In a statically typed language, the system is able to determine the type of each expression at compile time. This usually requires that the programmer provide type declarations for each variable, although a few languages, such as Miranda, also support type inference. In a dynamically typed language, the types of expressions (including variables) are not known until runtime.

   An advantage of static typing is that there is more checking at compile time — it is usually better to find errors (such as type errors) earlier than later. Also, if the programmer provides declarations, these are machine-checkable documentation. (Unlike comments, the compiler will flag an error if they become obsolete.)

   An advantage of dynamic typing is that it is less work for the programmer, and better supports rapid prototyping, in which decisions about the types of variables may change rapidly. Another advantage (OK, you only needed to give one, but here's another) is that it may be easier to implement heterogeneous data structures. For example, in Scheme one can have a list of integers and strings, while in Miranda this can't be done directly, but would require a special user-defined type.

2. Suppose that we had a version of Miranda, called V-Miranda, that used **call by value** rather than lazy evaluation. Would V-Miranda offer **referential transparency**? Answer "yes," "no," or "more information needed," and **explain why**.

   Yes, V-Miranda would offer referential transparency. Roughly, referential transparency means that equals can be substituted for equals. For example, if we have defined that x=y+3, in a referentially transparent language such as Miranda we can substitute y+3 for x anywhere in the program. This property continues to hold in V-Miranda — lazy vs. call by value does not affect it, since it stems from the purely functional nature of the language (no side effects) rather than the calling convention.

   (As an aside — not expected in your answer, Miranda and V-Miranda *will* have different semantics, since there are some programs that terminate correctly in Miranda, and that will give an error or not terminate in V-Miranda. However, this property is separate from referential transparency.)

3. Consider the following program in an Algol-like language.

```
begin
integer n;
procedure p(k: integer);
    begin
    k := k+10;
    n := n+k;
    print(n,k);
    end;
n := 20;
p(n);
print(n);
end;
```

(a) What is the output when `k` is passed by **value**?
50, 30
50

(b) What is the output when `k` is passed by **value-result**?
50, 30
30

(c) What is the output when `k` is passed by **reference**?
60, 60
60

4. What does the following Java program print?

```
import java.awt.Point;
class Test {
    public static void main(String[] args) {
        Point p = new Point(10, 20);
        System.out.println("before resetOne: p.x = " + p.x);
        resetOne(p);
        System.out.println("after resetOne: p.x = " + p.x);

        p = new Point(10, 20);
        System.out.println("before resetTwo: p.x = " + p.x);
        resetTwo(p);
        System.out.println("after resetTwo: p.x = " + p.x);
    }

    public static void resetOne(Point q) {
        q.x = 0;
        q.y = 0;
    }

    public static void resetTwo(Point q) {
        q = new Point(0,0);
    }
}
```

```
before resetOne: p.x = 10
after resetOne: p.x = 0
before resetTwo: p.x = 10
after resetTwo: p.x = 10
```

5. Control structures in Java and Smalltalk.

   (a) Compare how **conditionals** are handled in Java and Smalltalk.

   In Java, there is special language syntax for conditionals. In Smalltalk, there is no special syntax. Rather, conditionals are provided using blocks and message sending (by sending the messages `ifTrue:`, `ifFalse:`, and `ifTrue:ifFalse:` to a boolean with blocks as arguments.

   (b) Compare how **iterating through a collection** is handled in Java and Smalltalk.

   In Java, to support iterating through a collection, the collection provides an `iterator` method that returns an iterator over that collection. The iterator is a separate object that includes `next` and `hasNext` methods, to return the next element in the collection and answer whether there are any more elements. It keeps track of which element in the collection should be returned next. In Smalltalk, collections provide a method `do:` that takes a block as an argument. The block is called with each element in the collection. A separate iterator object is not required.

6. What is the principal problem with Java's type system that Pizza is designed to solve? Include an example in your description that illustrates the problem as it exists in Java, and that shows how it is solved by Pizza. (You don't need to write down code — just describe the example in English.)

   The problem in Java is that there are no parameterized types. Thus, for the collection classes in Java (e.g. Set), one can only declare that there is a collection of Objects, not a collection of Points, of Strings, etc. (Well, you could declare separate collections for each type, but that would result in a lot of redundant code.) As a result, even if the programmer is using a set of Points, the Java compiler doesn't know this — only that the set contains Objects. As a result, the programmer must usually include casts when extracting objects from collections, resulting in less clear code.

   Pizza solves this by providing parameterized types. For example, in Pizza one can declare a collection such as `Set<A>`. This allows the programmer to declare a variable as being of type `Set<Point>`, and the type system will know that the elements in this set are of type `Point`.

7. Discuss the primary reason **inner classes** are useful for defining iterators in Java.

   An iterator is usually defined in terms of the specific data representation and other non-public members of a collection class. Since an inner class can access non-public members of the containing class, inner classes offer a clean, controlled way of exposing non-public members to a privileged class (in the absence of a keyword like C++'s friend). Another benefit is keeping the global namespace from getting cluttered with iterator class names. For example, the name `VectorIterator` would only be known within the class `Vector`.

8. Consider the following Smalltalk class definitions of three classes.

```
Object subclass: #ClassOne
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

test
  Transcript show: 'test - ClassOne'.
```

```
ClassOne subclass: #ClassTwo
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

test
  Transcript show: 'test - ClassTwo'.
  super test.
  self snark.

snark
  Transcript show: 'snark - ClassTwo'.
```

```
ClassTwo subclass: #ClassThree
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

test
  Transcript show: 'test - ClassThree'.
  super test.

snark
  Transcript show: 'snark - ClassThree'.
```

What is printed when each of the following expressions is evaluated?

(a) `ClassOne new test`

```
test - ClassOne
```

(b) `ClassTwo new test`

```
test - ClassTwo
test - ClassOne
snark - ClassTwo
```

(c) `ClassThree new test`

```
test - ClassThree
test - ClassTwo
test - ClassOne
snark - ClassThree
```

9. Define a **recursive** Scheme function `double-list` that takes a list of numbers as an argument, and returns a new list of numbers, each twice the number in the original. For example, `(double-list '(1 2 3))` should return `(2 4 6)`. (This version should **NOT** use `map`.)

```
(define (double-list s)
  (if (null? s)
      ()
      (cons (* 2 (car s))
            (double-list (cdr s)))))
```

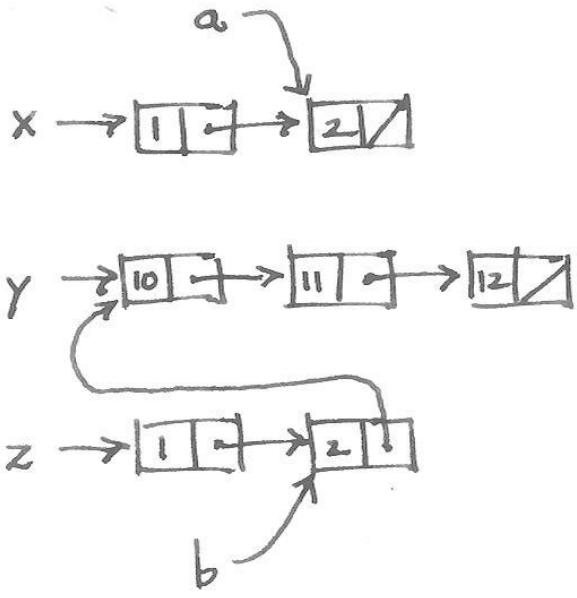
10. Define another version of `double-list` that is **NOT** recursive, using `map` and `lambda`.

```
(define (double-list s)
  (map (lambda (x) (* 2 x)) s))
```

11. Suppose we evaluate the following Scheme expressions:

```
(define x '(1 2))
(define y '(10 11 12))
(define z (append x y))
(define a (cdr x))
(define b (cdr z))
```

Draw a box-and-arrow diagram of the lists that `x`, `y`, `z`, `a`, and `b` are bound to, being careful that your diagram clearly shows what parts of the lists are shared, if any.



12. Consider the following function definitions in Scheme.

```
(define x 2)
(define y 3)

(define (octopus x)
  (+ x y))

(define (squid x y)
   (octopus 10))

(define (mollusc x)
   (lambda (y) (+ x y)))

(define (crab z)
   ((mollusc 10) z))

(define (complicated x)
   ((mollusc 10) x))
```

What do the following expressions evaluate to? (They all evaluate without error.)

(a) `(+ x y)`
`5`

(b) `(octopus 20)`
`23`

(c) `(squid 100 200)`
`13`

(d) `(crab 20)`
`30`

(e) `(complicated 20)`
`30`

13. True or False? (Circle one for each.)

(a) True or False: In both Smalltalk and Java, 3 is an instance of a class. **False**.

(b) True or False: Scheme, Java, and Smalltalk all pass parameters **by reference**. **False**.

(c) True or False: It is **NOT** possible to pass a function as a parameter in Scheme, because Scheme will evaluate the function **before** it is passed. **False**.

(d) True or False: In Smalltalk, the programmer **can** modify methods of the built-in classes that come with the system. **True**.

(e) True or False: In Java, the programmer **can** modify methods of the built-in classes that come with the system. **False**.