CSE 341 Midterm

(50 points total)
Winter 1999

Name: ______________________________

Section: ______________________________

1. (6 points) In a Scheme evaluator, the basic functions can be described as either primitive or special forms. What is the difference between these two kinds of functions?

   The most important distinction between *primitive functions* and *special forms* is that primitive functions are called by first evaluating all of the arguments to the function then invoking the function on the evaluated arguments. When a special form is invoked, on the other hand, none of the arguments are evaluated. The special form can pick and choose which of the arguments it wants to evaluate as it runs.

2. (5 points) In Java, an instance of class `Vector` is an indexed data structure that can be used to hold a collection of items. Suppose we have a class `Ball` and we want to store `Ball` instances in a `Vector`. We can do this as follows:

   ```
   Ball b1 = new Ball();
   Vector balls = new Vector();
   balls.addElement(b1);
   ```

   We can extract an element of the `Vector` as follows:

   ```
   Ball b2 = (Ball) balls.elementAt(0);
   ```

   The question is, why is the notation "`(Ball)`" needed in this assignment statement?

   The Java compiler does static type checking. This means that the left hand side of an `=` and the right hand side of the `=` must have the same static type. `Vector.elementAt(int)` is statically typed to return an `Object`; however, it may return any subclass of `Object`. In this case, the cast adds a runtime type check to make sure that the `Object` returned by `elementAt()` is in fact a `Ball` and it tells the compiler to treat the object being cast (in this case, the right hand side of the equation) is dynamically an object of type `Ball`. Thus, the static type of the right hand side and the left hand side of the `=` are the same and the code fragment passes the Java type checker.

3. (4 points) Is the following function tail-recursive? Why or why not? (It shouldn't really matter what this function actually does.)

```
(define (scale stuff by)
   (if (null? stuff)
       '()
       (cons (* (car stuff) by)
             (scale (cdr stuff) by)
       )
   )
)
```

No. It is not tail recursive because the final return value is the result of `cons` not the direct result of a recursive call to `scale`.

4. (5 points) Java implementations rely on garbage collection to reclaim dynamic memory that is no longer in use. C++ implementations, on the other hand, do not provide garbage collection, but instead rely on the programmer explicitly deleting dynamic memory when it is no longer needed. What property of the C++ language makes it difficult or impossible to provide automatic garbage collection? Give a brief, technical reason.

In C++, data is data. In a typical implementation, integers and pointers are the same size. There is no way to tell if a particular pointer sized piece of data is a pointer or some other data (like an integer). Thus, there is no way for a garbage collector to go through known data and determine which pieces of the data are really pointers to other data and which are just numbers. C++ also allows arbitrary pointer operations like pointer arithmetic which mean that a garbage collector cannot make assumptions about pointers even if it could tell which values were pointers.

5. (7 points) Write a recursive Scheme function `last` that yields as its value the last element of its argument. Assume that the argument will be a non-empty list (i.e., don't worry about what happens if `last` is applied to an empty list or an atom. Examples:

```
(last '(a b c))      => c
(last '(a))          => a
```

Your function should use only basic Scheme operations like `if`, `cond`, `car`, `cdr`, and appropriate recursion.

```
(define (last l)
   (if (null? (cdr l))
       (car l)
       (last (cdr l))
   )
)
```

6. (9 points) Here is a pseudo-Algol program

```
begin
   integer i;
   integer array b[3]; // b has elements b[0], b[1], and b[2]

   procedure q(int x)
   begin
      x = x + 2;
      b[i] = 10;
      i = 2;
      x = x + 2;
   end;

   b[0] = 1; b[1] = 1; b[2] = 1;
   i = 1;
   q(b[i]);
end.
```

For each of the following parameter passing mechanisms, execute the program and give the final values of `b[1]` and `b[2]`.

(a) Call by value

`b[1]` 10

`b[2]` 1

(b) Call by reference

`b[1]` 12

`b[2]` 1

(c) Call by value-result

`b[1]` 5

`b[2]` 1

7. (6 points) Consider the following Scheme program

```
(define x 10)
(define (squid y) (+ x y))
(define (clam x) (squid (* 2 x)))
```

(a) What is the result of evaluating

```
(clam 3)
```

using the normal static scoping rules of Scheme?

16

(b) Suppose we had a dialect of Scheme that used dynamic scoping. What is the result of evaluating

```
(clam 3)
```

using dynamic scoping?

9

8. (8 points) Some function definitions in Java *overload* another function definition, some others *override* another definition.

   (a) Give a short Java example that contains overloading but not overriding.

   (b) Give a short Java example that contains overriding but not overloading.

   You should define tiny classes and/or functions as needed in your examples..

   Example of Overloading: Below, the function `foo` is overloaded.

```
public class A
{
   int foo()
   {
      return 0;
   }

   int foo(int i)
   {
      return i;
   }
}
```

   Example of Overriding: Below, `C.bar()` overrides `B.bar()`.

```
public class B
{
   void bar()
   {
      System.out.println("called B.bar()");
   }
}

public class C extends B
{
   void bar()
   {
      System.out.println("called C.bar()");
   }
}
```