Final Review Sample Problems Solutions

For all of these problems, assume the compiler will not change the relative order of instructions within a thread, and only worry about interleaving of code between two threads.

- What is the following function checking? I.e. for what kind of inputs does funcA return true?

```
(define (funcA x)
  (cond
    ((null? x) #f)
    ((integer? x) #t)
    (#t (and (funcA (car x)) (null? (cdr x))))
  )
)
```

The result is easiest to describe as a property of the binary tree x.
funcA returns true when x is an integer, or x is a binary tree where the leftmost path ends with an integer and all branchings to the right are null (the empty list).
So inputs like 2, (2), (((2))) return true. Inputs like (1 2), (1 ()) return false.

- Consider the following code:

```
int data = 0;
int t = 10;
boolean flag = false;
Object m = new Object();
```

Thread 1

```
flag = true;




synchronized(m) {
    data = 10;
}
```

Thread 2

```
if (flag)
    synchronized (m) {
        t = data;
    }



print(t);
```

Here is the code with the vector clock for each step:

Thread 1

```
flag = true;     // (1, 0)

// When T1 locks get T2 clock.
// Note this is the time T2 had
// when unlocked, not current time.
synchronized(m) {
    data = 10;  // (1, 1)
}
```

Thread 2

```
if (flag)    // (0, 1)
    synchronized (m) {
        t = data;    // (0, 1)
    }


// T2 clock increments after unlock
print(t);    // (0, 2)
```

Looking at this, we have a read and a write to "flag" at time (0, 1) and (1, 0), so there is no happens before ordering to these accesses. Therefore, it is a data race.

This execution does not have atomicty. In either of the serial executions we would have data=10, t=10 and we would print 10. With the given interleaving, we would have data=10, t=0 and we print 0.

- Consider the following code:

```
int data = 0;
boolean flag = false;
Object m = new Object();
```

Thread 1

```
flag = true;
synchronize(m) {
    data = 10;
}
```

Thread 2

```
boolean f;
synchronize(m) {
    f = flag;
}
if (f) {
    data = 100;
}
print(data);
```

For this code, there is at least one execution order which has a data race, and at least one execution order which does not have a data race. Find an example of both.

-For no data race, it is easy to find such an execution order (for example, all of T1 runs before any of T2 runs.

-For a data race, consider the following interleaving of instructions:

Thread 1

```
flag = true;      // (1,0)
synchronize(m) {
    data = 10;    // (1, 1)
}
```

Thread 2

```
boolean f;    // (0,1)
synchronize(m) {
    f = flag;     // (0, 1)
}

if (f) {      // (0, 2)
    data = 100; // (0, 2)
}
print(data);      // (0, 2)
```

Here we have two data race: One is for the read and write to flag at times (1, 0), (0, 1) and the other is the write and write to data at times (1, 1) and (0, 2).