CS61A SUMMER 2010
FINAL REVIEW SESSION 2

George Wang, Jonathan Kotker, Seshadri Mahalingam, Steven Tang, Eric Tzeng
Derived from the notes of Chung Wu, Justin Chen and Carolen,
and past CS61A review sessions (Spring 2007)

QUESTION 1.
What are the possible values of x after the following is executed:

```
(define x 10)
(parallel-execute (lambda () (set! x (+ 5 x)) (set! x (* x 3)))
                  (lambda () (if (> x 16)
                                 (set! x 100)
                                 (set! x (- x 20)))))
```

QUESTION 2. (Question 13 of the Final Exam, Spring 2003)
Given the following definitions:

```
(define s (make-serializer))
(define t (make-serializer))
(define x 10)
(define (f) (set! x (+ x 3)))
(define (g) (set! x (* x 2)))
```

Can the following expressions produce an incorrect result, a deadlock, or neither? (By ”incorrect result” we mean a result that is not consistent with some sequential ordering of the processes.)

(a) (parallel-execute (s f) (t g))

(b) (parallel-execute (s f) (s g))

(c) (parallel-execute (s (t f)) (t g))

(d) (parallel-execute (s (t f)) (s g))

(e) (parallel-execute (s (t f)) (t (s g)))

QUESTION 3.
Which of the following interactions will execute faster or the same in the analyzing evaluator than in the original metacircular evaluator? Circle FASTER or SAME for each.

```
> (define (gauss-recur n) ;; sum of #s from 1 to n
    (if (= n 1)
        1
        (+ n (gauss-recur (- n 1)))))
> (gauss-recur 1000)
```

Analyzing will be: FASTER SAME

```
> (define (gauss n)
    (/ (* (+ n 1) n) 2))
> (gauss 1000)
```

Analyzing will be: FASTER SAME

QUESTION 4.
What are the first seven terms of the following stream definition?

```
(define mystery (cons-stream 1
                  (cons-stream 2
                    (stream-map (lambda (x y) (+ x (* 2 y)))
                                mystery
                                (stream-cdr mystery)))))
```

________ ________ ________ ________ ________ ________ ________

QUESTION 5.

Write code to generate **cxr-stream**, the stream of all the possible combinations of **car** and **cdr:**

```
(car cdr caar cdar cadr cddr …)
```

Each element of the stream is a *procedure*, so that, for example, we can write statements such as

```
((stream-ref cxr-stream 2) '((4 5) (foo bar)))
```

that would work (and in this case, return **4**). You may find the **compose** and **interleave** functions useful here.

QUESTION 6.

Ben Bitdiddle has conveniently defined stream-accumulate for you below:

```
(define (stream-accumulate combiner null-value s)
   (if (stream-null? s)
        null-value
        (combiner (stream-car s)
                  (stream-accumulate combiner null-value
                                     (stream-cdr s)))))
```

What happens when we do:

```
(a) (define foo (stream-accumulate + 0 integers))
```

```
(b) (define bar (cons-stream 1 (stream-accumulate + 0 integers)))
```

```
(c) (define baz (stream-accumulate (lambda (x y) (cons-stream x y))
                                   the-empty-stream
                                   integers))
```

QUESTION 7.
We would like to implement a cheating detection system for tests. We start by simulating a row of test-takers by a vector, where each element in the vector is the answers to each student's test. These answers are also simulated by a vector, where each index corresponds to a problem number, and the element at the index is the student's solutions to the problem. For example:

```
#( #('cs61a  8 'none)    ;; student 0 answered 'cs61a, 8, 'none
   #('mother 6 'a)
   #('cs61a  5 'b)    )
```

(The answers are aligned here for your benefit.) Here, the test requires three answers, and student 0 answered 'cs61a, 8, and 'none to the three questions. Student 1 is the only person with two neighbors.

Two students are suspected of cheating if they have at least half of the answers identical to a neighbor. So if there are three questions on the test, if two are identical in consecutive students, the students are suspected of cheating. Write **catch-cheaters** that takes in a vector of vectors and returns the index of the first student suspected of cheating. (So if students 1 and 2 cheated off each other, return 1.)

QUESTION 8.
At lines A, B, C, and D, how many times have + and * been called in lazy evaluation and in applicative order evaluation?

```
(define (foo n m)
   (if (> n 10)
       (begin (display m) n)
       (begin (display n) m)))

Lazy> (define z (* 8 4))
A
Lazy> (define x ((lambda (x) x) (+ 2 2)))
B
Lazy> (define y (foo z x))
C
Lazy> y
D
```

QUESTION 9.
Consider the following interactions in the lazy evaluator:

```
     (define w 100)
     (define (foo x y) (x y))
     (define q (foo (lambda (z) (set! w 50) z)
                    (begin (set! W 10) 3))))
```

What are the values of the following statements typed at the prompt immediately after?

(a) w

(b) q

(c) w

QUESTION 10.
A *magic square* is an arrangement of numbers in a square matrix, where the sum of each row, each column, and each main diagonal is the same. For example, here is a 3x3 magic square:

```
2 7 6 (For the 3x3 square, each row, column, and diagonal sums to 15)
9 5 1
4 3 8
```

Write a procedure called **magic-square** that uses the nondeterministic evaluator to find 3x3 magic square configurations.

In this implementation, a magic square is a list of lists, where each list is associated with a row of the magic square. So, for example, the magic square above is represented as

```
(list (list 2 7 6) (list 9 5 1) (list 4 3 8)).
```

If it helps, you may assume the **distinct?** procedure from your homework. Also, the **=** primitive can take more than two arguments, and returns true if all of the arguments are equal. Thus,

```
(= 3 3 3) is true, while
(= 3 3 4) is false.
```

QUESTION 11.

Consider the following Scheme program:

```
(let ((a (amb 1 2 3))
      (b (amb 4 5 6)))
  (display "hello")
  (require (= b (* a 2)))
  a)
```

How many times will *hello* be printed? What is the return value?

QUESTION 12.

What are the results of the following statements when entered into the nondeterministic evaluator? Write down all of the results after multiple **try-again** statements, until the evaluator claims that there are no more solutions.

(a) `(amb 1 2 3)`

(b) `(amb (list 1 2 3))`

(c) `(amb 1 (amb 2 (amb 3)))`

(d) `(amb (amb 1) (amb 2) (amb 3))`

(e) `(amb (amb 2 3) 1 (amb 4))`

(f)
```
(define (foo x)
  (cond ((not (pair? x)) (amb))
        ((word? (cdr x)) (cdr x))
        (else (amb (foo (car x))
                   (foo (cdr x))))))

(foo '(a (b c) (d e . f) (g (h . i) j) k))
```

QUESTION 13.
Rotating lists is fun, so let's keep doing it! For the following, assume that only the rule **append** has been defined, as in the lecture.

Implement a rule **rotate-forward** so that

```
(rotate-forward (1 2 3 4) ?what) ==> ?what = (2 3 4 1).
```

That is, the second list is the first list with the first element attached to the end instead. Assume the list is non-empty.

Let's get both sides of the story. We'd like a rule **rotate-backward** so that

```
(rotate-backward (1 2 3 4) ?what) ==> ?what = (4 1 2 3)
```

That is, the second list is the first list with the last element attached to the front instead. You may define other helper rules if you'd like.

QUESTION 14.
Write a rule or rules to determine if one integer is less than another. For example, the query

```
(less ?x (a a a))
```

should give the results

```
(less () (a a a)) (less (a) (a a a)) (less (a a) (a a a))
```