CS 61A, Fall 1997
Final
Professor Harvey

Problem #1

Given these definitions:

```
(define (square x) (* x x))

(define (inc) (set! count (+ count 1)) count)

(define count 5)
```

What is the value of the expression (square (inc)) under applicative order evaluation?

What is the value of the same expression under normal order evaluation (without memoization)?

Problem #2

True or false: If f is Theta(g) then f + g [the function that maps x to f(x) + g(x)] is Theta(f).

Problem #3

Given the tree abstract data type, with constructor make-node and selectors datum and children, consider this procedure:

```
(define (search tree pred)
  (define (helper tasks)
    (cond ((null? tasks) #f)
          ((pred (datum (car tasks))) (datum (car tasks)))
          (else (helper (append ___________ ___________)))))
  (helper (list tree)))
```

(a) Fill in the two dashes so that this procedure searches in breadth-first order.
(b) Fill in the two dashes so that this procedure searches in depth-first order.

Problem #4

Ben Bitdiddle says, "Brian keeps telling us that the idea of data directed programming is much more general that just using get and put. Anything where the data tells the program what to do is data-directed. Therefore, aren't all programs data-directed? The result of calling any procedure depends on what its arguments are."

(a) Name a procedure, not generally considered to be data-directed, that provides evidence in favor of Ben's view, and expain why(in no more than two sentences).

(b) Name a procedure that does not provide evidence for Ben's view, and explain why(in no more than two sentences).

Problem #5

(a) In one or two English sentences, describe something in the real world that would plausibly be implemented in the Adventure game as a class that inherits from both thing and place, and explain why.

(b) Dan Ingalls told us that in a real object-oriented language, everything must be an object. Help him out by writing a define-class implementation of pairs (including mutation) that does not depend on Scheme's built-in pair primitives.

Problem #6

Draw the environment diagram resulting from the evaluation of the following expressions:

```
(define (kons a b)
  (lambda (m)
    (if (eq? m 'kar) a b)))
(define p (kons (kons 1 2) 3))
```

Problem #7

Fill in the blanks to produce the given resulting value for p. Do not allocate any new pairs.

```
> (define p (cons (cons 1 2) (cons 3 4)))
p

> (set-c____r! ____________ _________________)
okay

>p
((1 3 . 4) 3 . 4)
```

Problem #8

Given the following definitions:

```
(define s (make-serializer))
(define t (make-serializer))

(define x 5)
(define y 8)
```

(a) Can the following expression 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.)

```
(parallel-execute (s (lambda () (set! x (+ x 1))))
                  (t (lambda () (set! x (* x 2)))))
```

(b) Can the following expression produce an incorrect result, a deadlock, or neither?

```
(parallel-execute (s (t (lambda () (set! x (+ x 1)))))
                  (t (s (lambda () (set! x (* x 2))))))
```

Problem #9

In the Logo interprester, instead of special forms we have primitive procedures that take the current environment as an extra argument.

Suppose we took the same approach with the Scheme metcircular evaluator. We invent a new data type, the special primitive. Apply will be modified so that it distinguishes among three kinds of procedures (user-defined, primitive, and special-primitive) and invokes a special-primitive with env as an extra first argument and with the other arguments unevaluated.

We then remove all the special form tests from eval, and instead we include special-primitives in the initial global environment. For example, instead of the clause

```
((if? exp) (eval-if exp env))
```

in eval, we include the folling in primitive-procedures:

```
(list 'if
      (make-special-primitive
       (lambda (env predicate consequent alternative)
          (if (true? (mc-eval predicate env))
              (mc-eval consequent env)
              (mc-eval alternative env)))))
```

(a) In addition to removing the rests for the original special forms, does eval also have to be modified to understand special-primitive procedures? Explain, in one sentence.

(b) In what user-visible way will the metacircular Scheme's behavior be different with this reimplementation of special forms? Explain in one sentence.

Problem #10

Consider the following procedure, defined in the nondeterministic evaluator:

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

Assume that pair?, word?, car, and cdr are all available as primitives in the nondeterministic Scheme.

Fill in the responses in the interaction below.

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

>try-again
_____________________________

>try-again
_____________________________
```

Problem #11

The text cheats by using lisp-value whenever it needs to do arithmetic in the query language. We are going to fix this by inventing integer arithmetic. The number n is represented by a list containing the letter a n times. For example, zero is represented by (); one is (a); two is (a a).

The rules for addition are just a special case of the rules for appending lists:

```
(rule (plus () ?y ?y))

(rule (plus (a . ?x) ?y (a . ?z))
      (plus ?x ?y ?z))
```

(a) Write the base case rule for multiplying zero by anything.

(b) Now write the rule for multiplying a nonzero number by anything. Don't worry about whether your rule will "run backwards"; all we require is that, for example, the query

```
(times (a a) (a a a) ?x)
```

should give the single result

```
(times (a a) (a a a) (a a a a a a))
```

Problem #12

The instructor's manual for SICP says: "Data abstraction isn't boring, although writing selectors is." We are going to eliminate the boredom by automating the creation of selectors.

(a) We'll represent an abstract data type by a list of the names of its components, like this:

```
(define binary-tree '(datum left-branch right-branch))
```

This means that a binary tree is a list containing three elements: the datum, the left branch, the right branch.

Recall that a selector for a component of a binary tree is a procedure that takes a binary tree as its argument and returns the desired component. For example, we would usually define selectors like this one:

```
(define (left-branch tree)
  (cadr tree))
```

Instead of that, you are to write a procedure select that takes two arguments, the name of a component and the list representing the type. It should return a procedure, the selector for that component. In our example, the invocation

```
(select 'left-branch binary-tree)
```

should return a procedure equivalent to the left-branch procedure defined above.

Write select here:

(b) Sometimes the componenets of an abstract data type have components of their own. For example, a line segment is two points, each of which has an x coordinate and a y coordinate. Suppose we represent a line segment as a list of two lists, each of which contains two numbers. So the segment from point (2,3) to point (5,9) would be represented as

((2 3) (5 9))

We want to be able to ask for the starting and ending points, but also for the individual numbers. (Yes, strictly speaking, this is a data abstraction violation.) So, we're going to extend the representation of abstract data types to allow that. We'll say

(define line-segment '((start start-x start-y) (end end-x end-y)))

When a list appears as one of the components of an abstract data type representation, the car of the list names the entire component, and the remaining elements identify subcomponents. For example,

((select 'end line-segment) '((2 3) (5 9)))

should return (5 9), while

((select 'start-y line-segment) '((2 3) (5 9)))

should return 3. Components may by arbitrarily deep. That is, sub- components can have sub-sub-components, and so on.

Rewrite select so that it implements this extended representation.

Problem #13

Consider this procedure:

```
(define (hanoi-stream n)
  (if (= n 0)
      the-empty-stream
      (stream-append (hanoi-stream (- n 1))
                     (cons-stream n (hanoi-stream (- n 1))))))
```

It generates finite streams; here are the first few values:

```
(hanoi-stream 1)        (1)
(hanoi-stream 2)        (1 2 1)
(hanoi-stream 3)        (1 2 1 3 1 2 1)
(hanoi-stream 4)        (1 2 1 3 1 2 1 4 1 2 1 3 1 2 1)
```

Notice that each of these starts with the same values as the one above it, followed by some more values. There is no reason why this pattern can't be continued to generate an infinite stream whose first 2^n-1 elements are (hanoi-stream n).

Generate this infinite stream; call it hanoi.
