CS 61A, Spring, 2004, Final, Harvey

Question 1 (2 points):

Circle TRUE or FALSE : Normal and applicative order of evaluation give the same result if there's no assignment (`set!`) or mutation (`set-car!, set-cdr!`).

Question 2 (6 points):

(a) The following procedure takes one argument, a nonempty list of words

```
(define (mystery list-of-words)
      (let ((pairs (map (lambda (x) (cons (count x) x))
                  list-of-words)))
      (cdr (accumulate (lambda (x y)
                        if (> (car x) (car y)) x y ))
                      (car pairs)
                      (cdr pairs)))))
```

In 20 words or less, what does `mystery` return?

(b) Write a procedure `maximizer` that takes one argument, which is a one-argument function `fun` whose range is numbers. `Maximizer` should return a function like `mystery`, but using `fun` to measure of the elements of the list.

Question 3 (4 points):

Consider the following procedure:

```
(define (repetitions sent)
      (define (reps s num)
            (cond ((= num 0) '())
                  ((empty? s) (reps sent num))
                  (else (se (first s) (reps (bf s) (- num 1))))))
      (reps sent 5))
```

(a) Fill in the proper return values:

```
> (repetitions '(hello there))
```

____

```
> (repetitions '(in the white room with black curtains))
```

____

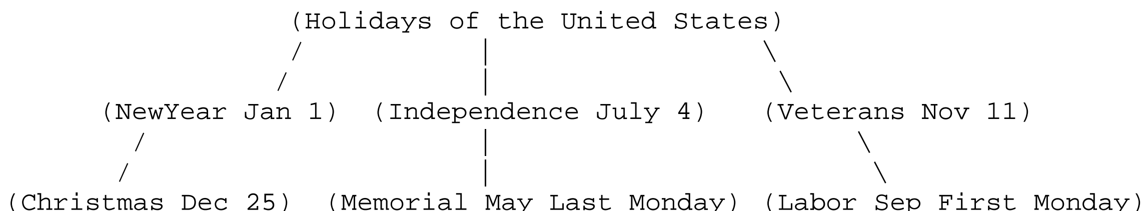
(b) Circle TRUE or FALSE : The time required for `repetitions` grows linearly with the size of its argument `sent`.

Question 4 (4 points):

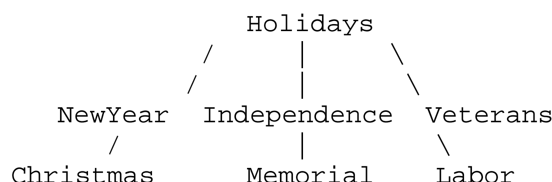
A database company is using trees to store sentences, using the Tree abstract data type:

```
(define make-tree cons)      (define datum car)      (define children cdr)
```

These trees are becoming too large, and in an effort to reduce memory usage the database programmers decide to store only the first word of the sentence in each datum. Below is a procedure that is designed to search through one of these trees of sentences and return a new tree with just the first word of each sentence. As an example, calling `reduce-tree` on the following tree:



results in smaller memory reduced tree:



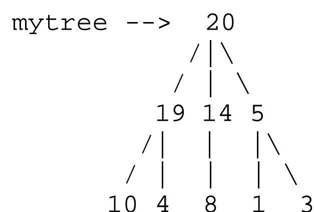
There are some data abstraction violations in the code for `reduce-tree` and its helper `reduce-forest`. Circle them and write the correct selector or constructor below the error.

```
(define (reduce-tree tree)
      (cons (car (car tree))
            (reduce-forest (cdr tree))))

(define (reduce-forest forest)
        (if (null? forest)
            '()
            (cons (reduce-tree (car forest))
                  (reduce-forest (cdr forest)))))
```

Question 5 (6 points):

For this question we're introducing a new kind of tree in which data are all numbers. This kind of tree also has the property that the children of any node have values strictly less than their parent. Here is a sample tree:



For example, the children of the 20 node have data 19, 14, and 5, all of which are less than 20.

Write the function bigs, which takes two arguments: a tree of this type, and a number n, and returns a list of all the data that are bigger than n, in any order. Your function should exploit the ordering

property of these trees, and not visit any nodes that it doesn't have to. Here is an example:

```
> (bigs mytree 9)
(20 19 10 14)
```

In this case, nodes 1 and 3 are not even visited because 5 is already less than 10.

Write bigs below. You may use helper procedures.

Question 6 (4 points):

Please circle TRUE or FALSE for each of the questions below.

In lecture, we organized different shapes and their respective area and perimeter operators using conventional style, data-directed programming, and message-passing.

(a) Circle TRUE or FALSE : Adding a new shape (like "rhombus") requires less modification of code in conventional style than in message-passing style.

(b) Circle TRUE or FALSE : Adding a new operator (like "length of shortest diagonal") requires less modification of code in data-directed programming than in message-passing style.

Question 7 (5 points):

We are going to simulate cell phones using OOP. There will be objects that represent individual cell phones, and objects that represent specific cell phone service providers. For example, Verizon would be one instance of the service-provider class.

Below we describe some aspects of cell phones and cell phone service providers. For each one, decide how to add it to the appropriate class and circle the correct choice. Only circle one choice for each.

For the service provider class:

(a) `customers`: Each provider must keep a list of the customers who use their services

```
Class variable          Instance Variable      Instantiation variable
```

(b) `sell`: A provider must be able to sell a phone to a customer and enroll the customer in the desired plan.

```
Default method          Initialize clause      Method
```

For the cell phone class:

(a) `Nokia-Cellphone`: Has all the possibilities of the cell phone class, but has extra features that are specific to a Nokia cell phone.

```
Parent of cell-phone class   Child of cell-phone class    Other class
```

(d) `address-book:` Each cell phone maintains a list of names and phone numbers that the customer can teach it.

```
Class variable                  Instance variable        Instantiation variable
```

(e) `phone-number:` When the phone is solid, it is assigned a telephone number.

```
Class variable                  Instance variable        Instantiation variable
```

Question 8 (5 points):

Fill in the blanks with the values returned:

```
> (define a (list 1 (list 2 3 )))
> (define b (cdr a))

> (set-car! a (cadr a))
> (set-cdr! a '())

> a
```

```
> b
```

```
> (eq? a b)
```

```
> (define a 10)
> (define (changer a ) (set! A (+ a 1)))
> (changer a)

> a
```

```
> (define x 10)
> (define y (+ 1 x))
> (set! x 50)

> y
```

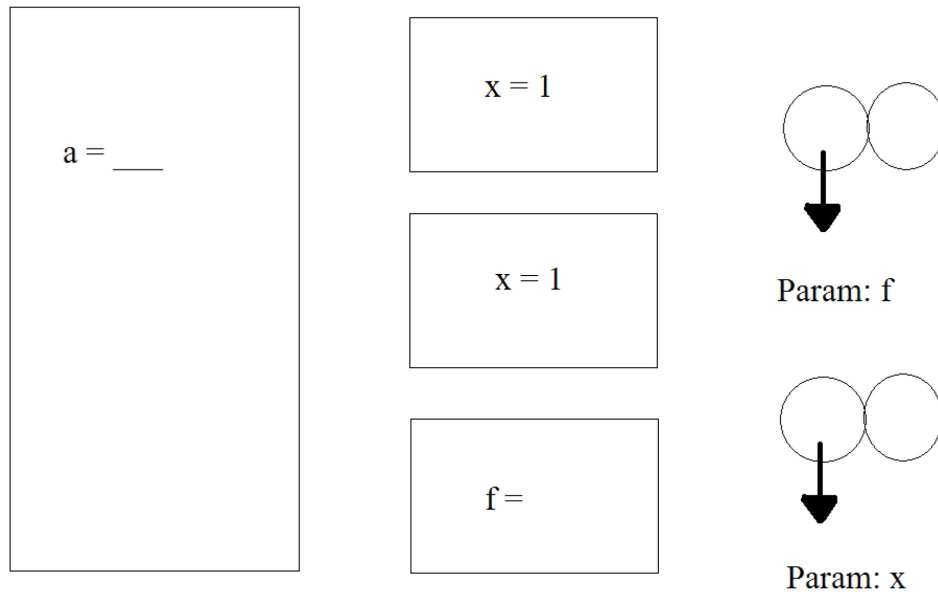
Question 9 (6 points):

For the following Scheme expressions:

```
(define a 0)

(let ((f (lambda (x) (set! A (+ a x)) x)))
      (f (f 1)))
```

complete the following environment diagram by drawing arrows from the three non-global frames to the environment each extends, from the right bubble of the procedures to an environment, and from the symbol f to the value to which it is bound.



Question 10 (4 points):

Here is a procedure that takes two arguments: a vector v and an integer i. It is supposed to swap the first element and the ith element of v:

```
(define (vector-swap! V i)
        (vector-set! V 0 (vector-ref v i)
        (vector-set! V i (vector-ref v 0)))
```

Here is an example of how it is supposed to work:

```
> (define myvec (vector 'a 'b 'c 'd 'e 'f 'g))
> (define-swap! myvec 3)
> myvec
#(d b c a e f g)
```

(a) The running time of this procedure is (circle one): theta(1) theta(i)

(b) Chose one of the following:
___This procedure works correctly.
___The procedure fails because the first element should be number 1, not number 0.
___The procedure fails because both elements get the value originally in the first element.
___The procedure fails because both elements get the value originally in the ith element.

Question 11 (2 points): Given the following definitions:

```
(define s1 (make-serializer))
(define s2 (make-serializer))
(define x 'x)
(define y 'y)
```

what are the possible results of each of the following:

```
(parallel-execute (s1 (s2 (lambda () (set! x y)) ))
                  (s1 (s2 (lambda () (set! x (word x y))) )) )
____ can produce incorrect results
____ can deadlock
____ both
____ neither
```

```
(parallel-execute (s2 (s1 (lambda () (set! y 'x)) ))
                  (s1 (s2 (lambda () (set! x 'y)) )) )
____ can produce incorrect results
____ can deadlock
____ both
____ neither
```

Question 12 (5 points):

For each of the following interactions, circle TRUE if the *analyzing* evaluator is more efficient than the regular metacircular evaluator, or FALSE if it isn't any faster.

(a) TRUE FALSE

```
> (* (+ 2 3) (+ 2 3))
```

(b) TRUE FALSE

```
> (define (square x) (* x x))
> (square 3)
```

(c) TRUE FALSE

```
> (define (double x) (* x 2))
> (+ (double 3) (double 5))
```

(d) TRUE FALSE

```
> (define foo 10)
> (* foo foo)
```

Question 13 (5 points):

Supposed we'd like to add the primitive procedure `procedure?` Into the metacircular evaluator. (Remember, procedure? Is a procedure that takes one argument, and returns #t if that argument is a procedure, or #f otherwise.)

(a) Ben Bitdiddle proposes adding the change in the definition of `primitive-procedures`, as follows:

```
(define primitive-procedures
      (list (list 'car car)
            ...
            (list 'procedure? procedure?)))
```

What would be the return values of the following expressions, typed into the modified evaluator?

```
MCE> (procedure? +)
```

____

```
MCE> (define (square x) (* x x))
MCE> (procedure? Square)
```

____

(b) Implement the `procedure`? primitive correctly:

```
(define primitive-procedures
      (list (list 'car car)
            ...
            (list 'procedure? ______________________________________________)))
```

Question 14 (3 points):

Define the stream nestings that looks like this (using braces to represent the stream):

```
{ () (()) ((())) (((()))) ... }
```

That is, the first element is a null list, the second is a list of the null list, the third is the list of the list of the null list, and so on. **Do not define any additional procedures.**

```
(define nestings ________________________________________________)
```

Question 15 (3 points):

```
> (define (foo x y)
      (if (> x y)
          (/ x y)
          x))
> (define (bar x y)
      (if (> x 1)
          (/ x y)
          x))
```

For each of the following interactions, circle TRUE if the *lazy* evaluator is more efficient than the regular metacircular evaluator, or FALSE if it isn't any faster.

(a) TRUE   FALSE   `> (foo (* 2 3) (* 5 2))`
(b) TRUE   FALSE   `> (bar (foo 10 5) 1)`
(c) TRUE   FALSE   `> (bar 1 (foo 10 5))`

Question 16 (3 points): Provide bindings for the patterns below that will unify each set of patterns. If no bindings exist that will unify the patterns, write "NOT POSSIBLE". For example:
Question: `(start to . ?finish) : (start ?end)`

Answer: `?finish -> (), ?end -> to`

```
(a) (?x ?x) : ((a ?y) ?z)
(b) (?x ?x) : ((a ?y) (?y b))
(c) (polly wanna ?cracker) : (polly ?morphic)
```

Question 17 (3 points): Fill in the blanks in the following rules to correctly implement the `coolerize` relation between the two lists. The `coolerize` relation is satisfied when every element of the second list is equal to the corresponding element of the of first list, but type tagged with the word "cooler". Example:

```
;;; Query input:
(coolerize (a b (c c)) (?uncoolA ?uncoolB ?uncoolC))

;;; Query results:
(coolerize (a b (c c)) ((cooler a) (cooler b) (cooler (c c)))
```

Notice that each member of the first list appears in a list with the `cooler` type tag in the second list.

```
(assert! (rule (coolerize () ())))

(assert! (rule (coolerize (?car . ?cdr) (_____________________ . ?z))
               (coolerize _______________ ________________)))
```