CS61A Solutions to mt3, Spring 2010

1. WWSP - Solutions are top to bottom, left to right (so top-left is a, top-right b, and so on)

a. (1 2)

The initial definition of x to be the list (1 2) is untouched because the (set!...) occurring in the mutate! function is defining its own local variable x, and then re-assigning that one. Therefore, even after calling mutate!, the global variable x is still (1 2).

b. (4 2)

Again, we have a global variable x pointing to the list (1 2), but this time, the function mutate! takes an argument whose parameter name is also x. When mutate! is called, it is called with the global x, so both the global x and the x local to the procedure call end up pointing to the same list. Therefore, the call to set-car! is actually mutating the /same/ list, so the global variable x is also affected.

c. (1 2)

This is similar to part b., but since /assignment/ is occurring instead of /mutation/ (set! instead of set-car!, essentially), the mutate! function merely changes the variable x that is local to the function, /not/ the global one! So, at the end of the mutate! function call, the local variable x now points to the number 3 (from the re-assignment), but the global variable x is untouched.

d. 3

Since the mutate! function does not take any param
eters in this case, there is only one defined x
variable in this case: the global x that is set to
 (1 2). Therefore, when the set! searches for the
first available x, it will find nothing in the fra
me belonging to the mutate! function call, and wil
l
thus choose the global x and re-assign that one to
3.

Grading: 1pt each

2. Analyzing Evaluator

a. Faster at evaluating some lambda expressions. <-
- FALSE

The analyzing evaluator is actually /slower/ at ev
aluating lambda expressions, since it has to do
the analyze the body of the procedure at the time
of procedure-creation, whereas the MC-Eval analyze
s
when the procedure is being called.

b. Faster at evaluating somce procedure calls. <--
TRUE

The pre-analyzed procedure bodies in the analyzing
 evaluator run faster than MC-Eval procedure bodie
s
which have to do the analysis step on every procedu
re call.

c. Represents primitive procedures the same way as
the MCE does. <-- TRUE

Primitive procedures are already taken care of by the underlying STk, so analysis is not required for those procedures. Therefore, analyzing eval represents them the same way as MCE.

d. Represents lambda-created procedures the same way as the MCE does. <-- FALSE

Lambda-created procedures have analyzed procedure bodies that accept an environment as their argument, while the MCE lambda-created procedures directly represent the procedure body, parameters, and environment as a tagged list.

e. Represents environment frames the same way as the MCE does. <-- TRUE

Though the analyzing evaluator uses environment slightly differently (in conjunction with the pre-analyzed procedure bodies), the representation of the environments is still the same.

Grading: 1pt each

3. Choose whether to use vectors or lists to implement the following procedures, and explain why you made your decision. You should take into account both programming ease and efficiency. You do *not* need to actually implement the procedure!

MEDIAN is a procedure that takes a *sorted* sequence of numbers and returns the median value.

List _____ Vector __X__

"Accessing the N/2th element (the definition of the median) takes Theta(N) time with a list but only Theta(1) or constant time with a vector. Both versions of MEDIAN are equally easy to program."

The main thing that was important here was the difference in the orders of growth; no matter what, the vector version is going to be faster. If you put that, you probably got full credit for the explanation.

Saying that MEDIAN could be easily implemented with VECTOR-LENGTH and VECTOR-REF only got you one point, since you could use the equivalent list procedures LENGTH and LIST-REF. We were lenient here since not everyone knew about LIST-REF.

```
; Sample implementation:
(define (median sorted)
  (let ((len (vector-length sorted)))
    (if (even? len)
        (/ (+ (vector-ref sorted (/ len 2))
              (vector-ref sorted (+ 1 (/ len 2)
))) )
           2)
        (vector-ref sorted (+ 1 (/ len 2))) )))
```

CONCATENATE is a procedure that takes two sequences of length N and appends them, returning a sequence of length 2N. (Assume you donâ€™t have append.)

List __X__ Vector _____

"CONCATENATE can be implemented using simple recursion for lists, whereas it requires an iterative helper for vectors. In addition, the list version only has to copy the elements of the first list, while the vector version has to iterate over *both* sequences to copy the elements into a newly created result vector."

This question is admittedly less clear-cut, but there are certainly better and worse answers. The main point for efficiency is that you only have to go over the first list to get your answer, while you'd have to copy elements from both vectors into a result vector.

The question did ask you to consider both efficiency and programming ease. The list version of CONCATENATE *is* APPEND, which is a plain old list recursion:

```
(define (concatenate left right)
  (if (null? left)
      right
      (cons (car left) (concatenate (cdr left) right)) ))
```

The vector version, on the other hand, requires some kind of VECTOR-COPY! helper procedure, in order to copy both inputs into the output.

```
(define (concatenate left right)
  (let ((left-length (vector-length left))
        (right-length (vector-length right)) )
    (let ((result (make-vector (+ left-length right-length))))
      (vector-copy! result 0 left 0 left-length)
      (vector-copy! result left-length right 0 right-length)
      result)))

(define (vector-copy! dest d-offset src s-offset count)
  (if (> count 0)
      (begin (vector-set! dest d-offset (vector-ref src s-offset))
             (vector-copy! dest (+ d-offset 1)
                           src (+ s-offset 1)
                           (- count 1)) )))
```

There are slightly simpler ways to do this, but none as simple as the list version. As with MEDIAN, simply stating "cleaner code" or "ease of programming" was only worth 1pt.

Some people interpreted the question to mean *mutate* the sequences to form one larger sequence, and correctly pointed out that you could do this *space*-efficiently with lists, by connecting the last pair of the first list to the second list. With vectors, you have to create a new vector to hold the result. This was a perfectly good answer (we didn't specify no mutation).

However, you had to be careful. An answer stating "it only requires one SET-CDR!" received one point: you still have to CDR to the end of the first list to perform that SET-CDR!. Saying that mutation was "simpler" than copying was also only worth 1pt.

The trouble came when people tried to use orders of growth in their answers. The shortest answers simply stated that growing a list was Theta(1), while growing a vector was Theta(N). Remember, this is only true if you're adding a /single/ element to the /front/ of the sequence! Any answer that implied the entire CONCATENATE took Theta(1) time received no credit.

Another version of this answer correctly pointed out that a version of CONCATENATE that used the hypothetical VECTOR-CONS would take Theta(N2) time. This would not be the correct way to implement CONCATENATE, though.

In fact, the order of growth of both versions is Theta(N): copying all N elements of the first list vs. copying all 2N elements of both vectors. Even on these grounds of "roughly equal efficiency", however, lists were still the better choice based on programming ease.

Scoring:
1pt List vs. Vector
2pts Explanation (roughly 1pt ease of use, 1pt efficiency)

4. Consider the simplified version of the PLACE class from the
adventure game. You're going to edit the PLACE class in various ways:

(a) Add a NAME instantiation variable to the PLACE class.

We gave two points for this problem. One was modifying the correct LAMBDA to take the new instantiation variable; this should've been done in the second LAMBDA, the one that currently has no arguments.

The second point was given for the accessor for the new NAME variable, otherwise the class has no way of understanding the NAME message: ((eq? message 'name) (lambda () name))

No credit was awarded for the accessor without the enclosing LAMBDA.

(b) Add the EXIT method.

Most of the time, this was an all-or-nothing answer. You needed to add (eq? message 'exit) to the COND clause, whose body was the code we provided, with a LAMBDA to take the method argument:

```
(lambda (person) (set! people (remove person people)) 'disappeared)
```

We took off one point if you were missing the outside LAMBDA, or somehow screwed up the body of the EXIT meth od.

(c) Add a DEFAULT-METHOD that returns the sentence
(i dont know how to message)

Notice that we want you to /return/ the sent
ence! Simply modifying
the argument to ERROR does not return the se
ntence, it fails. We
wanted you to replace the inner (else (error
"No such ...")) with
(else (lambda args (se 'I 'dont 'know 'how '
to message))), returning as
normal (i.e. with the enclosing LAMBDA) a se
ntence. We gave 1pt
for a decent effort (usually forgetting the
enclosing LAMBDA), and
nothing if you still used ERROR. We still ga
ve 2 pts if you did
(lambda () (se 'I 'dont ...)), although ther
e may be more args after
the message.

Scoring:
2pts each, partial credit described above

5. First we need to determine where the lock is ins
tantiated, i.e. how
granular it is.
Defining it as a global variable or as a class vari
able of stack means
that the lock prevents process 1 from using stack 1
while process 2 is
using stack 2, this is overly protective and ineffi
cient. Defining a
lock for the functions push and pop separately is c
learly not
sufficient protection because that means that proce
ss 1 can push onto

stack 1 while process 1 can pop from stack 1, so two processes read
and set items. Hence the mutex should be a instance variable:

```
(instance-vars (items '()) (m (make-mutex)))
```

Next we need to decide which sections have to be atomic. For the push
method, the atomic section is just the set!, so we enclose it in a
call to acquire and release the lock:

```
(method (push item)
   (m 'acquire)
   (set! items (cons item items))
   (m 'release))
```

In pop, the atomic sections starts with evaluating the value for top and
ends with the set!. If we acquire the lock after evaluating (car
items), the element pop removes from the stack and the element pop
returns is different when another process has changed the stack
between (car items) and set!. We have to remember the value of top
until after the lock is released, so (m 'release) is before the top
which is returned. This makes the code look like slightly asymmetric
because the release is nested into more parenthesis than the acquire,
but the mutex state is changed at the right time of execution.

```
(method (pop)
   (m 'acquire)
   (let ((top (car items)))
      (set! items (cdr items))
      (m 'release)
```

```
          top))
```

Putting it all together, we get the solution:

```
(define-class (stack)
      (instance-vars (items '()) (m (make-mutex)))
      (method (push item)
         (m 'acquire)
         (set! items (cons item items))
         (m 'release))
      (method (pop)
         (m 'acquire)
         (let ((top (car items)))
            (set! items (cdr items))
            (m 'release)
            top)))
```

The question asked specifically to use mutexes, but
 a working solution
with serializers looks like this:

```
(define-class (stack)
      (instance-vars (items '()) (s (make-serialize
r)))
      (method (push item)
         (s  (lambda ()
                (set! items (cons item items)))))
      (method (pop)
         (s  (lambda ()
                (let ((top (car items)))
                   (set! items (cdr items)))
                   top))))
```

Notice that s is a higher order function and we nee
d to thunk the code
sections we want to protect.

Grading:

* 6 pts if the implementation was working correctly
 and efficient (mutex is
        instance variable)

* 4 pts if the implementation was correct but ineff
icient (mutex is a
        global or class variable)
* 2 pts if the idea is there, but the implemntation
 does not work (a
        mutex per method or per call)
* 0 pts otherwise

Subtract from that

* -1 pt if in pop the (m 'release) and top were swi
tched so the method
        does not return the right result
* -1 pt if serializers were used

6.
```
(define (separate! a-list)
    (if (null? a-list)
        '()
        (let ((kv-pair (car a-list))
              (key (caar a-list))
              (value (cdar a-list))
              (next-pair (if (null? (cdr a-list)) '
() (cadr a-list))))
             (set-car! a-list key)
             (set-car! kv-pair value)
             (set-cdr! kv-pair next-pair)
             (separate! (cdr a-list))
             kv-pair)))
```

EDIT: The let next-pair line has been edited. The
point of this is to ensure
that the cadr is not going to be null. Note that th
is line can be avoided
by implementing all of the changes in the following
 paragraph. This is because
the null check in the beginning then fixes everythi
ng else.

The observant reader here may notice that, in fact, the requirement of using
let on everything is unnecessary. You can get away with just the first kv-pair
in the let. Furthermore, instead of setting the cdr to next pair, you can set
the cdr to the recursive call, as in (set-cdr! kv-pair (separate! (cdr a-list)))

The most common mistake here was to not understand how to separate the list into
two separate lists. I STRONGLY encourage you to draw out the first 4 pairs, ie
two pairs on top, and two on the bottom, and draw out the result of all the set!
on these pairs. Hopefully, that will give you some insight in how you might have
been able to come to the right answer if you didn't.

For grading, we determined that the criteria for getting the right idea was to
get the bottom set-cdr! correct. This was essentially worth 3 points. There was
a point each for the set-car! for setting the key and the value that you could
get independently of those 3 points. Trivial errors included not returning the
correct result, or missing the base case.

7 perfect
6 trivial
5 the idea: recursion fails b/c no return
4 the idea: constant number of new pairs, incorrect mutation order
3 the idea: correct CDRs on original kv-pair list
2 an idea
1 an idea
0 no idea: any non-constant number of new pairs