University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Summer 2002 Instructor: Kurt Meinz 2002-08-09

CS 61A Midterm #3

Personal Information

*First and Last **Name***	
*Your **Login***	cs61a-__ __ *(legibly!)*
*The **First Letter** of Your Login (please circle)*	a b c d e f g h i j k l m n o p q r s t u v w x y z
*The **Second Letter** of your login (please circle)*	a b c d e f g h i j k l m n o p q r s t u v w x y z
*Lab Section Time, TA, & Location **You Attend***	
*Discussion Section Time, TA, & Location **You Attend***	
"All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61a who have not taken it yet."	***(please sign)***

Instructions

- Partial credit may be given for incomplete / wrong answers, so please write down as much of the solution as you can.
- Feel free to use any Scheme function that was described in lecture or sections of the textbook we have read without defining it yourself. Do not use functions or constructs that we have not yet covered. Unless specifically prohibited, you are allowed to use helper functions on any problem.
- Please use "true" instead of #t , and "false" instead of #f. We have found that handwritten #t and #f unfortunately look too much alike.
- Please write legibly! If we can't read it, we won't grade it!

Grading Results

Question	*Max. Points*	*Points Earned*
1	**6**	
2	**10**	
3	**12**	
4	**12**	
Total	**40**	

Name: _____ Login: _____

Question 1: Stream Warm-up [6 Points]

Please define a procedure 'wave-maker' that takes two arguments (a lower and an upper bound) and returns an infinite stream that counts from lower to upper, then upper to lower, then lower to upper, ad infinitum.

E.g. (ss (wave-maker 1 4) 25)
➔ (1 2 3 4 4 3 2 1 1 2 3 4 4 3 2 1 1 2 3 4 4 3 2 1 2 …)

You may further assume that wave-maker will always be called with lower < upper. Since this is a relatively easy question, your aim should be elegance.

Name: _____ **Login:**_____

Question 2: Conc-urrrrrrrwin-cy **[10 Points]**

Consider the following:

```
(define jeffs-list '(1 1 1))

(define (inc-list-by-one! lst) ;; add 1 to all elements of lst
  (cond ((null? l) 'okay)
        (else (set-car! lst (+ (car lst) 1))
              (inc-list-by-one! (cdr l)))))

(define (inc-list! lst n)
  (cond ((= n 0) 'okay)
        (else (inc-list-by-one! lst)
              (inc-list! lst (- n 1)))))

(parallel-execute (lambda () (inc-list! jeffs-list 10)))
                  (lambda () (inc-list! jeffs-list 10))))
```

Part A: Does inc-list! have concurrency issues? If so, give concise and specific examples.

Part B: Kurt tries to correct inc-list! as follows:

```
(define (inc-list! lst n)
  (let ((my-mutex (make-mutex)))
    (my-mutex 'acquire)
    (cond ((= n 0) 'okay)
          (else (inc-list-by-one! lst)
                (inc-list! lst (- n 1))))
    (my-mutex 'release)))
```

In 20 words or less, why does this not solve the problem?

Part C: Now re-write inc-list! so that it does not suffer from concurrency issues. [You should use the mutex abstraction.]

Name: _____ **Login:**_____

Question 3: Mutable Structures [12 Points Total]

We would like you to implement a 'filter!' procedure that acts like the usual filter but mutates its argument list.

Part A: Please define 'filter!' It should take a predicate function and a list as arguments and mutate the list so that it contains only the elements for which the predicate function is true. You may assume that the list will be non-null and **(importantly)** that the predicate function will always return true on the first element of any list given to filter!. (Filter! itself may return any value you choose.)

Part B: Is it possible to implement filter! if we do not make any assumptions about the predicate function or the input list? Briefly explain why or why not. [Hint: we are looking for an insurmountable difficulty.]

Name: _____ **Login:**_____

Question 3: Mutable Structures --- Continued ---

Part C: We want you to create a new abstract data type for mutable lists such that it is possible to build a new filter! that will operate all lists made from this new ADT and will work with any predicate function. [You don't have to write the new filter!, just the new abstraction.]

You should provide implementations for 'list', 'car', 'cdr', 'set-car!', 'set-cdr!' and 'null?'. [Hint: for the 'list' constructor, use the dotted-tail notation.]

```
(define (list . elements)
```

Name: _____ **Login:**_____

Question 4: MCEs Ouch! **[12 Points]**

Kurt would like to add the 'map' procedure as a primitive to the MCE, but doesn't want to do it himself. So he asks all the TAs to do it, and they each try to do it in a different way. For each attempt, please say whether it is correct or not. If not, please explain, as concisely as possible, why not. Changes or additions to the existing code have been boxed.

a. Ilya's Attempt: The following clause is added to the `cond` in the `mc-eval` function. (This clause appears before the `application?` clause.)

```
((map? exp) (map (mc-eval (cadr exp) env)          ;; map added here
                 (mc-eval (caddr exp) env))
```

Where `map?` is defined as `(define (map? exp) (tagged-list? exp 'map))`, and `map` is the underlying Scheme higher-order function.

I think this **will** or **will not** (circle one) work because

b. Jane's Attempt: The following addition is made to the list of primitives:

```
(define primitive-procedures
  (list 'car car)
```

```
  (list 'map '(procedure (func lst)          ;; map added here
                         ((if (null? lst)
                              '()
                              (cons (func (car lst))
                                    (map func (cdr lst)))))
                         the-global-environment))
```

```
    . . .
  )
```

I think this **will** or **will not** (circle one) work because

Name: _____ **Login:**_____

Question 4: MCEs Ouch! **Continued**

c. Jeff's Attempt: Modify the setup-environment function as follows:

```
(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
```

```
(define-variable! 'map '(procedure           ;; map added here
                         (func lst)
                         ((if (null? lst)
                              '()
                              (cons (func (car lst))
                                    (map func (cdr lst)))))
                         the-global-environment)
                       initial-env)
```

```
    initial-env))
```

I think this **will** or **will not** (circle one) work because

d. Greg's Attempt: Modify the setup-environment function in a different way:

```
(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    . . .
```

```
(define-variable! 'map (make-procedure     ;; map added here
                        `(func lst)
                        `((if (null? lst)
                              '()
                              (cons (func (car lst))
                                    (map func (cdr lst)))))
                        initial-env)
                       initial-env)
```

```
    initial-env)
```

Name: _____ **Login:**_____

I think this **will** or **will not** (circle one) work because

Name: _____ **Login:**_____

Question 4: MCEs Ouch! **Continued**

e. Greg's Second Attempt: The mce function is modified as follows:

```
(define (mce)
  (set! the-global-environment (setup-environment))
  (define-variable! 'map                          ;; map added here
                    (map-procedure)
                    the-global-environment)
  (driver-loop))
```

Where the map-procedure function is defined as:

```
(define (map-procedure)
  (make-procedure '(func lst)
                  '((if (null? lst)
                        '()
                        (cons (func (car lst))
                              (map func (cdr lst)))))
                  the-global-environment))
```

I think this **will** or **will not** (circle one) work because

f. Erwin's Attempt: The extend-environment function is modified:

```
(define (extend-environment vars vals base-env)
   (if (= (length vars) (length vals))
       (let ((new-env (cons (make-frame vars vals) base-env)))
          (define-variable! 'map
                            (list 'procedure
                               '(func lst)
                               '((if (null? lst)
                                     '()
                                     (cons (func (car lst))
                                           (map func (cdr lst))))
                                the-global-environment)
                             new-env)
           new-env)
       (if (< (length vars) (length vals))
           (error "Too many arguments supplied" vars vals)
           (error "Too few arguments supplied" vars vals))))
```

I think this **will** or **will not** (circle one) work because