University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Summer 2003 Instructor: Kurt Meinz August 15, 2003

CS 61A Final

Personal Information

First and Last Name	
Your Login in Clear, Capital Letters	cs61a-__ __
Your TA's Name	
*Your **Real** Email Address*	
"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 look too much alike.
- Please write legibly! If we can't read it, we won't grade it!
- **This exam is difficult.** Just relax and do the best you can.

Grading Results

Question	*Max. Points*	*Points Earned*
1	**7**	
2a	**7**	
2b	**7**	
3	**7**	
4	**7**	
5	**7**	
6	**7**	
7	**7**	
8	**14**	
Total	**70**	

Name: _____ Login: _____

Question 1: Prove Kurt Wrong!

Kurt wanted to add apply to the MCE as a special form:

```
> (apply * (list 1 2 3 4)) ;; just like (* 1 2 3 4)
24
```

He inserted the following in the MCE's eval procedure:

```
;; MCE eval inserted just before the 'application?' clause:
((tagged-list? exp 'apply)
 (mc-eval (cons (cadr exp) (caddr exp)) env)))
```

Greg correctly pointed out that, in some cases, this implementation will crash or produce results different than those of the version of *apply* in STk.

Part A:

Please give a simple, well-formed scheme expression that starts with 'apply' and will not work with this implementation of apply. Be sure that your expression highlights the problem with this implementation of *apply*.

```
;;; MC-Eval Input:

(apply _____________________________________________________ )
```

Part B:

In the best English that you can muster, please give a clear and concise explanation of why this implementation of apply will not work as intended. No more than 16 words, please.

Name: _____ **Login:**_____

Question 2: Dynomite Dynamic Scoping!

Part A: Dynamically Scoped Procedures

Greg has started to add a special form to the **lazy** evaluator named 'dynamic-lambda'. Its function is to create a procedure that uses dynamic rather than lexical scoping to evaluate the body.

```
>(define x 4)
>(define lexical-square-x (lambda         () (* x x)))
>(define dynamic-square-x (dynamic-lambda () (* x x)))
>(let ((x 10)) (list (lexical-square-x) (dynamic-square-x)))
(16 100)
```

Here's what he did:

```
;; in eval, right before the 'application?' clause:
((tagged-list? exp 'dynamic-lambda)
 (make-procedure (lambda-parameters exp)
                 (lambda-body exp)
                 'dynamic))
```

Unfortunately, he got confused about how to invoke these procedures and gave up. We'd like you to finish the changes to the **lazy** interpreter so that these dynamic procedures are invoked correctly and have the proper scoping. Please write the procedure name and line numbers of anything that you change. (Should be 4 lines, tops.)

Name: _____ **Login:**_____

Question 2: Dynomite Dynamic Scoping!, cont.

Part B: Dynamic Promises

Carolen wants to extend the **lazy** evaluator to support dynamically scoped promises. In a way similar to the strict/non-strict problem on exam 3, she plans to extend the formal parameter specifications in the **non-memoizing, lazy interpreter** such that parameters are labeled as either 'lexical' or 'dynamic':

```
>(define x 4)
>(define foo (lambda ((lexical a) (dynamic b))
               (let (((lexical x) 10))
                  (list a b))))
>(foo x x)
(4 10)
```

The environment of evaluation for *a* will be its environment at creation, while the environment for *b* will be the current environment. She started to create these dynamically scoped promises as follows:

```
;; lazy's mc-apply lines 10 and 11 replaced with:
(map cadr (procedure-parameters exp)))
(map (lambda (param arg) (if (tagged-list? param 'dynamic)
                             (delay-it arg 'dynamic)
                             (delay-it arg env)))
     (procedure-parameters exp)
     arguments)
```

Unfortunately, she also got confused and gave up before implementing compound procedure application – so help her out. Again, write the proc name and line numbers of anything you change. [Can be done in 3 generous lines.]

Name: _____ **Login:**_____

Question 3: (adds-to-n? 70 (your-subscores)) ➔ #t

Use **vambeval** to solve *adds-to-n?*. You may assume that we have defined a procedure named 'subsets' that will return all of the sublists (including the empty list) of a given list:

```
>(subsets '(1 2 3)) ➔ ( () (1) (2) (3) (1 2) (2 3) (1 3) (1 2 3) )
```

In case you forgot, here is the regular-scheme solution to adds-to-n?:

```
> (define (adds-to-n? n nums)
    (cond ((= n 0)      true)
          ((null? nums) false)
          (else (or (adds-to-n? (- n (car nums)) (cdr nums))
                    (adds-to-n? n                (cdr nums))))))
> (adds-to-n? 5 '(2 10 3)) ➔ #t
> (adds-to-n? 5 '(2 10 4)) ➔ #f
```

Your solution **must** use 'amb' and 'require', and you are **NOT allowed** to use any helpers **other than** *subsets, an-integer-starting-from, an-integer-between, an-element-of*, *map*, *filter*, *accumulate*, and/or basic arithmetic primitives (all of which may be used without definition).

Your version of adds-to-n? should return the "winning subset" if found. 'try-again' should result in the next winning subset being found. If there are no more winning subsets, your definition should cause vambeval to print "No more values …":

```
;; Ambeval input
(adds-to-n? 6 '(3 6 3)) ➔ (6)
try-again               ➔ (3 3)
try-again               ➔ "No more values …"
```

Hint: Your solution should be 3 lines tops, and will not follow the form of the regular scheme solution. Take advantage of **amb**! Your procedure can return the winning subsets in any order, as long as it finds all of them.

```
(define (adds-to-n? n nums)
```

Name: _____ **Login:**_____

Question 4: Et Fail2, Brutus?

Alex was messing around with **vambeval**, and, just for giggles, inside of eval-sequence he replaced exactly **one** of the occurrences of 'fail2' with 'fail':

```
;;new eval-sequence:
(define (eval-sequence exps env succeed fail)
  (define (loop first-exp rest-exps succeed fail)
    (if (null? rest-exps)
        (ambeval first-exp env succeed fail)
        (ambeval first-exp
                 env
                (lambda (first-value fail2)  ;; unchanged
                  (loop (car rest-exps)
                        (cdr rest-exps) succeed fail))  ;; was fail2
                fail)))
  (if (null? exps)
      (error "Empty sequence")
      (loop (car exps) (cdr exps) succeed fail)))
```

Part A:

Please give a well-formed scheme expression that uses 'amb' and highlights why fail2 is necessary in order to get eval-sequence to behave properly.

```
;; Ambeval Input

(begin
```

Part B:

In your best English, please give a clear and concise explanation of why this implementation of eval-sequence will not work correctly. No more than 32 words, please.

Name: _____ **Login:**_____

Question 5: We liked the problem so much that we copyrighted it…

If you're not sick of seeing *adds-to-n?* yet, you will be after doing this problem. We'd like you to re-write *adds-to-n?*, this time in the query language. You must use the unary number language that we have been doing in class, and you can assume that we have pre-defined the *plus* relation as usual. Your *adds-to-n?* definition should work for the following sample calls:

```
(adds-to-n? (1 1 1) ((1) (1 1 1 1) (1 1)) ➔ Provable.
(adds-to-n? (1 1 1) ((1) (1 1 1 1) (1))   ➔ Not provable.
```

Hint: Your definition should follow the outlines of the regular scheme definition of *adds-to-n?*. The trickiness comes from representing *(- n (car nums))* and *(cdr nums)* in the logic language.

```
(define (adds-to-n? n nums)
   (cond ((= n 0)      true)
         ((null? nums) false)
         (else (or (adds-to-n? (- n (car nums)) (cdr nums))
                   (adds-to-n? n                (cdr nums))))))
```

As a courtesy for working so hard this summer, we'll start you out with the base case. (There is no false base case because everything that is not provably true in the system is considered false.)

```
(rule (adds-to-n? () ?any))
```

Fill in the following to finish the definition. Our solution is 3 lines.

```
(rule (adds-to-n? ?n (?car . ?cdr))
```

Name: _____ **Login:**_____

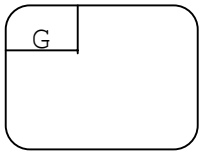
Question 6: Double Bubble Time!

Consider this expression typed in at the prompt:

```
(let ((a 5))
  ((let ((a a))
      (lambda (b)
        (set! b (* a 3))
        b))
   (let ()
      (set! a (+ a 1))
      (+ a 1))))
```

Part A:

Please draw the complete environment diagram for the evaluation of this expression. Don't forget to draw all double-bubbles (even unnamed ones) and label each frame in creation order.



Part B:

What is the return value of the expression? _____

Name: _____ **Login:**_____

Question 7: M U T A T I O N!

Part A:

Please examine this sequence of expressions:

```
>(define a (list 'everyone 'likes 'logo))
>(define (make-true x)
   (let ((y (cdr x)))
     (set! y (cons 'temp (cdr y)))
     (set-car! y 'greg)))
>(make-true a)
```

Please draw a box-and-pointer diagram for *a* after evaluation of the foregoing expressions.

Part B:

Now, take a look at this expression:

```
(let ((y (list 'hello 'nurse)))
  (let ((x (cdr y)))
    (set-cdr! (cdr x) '(betty))
    (cdr y)))
```

If this expression were evaluated in STk, what would the return value be? If you think it would be an error, please give an approximation of the reason for the error (i.e. what would STk say for the error message).

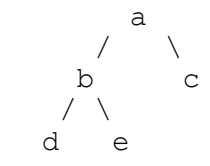
Name: _____ **Login:**_____

Question 8: And you thought you wouldn't have to do a tree problem!

Consider a *binary* tree whose interface is as follows:

```
(define (make-node datum left-child right-child)
   (list datum left-child right-child))
(define (make-leaf datum)
   (make-node datum nil nil))
(define get-datum car)
(define get-left-child cadr)
(define get-right-child caddr)
(define (is-leaf? node)
   (and (eq? (get-left-child node) nil)
        (eq? (get-right-child node) nil))
```

So, to construct the tree at right, you would call




```
(define my-tree
  (make-node 'a
             (make-node 'b
                        (make-leaf 'd)
                        (make-leaf 'e))
             (make-leaf 'c)))
```

The **distance** between two nodes in the tree is the number of branches (up and down) that must be crossed to form a path between the two nodes. Thus, in *my-tree*, the distance between *d* and *c* is 3, *d* and *b* is 1, and *d* and *e* is 2.

We'd like you to write a 'distance' procedure that takes a tree and two values and returns the distance between the nodes that contain those values as data. You may make these assumptions:

1. all the data in the tree will be distinct
2. distance will be called with values that are in the tree
3. distance will be called with distinct (non-equal) values

To make things easier, we'll give you two big hints:

1. If the two target nodes are in different subtrees of a given node, then the distance between them is the sum of their depths from the current node. E.g. since *d* and *c* are in different subtrees of *a*, the distance between *d* and *c* is the sum of their depths from *a*.

2. You'll probably want to write [and we'll give you partial credit for] procedures that compute 'is-in-tree?' and 'depth-of' (starting at **0**) for these binary trees.

Put your solution on the next page.

Name: _____ **Login:**_____

Question 8: And you thought you wouldn't have to do a tree problem!

If you use 'is-in-tree?' and/or 'depth-of' in your solution, you need to provide definitions for them here. Remember that these are binary trees – not the regular trees.

(define (is-in-tree? value node) **[optional]**

(define (depth-of value node) **[optional]**

(define (distance value1 value2 node) **[required]**