CS 61A Final — May 20, 1994

Your name _____

login cs61a–_____

This exam is worth 30 points, or 17% of your total course grade. **The exam contains eight questions, but you need only answer six of them!**

You must answer all of questions 1–4. Then there are two pairs of questions. Answer 5A or 5B, and 6A or 6B. Indicate below which you answered:

I answered 5_____ and 6_____. (Fill in A or B.)

This booklet contains sixteen numbered pages (both sides of six sheets) including the cover page. Put all answers on these pages, please; don't hand in stray pieces of paper. This is an open book exam.

When writing procedures, write straightforward code. Do not try to make your program slightly more efficient at the cost of making it impossible to read and understand.

When writing procedures, don't put in error checks. Assume that you will be given arguments of the correct type.

Our expectation is that many of you will not complete one or two of these questions. If you find one question especially difficult, leave it for later; start with the ones you find easier.

1	/5
2	/5
3	/5
4	/5
5	/5
6	/5
total	/30

Question 1 (5 points):

Draw a "box and pointer" diagram for the following Scheme expressions:

a (1 point).

```
(let ((x (cons '() '())))
  (set-car! x x)
  (set-cdr! x x)
  x)
```

b (2 points).

```
(define (funny! x)
  (if (null? x)
      '()
      (let ((temp (cdr x)))
        (funny! temp)
        (set-cdr! x (car x))
        (set-car! x temp)
        x)))
(funny (list 1 2))
```

Question continues on next page.

Your name _____ login cs61a–_____

Question 1 continued

c (2 points). Write a Scheme expression to construct the following structure.

Question 2 (5 points):

We are going to create an abstract data type called a *ternary tree*. It's just like a binary tree, except that each node can have a left branch, a middle branch, and a right branch. (Any of these might be empty for a particular node.) We are going to represent a node using three pairs, arranged like this:

(a) Write the constructor `(make-ternary datum left middle right)`; the selectors `datum`, `left-branch`, `middle-branch`, and `right-branch`; and the mutators `set-datum!`, `set-left-branch!`, `set-middle-branch!`, and `set-right-branch!`.

(b) Write a procedure `flip` that takes a ternary tree as its argument, and mutates the tree so that the left and right branches of *every* node are interchanged. (Don't move the middle branch.) Respect the data abstraction.

Your name _____ login cs61a–_____

Question 3 (5 points):

The following program has two kinds of errors: violations of data abstraction (that work but are bad style) and downright bugs. Fix them and indicate which are which. We are dealing with international finance and so we have amounts of money with manifest type to indicate the currency unit. (Note: none of the errors are about unbalanced parentheses!)

```
(define attach-type cons)
(define type car)
(define contents cdr)

(define (make-dollar amt)
  (attach-type 'dollar amt))

(define (make-franc amt)
  (attach-type 'franc amt))

(define (make-pound amt)
  (attach-type 'pound amt))

(define (+money amt1 amt2)
  (make-dollar (+ (contents (dollarize amt1))
                  (contents (dollarize amt2)))))

(define (dollarize amt)
  (* (conversion (car amt)) (cdr amt)))

(define (conversion type)
  (cond ((eq? type 'dollar) 1.0)
        ((eq? type 'franc) 0.2)
        ((eq? type 'pound) 2.0)))
```

Question 4 (5 points):

Write a function `regroup` whose argument is a list structure `L` containing nonnegative integers. Its return value should be a function that, given the list `(0 1 2 3 4 ...)` as argument, would return `L`. For example:

```
> (define f (regroup '((0 1) 4 (2 3))))
> (f '(the fool on the hill))
((THE FOOL) HILL (ON THE))

> ((regroup '(1 0 3 2 5 4)) '(being for the benefit of mister kite))
(FOR BEING BENEFIT THE MISTER OF)
```

Another way to say it is that each number in the argument to `regroup` indicates the starting position of an element in the argument to the returned function.

Your name _____ login cs61a–_____

Question 5A (5 points):

We are going to use OOP to simulate tape players and tapes. We can put a tape into a player, play it, fast forward it, and rewind it.

(a) We'll represent the music on a tape as a sentence containing the lyrics:

```
> (define help!
    (instantiate tape '(Help! I need somebody Help! not just ...)))
```

Each tape will also remember its position within that sentence. (Its initial position is before the first word.) A tape accepts two messages, `left` and `right`. `Left` means to move one position toward the beginning of the tape, returning the word passed over. `Right` is the same, but moving toward the end:

```
> (ask help! 'right)
HELP!
> (ask help! 'right)
I
> (ask help! 'right)
NEED
> (ask help! 'left)
NEED
```

If the tape is asked to move past the beginning or end of its sentence, it should return `#f` and not move.

Implement the `tape` class in OOP notation.

This question continues on the next page.

Question 5A continued.

(b) A tape player understands the messages `load`, `play`, `fast-forward`, `rewind`, and `eject`. Here's how they work:

```
> (define sony (instantiate tape-player))
> (ask sony 'load help!)
LOADED
> (ask sony 'play 5)                          ;; play five words
(HELP! I NEED SOMEBODY HELP!)
> (ask sony 'play 3)                          ;; play three more words
(NOT JUST ANYBODY)
> (ask sony 'fast-forward 4)                  ;; skip over four words
OK                                            ;; ["help! you know I"]
> (ask sony 'play 3)
(NEED SOMEONE HELP!)
> (ask sony 'rewind)                          ;; back to the beginning
OK
> (ask sony 'play 3)
(HELP! I NEED)
> (ask sony 'load she-loves-you)
ERROR -- TAPE ALREADY LOADED
> (ask sony 'eject)
OK
> (ask sony 'load she-loves-you)
LOADED
> (ask sony 'play 5)
(YOU THINK YOU LOST YOUR)
```

Implement the `tape-player` class in OOP notation.

You may continue your answer on the following (blank) page, if needed.

Your name _____ login cs61a–_____

Continue your answer to question 5A here.

Question 5B (5 points):

We're going to use streams for signal processing. Suppose you have an irreplaceable old, scratchy record, and you'd like to clean up the clicks and pops. First you digitize the information on the record, producing a stream of the voltages at each sampling interval (usually 44,000 per second, if the result will be recorded on a CD). Then we eliminate clicks by processing this stream using three procedures: The first procedure, `differences`, takes a stream of numbers as its argument and returns a stream of the differences between successive elements of the argument. If we represent a stream as `[element1 element2 ...]` then here's an example:

```
> (differences [0 5 8 6 3 7 26 9 5 ...])
[5 3 -2 -3 4 19 -17 -4 ...]
```

The second procedure, `limit`, takes as its arguments a stream of numbers and a limiting number. It returns a copy of its argument stream, but with every number greater than the limit replaced by the limit, and every number less than the negative of the limit replaced by the negative of the limit:

```
> (limit [5 3 -2 -3 4 19 -17 -4 ...] 6)
[5 3 -2 -3 4 6 -6 -4 ...]
```

The third procedure, `sums`, takes a stream of numbers as argument and returns a stream containing partial sums—that is, the first number, then the sum of the first two numbers, then the sum of the first three, and so on:

```
> (sums [5 3 -2 -3 4 6 -6 -4 ...])
[5 8 6 3 7 13 7 3 ...]
```

`Sums` is essentially the inverse of differences. For any stream `s`,

`(sums (differences (cons-stream 0 s)))`

will be the same as the original `s`.

With these tools we can write a declicking procedure:

```
(define (declick strm limit-value)
  (sums (limit (differences strm) limit-value)))
```

Your job is to write `differences`, `limit`, and `sums`.

Continue your answer on the following (blank) page.

Your name _____ login cs61a–_____

Continue your answer to question 5B here.

Question 6A (5 points):

Write a logic program to implement the `remove` relation that holds if one list is a copy of another with a particular value removed:

```
query==> (remove a (a b a c a d e f a x) ?what)
(REMOVE A (A B A C A D E F A X) (B C D E F X))
DONE

query==> (remove ?what (a b c a d e a b d) (a c a d e a d))
(REMOVE B (A B C A D E A B D) (A C A D E A D))
DONE

query==> (remove ?element (a b c a d e a b d) ?result)
(REMOVE A (A B C A D E A B D) (B C D E B D))
(REMOVE B (A B C A D E A B D) (A C A D E A D))
(REMOVE C (A B C A D E A B D) (A B A D E A B D))
(REMOVE D (A B C A D E A B D) (A B C A E A B))
(REMOVE E (A B C A D E A B D) (A B C A D A B D))
DONE
```

Do not use `lisp-value`**!** You may use the following rule:

```
(rule (same ?x ?x))
```

Your name _____ login cs61a–_____

Question 6B (5 points):

Sometimes you're writing a program and you can't find your bug, but you know that the value of a particular variable is changing to some wrong thing. But you don't know quite when the `set!` that changes the value is happening. What you'd like is to be able to say, "Whenever this variable changes its value, invoke this procedure with the old and new values as arguments." The procedure might print a message to help you debug, for example.

A variable with this magic behavior is called an *active* variable. We'd like to be able to do this:

```
> (define x 17)
> (activate x (lambda (old new)
                (print "Changing x from ")
                (princ old)
                (princ " to ")
                (princ new)
                (+ new 2)))
> x
17
> (set! x 3)
CHANGING X FROM 17 TO 3
> x
5
```

As this example shows, the new value of the variable should be whatever value the procedure returns.

(a) Is this primarily a change to `eval` or to `apply`?

(b) What specific procedure(s) will you change?

(c) Make the changes on the following pages.

This question continues on the following page.

Question 6 continued:

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((quoted? exp) (text-of-quotation exp))
        ((variable? exp) (lookup-variable-value exp env))
        ((definition? exp) (eval-definition exp env))
        ((assignment? exp) (eval-assignment exp env))
        ((lambda? exp) (make-procedure exp env))
        ((conditional? exp) (eval-cond (clauses exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence (procedure-body procedure)
                        (extend-environment
                         (parameters procedure)
                         arguments
                         (procedure-environment procedure))))
        (else (error "Unknown procedure type -- APPLY" procedure))))
```

```
(define (list-of-values exps env)
  (cond ((no-operands? exps) '())
        (else (cons (eval (first-operand exps) env)
                    (list-of-values (rest-operands exps)
                                    env)))))
```

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

This question continues on the following page.

Question 6 continued:

```
(define (extend-environment variables values base-env)
  (adjoin-frame (make-frame variables values) base-env))



(define (adjoin-frame frame env) (cons frame env))



(define (make-frame variables values)
  (cond ((and (null? variables) (null? values)) '())
        ((null? variables)
         (error "Too many values supplied" values))
        ((null? values)
         (error "Too few values supplied" variables))
        (else
         (cons (make-binding (car variables) (car values))
               (make-frame (cdr variables) (cdr values))))))


(define (make-binding variable value)
  (cons variable value))



(define (make-procedure lambda-exp env)
       (list 'procedure lambda-exp env))



(define (make-binding variable value)
  (cons variable value))

(define (binding-variable binding)
  (car binding))

(define (binding-value binding)
  (cdr binding))

(define (set-binding-value! binding value)
  (set-cdr! binding value))
```

This question continues on the following page.

Question 6 continued:

```
(define (lookup-variable-value var env)
  (let ((b (binding-in-env var env)))
    (if (found-binding? b)
        (binding-value b)
        (error "Unbound variable" var))))

(define (binding-in-env var env)
  (if (no-more-frames? env)
      no-binding
      (let ((b (binding-in-frame var (first-frame env))))
        (if (found-binding? b)
            b
            (binding-in-env var (rest-frames env))))))

(define (extend-environment variables values base-env)
  (adjoin-frame (make-frame variables values) base-env))

(define (set-variable-value! var val env)
  (let ((b (binding-in-env var env)))
    (if (found-binding? b)
        (set-binding-value! b val)
        (error "Unbound variable" var))))

(define (define-variable! var val env)
  (let ((b (binding-in-frame var (first-frame env))))
    (if (found-binding? b)
        (set-binding-value! b val)
        (set-first-frame!
          env
          (adjoin-binding (make-binding var val)
                          (first-frame env))))))

(define (eval-assignment exp env)
  (let ((new-value (mini-eval (assignment-value exp) env)))
    (set-variable-value! (assignment-variable exp)
                         new-value
                         env)
    new-value))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (mini-eval (definition-value exp) env)
                    env)
  (definition-variable exp))
```