Computer Science 61A, Fall 2004   University of California, Berkeley

Exam 2A November 8, 2004 8-10PM

<u>DRAFT SOLUTIONS 11/10/04 9:15AM</u> (points in {})

This is an open-book test. You have approximately two hours to complete it. You may consult any books, notes, or other paper-based inanimate objects available to you. To avoid confusion, read the problems carefully. If you find it hard to understand a problem, read it again, and do your best to answer it. If you think you have found a typographical error come to the front or the side of the room to ask about it. It would not be fair to give you extra personalized help during the test in understanding a question, and so we won't do it.

Partial credit may be given for partially correct answers.

Note that **some** questions on this exam offer the option to "punt". This is a way for you to receive partial credit for recognizing that you do not know the answer. For these questions, you have two choices. You may supply an answer, which, as usual, will receive anywhere between 0% and 100% of the points depending on how correct it is. Alternatively, you may check "punt" in which case you will receive 20% of points specified in exchange for us not having to grade the question.

Your exam should contain 8 problems (numbered 0 through 7) on 10 pages. Please write your answers in the spaces provided in the test. <u>DO NOT START UNTIL WE TELL YOU TO BEGIN.</u>

Question	MAX POINTS	YOUR POINTS	punt
0	1		
1	15		
2	15		
3	5		
4	12		
5	5		
6	14		
7	8		
TOTAL	75		/5

0. [1 point] Your first name ________________ Your last (family) name_______________
Your Teaching Assistant's name ___________
The day and time that your discussion section meets ________________
The seat number of your seat ______.
The row number (we will help you with this, later) _______

<u>CLEARLY PRINT YOUR CS61A LOGIN ON EVERY PAGE</u>!!!!   11/8; 9AM

1. Here are two programs, **filter** and **limit**, along with minor variations on them. You have already seen the first version in lecture. We point out places where programs differ from earlier ones by lines marked with ;;*.

```
(define (filter pred? L)
  (define (loop L)
    (if (null? L) '()
       (let ((hd (car L))
             (rst (cdr L)))
         (if (pred? hd)
             (cons hd (loop rst))
             (loop rst)))))
  (loop L))

;; list of all elements in L less than max
(define (limit L max)
  (filter (lambda(x)(< x max)) L))
;;.......................

(define (filter2 pred? L)
  (define (loop L)
    (if (null? L) '()
       (let ((hd (car L))
             (rst (cdr L)))
         (if (< hd max)  ;;*
             (cons hd (loop rst))
             (loop rst)))))
  (loop L))

(define (limit2 L max)
  (filter2 (lambda(x)(< x max)) L))
;;.......................

(define (filter3 max L)
  (define (loop L)
    (if (null? L) '()
       (let ((hd (car L))
             (rst (cdr L)))
         (if (< hd max)
             (cons hd (loop rst))
             (loop rst)))))
  (loop L))

(define (limit3 L max)  (filter3 max L)) ;;*
;;.......................

(define (filter4 L)  ;;*
  (define (loop L)
    (if (null? L) '()
       (let ((hd (car L))
             (rst (cdr L)))
         (if (< hd max)
             (cons hd (loop rst))
             (loop rst)))))
  (loop L))

 (define (limit4 L max)   (filter4 L)) ;;*
;;.......................
```

```
(define (filter5 pred? L) ;;*
  (define (loop L)
    (if (null? L) '()
         (if (pred? (car L)) ;;*
             (cons (car L)(loop (cdr L))) ;;*
             (loop (cdr L))))) ;) ;;* oops had one too many rpar
  (loop L))

(define (limit5 L max)
  (filter5 (lambda(x)(< x max)) L))

;;.......................

(define (filter6 pred) ;;*
  (define (loop L)
    (if (null? L) '()
         (if (pred)  ;;*
             (cons (car L) (loop (cdr L)))
             (loop (cdr L))))); oops had one too many rpar
  (loop L))

(define (limit6 L max)
  (filter6 (lambda ()(< (car L) max)))) ;;*
```

The questions consist of statements that may be **T [true],** or **F [false]** , *or need some explanation.* You must answer the question, as indicated. You should check (**F [false] describe**) if you believe a program does not return an answer but stops with an error. Then for full credit *you must describe clearly the reason for the error.* If the error has to do with the environment explain why in complete English sentences. You may draw a picture, but only to supplement, not replace, your explanation.

Some questions have another alternative (**punt**): Read the instructions on the cover page. A "punt" is worth 20%.

1A. Does limit work as described in the comment? {1 point}
_X_Yes, (limit '(1 2 3 4 5) 4) returns (1 2 3).
___Yes, (limit '(1 2 3 4 5) 4) returns (1 2 3 4).
___Yes, (limit '(1 2 3 4 5) 4) returns (4 3 2 1).
___Yes, (limit '(1 2 3 4 5) 4) returns (4 5).
___ No, it does something else (describe) _________________________
___ I don't know (punt)

1B. limit2 returns the same value as limit. {2}
___T
X__F Describe
___F Punt

The variable max is unbound in the environment in which filter2 is called.

1C. limit3 returns the same value as limit. {1}
_X_T
___F Describe
___F Punt

1D. limit4 returns the same value as limit. {2}
___T
_X_F Describe
___F Punt

The variable max is unbound in filter4.

1E. limit5 returns the same value as limit. {2}
_X_T
___F Describe
___F Punt
Unless you noticed one too many right parens, in which case filter5 can't work because it is not syntactically correct.

1F. limit6 returns the same value as limit. {2}
___T
_X_F Describe
___F Punt

variables L and max are both unknown inside filter6; also an extra rpar. If you said something about values returned you didn't get full credit.

1G**. filter5** might take a much longer time using **eval-1** than the other filters because it calls (car L) twice and L might be very long. (T/F _F__) {1}
1H**. loop** in **filter2** runs an iterative process. (T/F _F__) {1}

1I. **f1** and **f2** below seem to return the same values for any L, a list of integers. Why? {3}

```
(define (f1 L) (accumulate + 0 (filter odd? (map square L))))
(define (f2 L) (accumulate + 0 (map square (filter odd? L))))
```

```
SOLUTION. The squares of even numbers are even and the squares of odd numbers
are odd. Therefore the result is the same if you filter out the odd ones before
or after squaring: you get the same result.
```

2. Sometimes it's difficult to remember the order of arguments to a procedure. Consider the following contrived function:

```
 (define (f a b c)
   (list a (list b) c))
```

It is designed to be called like this:

```
STk> (f 1 2 3)  ;; a=1 b=2 c=3
(1 (2) 3)
```

We'd like to create a function **g** that is just like **f** except that each argument to **g** is preceded by a symbol which we shall call a keyword. This keyword makes explicit the parameter with which the actual argument is to be associated, eliminating the need to remember the exact order of the parameters:

```
STk> (g 'a 1 'b 2 'c 3)
(1 (2) 3)
STk> (g 'b 2 'a 1 'c 3)
(1 (2) 3)
STk> (g 'c 3 'b 2 'a 1)
(1 (2) 3)
```

The goal is to write a function **make-keyword-proc** that takes two arguments. The first is a regular Scheme procedure like **f**. The second is a list of the keywords in the order that the procedure expects. **make-keyword-proc** should return a function like **g** that takes keyword arguments and calls **f** with the arguments in the order **f** expects. Continuing the example above, to create the function **g** from **f** we can use make-keyword-proc like this:

```
STk> (define g (make-keyword-proc f '(a b c)))
```

The keywords do not have to match the names of the formal parameters of f:

```
STk> (define g2 (make-keyword-proc f '(cs61a cool is)))
 STk> (g2 'cs61a 1 'is 2 'cool 3) ;;note, keyword order irrelevant in call
 (1 (2) 3)
```

Additionally, a call like

```
STk> (g2 'is 2 'cs61a 1 'cool 3 'notcool 4)
```

must work because `notcool` is not a keyword and doesn't affect the call to **f**. Your program should work so

```
STk> (g2 'cs61a 1)
```

will call **f** on arguments `1, #f, #f.`

2A. First write a program, call it **evenmember**, with the following specification: ___Punt (20%) {3 points}

```
;; given an element x and a list L whose length is even,
;; return the sublist of L beginning with the first value eqv? to x,
;; checking only positions 0, 2, 4 ..  Examples
;;(evenmember 'b '(a x b y c z)) is  (b y c z)
;;(evenmember 'b '(a b b y c z)) is  (b y c z)
;;(evenmember 'b '(a b c d)      is ()

;;SOLUTION
 (define (evenmember x L)
  (if (pair? L)
      (if (eqv? (car L) x) L
                (evenmember x (cddr L)))
      '()))
```

2B. Next, using **evenmember**, write **make-keyword-proc**. In order to keep your program short, assume that the keyword argument list supplied is of even length, alternating keynames and values. ___Punt (20%) {8}

```
SOLUTION
(define (make-keyword-proc proc keylist)
  (lambda args
    (apply proc (map
             (lambda(keywd)(and (not (null?(evenmember keywd args)))
                                (cadr (evenmember keywd args))))
             keylist))))
```

2C. Finally, assume that we want to build keywords in to **eval-1**, and implement keyword argument functions by a new special form similar to **lambda**, but called **keyword-lambda**, say as
`(define f (keyword-lambda(a b c) ….)`

How would you change **eval-1** to accommodate this? You should use one or more complete English sentences. {4}
Describe
___Punt (20%)

```
SOLUTION
When a keyword-lambda expression is encountered, it will be converted
to a specially marked procedure that, if it is applied (using apply)
will match up the formal parameters and the arguments according the
keyword specifications like the inside of make-keyword-proc
```

3. In midterm 1 we asked about a version of lisp in which there were no pairs, but just triples that had three components. One common response was to point out that we can already make triples without rewriting lisp. But now we know of a strange encoding as programs. We can define a tree-cons this way

```
(define (tree-cons label left right)
  (lambda(which)
      (cond ((= which 0) label)
            ((= which 1) left)
            ((= which 2) right)
            (else (error "illegal access to tree-cons")))))
```

3A. Show us you understand this representation by defining the accessor function for
`right-tree.  {2 pts} [no partial credit here]`

```
(define (right-tree tc)(tc 2)) ; similar for other version of test.
```

3B. Also explain why the predicate **tree-cons?** would be difficult to write, using a complete English sentence.
(or punt____): get 20% for saying “I have no idea”. {3}

```
SOLUTION: The predicate tree-cons? must be able to distinguish an
instance of a tree-cons, a function, from other functions; however,
Scheme officially provides no tools for comparing “closure procedure”
parts.  Even if we could come up with a test that says “this is a
procedure of one argument allowing 0, 1, 2”: it might not be a tree-
cons. Oddly, you could try this: surround a call to the alleged tree-
cons function T with some “error catching” mechanism.  See if (catch (T
3)) returns a string “illegal access to tree-cons” and from that
conclude it is a tree-cons, since who else would say that?
```

4. (Option: you can punt all of question 4 _____ for 20%)

4A. Define a function **every-nth** that takes two arguments integer *k* and integer *n* > 0, and returns a *stream*: the infinite stream of numbers beginning at the specified integer *k* and whose succeeding elements are *k*+*n*, *k*+2*n*, *k*+3*n*, ... For example, if *k*=0 and *n*=3, the stream will begin with 0, 3, 6, 9, 12, 15, 18. {2 pts}

```
SOLUTION
(define (every-nth start n)(cons-stream start (every-nth (+ start n)
n)))
```

4B. Define a stream **r1** of all non-negative integer multiples of 3. {1, 1} [Why did some people not use every-nth for this??]

```
(define r1 (every-nth 0 3))
```

Define another stream **r2** of all non-negative integer multiples of 5.

```
(define r2 (every-nth 0 5))
```

4C. Define a function **both-streams** that given two streams s1 and s2 each of (increasing) numbers, returns a new stream s3 of those numbers that appear in both s1 and s2. For example, **(both-streams r1 r2)** would begin with 0, 15, 30, ... {5 pts}

```
SOLUTION
(define (both-streams s1 s2)
  (cond ((null-stream? s1) '())
        ((null-stream? s2) '())
        ((eqv? (stream-car s1)(stream-car s2))
         (cons-stream (stream-car s1)
                  (both-streams (stream-cdr s1)(stream-cdr s2))))
        ((< (stream-car s1) (stream-car s2))
         (both-streams (stream-cdr s1)s2))
        (else (both-streams s1 (stream-cdr s2))))))
```

4D. Are there any circumstances in which **both-streams** will not return? Explain, using examples as appropriate. {3}

```
Yes, if the streams have no numbers in common, both-streams will not
return. An example would be even and odd number generated by(every-nth
0 2) and (every-nth 1 2).
```

5. You are familiar with **set!** and **set-car!** One of them is a special form and the other is not. Explain why this is the case in complete English sentences.

```
set! is a special form because (set! x y) must not evaluate its first
argument. Set-car! is an ordinary function because it must evaluate x,
which presumably evaluates to a pair, and it changes the car of that
pair.
```

6. Henry wants to design a Scheme bank account, protecting his account with a password or personal identification number (PIN). *He is not as good a programmer as you are* and asks you to help him. He has written a first draft of his **new-account** program below. Be sure to notice that it is different from the one you have seen before. You should be able to help him to r**ewrite it all in one new version.**
(or you can punt the whole question _____ for 20%)

6A. Enhance the program in a minimal fashion so that he can only take money out if he remembers the correct PIN.
What is a command to extract $10?

```
(HA 'withdraw) 100 PIN) {1}
```

6B. Enhance the program below so that he can change the PIN.
What is a command to change the PIN to 1234?

```
(HA 'changePIN) PIN 1234) {1}
```

6C. Perhaps because of a design error, anyone who has (a pointer to) Henry's account can find out the balance without knowing Henry's PIN. (Hint: compare the **deposit** and **withdraw** programs). What command would you use in the given program to find out Henry's balance?

```
((HA 'deposit) 0)  {1}
```

6D. Change the program to protect Henry's privacy, but add a feature so that Henry's benefactors (perhaps his parents) can deposit money, *without knowing his PIN, or finding out his balance*. Write such a program, **give-money** so that (**give-money HA 100)** where HA is Henry's account, will deposit 100 dollars in his account.
see below

6E. Finally, change the program so that it causes an error to deposit or withdraw a negative amount of money.
see below

```
;;SOLUTION for problem 6  {11 more points. Many variations made this hard to
grade.…}

(define (new-account  balance origPIN) ; PIN is personal identification
number
  (define (withdraw  amount PIN) ;; TEST VERSION A  ;;{3}
    (cond((equal? PIN origPIN)
          (set! balance (- balance (abs amount)))
          balance)
         (else (error "bad PIN"))))
 (define (withdraw  amount PIN)  ;; TEST VERSION B
    (cond((not(equal? PIN origPIN)) (error "bad PIN"))
         ((< amount 0)(error "bad amount"))
         (else
          (set! balance (- balance  amount))
          balance)))
  (define (deposit amount) ;; VERSION A  ;;{2}
    (set! balance (+ balance (abs amount))) balance) ; never negative

  (define (deposit amount) ;; VERSION B
        (cond
         ((< amount 0)(error "bad amount"))
         (else (set! balance (+ balance  amount))
          balance))

  (define (pdep amount)  ;;{2}
    (set! balance (+ balance (abs amount)))  ;; or could check parents
;too.
    'Thanks) ;<<<<<

  (define (changePIN old new)  ;;{1}
    (if (equal? old origPIN)(set! origPIN new)(error "bad PIN")))

  (define (respond-to-msg msg)
    (cond
      ((eq? msg 'withdraw) withdraw)
      ((eq? msg 'deposit) deposit)
      ((eq? msg 'changePIN) changePIN)  ;;{1}
      ((eq? msg 'parentsdep) pdep)      ;;{1}
      (else (error "Bad message: ~s to account" msg))))
  respond-to-msg)

(define (give-money ACCT AMT)   ;;{1}
   ((ACCT 'parentsdep) AMT))
```

7. In eval-1, there is no "set!". Add it. (or punt___ for 20%)
You may wish to base your answer on the following program already included in eval-1.

Check here if you have tried to do this problem previously either during a discussion section, review session, or with friends _____. {0, just curious}|

```
....
(define (lookup-variable-value var env0 env1)
   (let ((match (or (assq var env0) (assq var env1))))
       (if match (cdr match)
          (error "Undefined variable: " var))))
```

and by recalling this part of eval

```
((eq? kind 'symbol)
 (lookup-variable-value exp
                  inner-env outer-env))
```

```
SOLUTION:
There are a bunch of ways of doing this, including putting set! in the special
forms list, and then doing (eq? kind 'set!)  as a test in eval-1.

;;A kind of minimal change to eval-1, ignoring most abstractions etc would be
….
((eq? (car exp) 'set!)(set-cdr!  ;{2}
             (lookup-variable-location (operand 1 exp) ;{2}
                                                      env0 env1)
                                 (eval-1 (operand 2 exp) env0 env1))      ;;{2}
....

;;where
(define (lookup-variable-location var env0 env1)   ;;{1,1}
  ;; return (name . val) or bomb.
    (or (assq var env0)
        (assq var env1)
        (error "Undefined variable: " var))) ; don't add it to env.


;;; many other (more elaborate) versions possible. failure to use
mutation is a bad sign. (max 4 points). Many people used the punt
option, making this easier to grade than expected.
```