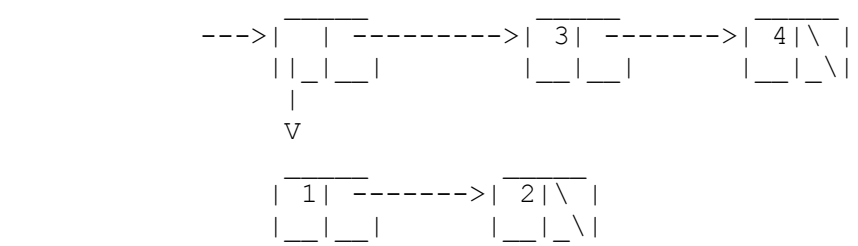
CS61A Midterm 1, Spring 2011

1. What would Scheme print? (6 points)

(a) (cons '(1 2) '(3 4))

What would Scheme print?
((1 2) 3 4)

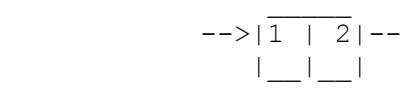
Box and pointer diagram:



Common Errors:

1. The most common error was to draw a dot between (1 2) and the remaining elements of the list: ((1 2) . 3 4), or between all of the elements ((1 2) . 3 . 4). CONS does not insert dots when the second element is a list. This often resulted in a box and pointer diagram like (b) below.

2. Another error was to have boxes that had numbers /and/ arrows in them.

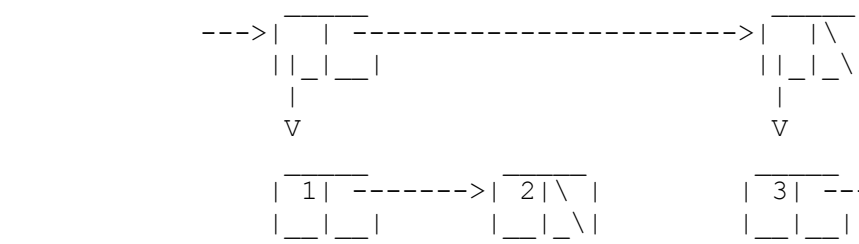


Remember, the car or cdr of a box can /either/ point to another box /or/ a simple value (like a word or #t/#f)!

(b) (list '(1 2) '(3 4))

What would Scheme print?
((1 2) (3 4))

Box and pointer diagram:



Common errors:

1. A common error was to flatten the list - to draw the box and pointer diagram for (1 2 3 4) instead of ((1 2)

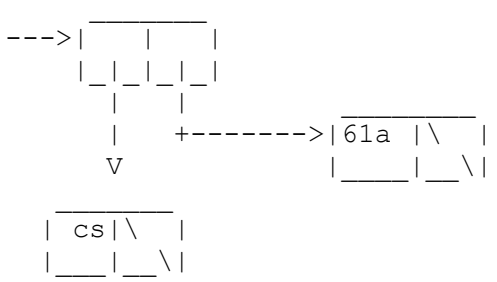
(3 4)). This comes from a misconception about the way that list works. It does not flatten its elements one level like APPEND. It simply takes them and inserts them into a new list.

2. Another error was similar, to claim that the answer to this question is the same as the answer to the first. In this case, the incorrect assumption is that LIST works the same way as CONS.

(c) ((lambda (x y) (cons y x)) '(61a) '(cs))

What would scheme print?
((cs) 61a)

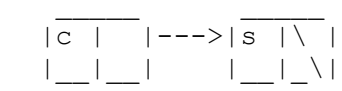
Box and pointer diagram



Common Errors:

1. The most common error was missing the switched x and y -- drawing ((61a) cs) instead of ((cs) 61a). Only one point out of two was taken off for this.

2. Another common misconception was that the individual letters of the symbols 61a and cs each take up an entire box slot. This resulted in an incorrect drawing, even if the scheme expression was correct, something like this:



3. Another error was writing ((cs) . 61a). A CONS expression does not produce a dot if the second part is a list, which it was in this case.

Scoring:
2pts each: 1 for the result and 1 for the diagram. However, if both were incorrect but your diagram matched your result, you still got 1 point.

2. Functional or not? (4 points)

Most students seemed to forget that something is "functional" in this

course if, given any input, always returns the same output. The usual example of something that is not functional is a random number function. As a result, things that use other non-functional things as part of computing their return value also aren't functional (like JORDY, which used CHRIS).

```
(define (chris x)
  (random x) )
```
NOT functional.

```
(define (jordy x)
  (+ 1 (chris x)) )
```
NOT functional.

```
(define (aditi x)
  3)
```
Functional.

```
(define (ingrid x)
  (lambda (y) (+ x y)) )
```
Functional.

Scoring: 1pt per procedure.

3. Iterative or Recursive? (3 points)

Most people got the first two procedures right, but got thrown off by the last one, which if you're not looking carefully doesn't appear to be recursive (since it returns the value of another call to DARREN), but happens to need another call to DARREN to finish before it can begin the "outside" recursive call to get its return value.

```
(define (eric sent)
  (if (empty? sent)
      0
      (+ (first sent) 10 (eric (bf sent))) ))
```
Recursive.

```
(define (tom lst n)
  (cond ((null? lst) n)
        ((= (car lst) n) (tom (cdr lst) (+ 1 n)))
        (else (tom (cdr lst) n)) ))
```
Iterative.

```
(define (darren lst1 lst2)
  (if (or (null? lst1) (null? lst2))
      '()
      (darren (cdr lst1)
              (darren (cdr lst1) (cdr lst2)) )))
```
Recursive.

Scoring: 1pt per procedure.

4. Orders of Growth (3 points)

(The "O"s below should technically be thetas, but there's no theta key.)

```
(define (sum sent)
  (if (empty? sent)
      0
      (+ (first sent) (sum (bf sent)))))
```

Runtime: O(n).

Each time, we recurse on the butfirst of the sentence. This means that we look at each element exactly once. Taking n as the length of the sentence, the runtime is O(n).

```
(define (mystery sent)
  (cond ((empty? sent) 0)
        ((even? (first sent)) (+ (sum sent) (mystery (bf sent))))
        (else (mystery (bf sent))) ))
```

Runtime: O(n^2)
The function takes the longest time to run when each element is even. When all the elements are even, the MYSTERY function makes a call to SUM on the sentence during each recursive call. From above, we know that SUM takes O(n). In the worst case, we will have to do a O(n) operation n times, which is O(n^2)

(To be more precise, in the worst case, we will do O(n), O(n-1), O(n-2)...O(1)
operations respectively, since we are taking the butfirst of the sentence each
time. But the sum of O(n), O(n-1), O(n-2)...O(1) comes out to O((n^2-n)/2),
which is O(n^2).)

```
(define (enigma sent)
  (if (empty? sent)
      1
      (first sent) (sum (bf sent)) (sum (bf sent))) ))
```

Runtime: O(n)
The function makes two calls to SUM and one call to FIRST. The call to the function FIRST is a constant time operation, while the calls to SUM are O(n). Therefore, the total runtime is O(n) + O(n) + O(1), which is just O(n).

Scoring: 1pt each.
Full credit was given for equivalent answers, like O(n^2 + n) for O(n^2).

5. LET to LAMBDA (3 points)

```
(define (distance x1 y1 x2 y2)
```

```
 (let ((a (- x1 x2))
       (b (- y1 y2)))
   (sqrt (+ (* a a) (* b b))))
```

The question ask us to rewrite the LET with LAMBDA. The correct solution is:

```
(define (distance x1 y1 x2 y2)
  ( (lambda (a b)
      (sqrt (+ (* a a) (* b b))))
    (- x1 x2)
    (- y1 y2) ))
```

The secret formula is to take all the variables in the LET and use them as
parameters for a lambda, and use all the values of the variables as the arguments to the lambda.

The most common mistake was:

```
(define (distance x1 y1 x2 y2)
  (lambda (a b)
    (sqrt (+ (* a a) (* b b))))
  (- x1 x2)
  (- y1 y2) )
```

Where there's a missing parenthesis before the lambda call. Since LET consist of both creating a lambda and invoking the lambda, this solution is incomplete. 2 pts for this solution.

Another very common mistake was:

```
(define (distance x1 y1 x2 y2)
  (define (f a b)
    (sqrt (+ (* a a) (* b b))))
  (f (- x1 x2) (- y1 y2)))
```

The solution basically defines the F variable to hold the function, which is in some sense what LET is doing, but not what we are looking for.
2 pts for this solution.

A close solution is:

```
(define (distance x1 y1 x2 y2)
  ( ( (lambda (a)
        (lambda (b)
          (sqrt (+ (* a a) (* b b))) )
      (- x1 x2) )
    (- y1 y2) ))
```

This is what a LET* expression would translate to; the following code would translate to the solution above:

```
(define (distance x1 y1 x2 y2)
  (let* ((a (- x1 x2))
```

```
         (b (- y1 y2)))
    (sqrt (+ (* a a) (* b b))))
```

Another mistake was:

```
(define (distance x1 y1 x2 y2)
  ( (lambda (x1 y2 x2 y2)
      (sqrt (+ (* (- x1 x2) (- x1 x2)) (* (- y1 y2) (- y1 y2)))))
    x1
    x2
    y1
    y2))
```

This solution basically did nothing, since the solution below does the same:

```
(define (distance x1 y1 x2 y2)
  (sqrt (+ (* (-x1 x2) (-x1 x2)) (* (- y1 y2) (- y1 y2))) )
```

The solution with the lambda gets 1 pt, while the second one gets none. It's not because the first one is more correct, but rather the student attempts to recreate a lambda.

Another mistake is:

```
(define (distance x1 y1 x2 y2)
  (define a (- x1 x2))
  (define b (- y1 y2))
  (sqrt (+ (* a a) (* b b))))
```

The student basically defined the variables to use them, which is in some sense what LET is doing, but not what we are looking for (and not how it works). 1pt for this solution.

Scoring
3 - Perfect
2 - missing a set of calling parenthesis
  - internal definition of the function
  - converting as let* instead of let
1 - internal definition of variables
  - lambda for 4 variables
0 - entertaining (or not) solution that doesn't work

6. Louis's Calculator (2 points)

Louis Reasoner changed CALC-EVAL from:

```
  (define (calc-eval exp)
    (cond ((number? exp) exp)
          ((list? exp) (calc-apply (car exp) (MAP CALC-EVAL (CDR
EXP))))
          (else (error "Calc: bad expression:" exp)) ))
```

to:

```
  (define (calc-eval exp)
    (cond ((number? exp) exp)
```

```
            ((list? exp) (calc-apply (car exp) (CDR EXP)))
            (else (error "Calc: bad expression:" exp)) ))
```

Give an example of a valid calculator input that will break Louis's CALC-EVAL.

Answer: any list with a list as an argument, i.e. (+ (+ 1)).

Why?
The procedure CALC-EVAL takes an expression and returns a number, which is the result of completely evaluating the expression. A valid expression for the calculator program can either be a number or a list. For a valid list expression, the first item should be a word denoting a math operator (+, -, *, /), and the remaining elements should be expression arguments for the math operator *that when evaluted by CALC-EVAL, returns a number*. This means that the arguments themselves can be list expressions. However, CALC-APPLY takes two arguments: a math operator word and a list of numbers. To satisfy CALC-APPLY's argument requirement, CALC-EVAL is applied to all the arguments of the math operator -- (cdr exp) -- using (map calc-eval (cdr exp)).

In Louis's midguided program, CALC-APPLY is given the math operator word and the arguments of the math operator word unsimplified...and if any of the arguments is a list, CALC-APPLY will not work, and thus, CALC-EVAL will not work.

(See the code for CALC-APPLY if you don't know why it takes a math operator word and a list of numbers: ~cs61a/lib/calc.scm).

Common wrong answers:
- Infix notation: i.e. (+ 1 * 2)
  This case would be caught in CALC-EVAL as a "bad expression" error. When CALC-EVAL is called by (map calc-eval (cdr exp)) for the arguments of + (which are {1, *, 2}) (calc-eval *) will return an error, since * is not a number or a list.
- Any other invalid expression (for calc-eval) as the argument of a math operator: i.e. (+ (1 2) 3)
  This case would be caught in CALC-APPLY as a "bad expression" error. When CALC-EVAL is called by (map calc-eval (cdr exp)) for the arguments of + (which are {(1 2), 3}) (calc-eval (1 2)) will call CALC-APPLY with 1 as its math operator word, which will make CALC-APPLY return an error.

Scoring: 2 points, all or nothing.

7. True/False (4 points)

F: All lists are pairs. Counterexample: ()
F: All pairs are lists. Counterexample: (1 . 2)

For these, it's important to recall the definition of a list. A list can be
either:
(a) a pair with ANYTHING as its car and a LIST as its cdr.
(b) the empty list (), which is NOT a pair.
    (Ways to get the empty list: NIL, '(), (list))
Students seemed to have difficulty with these two T/F questions. (Moral: be
rigorous about your data type definitions! It is hard to work with a data type
when you don't know what it is exactly.)

T: It's valid to have a list of sentences
   This question most students did well at. A list can contain anything,
   including sentences.

F: It's valid to have a sentence of lists
   This questions had slightly less luck than the previous question. A sentence
   can ONLY contain words, including numbers (respect the data abstraction!).
   This allows it to have the structure of a list, but remain flat.

Scoring: 1pt each

8. REMOVE-ALL and COUNTS (8 points)

  (a) Write a procedure REMOVE-ALL that takes two arguments. The second
  argument should be a list. The result is that list with all occurences of
  the first argument removed.

There are essentially two standard ways of writing REMOVE-ALL. The first is
by using the higher-order procedure FILTER.

```
(define (remove-all item lst)
  (filter (lambda (x) (not (equal? item x))) lst))
```

The second is to use recursion.

```
(define (remove-all item lst)
  (cond ((null? lst) '())
        ((equal? item (car lst)) (remove-all item (cdr lst)))
```

```
        (else (cons (car lst) (remove-all item (cdr lst))))))
```

A common wrong answer was:

```
(define (remove-all item lst)
  (filter (not (equal? item x))) lst))
```

FILTER must take a procedure as its first argument, and here NOT would return
#t or #f. (STk will actually complain that X is not defined, of course.)

We took off -1 over the whole problem if you committed a data abstraction violation (usually using sentence procedures).

Scoring:
3 correct
- including deep list removal
2 "the idea"
- tests for equality
- attempt to include the element in one branch and not in the other OR using filter (not map!)
- list combiner /may/ be incorrect (like using APPEND instead of CONS)
1 "an idea"
- recursion or some HOP
0 other

(b) Write a procedure COUNTS that, given a list, returns a new list that
encodes the number of times each element appears as a pair of the form
(<element> . <# of occurences>)

First, I'll give one solution:

```
; Using remove-all from 9a
(define (counts lst)
  (if (null? lst)
      '()
      (cons (cons (car lst)
                  (- (length lst) (length (remove-all (car lst) lst))) )
            (counts (remove-all (car lst) lst) ))))
```

This solution is nice, because it's a simple way to tackle the main aspect of
this problem - mainly, counting each element of a list, while creating a new
list at the same time. Note that this is a case where the recursive call isn't
something simple like (counts (cdr lst)) - we don't have to cdr down the list,
otherwise we'd re-count elements! (or, you could "rememeber" if you've seen

certain list elements as you cdr down the list, but that's just messy and not a
good idea).

A very common problem was to use an incorrect combiner, such as list or append rather than cons, i.e:

```
    ...
    (list (cons (car lst)
                (- (length lst) (length (remove-all (car lst) lst))) )
          ...)
    ...
```

You should pretty much never have to use (list ...) as the combiner in a recursive function, because list will create /lots/ of nested parenthesis in the
final output.

For those who didn't think to do
`(- (length lst) (length (remove-all (car lst) lst)))`,
an alternate solution (which was very commonly done) was to define a helper
procedure that counted up how many times an element appeared in a given list,
returning the pair (<element> . <# of occurrences>):

```
(define (counts lst)
    (define (counts-helper el lst occur)
        (cond ((null? lst) (cons el occur))
              ((equal? el (car lst)) (counts-helper el (cdr lst) (+ 1 occur)))
              (else (counts-helper el (cdr lst) occur))))
    (if (null? lst)
        '()
        (cons (counts-helper (car lst) lst 0) (counts (remove-all (car lst) lst)))))
```

With this solution, there's the danger of being off-by-one - if you had done:

```
        (cons (counts-helper (car lst) lst 1) ... )
```

Then you just counted the first element twice - whoops! The 'right' way to do
that would have been:

```
        (cons (counts-helper (car lst) (cdr lst) 1) ... )
```

But that looks a bit strange (to us) - better to keep things simple and start from 0!

Oh, and those who love Higher-Order Functions (like Eric!);

```
(define (counts lst)
    (if (null? lst)
        ()
        (cons (cons (car lst) (count-of (car lst) lst))
              (counts (remove-all (car lst) lst)) )))

(define (count-of item lst)
```

```
    (length (filter (lambda (x) (equal? item x)) lst)) )
```

Note: Some students may have incorrectly interpreted the question in the following way:

```
    > (counts '(a a a a b b b c d d a))
    ((a . 4) (b . 3) (c . 1) (d . 2) (a . 1))
```

In other words, a procedure that condenses consecutive runs of elements. This isn't what we asked - but, we gave 3 points to a /correct/ solution to this problem.

Grading Rubric
5 perfect
4 trivial mistakes
3 "the idea"
- must use CONS to make elements of output
- recursive call includes a call to removes-all of current element
- list combiner /may/ be incorrect
- can incorrectly use (equal? (car lst)) instead of
  (lambda (x) (equal? (car lst) x)) as "predicate" for filter
- a fully correct solution for COMPRESS (i.e. condensing runs)

1 "an idea"
- either makes pairs properly OR makes recursive call properly

0 other

-1 DAVs (floored at 1)

9. HALF-DONE-CALC (6 points)

Write a procedure HALF-DONE-CALC that takes a procedure (either + or *), and a list of numbers. It should return a new procedure that takes a second list of numbers, and performs the + or * calculation on the first and second list of numbers.

The domain and range of HALF-DONE-CALC was very clearly laid out in the problem description, so we didn't have to worry about that.

```
(define (half-done-calc proc first-list)
  (lambda (second-list)
    ; perform the + or * calculation on both lists
  ))
```

The cleanest way to actually do the calculation was to put the two lists together using APPEND, then use APPLY:

```
(define (half-done-calc proc first-list)
  (lambda (second-list)
    (apply proc (append first-list second-list)) ))
```

If you forgot about APPLY, you could have used ACCUMULATE, but then you have to
pick a base case. For + it should be 0, but for * it should be 1!

Fortunately, a simple IF can take care of that:

```
(define (half-done-calc proc first-list)
  (lambda (second-list)
    (accumulate proc
                (if (equal? proc +) 0 1)
                (append first-list second-list) )))
```

If you were very clever, you might have remembered the first lab, in which we
found out that (+) is 0 and (*) is 1, and then solved the problem like this:

```
(define (half-done-calc proc first-list)
  (lambda (second-list)
    (accumulate proc
                (proc)
                (append first-list second-list) )))
```

Finally, a number of people just used the two-argument version of ACCUMULATE
from CS3, which was also given full credit.

The most common mistake was probably comparing PROC against the /word/ +,
rather than the addition primitive itself. That is:

```
(if (equal? proc '+) 0 1)
```

Why is this wrong? Well, if PROC /was/ the word +, then we couldn't use it
with APPLY or ACCUMULATE to actually add up the items in the list. (And the
same for *, of course.)

6 for a correct solution
4 "The Idea", including:
- comparing against the procedure name, as above
- one extra set of parentheses around the body of HALF-DONE-CALC
  (remember, parentheses aren't for grouping in Scheme!)
- forgetting that the base cases for + and * are different
3 Using an /extra/ ACCUMULATE where you should have just used PROC
2 "An Idea", meaning you had the right domains and roughly the right range,
  and tried to use the function somehow.
- a couple of solutions used (cons proc first-list) to build an expression...but
  remember this expression won't actually be evaluated! Use APPLY instead.
1 HALF-DONE-CALC returns a lambda of one argument.

0 other