CS 61A   Fall 2007   Midterm 2 solutions

1. What will Scheme print?

```
> (append (cons (list 1 2) (list 2 3)) '(5 6))

((1 2) 2 3 5 6)

---->XX------>XX------>XX------>XX------>X/
     |        |        |        |        |
     |        V        V        V        V
     |        2        3        5        6
     V
     XX------>X/
     |        |
     V        V
     1        2
```

CONS makes one new pair, but from a list point of view, what it does is take a list as its /second/ argument, and extend that list at the left with one new element, the /first/ argument, whether that's an atom or a list. So (LIST 1 2) returns (1 2), and that becomes the first element of an extension of (2 3). So the CONS returns

```
        ((1 2) 2 3)
```

APPEND takes the three elements of that list, and combines them with the two elements of (5 6), producing the five-element list above.

By the way, the number 2 appears twice in this list. It doesn't matter if you have two 2s, as above, or one 2 that both pairs point to.

```
> (let ((y (list '(1) 2 3)))
    (cons '(7 . 8) (cdr y)))

((7 . 8) 2 3)

---->XX------>XX------>X/
     |        |        |
     |        V        V
     |        2        3
     V
     XX
    /  \
   V    V
   7    8
```

The main point here was to see if you could draw a correct box and

pointer diagram for a pair that isn't a list, the one that
prints
as (7 . 8). (By the way, is this result a list? Should LIST?
of it
return true? Yes, because the definition of a list only
mentions the
CDRs of the spine pairs: a list is either the empty list or /a
pair
whose CDR is a list/. That's true of this structure, even
though the
/CAR/ of one of the pairs isn't a list.

Many people thought that (7 . 8) was a 3-element list with 7, a
period, and 8. The dot in "dotted pair" notation is part of the
notation, just as the parentheses are part of the notation, not
part of the actual data.

Scoring: One point per printed form, one point per diagram. At most
one
point lost for quotation marks anywhere in the answer.

2. Another box and pointer diagram.

```
(3 (9 (2 7) 8) ())
 - ----------- --


---->XX------>XX------>//
     |        |
     V        |
     3        |
              V
              XX-------->XX-------->X/
              |          |          |
              V          |          V
              9          |          8
                         V
                         XX---->X/
                         |      |
                         V      V
                         2      7
```

This is a three-element list (the elements are underlined in the
print
form shown above the diagram). So its spine is three pairs.
The only
slightly tricky part is the third element, which is the empty
list.
This is shown with a slash through the CAR of the pair, just as
an
empty list in the CDR of a pair is shown with a slash. Many
people
made the CAR point to another pair (e.g., of empty lists)! Some
people had the CDR of the second pair in the spine point to the
empty

list; that would make it a two-element list.

3. Abstract data types.

(a) This part should have been easy; a hider is a list of two elements, and the three selectors have to return the first element, the CAR of the second element, and the CDR of the second element:

```
(define (hider-description hider)
  (CAR HIDER))                ; the first element

(define (encoder hider)
  (CAADR HIDER))              ; the CAR of the second element (CADR)

(define (decoder hider)
  (CDADR HIDER))              ; the CDR of the second element
```

Many people used CADR and CDDR for encoder and decoder, respectively. This
would be correct if the the LIST in the constructor had been a CONS instead,
but to select the second element of a list, use CADR.

Scoring: one point each.

(b) The whole point of this part of the problem is that it should use the
selectors you wrote -- no CARs or CDRs here:

```
(define (works? hider val)
  (EQUAL? VAL ((DECODER HIDER) ((ENCODER HIDER) VAL))))
```

Why two parentheses before DECODER and ENCODER? Those selectors return /procedures/, and we have to /invoke/ the procedures to do the work of this
procedure. So the inner parentheses are for the calls to the selectors, and
the outer parentheses are for the calls to the actual encoder and decoder
functions.

We didn't care if you used EQ? or = instead of EQUAL?, but the latter is the
best choice for a general equality test when you're not given any special
information about the types of values you'll get.

Scoring:

4 Correct.
3 One missing open parenthesis, or swapping the encoder and decoder.
2 Two missing open parentheses, or just encoding and decoding without testing for equality.
0 Anything else.

4. Scheme-1.

Of course Louis's change doesn't work.

Even if you didn't know anything about Scheme-1, you should be able to
figure
out part of this problem just by examining the change Louis makes. His
version handles Scheme-1 primitive procedures (the ones for which STk's
PROCEDURE? is true) the same way that the real version does: by calling
STk's APPLY. But Scheme-1 defined procedures, the ones for which
LAMBDA-EXP? returns true, are handled differently. So the thing that's
going
to break is invoking (because that's what calls APPLY-1) a defined
procedure!

```
Scheme-1> car                         =>   #[SUBR CAR]

Scheme-1> (* 3 7)                     =>   21

Scheme-1> (lambda (x) (+ x 1))        =>   (LAMBDA (X) (+ X 1))

Scheme-1> ((lambda (x) (+ x 1)) 3)    =>   ERROR
```

The first two are about primitive procedures, so they work correctly.
The
third is about a lambda-defined procedure, but it doesn't /invoke/ that
procedure; it just creates one. It's the fourth example that invokes a
defined procedure, and so that's the one that fails; Louis is trying to
use that Scheme-1 procedure, which is just a list as far as STk is
concerned,
as if it were an STk procedure (by calling STk's APPLY).

Knowing what Scheme-1 returns for the third example was the only part
that
really required knowing specific details about Scheme-1, namely the
fact that
Scheme-1 represents a defined procedure using the LAMBDA expression
itself.

Many people said that the first one would return an error. #[SUBR CAR]
is not
an error! This is just STk's way of printing a procedure. Many people
also
said CAR, without indicating it was a procedure. That's how Scheme (or
Scheme-1) would print the /word/ "CAR," which is a different thing.
Scheme-1
will evaluate the symbol CAR, and return the procedure. A few people
made
arithmetic errors in the second expression, but we didn't take off for
that.

Scoring: One point each.

5. Trees.

This problem is a little less elegant than many Tree problems because

it can't
have the empty word as its base case, if you write it in the most
straightforward possible way; the empty word would correspond to an
empty
Tree, and there are no empty Trees. So the base case is a one-letter
word.

The solution we expected examines every child to find a match for the
butfirst
of a word of more than one letter:

```
(define (contains-word? Tree wd)
  (cond ((and (empty? (butfirst wd))
              (equal? (first wd) (datum Tree))
              (leaf? Tree))
         #t)                   ; base case
        ((or (empty? (butfirst wd))
             (leaf? Tree))
         #f)                   ; one ran out, but not the other
        (else (and (equal? (first wd) (datum Tree))
                   (not (null? (filter (lambda (child)
                                         (contains-word? child (butfirst
wd)))
                                       (children Tree)))))))))
```

Many variants on this general plan are possible. One is that the base
case
can be simplified as follows:

```
  (cond ((and (equal? WD (datum Tree))
              (leaf? Tree))
         #t)                   ; base case
        ...)
```

Since each datum in the tree is just a single letter, comparing WD to
the
datum, instead of (FIRST WD), combines the same-first-letter and
length-is-one
tests.

Another common variant was to use a helper procedure for forests,
instead of
using FILTER. This allows a bit of a simplification because the base
case
test can be in the helper instead of the main procedure, and can then
deal
with empty words (because we called the forest helper with (BUTFIRST
WD) as
its argument, and if the butfirst is empty, then the original word has
one
letter, as in the version above):

```
(define (contains-word? Tree wd)
  (and (equal? (first wd) (datum Tree))
       (cw-forest? (children Tree) (butfirst wd))))

(define (cw-forest? forest wd)
  (cond ((and (empty? wd) (null? forest)) #t)
        ((or (empty? wd) (null? forest)) #f)
        (else (or (contains-word? (car forest) wd)
                  (cw-forest? (cdr forest) wd)))))
```

Another approach relies on the assumption (implicit in the problem, although
not explicitly stated) that the same letter won't appear as the datum of two
different children of the same node. So the /second/ letter of the word can
be used to determine which child to traverse, instead of traversing all of
the children:

```
(define (contains-word? Tree wd)
  (if (equal? (first wd) (datum Tree))
      (let ((child (find-next (children Tree) (butfirst wd))))
        (cond ((equal? child #t) #t)   ; last letter at leaf
              ((equal? child #f) #f)   ; mismatch
              (else (contains-word? child (butfirst wd)))))
      #f))

(define (find-next forest wd)
  (cond ((and (empty? wd) (null? forest)) #t)
        ((or (empty? wd) (null? forest)) #f)
        ((equal? (first wd) (datum (car forest)))
         (car forest))
        (else (find-next (cdr forest) wd))))
```

Another approach was to construct a sentence of all the words in the tree,
then just use MEMBER? to see if the word we want is there:

```
(define (contains-word? Tree wd)
  (member? wd (words Tree)))

(define (words Tree)
  (if (leaf? Tree)
      (sentence (datum Tree))
      (every (lambda (wd) (word (datum Tree) wd))
             (accumulate sentence
                         (map words (children Tree))))))
```

This isn't my favorite solution, because it builds an unnecessary data
structure (the sentence of words in the tree), but some students had seen
enumerating words as an example problem and thought it would be a good idea to
reuse that solution. This can be a good strategy, but it's easy to get the
WORDS procedure wrong.

Scoring: We tried to stick to even numbers of points, on this scale:

```
        10  correct
         8
         6  has the idea
         4
         2  has an idea
         0  other
```

We sometimes gave 9 for really tiny errors. Here are some specific common errors and their scores:

```
         9  (apply or ...) or (accumulate or ...)
            [You can't do this because OR is a special form, not a
procedure.]
         8  Base case error.
         6  Tries to be tree recursive, but something wrong with
structure.
         6  (map (lambda (child) (contains-word? child wd))) without
            combining the boolean values in that list.
         4  No attempt to check all the children.
         4  Tries and fails to enumerate words.
         4  Treats forest as a list of letters, not Trees.
         2  Takes CAR or CDR of a Tree, or has other glaring
domain/range
            errors.
```

6. Lists.

Most people took the hint (thank you!), although a few thought it meant that
there was an /existing/ procedure nth-cdr you could use.

Here's the solution we expected:

```
(define (list-split lst num)
  (if (null? lst)
      '()
      (cons (first-n num lst)
            (list-split (nth-cdr num lst)))))

(define (first-n num lst)
  (cond ((null? lst) '())
        ((= num 0) '())
        (else (cons (car lst) (first-n (- num 1) (cdr seq))))))

(define (nth-cdr num lst)
  (cond ((null? lst) '())
        ((= num 0) lst)
        (else (nth-cdr (- num 1) (cdr lst)))))
```

Note that the two helpers are written to be /safe/, meaning that if the list
is too short to collect or skip NUM elements, they return useful values rather
than give error messages about taking the CDR of the empty list. This is
necessary given the base case I used in the main procedure, because the list
can be non-null and still not have enough elements for a complete set

of NUM
of them. Another approach is to use a more conservative base case:

```
(define (list-split lst num)
  (cond ((null? lst) '())
        ((<= (length lst) num)
         (list lst))           ; note, a one-element list of a list!
        (else (cons (first-n num lst)
                    (list-split (nth-cdr num lst))))))

(define (first-n num lst)
  (if (= num 0)
      '()
      (cons (car lst) (first-n (- num 1) (cdr lst)))))

(define (nth-cdr num lst)
  ((repeated cdr num) lst))
```

With this base case, we can use /unsafe/ helpers.

Note the base case (LIST LST) in the second version. If there are NUM
or
fewer elements, we /don't/ want to return LST! That would return a
list
whose elements are the elements of LST, whereas we want to return a
list whose
elements are /subsets/ of LST:

```
> (list-split '(a b c d e) 4)
((a b c d) (e))
> (list-split '(a b c d) 4)
((a b c d))
```

Some students argued at the exam that we got the last example wrong,
and that
it should have been

```
> (list-split '() 5)   ; WRONG
(())
```

so that it would contain a set of elements of the empty list. But we
never
have empty subsets in the return value for other cases with an exact
multiple
of the split length:

```
> (list-split '(a b c d) 3)
((a b c) (d))
> (list-split '(a b c) 3)
((a b c))      ; NOT ((a b c) ())
```

If there aren't going to be any elements in a subset, we drop the
subset.
Same for the transition between one element and no elements:

```
> (list-split '(d) 3)
((d))
```

```
> (list-split '() 3)
()              ; NOT (())
```

A word about the CONS in

```
      (cons (first-n num lst)
            (list-split (nth-cdr num lst)))
```

This is the common, common, almost universal form of a recursive
procedure
that processes a sequential list:

```
        (cons (something (car lst))
              (recursive-call (cdr lst)))

        (cons new-element some-list)
```

In this program, instead of CAR and CDR of the list argument, we take
FIRST-N and NTH-CDR of the list. But the idea is the same: we're
adding one
new element to a (recursively generated) list. Too many people said

```
      (APPEND (LIST (first-n num lst))
              (list-split (nth-cdr num lst)))
```

This works, but it's needlessly inefficient and makes your code harder
to
read and understand.

A few people, not a lot, but still too many, saw the hint about NTH-CDR
and
decided it meant to write a procedure that generates the /name/ of a
function
that takes the Nth CDR:

```
(define (nth-cdr num)
  (word 'C ((repeated (lambda (wd) (word 'D wd)) num) 'R)))
```

Then they tried to invoke the name:

```
        ((nth-cdr num) lst)
```

Names aren't procedures! You can't invoke a word, even if /you/ think
that
that word is the name of a procedure. Scheme just thinks it's a word.

Note: During the exam, several students asked, "Can we assume that
it's a
flat list?" The answer to that question is no; the problem says "a
list,"
not "a sentence." The example used letters as elements just to make
them
easy to read; if the example had been something like

```
STk> (list-split '((a (b)) c (((d))) (e f) g) 3)
```

you wouldn't have been very happy. But the answer to a different

question,
namely "Can we assume that we don't have to split elements that are lists?"
is yes. The difference is that you have to preserve the structure of the
elements; the answer to the ugly problem above should be

```
(((a (b)) c (((d)))) ((e f) g))
```

and not

```
((a b c) (d e f) (g))
```

So any solution that starts by flattening the list is wrong.

Scoring: As in question 5, we tried for an even-number-of-points scale.
Some specific examples:

```
        8  Base case wrong.
        8  LIST instead of CONS.
        8  Reverses sublists, but preserves the order of the overall
list.
        6  Unsafe helpers with the base case that needs safe ones.
        4  Flattens the list.
        4  Generates the word CDDD...DR, otherwise good.
        2  Generates the word CDDD...DR, rest of code bad too.
```

7. Data Directed Programming

We wanted a solution that works no matter what's in the new table entry:
a procedure, another type signature, or nothing at all. (By the way, many
students didn't recognize the name "type signature," which I used in lecture,
which means a list of types, such as (INTEGER INTEGER) or (INTEGER REAL).)
This is best done with a helper procedure that takes the signature as an
argumment:

```
(define (apply-generic op . args)
  (ag1 op (map type-tag args) args))

(define (ag1 op type-tags args)
    (let ((proc (get op type-tags)))
      (COND ((PROCEDURE? PROC)
             (apply proc (map contents args)))
            ((LIST? PROC)
             (AG1 OP PROC ARGS))       ; This is the new feature
            (ELSE (error
                   "No method for these types -- APPLY-GENERIC"
                   (list op type-tags))))))
```

Note how using a helper procedure can solve the problem of the variable

number
of arguments to apply-generic; the helper is a plain old three-argument
procedure, which can easily be called recursively.

It's also possible to avoid the helper by creating a new ARGS list that
combines the new type tags with the old contents. Note that the call
to MAP
that accomplishes this is a little complicated; it's not just

```
        (attach-tag proc (contents args))  ; WRONG
```

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (COND ((PROCEDURE? PROC)
             (apply proc (map contents args)))
            ((LIST? PROC)
             (APPLY APPLY-GENERIC
                    (CONS OP
                          (MAP ATTACH-TYPE
                               PROC
                               (MAP CONTENTS ARGS)))))
            (ELSE (error
                   "No method for these types -- APPLY-GENERIC"
                   (list op type-tags)))))))
```

But, as always, solutions that generate unnecessary data structures are
unaesthetic. In this case, it's a little worse than usual, because the
new
ARGS is full of lies; it attaches type tags to data that don't match,
as in

```
        (integer . 3.14159)
```

A nicer attempt to minimize the amount of change required just uses a
new
helper to get the "real" table entry:

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (GET-PROC op type-tags)))     ; This line changed
      (if proc
          (apply proc (map contents args))
          (error
            "No method for these types -- APPLY-GENERIC"
            (list op type-tags))))))

(define (get-proc op types)
  (let ((proc (get op types)))
    (cond ((procedure? proc) proc)
          ((list? proc) (get-proc op proc))
          (else #f))))
```

Scoring: Note that this problem was worth 8 points, as marked on the
front
of the exam, not 10 points, as marked on the problem itself. So the
general
scale was

8	correct
5-7	has the idea
2-4	has an idea
0-1	other

Some specific common errors:

7	Problem using APPLY to call APPLY-GENERIC.
7	No recursion; works only if new entry has a procedure.
6	(attach-tag proc (contents args)).
4	Assumes the first value in the table will always be a list.
2	(apply-generic op proc).