CS 61A Final
Spring 2010

Question 1:

What is the order of growth in time of the followin
g function? Check all that
apply.

```
(define (foo x y)
   (cond ((= x 0) y)
         ((= x 1) (foo (- x 1) (* x y)))
         (else    (foo (- x 1) (foo 1 (+ x y))))))
```

  At first blush, this might seem exceptionally dif
ficult because of the
  recursive call as one of the arguments to FOO in
the ELSE clause. However,
  notice that the logic of FOO only depends on the
value of X; when X is 0,
  return Y, when X is 1, do something else, etc. Th
e recursive call to FOO:
  (foo 1 (+ x y)) in the ELSE clause has X set to 1
 for the next call, which
  causes it to call (foo (- 1 1) (* 1 y)) for the n
ext call, which will
  eventually return Y. That means that the recursiv
e call (foo 1 (+ x y))
  always takes constant time, so we can essentially
 ignore it. Then we're
  left with X decreasing by 1 at each recursive cal
l, so it takes O(x) time.

Scoring: 2pts, all or nothing

Question 2:

When you reserve a book at the library, you can giv
e them your e-mail address,
so they can send you an e-mail when your book comes

in. This is an example of:

a callback

The answer would have been a bit too obvious if we said the library "calls you back", so we used e-mail instead. The callback here is your e-mail address; once something happens that will require them to "execute" the callback (your book comes back in), the original thing that requested the information (you) gets notified.

A particular kindergarten class has the rule that only the person holding a certain special sock can talk. The sock is an example of:

a mutex

Everyone is waiting on the sock to be able to talk, so it acts as a mutex in this case. Note that it couldn't be a deadlock; the person speaking could simply hand off his sock if someone else ever had to talk!

Scoring: 2pts each, all or nothing

Question 3:

(Based on a true story!) A certain model of fighter plane had an abnormally high number of crashes on landing -- blamed on pilot error. It turned out the controls for the flaps and the landing gear were right next to each other, causing pilots to accidentally retract the landing gear when they meant to extend the flaps to slow the plane down. The fix? A round knob on the landing gear lever, and a rectangular one on the flaps control. The change stopped these accidents almost completely.

How is this story similar to the case of the Therac-25?

Answer: "Operator error" was a result of a poor user interface.

While it's true that it is the pilot that was at fault, the core misunderstanding that caused the problem is at the user interface. The two controls are operated at nearly the same time, and have the same look; naturally problems would occur! The Therac-25 also had user interface problems, namely cryptic error messages that the operator would simply skip once he/she got used to it, and fields that allowed you to type over previous settings without updating them.

The other answers:

The problem was easy to reproduce.
True of the airplane, not true of the Therac-25.

Incorrect concurrency was one of the main issues.

True of the Therac-25, not true of the airplane.

The makers of the machines actively tried to hide the problem.
True of neither.

Scoring: 2pts, all or nothing

Question 4:

Given the following code:

```
(define x 10)
(define y 10)

(define foo
   (let ((x 5))
      (lambda (y) (+ x y))))

(foo 7)
```

What is the result of the final expression using le
xical scope? 12

What is the result of the final expression using dy
namic scope? 17

Which does Scheme use? lexical scope

Note that FOO is defined as the return value of i
ts LET; the LET here
returns the LAMBDA whose lexical environment (rig
ht bubble) points to the
frame created by the LET. In other words, in lexi
cal scope, the X that the
LAMBDA will access is the LET's X. So when you ca
ll (foo 7), X is 5, and
Y is 7, giving you 12.

In dynamic scope, when we call FOO with 7, the X
the body of the LAMBDA gets
is the one in the current (global frame). X is 10
, Y is 7, giving us 17.

Finally, the question about Scheme should've been
a freebie if you've been
paying attention in class :)

Scoring: 5pts - 2 for each blank and 1 for "Which d
oes Scheme use?"

  (If you managed to get the entire thing reversed, we still gave you a single
  point for recognizing which answer Scheme should get.)

Question 5:

Here is the code with the blanks filled in:

```
; Assume x, y, and z have already been defined.
> (define a (append (list x y) z))
a
> (define three (list 3))
three
> a
((1 2) (1 2) 1 2)

> (set-cdr! (cdr z) THREE)
okay
> a
((1 2) (1 2) 1 2 3)

; either (set-cdr! X (CDR Z)), or
> (set-cdr! (CDR X) THREE)
okay
> a
((1 2 3) (1 2 3) 1 2 3)

> (set-car! THREE 6)
okay
> (set-car! X 4)
okay
> a
((4 2 6) (4 2 6) 1 2 6)
```

  The trick was to realize that while X, Y, and Z all print as (1 2), X and Y
  are EQ?, meaning they point to the same box in a box-and-pointer diagram.
  After that it was a relatively simple matter of keeping your cars and cdrs

straight.

Most of the blanks had more than one correct answer -- for example, you
could have said (cdar a) instead of (cdr x). Except for the "either-or"
commment above, though, there was only one correct /value/ (usually a pair)
that could go in each slot. We would have given credit even to answers like
(set-cdr! (cdadr a) (cddddr a)), which is just (set-cdr! (cdr y) three).

Scoring: 1pt per blank. Because some blanks were harder than others, this
kept people from losing all points on this problem.

Question 6:

To answer the question, you were not supposed to solve the puzzle,
but to translate the puzzle into an evaluation through ambeval.

In the puzzle, you were supposed to find the order in which Alice,
Bob, Carol and Dave arrived. This was encoded into the variables a, b,
c and d which could take values from 1 to 4.

The code has three parts to it: binding the variables to the ambiguous
values (1-4), checking all the conditions posed in the puzzle (5-10)
and then return the solution (that part was already given to you) in
proper format.

The first of the seven bullet points in the quiz is just general information, the next 6 translate to a (require ...) expressions each:

```
(let
    ((a (amb 1 2 3 4))                    ; 1
     (b (amb 1 2 3 4))                    ; 2
     (c (amb 1 2 3 4))                    ; 3
     (d (amb 1 2 3 4)))                   ; 4
    (require (distinct? (list a b c d)))  ; 5
    (require (or (> c a) (> c b)))        ; 6
    (require (not (and (< a c) (< b c)))) ; 7
    (require (> d a))                     ; 8
    (require (not (= d 4)))               ; 9
    (require (= b 1))                     ; 10
    (list (list 'Alice a)  (list 'Bob   b)
        (list 'Carol c)  (list 'Dave  d)))
```

Alternatively, you can drop line 9 by changing 4 to (d (amb 1 2 3)) and
you can drop line 10 by changing 2 to (b (amb 1)) or just (b
1). Instead of (require ...), you can write (if (not ...) (amb)).

Scoring: 5 points in total.

Lines 1-4 were worth 2 points in total. Each line 5-10 was worth half
a point each. The result was rounded down.

Remark: The puzzle actually has no a solution!. So just writing
((a (amb)) (b (amb)) (c (amb)) (d (amb))) gives the right result, but
it would not be worth credit.

Question 7:

Here is the mystery code again:

```
  (define (helper s n)
```

```
    (if (= n 0)
        the-empty-stream
        (cons-stream (stream-car s)
                     (helper (stream-cdr s) (- n 1)
))))

  (define (mystery s)
    (define (foo n)
      ; like STREAM-APPEND, but takes a stream prom
ise instead
      (stream-append-delayed (helper s n)
                             (delay (foo (+ n 1))))
)
    (foo 1))
```

Taking the hint, let's see what HELPER does. If N i
s 0, it returns the empty
stream. Otherwise, it takes the first element of th
e stream, CONS-STREAMed
with HELPER of the rest of the stream. The recursiv
e call to HELPER decreases
N by 1, so it will take N calls to reach the base c
ase. So HELPER just takes
the first N elements of the stream S. Notice that T
HE-EMPTY-STREAM is not an
/element/ of the stream; it's the value that marks
the end of a finite
stream...just like the empty list marks the end of
a list.

Now that we know what HELPER does, we can look at M
YSTERY...which just calls
FOO. FOO is going to append two streams: the first
is (helper s n), and the
second is (foo (+ n 1)). (Ignore the DELAY for righ
t now.) So we're appending
the first N elements of the input stream with (foo
(+ n 1)). What does that
stream start with? The first n+1 elements of the in
put stream. And so on.

So, MYSTERY takes a stream, and returns a stream of

```
 the first element, the
first two elements, the first three elements...appe
nded together. The first
ten elements are as follows:

1 1 4 1 4 9 1 4 9 16
^ ^^^ ^^^^^ ^^^^^^^^
|  |    |       |
|  |    |       +-- (helper s 4)
|  |    |
|  |    +-- (helper s 3)
|  |
|  +-- (helper s 2)
|
+-- (helper s 1)

This was enough to get the answer, but one question
 remains: why the DELAY?
Because Scheme is applicative order, and STREAM-APP
END isn't a special form!
If we used regular STREAM-APPEND and a non-delayed
recursive call, we would
end up with infinite recursion trying to find (foo
(+ n 1)).

Scoring: 5 points total
* 2 for having subsequences that increased in size
* 1 for having the correct numbers in those subsequ
ences
* 2 for making a flat stream instead of a stream of
 streams


Question 8:

First off, the solution we were looking for:

(define (fib-vector n)
  (let ((result (make-vector n 1)))  ; initialize a
ll values to 1
    (define (iter x)
      (if (>= x n)
```

```
          result
          (begin (vector-set! result x (+ (vector-r
ef result (- x 1))
                                         (vector-r
ef result (- x 2)) ))
                 (iter (+ x 1)) )))
    (iter 2) ))
```

Notice how we use lexical scope to minimize the num
ber of parameters that get
passed around! The helper only needs to take one ar
gument, the index of the
current Fibonacci number to compute, and has access
 to the result vector and
the limit anyway.

The interesting part of the problem is handling the
 base cases, which is what
prompted us to announce that it was okay to assume
n >= 2. (The version above
happens to work for any non-negative n.) There were
 three ways to do this:

 - Like the solution above, use MAKE-VECTOR's impli
cit initialization.
 - VECTOR-SET! the first two elements to 1, then ca
ll the helper.
 - Have special cases in the helper for the first t
wo elements.

Some people misinterpreted the question as suggesti
ng the use of a memoized
version of FIB or FIB-VECTOR -- i.e. numbers should
n't be calculated more than
once, ever! These were mostly graded on a case-by-c
ase basis, but still had
the same efficiency and usage criteria (e.g. using
1-based indexing still cost
a point, etc.).

Scoring: 8 points total
* 1 for creating a vector of the proper size.

* 2 for getting the first two elements right.
* 1 for properly calling your helper procedure, based on how you initialized
     the first two elements.
* 2 for properly calculating the rest of the elements.
* 1 for knowing when to end the recursion (no off-by-one errors).
* 1 for remembering to return the vector at the end.

* -1 for using 1-based indexing; the first element of a vector is element 0.
* -2 for non-efficient uses of vectors, such as using the hypothetical
     procedure VECTOR-APPEND.
* -4 for calculating a specific Fibonacci number more than once, since the
     instructions specifically said not to. This included implementations that
     just called FIB.

Question 9:

Write rules for RELATED-BY which take two people and a list of relationships.
The rules should match if and only if the two people are related through the
sequence of relationships in the list.

Hint: What two people satisfy RELATED-BY if the list is empty?

The hint was intended to suggest the base case: if it doesn't take any
relationships to get from person A to person B, then A and B must be the
same person!

```
(assert! (rule (related-by ?self ?self ())))
```

How about the rest of it? Like most query system rules dealing with lists, we want to split the list of relationships up into the "first" relationship and the "rest" of the list. So the rule header we want is:

```
(assert! (rule (related-by ?a ?b (?first-rel . ?rest)) ...))
```

What are the conditions on this rule? That is, what must be true for the rule to match properly? We know that ?a has to be related to someone through the first relation, AND that that person is then related to ?b through the rest of the relations.

```
(assert! (rule (related-by ?a ?b (?first-rel . ?rest))
               (and (?first-rel ?a ?someone)
                    (related-by ?someone ?b ?rest)
)))
```

Actually, that's it! The "trick" was remembering that the query system isn't doing anything smart -- it's just pattern-matching. That means you can have two things related by ?first-rel, and ?first-rel will match PARENT or GRANDPARENT, or any other pattern in the query system.

The other important step was realizing that you needed to introduce a "middle" person, someone who was related to both ?a and ?b. This is also something we've seen before.

Some people took the additional step of adding (not (same ?a ?b)) as a condition. This isn't a bad idea, particularly

since SAME is a
    "relationship" of sorts, and you could get your self into an infinite loop
    that way: (related-by bart bart (same same same same same ...))! But we
    said not to worry about infinite loops.

    We didn't take any points off for reversing the order of the relations,
    since we didn't specify it!

Scoring: 8pts total
* 2 for the base rule
    (1 for an attempt)
* 6 for the recursive rule
    6 - correct
    5 - trivial errors
    4 - the idea: a recursive rule that goes one relationship at a time and
        introduces a "middle" person that's related to both ?a and ?b
    2 - an idea
    0 - other (hardcoding "parent" or "grandparent" in your rule(s), or trying
        to use Scheme-style composition of functions)

Question 10:

Instead of the chain of definitions yielding to the result of (best-song-ever 'artist) we could have written:

```
(
   (
      (
         generate-adt
         '(title artist)     ; let this be "names"
      )
      '(never gonna give you up) 'astley ; let this be "args"
```

```
    )
    'artist ; let this be "message"
)
```

We see that generate-adt's domain is lists (names, here (title
artist)), its range is actually procedures (one of them is make-song),
whose domain is a number of arguments (args, here (never ...) and
astley) and whose range is again procedures (one of them is best-song)
whose domain is a word (message, here artist). It can be thought of
as nested message-passing. It is three steps deep, so the solution
should involve three lambdas. Furthermore, the body of the function
has to figure out where the "message" appears in "names" with
(position message names) and then return the corresponding element
from args with (list-ref args (position message names)).

A more formal way to write the domain and range of this curried
function is:

```
generate-adt:
   lists ----> (objects^* ----> (strings ---> objects))
   names |---> ( args |-------> (message |--> (list-ref args

     (position

        message

        names))))
```

And here is the scheme code:

```
(define generate-adt
    (lambda (names)
        (lambda args
            (lambda (message)
                (list-ref args (position message na
mes))))))
```

or using the header given in the exam:

```
(define (generate-adt names)
    (lambda args
        (lambda (message)
            (list-ref args (position message names)
))))
```

Notice that in the above lambdas, there are no pare
nthesis around
args. This is intentional. Recall that scheme allow
s the following
patterns for lambda:

```
   lambda expression applied           what will va
riables bind to
        to (1 2 3)

   ((lambda args args) 1 2 3)          args = (1 2
3)
   ((lambda (head . tail)) 1 2 3)      head = 1, ta
il = (2 3)
   ((lambda (a b c)) 1 2 3)            a = 1, b = 2
, c = 3
```

Hence the following is also a valid solution (excep
t for the degenerate case of
(generate-adt '())):

```
(define (generate-adt names)
    (lambda (arg1 . args)
        (lambda (message)
            (list-ref (cons arg1 . args) (position
message names)))))
```

And another, more lengthy solutions:

```
(define (generate-adt names)
    (define (class-like . args)
        (define (object-like message)
            (list-ref args (position message names)
))
        object-like)
    class-like)
```

Solutions that didn't use LIST-REF and POSITION but
 still worked also
received credit.

Grading: 8 points total.
* 4 for recognizing and formalising the domain and
range of
    generate-adt correctly, i.e. writing (lambda ar
gs (lambda (message) ...))
    2 points for writing two lambdas matching the a
rguments in the
      right order
    2 points for writing args without parenthesis o
r writing (arg1 . args)
      (and failing to use it correctly)
* 4 points for the body (list-ref args (position me
ssage names))
    2 points for composing list-ref and position
    2 points for using the right arguments to list-
ref and position
    0 points subtracted for swapping the list and t
he index argument
      to list-ref

Question 11:

Write a function PATH-ACCUMULATE that, given a Tree
, a function of two
arguments, and a base case for that function, retur
ns a new Tree in which the
datum of every node is the accumulation of the path

 in the original Tree from
the root to that node.

```
(define (path-accumulate fn base tree)
  (let ((new-datum (fn (datum tree) base)))
    (make-tree new-datum
               (map (lambda (child) (path-accumulat
e fn new-datum child))
                    (children tree) ))))
```

    This was a pretty straightforward tree recursio
n question. As with most
    such problems, it could be done either using MA
P, or a separate
    FOREST-ACCUMULATE helper function. The basic id
ea of such a question is
    "we're making a new Tree, so what's the new dat
um, and what should I do
    to the children?"

    In this case, the new datum is what you get by
combining the old datum
    with the running accumulation -- which is BASE.
 The children aren't
    filtered in any way, just PATH-ACCUMULATED vers
ions of the original
    children. However, you have to remember to use
the new datum as the new
    base as well -- otherwise there's not actually
any accumulation!

Scoring: 8 points total
* 8 - perfect
* 7 - trivial errors, such as using + and 0 instead
 of FN and BASE
* 6 - correct, but forgot to use the new base in th
e recursive call
* 5 - a good attempt at Tree recursion, plus applyi
ng FN to the datum and BASE
* 2 - applying FN to the datum and BASE, but no cle
ar tree recursion
* 0 - other

Question 12:

There were four key parts to this question. The fir
st was realizing that the
input to LOOKUPSTATIC was going to be a procedure /
name/, not an actual
procedure. The second was knowing that the procedur
e statics were stored in
a frame inside the procedure (once you had looked i
t up), as specified in
project 4. The third was knowing how to get the val
ue out of the statics
frame. Last was knowing how to add LOOKUPSTATIC as
a Logo primitive.

One correct implementation:

```
(define (lookupstatic proc-name var-name)
  (let ((proc (lookup-procedure proc)))
    (lookup-variable-value var-name (list (statics-
frame proc))) ))
```

The calls to LOOKUP-PROCEDURE and STATICS-FRAME wer
e pretty much the same for
everybody. The trouble came with getting the variab
le value /out/ of the
statics frame. Many people forgot that LOOKUP-VARIA
BLE-VALUE takes an
/environment/, not a frame.

There were three ways around this that came up in a
nswers:
1. Like the sample solution above, make an environm
ent containing a single
   frame. Probably an even better solution would ha
ve been to use
   (adjoin-frame (statics-frame proc) the-empty-env
ironment),
   and a few people did do this. But we have said a
n environment is just a

```
   list of frames, not a proper ADT.
2. Copy the SCAN procedure from LOOKUP-VARIABLE-VAL
UE, and tweak it for use
   here.
3. Use the FRAME-VARIABLES and FRAME-VALUES selecto
rs to extract the variable
   names and values, then use POSITION annd LIST-RE
F as in Question 10.
```

Last, you had to add your procedure to Logo:

```
(add-prim 'lookupstatic 2 lookupstatic)
```

Grading: 8 points total

```
* 1 for the call to ADD-PRIM, all-or-nothing
  - No points off for using a count of (2), as long
 as your procedure actually
    takes an environment as a parameter.
* 7 for the implementation of LOOKUPSTATIC
    7 - perfect
    5 - treated (statics-frame proc) as an environm
ent instead of a frame
    4 - attempted to look up the procedure, extract
 the statics frame from the
        procedure, and look up the variable in the
statics frame.
    2 - only did one or two of the above.
    0 - anything else
```