CS 61A Final Exam — May 18, 2010

Your name _____

login: cs61a–_____

This exam is worth 70 points, or about 23% of your total course grade. The exam contains 12 questions.

This booklet contains 14 numbered pages 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, don't put in error checks. Assume that you will be given arguments of the correct type.

If you want to use procedures defined in the book or reader as part of your solution to a programming problem, you must cite the page number on which it is defined so we know what you think it does.

***** IMPORTANT *****

Check here if you are one of the people with whom we arranged to replace a missed/missing exam with other exam scores: ____

***** IMPORTANT *****

If you have made grading complaints **that have not yet been resolved**, put the assignment name(s) here:

READ AND SIGN THIS:

I certify that my answers to this exam are all my own work, and that I have not discussed the exam questions or answers with anyone prior to taking this exam.

If I am taking this exam early, I certify that I shall not discuss the exam questions or answers with anyone until after the scheduled exam time.

_____

1–2	/6
3	/2
4	/5
5	/7
6	/5
7	/5
8	/8
9	/8
10	/8
11	/8
12	/8
total	/70

Question 1 (2 points):

What is the order of growth in time of the following function?

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

_____ $\Theta(x)$ _____ $\Theta(y)$ _____ $\Theta(x + y)$ _____ $\Theta(x \cdot y)$

Question 2 (4 points):

Pick the *best* choice:

When you reserve a book at the library, you can give them your e-mail address, so that they can send you an e-mail when your book comes in. Your address is an example of:

_____ a mutex _____ a callback _____ a packet _____ a deadlock

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 _____ a callback _____ a packet _____ a deadlock

Your name _____ login cs61a–_____

Question 3 (2 points):

(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? (Choose the best answer.)

_____ The problem was easy to reproduce.

_____ Incorrect concurrency was one of the main issues.

_____ The makers of the machines actively tried to hide the problems.

_____ "Operator error" was a result of a poor user interface.

Question 4 (5 points):

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 lexical scope? _____

What is the result of the final expression using dynamic scope? _____

Which does Scheme use? _____ Lexical scope _____ Dynamic scope

Hint: You may find it helpful to draw an environment diagram.

Your name _____ login cs61a–_____

Question 5 (7 points):

Fill in the blanks below to make STk behave as shown. You may **not** create any new pairs. Solutions that do so will receive no points. In other words, you may not fill in the blanks with `append`, `cons`, `list`, or anything else that creates pairs.

```
; 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) _____________)
okay

> a
((1 2) (1 2) 1 2 3)

> (set-cdr! _____________  _____________)
okay

> a
((1 2 3) (1 2 3) 1 2 3)

> (set-car! _____________  _____________)
okay

> (set-car! _____________  _____________)
okay

> a
((4 2 6) (4 2 6) 1 2 6)
```

Question 6 (5 points):

Write a program using the nondeterministic evaluator (ambeval) that solves the following logic puzzle. The puzzle itself may or may not have a solution. Your program should **not** include your own insights into the logic puzzle itself but should include all the facts below:

- Alice, Bob, Carol and Dave went to a restaurant and arrived in some order.
- Everybody arrived at a different time.
- Carol arrived after Alice or after Bob.
- Carol is sure that Alice and Bob were not both there before her.
- Dave arrived after Alice.
- Dave did not arrive last.
- Bob arrived first.

What order did they arrive in? The final output has been done for you: it should be of the form `((alice 4) (bob 3) (carol 2) (dave 1))`. (This is not a valid solution—it's just to show you what a solution should look like!)

You may remember `distinct?`, a procedure that, given a list, returns `#t` if and only if all the elements of the list are different.

```
(let
```

```
  (list (list 'Alice  a)  (list 'Bob   b)
        (list 'Carol  c)  (list 'Dave  d) ))
```

Your name _____ login cs61a–_____

Question 7 (5 points):

The following code operates on streams:

```
(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 promise instead
    (stream-append-delayed (helper s n)
                           (delay (foo (+ n 1)))))
  (foo 1))
```

What stream would `mystery` return when given the stream

`(1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 ...)`?

Write the first ten elements below:

_____ _____ _____ _____ _____ _____ _____ _____ _____ _____

Hint: Figure out what `helper` does first.

Question 8 (8 points):

The Fibonacci numbers are defined by

$$a_0 = a_1 = 1, \quad a_n = a_{n-1} + a_{n-2}$$

Define a procedure `(fib-vector n)` which returns a vector of the first n Fibonacci numbers. Make it efficient – you should only calculate each number once. Do not use `vector->list` or `list->vector`.

```
> (fib-vector 5)
#(1 1 2 3 5)
```

Hint: Go from smallest to largest!

```
(define (fib-vector n)
```

Your name _____ login cs61a–_____

Question 9 (8 points):

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. (Do not worry about infinite loops caused by circular relations, e.g., cousins.) For example:

```
> (assert! (rule (grandparent ?x ?y)
                 (and (parent ?x ?z)
                      (parent ?z ?y)))
> (assert! (parent lisa homer))
> (assert! (parent homer abraham))
> (assert! (parent abraham orville))

> (related-by lisa orville ?how)
(related-by lisa orville (grandparent parent))
(related-by lisa orville (parent grandparent))
(related-by lisa orville (parent parent parent))
```

The rules should work for any relationships (e.g. `uncle`), not just the ones shown here, as long as there is enough information in the database to determine the result (e.g., rules for `uncle`). Read the paragraph again—we're not asking you to write rules for `uncle`!

Hint: What two people satisfy `related-by` if the list of relationships is empty?

Question 10 (8 points):

As we know, abstract data types are defined by their constructors and selectors. Instead of having to define them ahead of time, though, we'd like to be able to generate new data types "on the fly", that is, as part of our programs.

`generate-adt` is a procedure that takes a *list* of selector names for a new ADT and returns a constructor for that ADT:

```
> (define make-song (generate-adt '(title artist)))
make-song
```

You then call this constructor with values corresponding to those arguments.

```
> (define best-song-ever (make-song '(never gonna give you up) 'astley))
best-song-ever
```

The data is then represented by a procedure, which takes a selector name as an argument.

```
> (best-song-ever 'title)
(never gonna give you up)
> (best-song-ever 'artist)
astley
```

Implement `generate-adt`. Remember the syntax for a procedure that takes any number of arguments: `(lambda args` *body*`)` – `args` will be a list of the actual arguments. You will need this for the constructor, since different constructors will have different numbers of arguments.

The created data should be immutable – you should not be able to change the `title` or `artist` of `best-song-ever` once it's been created. You do not need to do error-checking.

Hint: The primitive procedures `position` and `list-ref` may be useful for this problem. You are not required to use them, however.

```
> (position 'c '(a b c d))
2
> (list-ref '(w x y z) 3)
z
> (list-ref '(w x y z) 0)
w
```

Write your answer on the next page.

Your name _____ login cs61a–_____

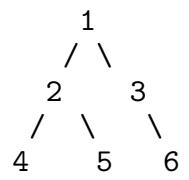
Write your answer to question 10 here:

```
(define (generate-adt names)
```

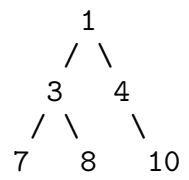
Question 11 (8 points):

Write a function `path-accumulate` that, given a Tree, a function of **two** arguments, and a base case for that function, returns 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. Here is an example:

`tree`



`(path-accumulate + 0 tree)`



Notice that the bottom-left leaf is 7, because the path to the node in the original Tree is $1 \rightarrow 2 \rightarrow 4$, so when we accumulate with $+$, we get the sum, $[0+]1+2+4 = 7$.

```
(define (path-accumulate fn base tree)
```

Your name _____ login cs61a–_____

Question 12 (8 points):

In project 4, we added static variables to our Logo evaluator—variables for a specific procedure that keep their values across calls. For debugging purposes, we may want to get the values of these variables without calling the procedure. To do this, we're going to add a new primitive `lookupstatic` to the Logo evaluator, which behaves as follows:

```
? to count :more static :saved 0 :calledbefore "False
> make "saved :saved + :more
> make "calledbefore "True
> end
? print lookupstatic "count "calledbefore
False
? count 5
? print lookupstatic "count "calledbefore
True
```

Implement `lookupstatic` (as a primitive) and **add it to the Logo evaluator**. Some relevant procedures from the evaluator have been included on the next page.

```
(define (add-prim name count proc)
  (set! the-primitive-procedures
        (cons (list name 'primitive count proc)
              the-primitive-procedures)))

(define (lookup-procedure name)
  (assoc name the-procedures))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((equal? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable " var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))

(define (arg-count proc) ...)
(define (parameters proc) ...)
(define (procedure-body proc) ...)
(define (statics-frame proc) ...)

(define (frame-variables frame) ...)
(define (frame-values frame) ...)
```