CS 61A Midterm #2 — March 10, 2010

Your name _____

login: cs61a–_____

Discussion section number _____

TA's name _____

This exam is worth 40 points, or about 13% of your total course grade. The exam contains six substantive questions, plus the following:

Question 0 (1 point): Fill out this front page correctly and put your name and login correctly at the top of each of the following pages.

This booklet contains ten 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.

Our expectation is that many of you will not complete one or two of these questions. If you find one question especially difficult, leave it for later; start with the ones you find easier.

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.

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.

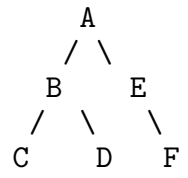
If I am taking this exam early, I certify that I shall not discuss the exam questions or answers with anyone until after the scheduled exam time.

_____

0	/1
1	/8
2	/5
3	/5
4	/4
5	/7
6	/10
total	/40

Question 1 (8 points):

Let `tree` be the following Tree:



What will Scheme print in response to the following expressions? If the expression produces an error message, you may just write "error"; you don't have to provide the exact text of the message. If the expression does not produce an error message but violates the Tree data abstraction, write "DAV". If the result involves a Tree, draw the Tree and draw a box around it. `eval-1` is defined in `scheme1.scm` – parts of it can be found in the back of the exam.

```
> (car tree)
```

```
> (cadr (children tree))
```

```
> (datum tree)
```

```
> (make-tree 'a '())
```

```
> (children (datum tree))
```

```
> (make-tree (datum tree)
             (children tree))
```

```
> (eval-1 '(sentence a b))
```

```
> (eval-1 '(sentence 'a 'b))
```

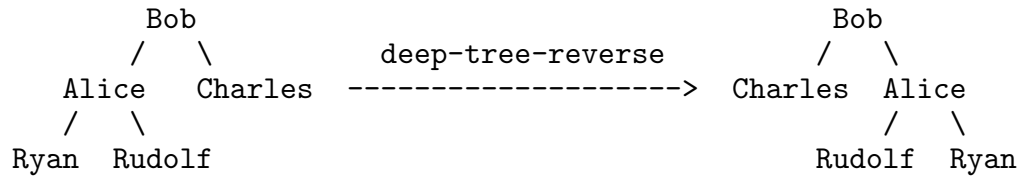
Your name _____ login cs61a–_____

Question 2 (5 points):

Write a function `deep-tree-reverse` which deep reverses a Tree, i.e. it reverses the order of the children of each node. Use `make-tree`, `datum` and `children` for the Tree data structure. You can use the `reverse` procedure which reverses a list:

```
> (reverse '(1 2 3 (4 5)))
((4 5) 3 2 1)
```

Here is an example of `deep-tree-reverse`:




```
(define (deep-tree-reverse tree)
```

Question 3 (5 points):

Let L be a deep proper list **built only of pairs and empty lists**, e.g. `()`, `(())`, `(()())`, `((()))`. Write a function `deep-null-count` which takes a list and counts the number of empty lists in the list. For example:

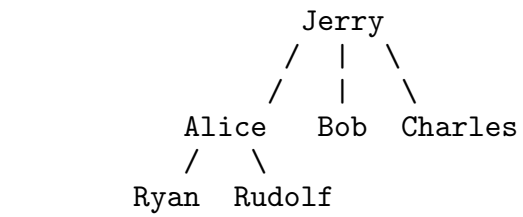
```
> (deep-null-count '(()))
2
> (deep-null-count (cons '() '()))
2
```

```
(define (deep-null-count L)
```

Your name _____ login cs61a–_____

Question 4 (4 points):

What will `mystery` applied to the following Tree (which uses the usual Tree data abstraction using the `datum` and `children` selectors) give? If the result is a Tree, you can just draw the Tree.




```
(define (mystery tree)
    (make-tree
        (foo tree)
        (map mystery (children tree))))

(define (foo tree)
    (+ 1 (bar (children tree))))

(define (bar forest)
    (accumulate + 0 (map foo forest)))
```

Question 5 (7 points):

Let a set of numbers be represented as an ordered list, e.g. `(4 5 10)`, where each element appears only once in the list. Write a function `(set-difference left right)` which returns the set of all elements in `left` which are not in `right`. Here is an example:

```
> (set-difference '(4 5 10 16) '(1 2 5 10))
(4 16)
```

Write `set-difference` as efficiently as possible using the orderedness of the list, i.e. do **not** use `member` or any helper functions.

```
(define (set-difference left right)
```

Your name _____ login cs61a–_____

Question 6 (10 points):

We would like to build a system for registering participants in a course, much like you did at the beginning of the semester.

We will use our OO syntax. First we define a class `participant`. A login is of the form "cs61a-xx", so a `participant` object's `course` can be figured out from the `login`.

```
(define-class (participant name login)
    (method (course) (bl (bl (bl login)))))
```

(a) Not all participants are the same! Define a class `student` and a class `ta`, both of which are `participant`s. Each `ta` has a list of `student`s, and we will call this list `section` (which is initially empty). You should also provide an `add-student` method which takes one argument `stud` of type `student` and adds the student to a `ta`'s `section`.

Question 6 continues on the next page.

(b) Write a sequence of Scheme expressions: 1. to create a `student` object for yourself; 2. to create a `ta` object for your `ta`; 3. to add your `student` object to your `ta`'s `section`.

(c) Write a procedure `make-roster` that, given a `ta`, will return a list of the `name`s of the students in his or her `section`.

```
(define (make-roster ta-obj)
```

Your name _____ login cs61a–_____

This page intentionally almost blank.

Parts of `scheme1.scm`

The parts of `scheme1.scm` shown here are the core of the file and are enough to answer Question 1.

```
;; Comments on EVAL-1:

;; There are four basic expression types in Scheme:
;;    1. self-evaluating (a/k/a constant) expressions: numbers, #t,
;;    etc.
;;    2. symbols (variables)
;;    3. special forms (in this evaluator, just QUOTE, IF, and LAMBDA)
;;    4. procedure calls (can call a primitive
;;                        or a LAMBDA-generated procedure)

(define (eval-1 exp)
  (cond ((constant? exp) exp)
        ((symbol? exp) (eval exp))        ; use underlying Scheme's EVAL
        ((quote-exp? exp) (cadr exp))
        ((if-exp? exp)
         (if (eval-1 (cadr exp))
             (eval-1 (caddr exp))
             (eval-1 (cadddr exp))))
        ((lambda-exp? exp) exp)
        ((pair? exp) (apply-1 (eval-1 (car exp))      ; eval the operator
                              (map eval-1 (cdr exp))))
        (else (error "bad expr: " exp))))

;; Comments on APPLY-1:
;; There are two kinds of procedures: primitive and LAMBDA-created.
;; We recognize a primitive procedure using the PROCEDURE? predicate
;; in the underlying STk interpreter.

;; If the procedure isn't primitive, then it must be LAMBDA-created.
(define (apply-1 proc args)
  (cond ((procedure? proc)        ; use underlying Scheme's APPLY
         (apply proc args))
        ((lambda-exp? proc)
         (eval-1 (substitute (caddr proc)   ; the body
                             (cadr proc)    ; the formal parameters
                             args           ; the actual arguments
                             '()))            ; bound-vars, see below
        (else (error "bad proc: " proc))))
```