CS 61A Midterm #2 — March 12, 2003

Your name DAVID TA-HSIANG LIU

login: cs61a-ce

Discussion section number 15/115

TA's name Ben

This exam is worth 40 points, or about 13% of your total course grade. The exam contains 5 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 six 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.

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.

[illegible]

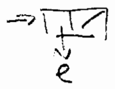
0	/1
1	/8
2	/8
3	[illegible] /8
4	/8
5	7 /7
total	/40

Question 1 (8 points):

What will Scheme print in response to the following expressions? If an expression produces an error message, you may just write "error"; you don't have to provide the exact text of the message. **Also, draw a box and pointer diagram for the value produced by each expression.**

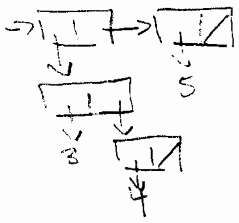
```
(cadadr '((a (b) c) (d (e) f) (g (h) i)))
```

(e)



```
(cons (append '(3) '(4)) '(5))
```

((3 4) 5)



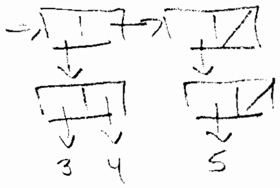
```
(cons (append 3 4) 5)
```

error

```
(list (cons 3 4) '(5))
```

~~(3.4)~~

((3.4) (5))



Your name DAVID LIN login cs61a–ce

Question 2 (8 points):

Write a procedure `lastfirst` whose argument is a *list of sentences* (*not* a deep-list that could contain lists of lists of lists). It should return a list of sentences in which every two-word sentence in the argument is included with the two words interchanged, and sentences with other than two words are omitted:

```
> (lastfirst '((john lennon) (james paul mccartney)
               (george harrison) (ringo starr)))
((lennon john) (harrison george) (starr ringo))
```

Respect the data abstraction; use the appropriate selectors and constructors for sentences and for non-sentence lists.

(a) Write a recursive version of `lastfirst`. Write only one procedure, without helper functions, and without using higher-order procedures.

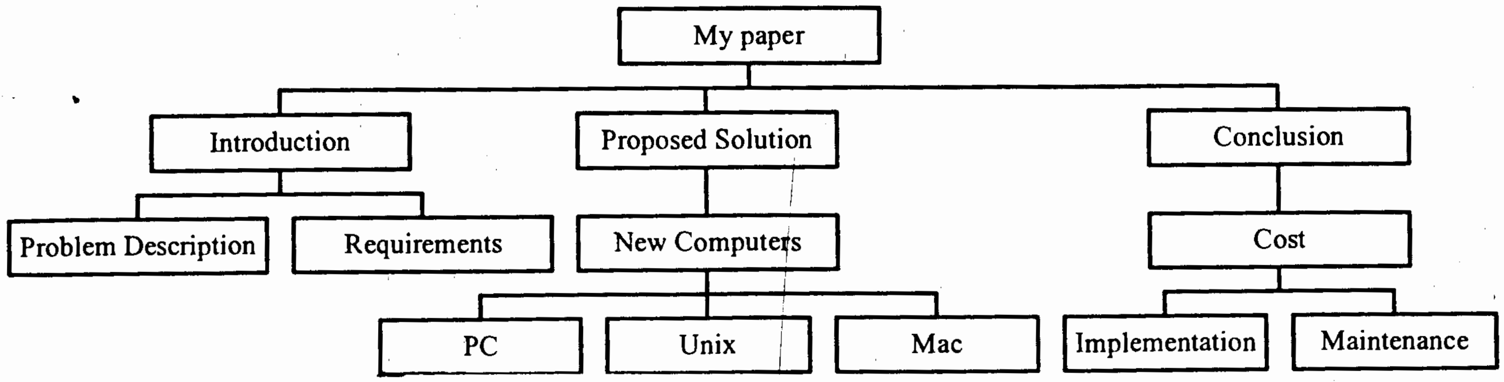
```
(define (lastfirst lst)
   (let ((first-se (car lst)))
      (cond ((null? lst) '())
            ((= (count first-se) 2) (cons (se (last first-se)
                                               (first first-se))
                                          (lastfirst (cdr lst))))
            (else (lastfirst (cdr lst))))))
```

(b) Write a version of `lastfirst` that uses higher-order procedures and does not use recursion.

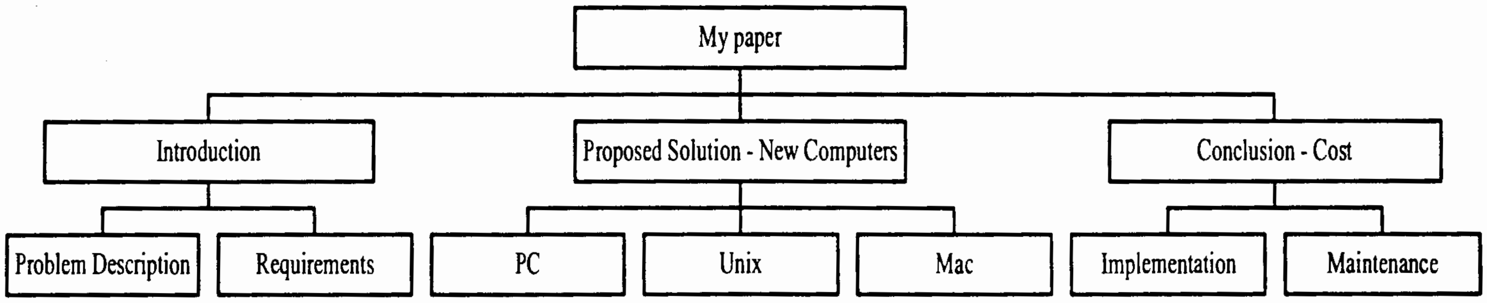
```
(define (lastfirst lst)
   (accumulate append '() (map (lambda (sent)
                                  (if (= (count sent) 2)
                                      (list (se (last sent)
                                                (first sent)))
                                      '()))  lst)))
```

Question 3 (8 points):

We are going to use the Tree abstract data type to represent the subdivisions of a paper.



However, when a heading has only one subheading (for example, "Proposed Solution" has only "New computers" as a subheading), it doesn't actually make sense to subdivide the topic. You will write `divisions-reduced`, which takes in a Tree (using the `datum/children` abstract data type discussed in lecture) in which the data are sentences. It outputs a similar tree except that each node that has only a single child is replaced by a node with the datum of the parent, followed by a hyphen, followed by the datum of the child. The node's children are the child's children, *with their divisions reduced.*



Write `divisions-reduced`.

```
(define (divisions-reduced tree)
  (cond ((leaf? tree) tree)
        ((= (length (children tree)) 1) (make-node (se (datum tree) '(-)
                                                        (datum (car (children tree))))
                                                  (divisions-reduced-for (children (car
                                                                          (children tree))))))
        (else (make-node (datum tree) (divisions-reduced-for (children tree))))))
(define (divisions-reduced-for forest)
  (if (null? forest)
      '()
      (cons (divisions-reduced (car forest)) (divisions-reduced-for (cdr forest)))))
```

[illegible]

Your name ______DAVID LIN______ login cs61a-__ce__

Question 4 (8 points):

Write a predicate procedure `deep-car?` that takes a symbol and a deep-list (possibly including sublists to any depth) as its arguments. It should return true if and only if the symbol is the car of the list or of some list that's an element, or an element of an element, etc.

```
> (deep-car? 'a '(a b c))
#t

> (deep-car? 'a '(b a c))
#f

> (deep-car? 'a '((x y) (z (a b) c) d))
#t

> (deep-car? 'a '(((a))))
#t
```

Fill in the blanks below to complete the definition.

```
(define (deep-car? symbol lst)
  (if (pair? lst)
      (or (eq? symbol ___(car lst)___)
          (helper symbol ___lst___))
      ___#f___))

(define (helper symbol lsts)
  (cond ((null? lsts) ___#f___)
        ((deep-car? symbol (car lsts))
         ___#t___)
        (else ___(helper symbol (cdr lsts))___)))
```

Question 5 (7 points):

In the early 1960s, the IBM 1620 computer did arithmetic one digit at a time, just the way you learned in elementary school, looking up the digit-by-digit sums or products in a table. We're going to simulate that.

You are given a table in which the rows and columns are digits (0 to 9) and what's stored in each table entry is a two-element ~~list~~ sentence representing the two-digit sum. For example:

```
(put 2 3 '(0 5))
(put 8 9 '(1 7))
```

We want a procedure add that takes two nonnegative integers, represented as ~~lists~~ sentences of digits, and adds them, returning another ~~list~~ sentence of digits:

```
> (add '(3 7) '(4 5))
(8 2)
```

because 37 + 45 = 82. Note: Don't worry about extra leading zeros in the result. For example, this is okay:

```
> (add '(3 7) '(4 5))
(0 0 8 2)
```

Fill in the blanks below:

```
(define (add n1 n2)
  (cond ((empty? n1) n2)
        ((empty? n2) n1)

        (else (let ((result (get (last n1) (last n2)) ))

                (sentence (add (add (bl n1) (bl n2)) (se (first result)))

                          (bf result) )))))
```

7