CS 61A Midterm #2 - October 5, 1998

Your name

login: cs61a-

Discussion section number

TA's name

This exam is worth 20 points, or about 13% of your total course grade. The exam contains four 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 eight 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.

Question 1 (4 points):

What will Scheme print in response to the following expressions? If an expression produces an error message, you may just say ``error''; you don't have to provide the exact text of the message. If the value of an expression is a procedure, just say ``procedure''; you don't have to show the form in which Scheme prints procedures. **Also, draw a box and pointer diagram of the value produced by each expression.**

```
(append (list 'a 'b) '(c d))

(cons (list 'a 'b) (cons 'c 'd))

(list (list 'a 'b) (append '(c) '(d)))

(cdar '((1 2) (3 4)))
```

Your name login cs61a-

Question 2 (5 points):

An *association list* is a list of pairs, each of which associates a name (the *key*) with a value. For example, the association list

```
((a . 3) (b . 7) (c . foo))
```

associates the key `A` with the value `3`, the key `B` with the value `7`, and the key `C` with the value `FOO`.

Here are a constructor and two selectors for an *association* abstract data type:

```
(define make-association cons)
(define association-key car)
(define association-value cdr)
```

Note that this is an ADT for an association, not for an association list, which is just a sequence of associations.

(a) Scheme provides the procedure `assoc` for looking up a key in an association list. It returns the entire association, not just the value. Here is an implementation of `assoc`:

```
(define (assoc key a-list)
  (cond ((null? a-list) #f)
        ((equal? key (caar a-list)) (car a-list))
        (else (assoc key (cdr a-list)))))
```

(Both `caar` and `car` in the next-to-last line above are correct!)

Rewrite `assoc` to use the association ADT defined above. Don't make any unnecessary changes; your program should look much like the one above, but with some procedure calls renamed.

This question continues on the next page.

Question 2 continued

(b) We are given an association list in which the keys are names of musical groups and the values are *sentences* of names of the group members, like this:

```
((Beatles . (John Paul George Ringo))
 ((Buffalo Springfield) . (Steve Neil Ritchie Dewey Bruce))
 (Who . (Pete Alec Roger Keith))
```

What follows is a procedure to make a ``backwards'' association list in which the keys are the musicians and the values are their groups, like this example:

```
((John . Beatles) (Paul . Beatles) (George . Beatles) (Ringo . Beatles)
 (Steve . (Buffalo Springfield)) (Neil . (Buffalo Springfield))
 (Richie . (Buffalo Springfield)) (Dewey . (Buffalo Springfield))
 (Bruce . (Buffalo Springfield)) (Pete . Who) (Alec . Who)
```

```
 (Roger . Who) (Keith . Who))
```

(Note to '60s rock fans: I've listed John Alec Entwistle by his middle name so that there won't be two people named John in this example. You can ignore the possibility of people with the same name; assume that won't happen.)

Here is the program:

```
(define (index groups)
  (if (null? groups)
      '()
      (append (index-one (car groups)) (index (cdr groups)))))
```

(define (index-one group) (define (help groupname people) (if (null? people) '() (cons (cons (car people) groupname) (help groupname (cdr people))))) (help (car group) (cdr group)))

On the next page, rewrite `index` and `index-one` to respect both the association ADT and the sentence ADT. Don't make any unnecessary changes.

Put your answer on the next page!

Your name login cs61a-

Question 2 continued

Here again is the program you're modifying:

```
(define (index groups)
  (if (null? groups)
      '()
      (append (index-one (car groups)) (index (cdr groups)))))
```

(define (index-one group) (define (help groupname people) (if (null? people) '() (cons (cons (car people) groupname) (help groupname (cdr people))))) (help (car group) (cdr group)))

Question 3 (5 points):

You are implementing a calculator for physicists, in which arithmetic is performed on numbers with units attached, e.g., 3 volts times 4 amps equals 12 watts. For this exam, we are only thinking about multiplication. We're going to start with a simplified version, and then extend its capabilities. To make this work, you are using `attach-tag` to attach a unit to a number.

(a) To start, write a version of the procedure `(times x y)` that multiplies two typed quantities, but doesn't know anything about relationships among units. Instead, if two numbers with the same unit are multiplied, the result should be in units of square-whatevers, like this:

```
> (times (attach-tag 'ft 4) (attach-tag 'ft 6))
```

```
(sq-ft . 24)
```

If two numbers with different units are multiplied, make a new unit name by putting a hyphen between the two given unit names, like this:

```
> (times (attach-tag 'ft 3) (attach-tag 'sec 7))
(ft-sec . 21)
```

Continued on next page.

Your name login cs61a-

Question 3 continued.

(b) Now extend your program to recognize known unit names for products of known units, using data-directed programming. These product units will be represented by table entries such as these:

```
(put 'volt 'amp 'watt)
(put 'dyne 'cm 'erg)
```

If the two arguments to `times` are of the same type, the program should work as in part (a). If the two arguments are of different types, look them up with `get`. (Don't forget that the two arguments may not be in the same order as the types in the table entry. That is,

```
(times (attach-tag 'volt 2) (attach-tag 'amp 6))
(times (attach-tag 'amp 6) (attach-tag 'volt 2))
```

should both work with the volt-amp-watt table entry above.) If you find another unit, as in the watt example, use that unit for the result, so that 3 volts times 4 amps gives a result of 12 watts. If you find no table entry at all for the given units, create a new unit with the two given units separated by a hyphen, as in part (a).

Question 4 (5 points):

The following definition comes from page 116 of *SICP* :

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
```

In this problem you're going to extend the idea of accumulation from sequences to ``deep lists'': lists of lists of lists to arbitrary depth. Write `deep-accumulate`, a function of three arguments: a two-argument function, an initial value, and a list structure. It should work like this:

```
> (deep-accumulate + 0 '(3 (4 (5) ((6) 7) 8) (9 10)))
52
```