CS61A, Spring 1995
Final

Question 1 (5 points):

(a) Write a function `prefix-to-infix` that takes a Scheme arithmetic expression as its argument and returns a list containing the equivalent expression in the ordinary arithmetic notation with operators between the operands, like this:

```
> (prefix-to-infix '(+ (* 2 3) (- 7 4)))
((2 * 3) + (7 - 4))
> (prefix-to-infix '(* (remainder 9 2) 5)
((9 remainder 2) * 5)
```

You may assume that every function in the argument expression has exactly two arguments.

(b) The procedure you wrote in part (a) carries out a tree reordering; in each sublist, the three elements are rearranged from the original order `(0 1 2)` to the new order `(1 0 2)`. That is, what used to be element number 1 now comes first; what used to be element number 0 now comes second, and what used to be number 2 remains third. We can represent this ordering by the list `(1 0 2)`. We'd like to generalize this by writing a procedure that takes an ordering as an additional argument, so that we could say

```
(define (prefix-to-infix tree)
  (tree-reorder '(1 0 2) tree))
```

Here's an example of a `tree-reorder` that isn't a `prefix-to-infix` :

```
> (tree-reorder '(2 1 2 1) '((a b c) (d e f) (g h i)))
((i h i h) (f e f e) (i h i h) (f e f e))
```

What follows is a partial implementation; your job is to fill in the blank.

```
(define (tree-reorder ordering tree)
  (if (atom? tree)
      tree

      (map _____________________________________________________
           ordering)))
```

Assume that no number in the ordering is bigger than the length of any sublist; no error checking is needed.

Question 2 (5 points):

You are given a possibly infinite stream of lists of numbers. In the following example, the notation `{...}` represents a stream, while `(...)` represents a list:

```
{(0 1 0 0 3) (1 2 3 0 4 2) (0 0 0 5 0) () (3) ...}
```

Write a procedure `positions` that, given such a stream as its argument, returns a stream of two-element lists showing the positions of the nonzero numbers within the argument. Each two-element list has the form

```
(list-number position-within-list)
```

so for the stream shown above you would produce a stream with elements

```
{(0 1) (0 4) (1 0) (1 1) (1 2) (1 4) (1 5) (2 3) (4 0) ...}
```

but not necessarily in the order shown. The elements may appear in any order in your result stream, provided that any specific element of it is reachable in finite time. Be sure not to confuse lists and streams! **Question 3 (5 points):**

(a) As you know, Logo is dynamically scoped. For each of the following Logo procedures, indicate whether that procedure

{1:} depends on dynamic scope to be able to do its job.

{2:} would work exactly the same under lexical scope.

{3:} might under some conditions give different results under lexical scope. Indicate your answer by checking one of the choices to the right of each procedure.

```
to circle.area :radius                    __ needs dynamic
output :pi * :radius * :radius            __ same either way
end                                       __ might matter



to square :num                            __ needs dynamic
output :num * :num                        __ same either way
end                                       __ might matter



to repeated :action :times                __ needs dynamic
if :times=0 [stop]                        __ same either way
run :action                               __ might matter
repeated :action :times-1
end
```

(b) As we discussed in lecture, the metacircular evaluator evaluates argument subexpressions either left to right or right to left, depending on the evaluation order of the underlying Scheme. We can tell this by examining the procedure `list-of-values`. For each of the following statements, say whether it's true or false, give a *one-sentence* explanation, and indicate which procedure(s) in the metacircular evaluator determine your answer. This question refers to the evaluator as presented in Abelson and Sussman section 4.1, not any modified versions.

1. The metacircular evaluator will implement dynamic scope if and only if the underlying Scheme uses dynamic scope.

2. In some versions of Scheme, the empty list counts as false. (That part is true!) The metacircular evaluator will consider the empty list to be false if and only if the underlying Scheme interpreter does.

3. The metacircular evaluator will understand the notation

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

for an improper list if and only if the underlying Scheme does.

4. The metacircular evaluator will understand the notation

```
(lambda (arg1 arg2 . rest) ...)
```

for a procedure with a variable number of arguments if and only if the underlying Scheme does.

Question 4 (5 points):

(a) Write query system rules to implement the {\tt assq} relation, as in this example:

```
query==> (assq bbb ((aaa . 5) (bbb . 6) (ccc . 7)) ?what)
(assq bbb ((aaa . 5) (bbb . 6) (ccc . 7)) (bbb . 6))
```

Do not use `lisp-value`!

(b) Which of the following queries will go into an infinite loop?

```
___ (assq bbb ((aaa . 5) (bbb . 6) (ccc . 7)) ?what)

___ (assq ?who ((aaa . 5) (bbb . 6) (ccc . 7)) ?what)

___ (assq bbb ?which (bbb . 6))

___ (assq ?who ((aaa . 5) (bbb . 6) (ccc . 7)) (bbb . 6))
```

Question 5 (5 points):

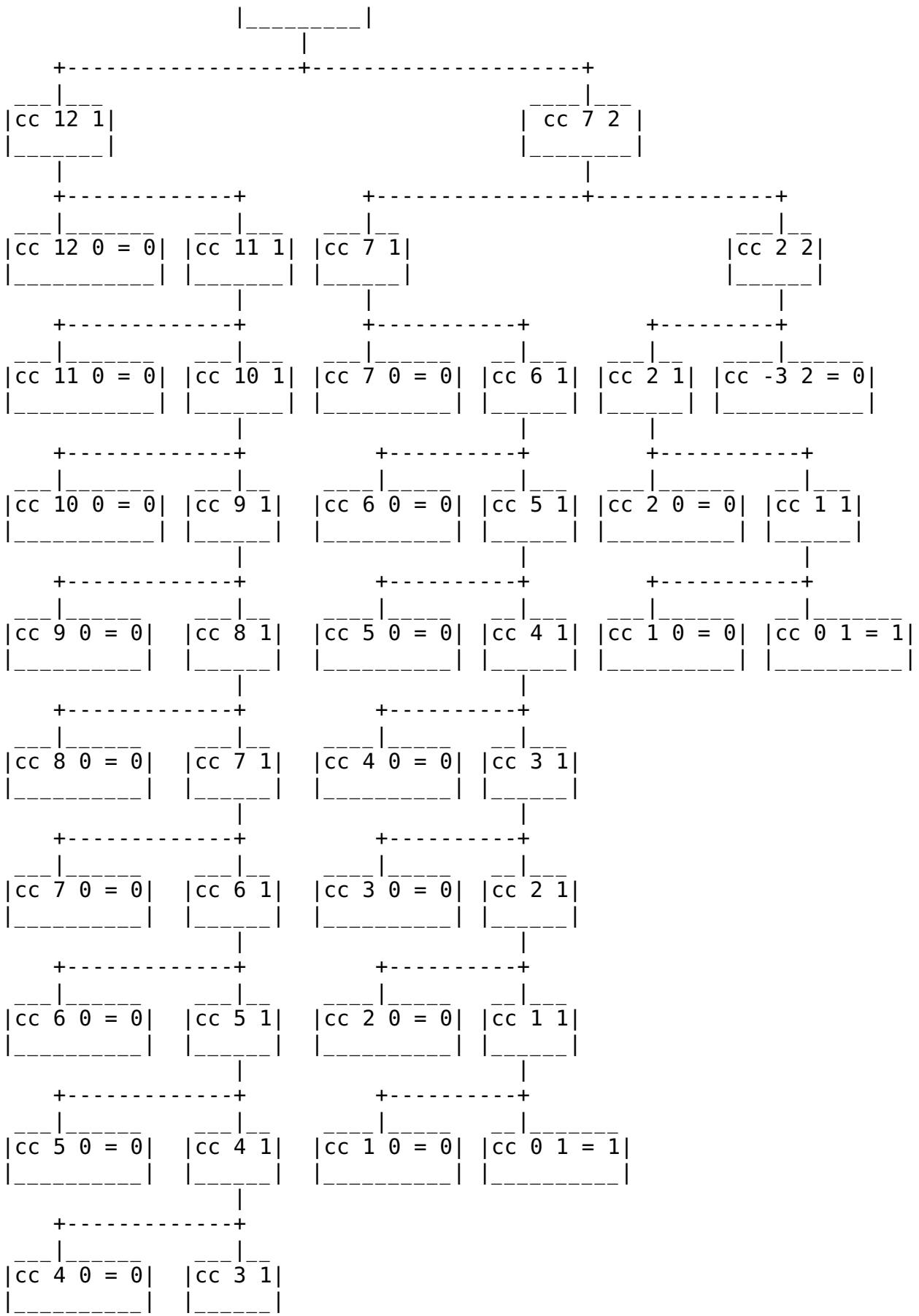
(a) Memoize this `cc` function (from page 38 of Abelson and Sussman):

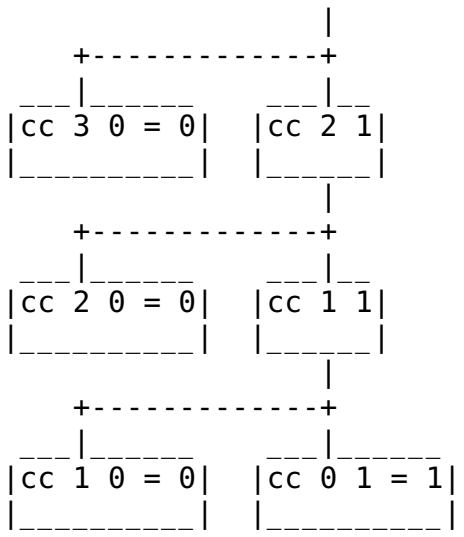
```
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (<amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc (- amount (first-denomination kinds-of-coins))
                     kinds-of-coins)
                 (cc amount (- kinds-of-coins 1)) ))))
```

You may use procedures from the book if you clearly identify where you found them.

(b) If the un-memoized `cc` is used to evaluate the expression `(cc 12 2)`, `cc` is called 49 times, as shown in the chart below. How many calls to the memoized `cc` are made as a result of evaluating the same expression? Briefly explain your answer.

```
          _________
        | cc 12 2 |
```


cc 12 1
cc 7 2
cc 12 0 = 0
cc 11 1
cc 7 1
cc 2 2
cc 11 0 = 0
cc 10 1
cc 7 0 = 0
cc 6 1
cc 2 1
cc -3 2 = 0
cc 10 0 = 0
cc 9 1
cc 6 0 = 0
cc 5 1
cc 2 0 = 0
cc 1 1
cc 9 0 = 0
cc 8 1
cc 5 0 = 0
cc 4 1
cc 1 0 = 0
cc 0 1 = 1
cc 8 0 = 0
cc 7 1
cc 4 0 = 0
cc 3 1
cc 7 0 = 0
cc 6 1
cc 3 0 = 0
cc 2 1
cc 6 0 = 0
cc 5 1
cc 2 0 = 0
cc 1 1
cc 5 0 = 0
cc 4 1
cc 1 0 = 0
cc 0 1 = 1
cc 4 0 = 0
cc 3 1



(c) Name a tree-recursive function that would *not* gain significant efficiency from being memoized. Explain briefly.

Question 6 (5 points):

We would like to add *clubs* to the adventure game. A club is a place in which only members are allowed. If someone who is not a member tries to enter a club, the club will `ask` the person to go to another place, the club's `outside`. For example:

```
(define Cavern (instantiate club 'Cavern Telegraph-Ave))
(can-go Telegraph-Ave 'east Cavern)
(can-go Cavern 'west Telegraph-Ave)
```

To join a club, a person sends it an `enroll` message:

```
(define George (instantiate person 'George Telegraph-Ave))
(ask Cavern 'enroll George)
```

Your job is to define the `club` class.

Do not modify any existing class definitions.

Take a peek at the solutions