CS 61A Spring 1995 Final exam solutions

1. prefix-to-infix, tree-reorder

(a)

```
(define (prefix-to-infix tree)
  (if (atom? tree)
      tree
      (list (prefix-to-infix (cadr tree))
            (car tree)
            (prefix-to-infix (caddr tree)))))
```

Scoring: 2 if correct, 1 if a tree recursive function, 0 if mutation used.

Most people did okay on this part, but many solutions were more complicated than necessary because of not believing in leaf nodes as the base case.

(b)

```
(define (tree-reorder ordering tree)
  (if (atom? tree)
      tree
      (map (LAMBDA (POS) (TREE-REORDER ORDERING (NTH POS TREE)))
           ordering)))
```

Scoring: 3 if correct, 2 if tree recursive, 1 if (lambda (number) ...),
1 if map inside map, 0 if lambda's argument unused (or no arg),
0 if (lambda (list) ...) or no lambda at all,
0 if code works only for examples shown.

A fairly common wrong answer was

```
   (lambda (pos) (nth pos (tree-reorder ordering tree)))
```

which of course is an infinite recursion since the problem never gets smaller!

2. streams

The solution we were expecting was this:

```
(define (positions str)
  (FLATTEN (pos-help str 0)))

(define (pos-help str index)
  (if (empty-stream? str)
      the-empty-stream
      (CONS-STREAM (nonzeros (head str) index)
```

```
                  (pos-help (tail str) (+ index 1)))))

(define (nonzeros lst index)
  (nz-help lst index 0))

(define (nz-help lst index subindex)
  (cond ((null? lst) the-empty-stream)
        ((= (car lst) 0) (nz-help (cdr lst) index (+ subindex 1)))
        (else (cons-stream (list index subindex)
                           (nz-help (cdr lst) index (+ subindex 1))))))
```

The crucial point in this solution is that the capitalized CONS-STREAM in pos-help can't be append-streams or interleave, because those would fail to delay the recursive call. The call to FLATTEN in positions turns the stream of streams of positions into a flat stream of positions as desired.

Another possible solution essentially does the flattening as it goes:

```
(define (positions str)
  (if (empty-stream? str)
      the-empty-stream
      (pos-help str (head str) 0 0)))

(define (pos-help str lst index subindex)
  (cond ((empty-stream? str) the-empty-stream)
        ((null? lst)
         (pos-help (tail str) (head (tail str)) (+ index 1) 0))
        ((= (car lst) 0)
         (pos-help str (cdr lst) index (+ subindex 1)))
        (else (cons-stream (list index subindex)
                           (pos-help str (cdr lst) index (+ subindex 1))))))
```

Scoring:

```
5  ok
4  uses interleave or append-streams, not flatten or interleave-delayed
3  gives stream of streams as result, or uses (list) append
2  leaves out some positions, e.g., only gets one per list
1  solves wrong problem: no positions at all
0  uses mutation

-1 from above if no base case (stream must be infinite)
-1 from above if gets lists and streams mixed up (e.g., takes car of stream)
```

3. Metacircular evaluator and Logo

(a) How do these procedures interact with dynamic scope?

```
to circle.area :radius
output :pi * :radius * :radius
```

```
end
```

MIGHT MATTER -- if this procedure is called by another procedure that has a local value for the variable PI, then the result won't be what we expect.

```
to square :num
output :num * :num
end
```

SAME EITHER WAY -- this procedure makes no references to variables other than its own formal parameter, so it doesn't matter what environment its new frame extends.

```
to repeated :action :times
if :times=0 [stop]
run :action
repeated :action :times-1
end
```

NEEDS DYNAMIC -- if a calling procedure wants to carry out an action that involves its local variables, then REPEATED must have dynamic scope in order to be able to evaluate the expressions that the caller gives it.

A few people made a good argument for "might matter" in the last example because the action to be repeated might not refer to any local variables. But we feel that "needs dynamic" is the best answer because *in general* this procedure can't do what it claims to do without dynamic scope.

Scoring: 1 point each.

(b) Are these metacircular evaluator (MCE) features inherited from the underlying Scheme (US)?

General comment: The instructions specifically refer to the MCE as presented in the textbook, without modification, and each question asks if *that* MCE will have a certain behavior if and only if the US does. It's not responsive to say, "false, because we could modify the MCE..."

1. Dynamic scope. FALSE. An environment in the MCE is just a list structure, not an environment, in the US. The scope rule is determined in APPLY, by the third argument given to EXTEND-ENVIRONMENT.

2. Empty list is false. FALSE. In the MCE as presented in A&S, the empty list is always false, because of the procedure

```
(define (true? x) (not (null? x)))
```

However, if the empty list is different from #f in the US, then
#f will not be considered false in the MCE!

Notice that the question was not, "is () eq? to #F" but rather
"is () considered false".

3. Improper list notation. TRUE. The MCE's read-eval-print loop uses the READ function from the US, and that's what turns dot notation into improper lists.

Although the use of the phrase "improper list" in the question should have made this clear, some people thought that the question was asking about the ' notation for a quoted list. If that were the question, the answer would be complicated: The MCE inherits from the US the fact that 'xxx is turned into (quote xxx), because that's done by the READ procedure, just as the dot notation is turned into an improper list by READ. Whether (quote xxx) means to quote xxx, though, is not inherited; it's determined by the call to text-of-quotation in EVAL. We accepted a FALSE answer if it was accompanied by an explanation that made it clear that you were answering this question with this understanding, but there weren't very many of those. (Lots of people mentioned text-of-quotation but with rather confused reasoning.)

4. Rest args. FALSE. The dot notation will be turned into an improper list by READ, but the MCE's procedure MAKE-FRAME looks only at the CARs of the formal parameter list.

Many people gave handwavy reasons for #3, and a few people confused #4 with a question from an old exam about dot notation in the *actual argument* list.

Scoring: 1/2 point each, no credit if no explanation, rounding down.

4. assq in logic programming

(a) If you don't mind some duplicate answers you can do it this way:

```
(rule (assq ?var ((?var . ?val) . ?more) (?var . ?val)))

(rule (assq ?var (?other . ?more) ?result)
      (assq ?var ?more ?result))
```

To make sure that only the first match in an a-list counts, you could instead write the rules this way:

```
(rule (assq ?var ((?var . ?val) . ?more) (?var . ?val)))
```

```
(rule (assq ?var ((?other . ?val) . ?more) ?result)
      (and (assq ?var ?more ?result)
           (not (same ?var ?other))))
```

Many people wanted a third rule

```
    (rule (assq ?var () #F))
```

to make this work more like Scheme's assq, but that's actually rather non-logic-programmingy. Ordinarily you would want no match at all if the name you're looking for isn't in the a-list. For example, consider writing the rule

```
(rule (in-alist ?thing ?alist)
      (assq ?thing ?alist ?anything))
```

This rule will always succeed if the #F rule is used.

Even worse, and actually more common, was

```
    (rule (assq ?var () ()))
```

which makes no sense at all. Probably people did this because of a mechanical idea that everything has to have a base case for the empty list!

By the way, DON'T say (assq ?var '() '#F)! The query language isn't Scheme; there's no need to quote anything because nothing is ever evaluated.

Scoring:

4 OK
3 Basically okay, e.g., includes #F rule
2 One rule right
1 Solved the wrong problem
0 Not logic programming, e.g., uses composition of functions.

(b) Which query won't work?

The third one, (assq bbb ?which (bbb . 6)). How on earth is the query system supposed to make up an a-list for this? Depending on the order in which you enter the rules, you'll get either no answer at all or an infinite number of partial answers like

```
(ASSQ BBB ((BBB . 6) . ?MORE-1) (BBB . 6))
(ASSQ BBB (?OTHER-2 (BBB . 6) . ?MORE-3) (BBB . 6))
(ASSQ BBB (?OTHER-2 ?OTHER-4 (BBB . 6) . ?MORE-5) (BBB . 6))
(ASSQ BBB (?OTHER-2 ?OTHER-4 ?OTHER-6 (BBB . 6) . ?MORE-7) (BBB . 6))
...
```

All the other queries are fine. In particular,

```
query==> (assq ?who ((aaa . 5) (bbb . 6) (ccc . 7)) ?what)
(ASSQ AAA ((AAA . 5) (BBB . 6) (CCC . 7)) (AAA . 5))
(ASSQ BBB ((AAA . 5) (BBB . 6) (CCC . 7)) (BBB . 6))
(ASSQ CCC ((AAA . 5) (BBB . 6) (CCC . 7)) (CCC . 7))
DONE
```

This is the sort of thing that shows the power of logic programming.

Scoring: One point, all or nothing!

5. Memoization

(a) Memoize cc.

The key point is that you CAN'T use the memoize function from page 218 of A&S, because that function only works with functions of one argument. You could rewrite a new version of memoize, but the simplest solution is probably to use GET and PUT:

```
(define (memo-cc amount kinds-of-coins)
  (let ((result (get amount kinds-of-coins)))
    (if result
        result
        (let ((new (real-cc amount kinds-of-coins)))
          (put amount kinds-of-coins new)
          new))))

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

Scoring:

3 OK
2 Call memoized version in recursion, try to save state somehow
1 Blindly use MEMOIZE from the book
0 Not memoized at all

(b) How many calls?

The calls that are duplicated are (cc 7 1) and (cc 2 1). So the ones we eliminate are 14 "children" of the second (cc 7 1) and 4 "children" of the second (cc 2 1). So the answer is 49-18, or 31.

The most common wrong answer was 29, because people thought that the second call for (cc 7 1) wouldn't happen. But of course it does happen, and that's where we notice that we already know the answer. But we gave credit for this almost-right answer.

The next most common wrong answer was "It makes no difference the first time, but there's only one call the second time." This answer misses the main point about memoization, which is that it avoids redundant computation of subproblems even the first time we use it.

Scoring: one point, all or nothing.

(c) Name a tree-recursive function that wouldn't gain from memoization.

Any tree-recursive function that has to look at every node of the tree, even if two equal nodes appear within it: countatoms, equal?, deep-reverse, and subsets are examples.

Scoring: one point, all or nothing. You had to *name a procedure,* not just give an argument (out of the book) about trees without much redundancy. In particular, referring to binary search trees is especially bad, because the most common operations on a BST, namely lookup and insert, are not tree recursive! The whole point of a BST is that you only have to look at one branch in any particular case.

6. OOP

```
(define-class (club name outside)
  (parent (place name))
  (instance-vars (members '()))
  (method (enroll who)
    (set! members (cons who members)))
  (method (enter who)
    (if (member who members)
        (usual 'enter who)
        (ask who 'go-directly-to outside))))
```

Scoring:

5 OK
4 Small errors, e.g., GO instead of GO-DIRECTLY-TO, APPEAR instead of ENTER
3 inherits from place, uses it, has a state variable for members,
  but something wrong, e.g., no SET! in the enroll method.
2 Screwy object hierarchy, e.g., inherits from PERSON as well as PLACE.
2 Copy code from place method instead of using USUAL.
1 Solves the wrong problem

```
0  Even worse.
```

Go back to the questions
Back to the Midterm/Final page