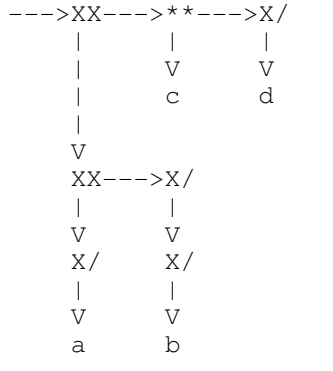
CS 61A Spring 2006 Midterm 2 solutions

1. Box and pointer.

Note: Please draw actual boxes, as in the book and the lectures, not XX and X/ as in these ASCII-art solutions.

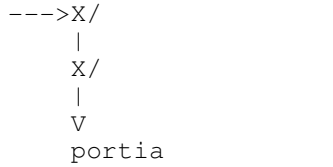
Also, please put in the start arrows! Sometimes it's hard to know where your diagrams are supposed to begin, especially if you use leftward and upward pointing arrows.

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



The pair marked ** in the diagram above is the value of the subexpression (APPEND '(C) '(D)), namely, the list (C D). The CONS sticks one pair in front of that, the pair at the start arrow. Its car, the first element of the new list, is the value of (LIST '(A) '(B)), which is ((A) (B)).

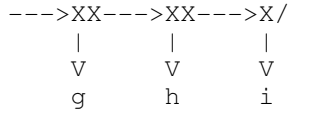
```
> (car (cons '((portia)) '(black . satin)))
((PORTIA))
```



This should have been really easy, because CAR is an inverse function of CONS; that is, (CAR (CONS X Y)) is always X, whatever X might be. So you could entirely ignore the second argument to CONS.

The double parentheses in ((PORTIA)) mean that it's a list of one element, which is itself a list of one element, a word.

```
> (caadr '( ((a b c) (d e f)) ((g h i) (j k l)) ((m n o) (p q r)) ))
(g h i)
```



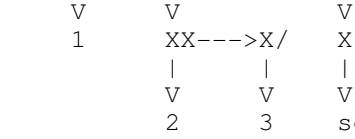
This was mainly a test of whether you understand the order of operations in the C...R functions. CAADR means the CAR of the CAR of the CDR, not "first do CAR then CAR then CDR"! So we have

```
(cdr '(((a b c) (d e f)) ((g h i) (j k l)) ((m n o) (p q r))))
(car '(((g h i) (j k l)) ((m n o) (p q r))))
(car '((g h i) (j k l)))
(g h i)
```

A shortcut to this solution is to remember that CADR returns the second element of a list, so you can get to ((G H I) (J K L)) in one step, then take the CAR (the first element) of that.

If you did the operations in reverse order, you got the wrong answer (B C).

```
> (filter (lambda (x) (if (list? x) (pair? x) (number? x)))
          '(1 () (2 3) (so) what))
(1 (2 3) (so))
```



This was an exercise in understanding FILTER, IF, and predicates. The IF subexpression returns true for a list that's also a pair (i.e., not the empty list) or for a non-list that's a number (not a non-numeric word, for example). So these tests are made:

element	first test	result	second test	result
1	LIST?	#F	NUMBER?	#T
()	list?	#t	pair?	#f
(2 3)	LIST?	#T	PAIR?	#T
(SO)	LIST?	#T	PAIR?	#T
what	list?	#f	number?	#f

The lines in capital letters show the elements that FILTER keeps. To solve this problem correctly you also had to understand that there is no recursion here, so this is not a deep-list problem: the elements of the elements are not tests (which would have rejected the word SO).

Scoring: One point for each printed result; one point for each diagram, except that you lost at most one point for putting quotation marks in the printed results.

2. Data abstraction.

This should have been an easy one, but a lot of people had trouble with it. It's important to know the domains of procedures; FIRSTS's domain is Trees, but HELPER's domain is forests, not trees, so (DATUM X) makes no sense.

The argument to FIRSTS is a Tree, so its proper selectors are DATUM and CHILDREN, and to construct a new one you use MAKE-TREE (or you could call it MAKE-NODE, equivalently). DATUM of this Tree is a sentence, so its selectors are FIRST and BUTFIRST. CHILDREN of a Tree (the argument to HELPER) is a forest, which is just a sequence (a list), so its selectors are CAR and CDR, and its constructor is CONS.

```
(define (firsts tree)
  (MAKE-TREE (FIRST (DATUM tree))
             (helper (CHILDREN tree))))

(define (helper x)      ; We chose this bad parameter to hide the type!
  (if (NULL? x)         ; Don't use X in your own programs.
      '()
      (CONS (firsts (CAR x))
            (helper (CDR x)))))
```

Scoring: One point off for each error, either failing to change something that should be changed, or changing something that shouldn't.

3. Understanding Scheme-1.

EVAL-1 deals with *expressions* -- with the notation in which programs are expressed. APPLY-1 deals with *values*, including procedures as well as their arguments, and knows how to call a procedure.

So (a), which is about the *notation* for a procedure call, and (c), which is about a special form, are changes to EVAL-1. For (a), we'd add a clause before line 10, something like

```
        ((and (pair? exp)                                ; exp is a list
              (not (null? (cdr exp)))                    ; of length > 1
              (arithmetic-operator? (cadr exp)))         ; 2nd elt. is +-*/
         (apply-1 (eval-1 (cadr exp))                    ; apply val of oper
                  (list (eval-1 (car exp))               ; to first elt
                        (eval-1 (caddr exp)))))          ; and third elt
```

For (c), we'd add a clause somewhere before line 10 saying

```
        ((define-exp? exp)
         (add-to-global-table! (cadr exp)                ; name is 2nd elt
                               (eval-1 (caddr exp))))    ; value exp is 3rd
```

and we'd change line 3 to

```
        ((symbol? exp)
         (if (found-in-global-table? exp)       ; if we've defined this name
             (value-in-global-table exp)        ; get value we saved
             (eval exp)))                       ; else ask STk as before
```

But (b) is about how to call a procedure; we'd change line 17 of APPLY-1 to:

```
        (eval-body (substitute (caddr proc)
                               ...
```

where EVAL-BODY evaluates the expressions in turn:

```
(define (eval-body exprs)
  (if (null? (cdr exprs))               ; if only one expression left,
      (eval-1 (car exprs))              ; return its value
      (begin (eval-1 (car exprs))       ; otherwise eval the first one
             (eval-body (cdr exprs)))))) ; but keep going
```

The unusual base case reflects the fact that we have to return the value of the last expression; that's why we can't just use FOR-EACH to evaluate all the expressions in the body.

Scoring: 2 points each, all or nothing.

4. Function overloading with DDP

Changing the syntax of the parameter list in STk's lambda expressions would be possible only using tools beyond the scope of this course. We could build an overloaded DEFINE into Scheme-1, but that's not what the question asked for. Instead we create two new procedures, OVERLOADED and DEFINE-OVERLOADED.

But these are really just like SICP's implementation of DDP for functions of possibly more than one argument, using the type signature to find the correct method. So all you had to do for full credit on this question is

```
(define define-overloaded put)

(define overloaded apply-generic)       ; the one on SICP page 184
```

Other correct solutions basically involved rewriting one or both of PUT and APPLY-GENERIC.

Scoring: We allocated 2 points to DEFINE-OVERLOADED, which was pretty much all or nothing. We allocated 3 points to OVERLOADED, as follows:

3 Correct.

2 Passes typed data to method (doesn't call CONTENTS).
2 Mistakes in dot notation for variable number of arguments.
2 (method args) instead of (apply method args).

```
1  Only works for two arguments.
1  Checks only one type tag.

0  Anything worse.
```

5. Deep lists.

(a) Car/cdr recursion means two recursive calls for each pair, one for the car, and one for the cdr!

```
(define (deepen lst)
  (cond ((null? lst) '())
        ((pair? lst) (cons (deepen (car lst)) (deepen (cdr lst))))
        (else (list lst))))
```

(b) Using MAP means that the recursion is indirect, through a call to MAP. Improper lists (pairs whose cdrs are non-null atoms) aren't in the domain of this procedure; you could have an error check if you wanted, but we don't require error checks on exams:

```
(define (deepen lst)
  (cond ((null? lst) '())       ;; This test is unnecessary but okay.
        ((list? lst) (map deepen lst))
  ;;    ((pair? lst) (error "no improper lists"))  ; Optional.
        (else (list lst))))
```

A common error on (b) was to try to combine car/cdr recursion with MAP, like this:

```
(define (deepen lst)
  (cond ((null? lst) nil)
        ((atom? (car lst)) (cons (list (car lst)) (map deepen (cdr lst))))
        (else (map deepen lst))))
```

This doesn't work, because the elements of (cdr lst) might be atoms, in which case the map will fail. But even if it did work, this problem is about understanding different styles of programming, and you should know how to write a MAP-based tree recursion cleanly. Such solutions got 2 points out of 4 for part (b).

Scoring: 4 points for each part, as follows:

```
4  Correct.
3  Has the idea.
2  Has an idea.
0  Other.
```

Some particular common examples: -2 for LIST or APPEND instead of CONS; -2 for SYMBOL? instead of ATOM?.

6. Tree search.

The procedure should return #T in two cases:

(1) The two desired data are directly below this node;
(2) A recursive call for some child returns #T.

```
(define (siblings? tree datum1 datum2)
  (define (helper forest)
    (cond ((null? forest) #f)
          ((siblings? (car forest) datum1 datum2))
          (else (helper (cdr forest)))))
  (or (and (member datum1 (map datum (children tree)))
           (member datum2 (map datum (children tree))))
      (helper (children tree))))
```

An alternative to using a helper function would be to replace the last line with

```
       (not (null (filter (lambda (child) (siblings? child datum1 datum))
```

```
                        (children tree))))
```

This is perfectly acceptable, but less efficient; if the first child satisfies the condition, it checks the other children anyway. (A useful higher order function we should have in our library is FIND-FIRST, like FILTER except that it returns only the first element that satisfies the condition.)

You *can't* avoid a helper by calling SIBLINGS? itself with (CHILDREN TREE) as its first argument. That argument has to be a tree, not a forest!

You could make this solution slightly more efficient by using

```
        (let ((data (map datum (children tree))))
          (or ...))
```

to avoid calling MAP twice.

Some people, taking advantage of the assumption that an element appears at most once, used my find-place procedure from lecture to find the path from the root to each datum, and then see if the butlasts of the paths are the same. This is a fairly reasonable "use what you have to get what you need" approach for an exam, if you don't know how to solve the problem properly, but in real programming it's really inefficient and ugly to create data structures (the paths) that you don't really need to solve the problem. And this solution isn't quite correct; it's a data abstraction violation to use BUTLAST on a path, which isn't a sentence. [You could instead say (reverse (cdr (reverse path))), making this solution even uglier, but correct.] Also, your program will be more robust if it doesn't rely on the uniqueness of the data, a condition that isn't necessary in our solution above.

Scoring:

8 Correct.

7 (NULL? TREE).

6 Tests each node correctly but doesn't accumulate the results correctly.
6 No base case in helper.

5 Forgets the (MAP DATUM ...).
5 Data abstraction violation, but does the right thing.

2 (SIBLINGS? (CHILDREN TREE) DATUM1 DATUM2).

0 Even worse (e.g., no tree recursion).

If you don't like your grade, first check these solutions. If we graded your paper according to the standards shown here, and you think the standards were wrong, too bad -- we're not going to grade you differently from everyone else. If you think your paper was not graded according to these standards, bring it to Brian or your TA. We will regrade the entire exam carefully; we may find errors that we missed the first time around. :-)

If you believe our solutions are incorrect, we'll be happy to discuss it, but you're probably wrong!