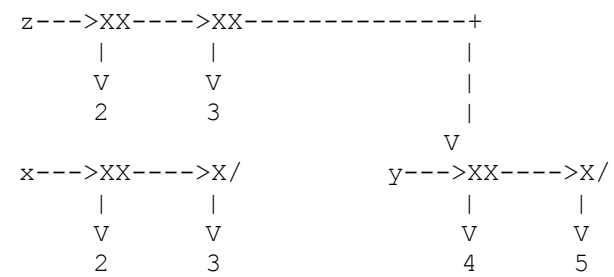
61A Spring 2009 Midterm 2 solutions

1. Box and pointer diagram.

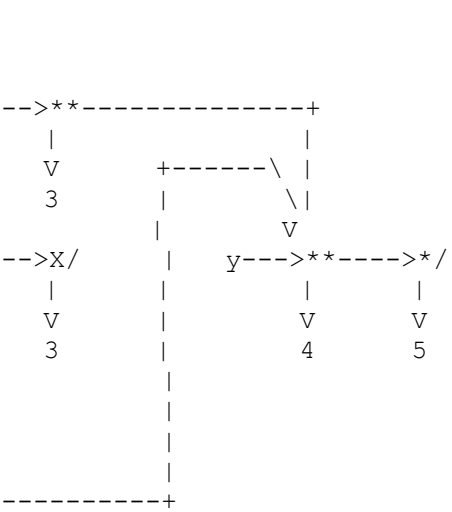
Before the APPEND we have this:

After the APPEND we have this:



because APPEND copies the spine pairs of all but its last argument. So in
this case the result of the APPEND shares spine pairs with list Y, but not
with list X.

Then after the CONS we have this:



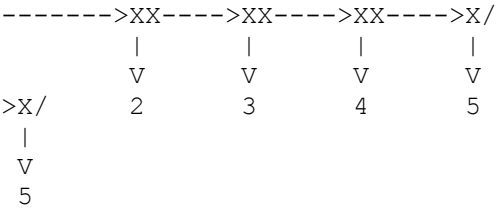
The start arrow is in the bottom left corner of this diagram. The pairs marked as ** and */ form the spine of the resulting list; you can tell this by following the CDRs from one pair to another, starting with the

start pair, the one made by the CONS.

You can see from the diagram that this is a list of five elements:

((4 5) 2 3 4 5)

Because we're not yet at the point in the course at which it really matters
whether or not the two occurrences of the list (4 5) in the result, first as
an element and then as a sublist, are the same pairs, we also accepted
diagrams like this:



But you should be aware that on midterm 3 there will probably be a similar
question, and that time you have to get the distinction between equality
and identity (to be discussed next week) right!

Scoring: 2 points for the diagram, 2 points for the print form.

2. OOP categories.

(a) A chimney is /part of/ a house, not a kind of house, so this is
an INSTANCE VARIABLE.

(b) DRIVE is something you can /do with/ an automobile, so it's
a METHOD.

(c) A comic book is a /kind of/ book, so it's a CHILD CLASS.

(d) Although you might use binoculars to look at animals in the jungle,
binoculars are neither part of an animal nor a kind of animal, so it's
NONE OF THE ABOVE.

(e) A keyboard is /part of/ a computer, so it's an INSTANCE VARIABLE.

This should have been easy, if you understand how objects fit together.

Scoring: One point each.

3. Scheme-1.

You are asked to make TWICE a /primitive procedure/ in Scheme-1. The
primitives of Scheme-1 are STk procedures, not built into EVAL-1 or
APPLY-1. So you're going to define an STk procedure:

```
(define (twice f)
  (lambda (x) (apply-1 f (list (apply-1 f (list x))))))
```

The two calls to LIST are there because APPLY-1's second argument has to be
a /list of argument values/; in this case it's a list of length 1, because
F takes one argument.

More importantly, the calls to APPLY-1 are there because F can be any Scheme-1 procedure, which might or might not be an STk procedure! If you
just say (LAMBDA (X) (F (F X))) then F has to be an STk procedure, i.e., a Scheme-1 primitive. So this won't work:

```
        Scheme-1> ((twice (lambda (x) (* x x))) 5)
```

You were asked to make TWICE a procedure, not a special form. So a solution
that involves modifying EVAL-1 by adding a clause

```
        ((twice-exp? exp) (eval-twice exp))
```

doesn't really do what you were asked. But you'd get part credit if you make
it work, like this:

```
(define (eval-twice exp)
  (lambda (x) (apply-1 (eval-1 (cadr exp))
                       (list (apply-1 (eval-1 (cadr exp))
                                      (list x)))))))
```

That is, your special form has to do the same thing the procedure would do,
but it also has to evaluate the argument to the TWICE form in order to get
an actual Scheme-1 procedure.

In both these solutions, what TWICE returns is an STk procedure, i.e., a new Scheme-1 primitive. It's almost possible, but not quite, to write a solution in which TWICE returns a Scheme-1 defined (compound) procedure:

```
(define (twice f)              ;; almost right
  (list 'lambda '(x)
        (list 'apply-1 f (list 'list
                               (list 'apply-1 f '(x))))))
```

This works if F is a Scheme-1 defined procedure, but not if it's a Scheme-1 primitive, because you'll then try to EVAL-1 the body of the procedure that TWICE creates and one piece of that body is an actual STk procedure, rather than the name of one. It /is/ possible to use this approach successfully if you make TWICE a special form:

```
(define (eval-twice exp)
  (list 'lambda '(x)
        (list 'apply-1 (cadr exp) (list 'list
```

```
                                        (list 'apply-1 (cadr exp)
'(x))))))
```

Scoring:

We started with 6 points and subtracted for errors:

-1 for making TWICE a special form instead of a procedure.
-1 for the almost-right constructed lambda expression above.
-1 for forgetting to use LIST for the second argument to APPLY-1.
-2 for 'F instead of F.
-3 for leaving out the calls to APPLY-1 or to EVAL-1 where needed.
-4 for leaving out /both/ APPLY-1 and EVAL-1.
-4 for thinking that just saying (LAMBDA ...) will create a Scheme-1
   compound procedure, as opposed to an STk procedure.

Solutions that didn't even come close to one of these patterns got 0.

4. Trees.

It's an important simplification that the Tree contains only positive numbers; this means that you can use zero as the identity element for MAX.
This in turn means that you don't have to worry about special cases such as a
node with no children. The problem would be conceptually harder otherwise.

```
(define (max-sum-children tree)
  (max (accumulate + 0 (map datum (children tree)))
       (accumulate max 0 (map max-sum-children (children tree)))))
```

The first argument to MAX is the sum of the data of the children of the root node; the second argument is the largest sum-of-data-of-children of any node below the root node.

Note that (CHILDREN TREE) returns a list of Trees, not a list of numbers,
so you can't (ACCUMULATE MAX 0 (CHILDREN TREE))!

If a node has no children, then (CHILDREN TREE) returns the empty list, so
(MAP ... (CHILDREN TREE)) returns the empty list, without calling the function
argument at all. So (ACCUMULATE +-or-MAX 0 (MAP ...)) returns 0, the base
case value, without calling + or MAX.

Note that we don't call (DATUM TREE) at all, because the root datum can't be
part of a sum-of-children. But we do /map/ DATUM over the children of TREE.

Another solution, that works even for trees that include negative numbers,

uses the sum of the numbers just below the root as the /base case value/ for
finding the MAX of the grandchildren:

```
(define (max-sum-children tree)
  (accumulate max
              (accumulate + 0 (map datum (children tree)))
              (map max-sum-children (children tree))))
```

This is no longer than the other solution, but it's much cleverer. It recognizes that although MAX has no universal base case (it would have to be
negative infinity), you can use any one of the candidate values as the base
case for any particular MAX computation, and we happen to have one candidate
value that's a natural choice because it's computed differently from all the
others, without recursive calls to max-sum-children.

Scoring:

8 correct
7 trivial mistake (e.g., includes (DATUM TREE) in the computation)
5 has the idea (e.g., has one ACCUMULATE but not the other)
2 has an idea (e.g., serious domain/range error)
0 other

"Has the idea" means that you do some accumulation over the data-of-children
of every node in the tree, even if you get the arithmetic wrong.

"Has an idea" means that you at least visit every node and accumulate something; a typical case would be to try to take the sum of the children,
rather than the sum of the data of the children.

5. Generic operators.

(a) This one was just to make sure you understand what a constructor and a type predicate are.

```
(define (make-money money)
  (attach-tag 'money money))

(define (make-clothes clothes)
  (attach-tag 'clothes clothes))

(define (money? thing)
  (and (pair? thing) (eq? (type thing) 'money)))

(define (clothes? thing)
  (and (pair? thing) (eq? (type thing) 'clothes)))
```

For the type predicates, we accepted solutions that left out the PAIR? test,
but you should understand that type predicates should accept anything at all
as argument. For example, INTEGER? doesn't error out if given a non-number.

Surprisingly many people got this wrong by answering some other question,
e.g., defining the tagged-data abstraction.

(b) This is to see if you can use GET to get what you need. In this part
you are given explicit type tags and numbers to work with, not type-tagged
generic values, so it's relatively easy:

```
(define (convert amount from to)
  (let ((from->to (get from to))
        (to->from (get to from)))
    (cond (from->to (* amount from->to))
          (to->from (/ amount to->from))
          (else (error "Can't convert types")))))
```

It's not until part (c) that we're going to need all those type-related
functions! Nevertheless, we accepted solutions that did something more
complicated, such as adding a type tag to the result, provided that the
use of CONVERT in part (c) is consistent with its definition.

The most common wrong answer was to assume that the conversion would be in
the table in the direction you want. We gave these one (out of two) point.

(c) This is the complicated part. If they're both money, we use CONVERT
to try to convert. If they're both clothing, then the subtypes have to be
the same and we just add. Otherwise (mixed money and clothing, or none of
the above) it's an error.

If adding two different money subtypes, it doesn't matter which one you pick
for the type of the result, since CONVERT will handle either direction of
conversion. Below I've picked the subtype of the first argument.

Despite what the question tells you that you have available as possible
helper procedures, you really can't use things like DOLLARS? in the solution,
since it has to work for /any/ money types, not just dollars and yen.

```
(define (+possession x1 x2)
  (cond ((and (money? x1) (money? x2))
         (let ((subx1 (contents x1))
```

```
              (subx2 (contents x2)))
          (let ((type1 (type subx1))
                (type2 (type subx2)))
            (if (eq? type1 type2)
                (make-money
                 (attach-type type1
                              (+ (contents subx1) (contents subx2))))
                (make-money
                 (attach-type type1
                              (+ (contents subx1)
                                 (convert (contents subx2)
                                          type2 type1)))))))))
       ((and (clothes? x1) (clothes? x2))
        (let ((subx1 (contents x1))
              (subx2 (contents x2)))
          (let ((type1 (type subx1))
                (type2 (type subx2)))
            (if (eq? type1 type2)
                (make-clothes
                 (attach-type type1
                              (+ (contents subx1) (contents subx2))))
                (error "Can't add unlike types")))))
       (else (error "Not possessions"))))
```

Hardly anyone got every detail of this. For grading purposes, we gave one point for using two levels of type tags (first money/clothes, and within that, the particular kind of money or clothes); one point for getting the money part basically right (using CONVERT, not necessarily realizing that if the two money amounts are in the same currency there won't be a table entry and no conversion is necessary); and one point for getting the clothing part basically right (disallowing mixed subtypes). Then we took off a point for egregious data abstraction violations (CAR and CDR everywhere instead of the appropriate selectors).

(d) This part of the question was really mainly to see if you read the book! Here's a procedure from the book:

```
(define (install-complex-package)
  ;; imported procedures from rectangular and polar packages
  (define (make-from-real-imag x y)
    ((get 'make-from-real-imag 'rectangular) x y))
  ; ...
  ;; internal procedures
  (DEFINE (ADD-COMPLEX Z1 Z2)
    (MAKE-FROM-REAL-IMAG (+ (REAL-PART Z1) (REAL-PART Z2))
                         (+ (IMAG-PART Z1) (IMAG-PART Z2))))
  ; ...
  ;; interface to rest of the system
  (define (tag z) (attach-tag 'complex z))
  (put 'add '(complex complex)
       (lambda (z1 z2) (tag (add-complex z1 z2))))
```

```
  ; ...
  'done)
```

The capitalized part provides the template for what you're supposed to write. Since your procedure returns a true/false value, not a complex number, there's no need to add any type tags to it:

```
(define (=c z1 z2)
  (and (= (real-part z1) (real-part z2))
       (= (imag-part z1) (imag-part z2))))
```

That's it! The workings of REAL-PART and IMAG-PART are really complicated, but they work -- you don't have to write them. And you don't have to look at the subtypes of the two complex numbers; REAL-PART and IMAG-PART do that for you.

Since the point of this part of the question was to see if you understand that you can rely on generic operators to work regardless of whether the complex numbers are in rectangular or polar form, we took off one point (out of two) for any explicit checking of the subtypes, even if your code would work.

By the way, some people checked for the same subtype as a special case, then using (EQUAL? (CONTENTS Z1) (CONTENTS Z2)) to test for equality. But this is mathematically wrong! If the numbers are in polar form, then one might have an angle of 0, and the other might have an angle of 360, and they'd still be the same number. But we didn't take off for this.

Scoring:

1 point for part (a).
2 points for part (b).
3 points for part (c).
2 points for part (d).

The common wrong answers and their scoring are explained above.

If your part (b) does too much, we included the extra parts as part of the solution to (c).

6. Deep lists.

You needed a helper procedure that takes a number DEPTH as an extra argument. It's easiest if you extend the domain of the helper to include words as well as lists-of-lists:

```
(define (depth-tag lol)
  (define (help lol depth)
    (if (list? lol)
        (map (lambda (elt) (help elt (+ depth 1)))
             lol)
        (word depth lol)))
  (help lol 0))
```

Many students were reluctant to include words in the domain of the helper
because that would also, in this implementation, extend the domain of DEPTH-TAG itself:

```
> (depth-tag 'foo)
0foo
```

It's not usually a problem to write a procedure whose domain is a /superset/
of the desired domain. But if it worries you, just change (HELP LOL 0) on
the last line of the definition above to

```
  (map (lambda (elt) (help elt 1)) lol)
```

Most student solutions were more complicated than this, usually because you
eschewed the use of higher order functions. Another way to complicate the
solution is to look explicitly one level deeper than the level you're on:

```
  (if (list? (car lol)) ...)
```

This is a sign of not really believing in recursion.

Scoring:

8 correct
7 trivial error
5 has the idea
2 has an idea
0 other

Here are some typical "has the idea" solutions:

* Tree recursion correct, and depth computation correct, but something very
wrong with the construction of the result list (usually because of an attempt
to do it iteratively -- /never/ try that for a tree-recursive problem!) or
a serious failure to distinguish words from lists properly.

* Adding one to the depth for the CDR of the list, as well as for the CAR.