Exercise 2.81.

Louis Reasoner has noticed that `apply-generic` may try to coerce the arguments to each other's type even if they already have the same type. Therefore, he reasons, we need to put procedures in the coercion table to "coerce" arguments of each type to their own type. For example, in addition to the `scheme-number->complex` coercion shown above, he would do:

```
(define (scheme-number->scheme-number n) n)
(define (complex->complex z) z)
(put-coercion 'scheme-number 'scheme-number
              scheme-number->scheme-number)
(put-coercion 'complex 'complex complex->complex)
```

a. With Louis's coercion procedures installed, what happens if `apply-generic` is called with two arguments of type `scheme-number` or two arguments of type `complex` for an operation that is not found in the table for those types? For example, assume that we've defined a generic exponentiation operation:

```
(define (exp x y) (apply-generic 'exp x y))
```

and have put a procedure for exponentiation in the Scheme-number package but not in any other package:

```
;; following added to Scheme-number package
(put 'exp '(scheme-number scheme-number)
     (lambda (x y) (tag (expt x y))))  ; using primitive expt
```

What happens if we call `exp` with two complex numbers as arguments?

b. Is Louis correct that something had to be done about coercion with arguments of the same type, or does `apply-generic` work correctly as is?

c. Modify `apply-generic` so that it doesn't try coercion if the two arguments have the same type.

Answer.

a. With Louis's coercion procedures installed, whenever `apply-generic` is called with two arguments of type `scheme-number` or two arguments of type `complex` for an operation that is not found in the table, it drops into an infinite loop, instead of reporting error explicitly. For example, when we call `exp` with two complex numbers z_1 and z_2 as been postulated above, the process generated during the evaluation turns out to be:

```
(exp z1 z2)
(apply-generic 'exp z1 z2)
; coerce the type of z1 to itself
(apply-generic 'exp z1 z2)
; coerce the type of z1 to itself
(apply-generic 'exp z1 z2)
; coerce the type of z1 to itself
...  ; infinite recursion of apply-generic
```

b. Louis is wrong. `Apply-generic` just works fine as originally to be. It is reasonable to regard the case where `apply-generic` is called with two arguments of the same type for an operation that is not found in the table as a special case in which there is no known way to coerce either type to each other type. This case had been addressed by the `else` clause in the `cond` predicate, where the program gives up and reports an error. Louis, however, gild the lily by keeping the program tied to coerce arguments of each type to their own type, which in turn makes the `apply-generic` procedure fail to address this case.

c. We can meet this requirement by making `apply-generic` report an error whenever it encounter two arguments of the same type but correspond to no operations in the table:

```
(define (apply-generic op . args)
```

```
(let ((type-tags (map type-tag args)))
  (let ((proc (get op type-tags)))
    (if proc
        (apply proc (map contents args))
        (if (= length 2)
            (let ((type1 (car type-tags))
                  (type2 (cadr type-tags))
                  (a1 (car args))
                  (a2 (cadr args)))
              (if (eq? type1 type2)
                  (error "No method for these types"
                         (list op type-tags))
                  (let ((t1->t2 (get-coercion type1 type2))
                        (t2->t1 (get-coercion type2 type1)))
                    (cond (t1->t2
                           (apply-generic op (t1->t2 a1) a2))
                          (t2->t1
                           (apply-generic op a1 (t2->t1 a2)))
                          (else
                           (error "No method for these types"
                                  (list op type-tags)))))))
            (error "No method for these types"
                   (list op type-tags)))))))
```