**Structures and Interpretation of Computer Program**

**Exercise Chapter 2.5 Name:** Wan Huzaifah bin Wan Azhar

**Exercise 2.5.2 Inadequacies of Hierarchies**



* If exp is called with two complex numbers, the procedure will fail.
  + This is because the exp procedure is defined in the scheme number package only.
  + When exp is called with two complex number, it will coerce to complex->complex
  + Since exp is not defined in complex packages, it will not know how to handle exp.
  + The only way to make this work is to coerce complex into scheme-number or implements exp in complex packages.
* Apply generic should work as it is.
* Add one more condition in cond at apply-generic
  + (cond (equal? type1 type2) (error-same-type op a1 a2))
  + Error-same-type is a procedure to warn user if there is a coercion of same type in the table.
  + The warning will notify the user if the attempted coercion is the same type and it should be removed from the table.
  + As apply-generic is work as intended, installing same type coercion and using it should be prevented.



Taken from <http://community.schemewiki.org/?sicp-ex-2.82>

(define (any-false? items)

(cond ((null? items) #f)

((not (car items)) #t)

(else (any-false? (cdr items)))))

(define (coerce type-tags args)

(define (iter tags)

(if (null? tags)

false

(let ((type-to (car tags)))

(let ((coercions

(map (lambda (type-from)

(if (eq? type-from type-to)

(lambda (x) x)

(get-coercion type-from type-to)))

type-tags)))

(if (any-false? coercions)

(iter (cdr tags))

(map (lambda (coercion arg) (coercion arg))

coercion

args))))))

(iter type-tags))

(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))

(let ((coerced-args (coerce type-tags args)))

(if coerced-args

(let ((coerced-type-tags (map type-tag coerced-args)))

(let ((new-proc (get op coerced-type-tags)))

(apply new-proc (map contents coerced-args))))

(error "No method for these types"

(list op type-tags))))))))

* Example of weakness of first-come coercion is with integer and real number.
  + Rather than coercing real number to integer, it is much easier and straightforward to coerce integer to real number.



(define (install-raise-package)

(define (raise-scheme z)

(make-rational z 1))

(define (raise-rational z)

(make-real (exact->inexact (/ (numerator z) (denumerator z)))))

(define (raise-real z)

(make-complex-from-real-imag z 0))

(define (raise-complex z)

#f)

(put 'raise 'scheme-number raise-scheme)

(put 'raise 'rational raise-rational)

(put 'raise 'real raise-real)

(put 'raise 'complex raise-complex)

'done)

(define (raise z) (apply-generic 'raise z))



* This assumes that highest level of type will return false if raised (in 3.), as it cannot be raised further.
* This will coerce both argument into same type, scheme-number and scheme-number, and complex with complex. It will cause issue like in 1. However, I’m ignoring it because the question explicitly asks me to coerce both argument into same type without making me solve the issue.

(define (raise-to tag z)

(cond

((equal? (type-tags z) tag)

z)

((equal? z #f)

(error "No possible raise: Maximum reached or invalid type raise"))

(else

(raise-to tag (raise z)))))

(define (find-higher-tag tag-list obj1 obj2)

(define (raise-then-compare obj tag-from tag-to)

(let ((new-tag (type-tags (raise obj))))

(cond

((equal? new-tag tag-to) #t) ;True means tag-from is lower than tag-to

((equal? new-tag #f) #f) ; Maximum reached assuming that the highest raise will return false

(else (raise-then-compare (raise obj) new-tag tag-to)))))

(let ((type1 (car tag-list))

(type2 (cadr tag-list)))

(cond ((equal? type1 type2)

type1)

((raise-then-compare obj1 type1 type2)

type2)

((raise-then-compare obj2 type2 type1)

type1)

(else (error "No highest tag")))))

(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 args) 2)

(let ((highest-tag (find-higher-tag type-tags (car args) (cadr args)))

(a1 (raise-to highest-tag (car args)))

(a2 (raise-to highest-tag (cadr args))))

(apply-generic op a1 a2))

(error "No method for these types"

(list op type-tags)))))))



* Project package

(define (install-project-package)

(define (project-scheme z)

#f)

(define (project-rational z)

(make-scheme-number (round (/ (numerator z) (denumerator z)))))

(define (project-real z)

(let

((fraction (inexact->exact (real-part z)))

(num (numerator fraction))

(denum (denumerator fraction)))

(make-rational num denum)))

(define (project-complex z)

(make-real (real-part z)))

(put 'project 'scheme-number project-scheme)

(put 'project 'rational project-rational)

(put 'project 'real project-real)

(put 'project 'complex project-complex)

'done)

(define (project z) (apply-generic 'project z))

* Drop package

(define (install-drop-package)

(define (can-drop? y)

(if (equal? (type-tag y) 'scheme-number)

#f

(let

((projected (project y))

(projected-raised (raise projected)))

(if (equ? projected-raised y)

#t

#f)))

)

(define (drop-generic z)

(if (can-drop? z)

(drop-generic (project z))

z))

(put 'drop 'scheme-number drop-generic)

(put 'drop 'rational drop-generic)

(put 'drop 'real drop-generic)

(put 'drop 'complex drop-generic)

'done)

(define (drop z) (apply-generic 'drop z))

* New apply-generic

(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 args) 2)

(let ((highest-tag (find-higher-tag type-tags (car args) (cadr args)))

(a1 (raise-to highest-tag (car args)))

(a2 (raise-to highest-tag (cadr args))))

(drop (apply-generic op a1 a2)))

(error "No method for these types"

(list op type-tags)))))))