

#### Exercise 4.9.

Many languages support a variety of iteration constructs, such as `do`, `for`, `while`, and `until`. In Scheme, iterative processes can be expressed in terms of ordinary procedure calls, so special iteration constructs provide no essential gain in computational power. On the other hand, such constructs are often convenient. Design some iteration constructs, give examples of their use, and show how to implement them as derived expressions.

#### Answer.

`Do` is an iteration construct that has the form<sup>1</sup>

```
(do ((<variable1> <init1> <step1>)
    (<variable2> <init2> <step2>)
    ...
    (<variablen> <initn> <stepn>))
    (<test> <expression1> <expression2> ... <expressionn>)
    <command> ...)
```

It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition is met, the loop exits after evaluating the `<expression>s`.

Do expressions are evaluated as follows: The `<init>` expressions are evaluated (in some unspecified order), the `<variable>s` are bound to fresh locations, the results of the `<init>` expressions are stored in the bindings of the `<variable>s`, and then the iteration phase begins.

Each iteration begins by evaluating `<test>`; if the result is false, then the `<command>` expressions are evaluated in order for effect, the `<step>` expressions are evaluated in some unspecified order, the `<variable>s` are bound to fresh locations, the results of the `<step>s` are stored in the bindings of the `<variable>s`, and the next iteration begins.

If `<test>` evaluates to a true value, then the `<expression>s` are evaluated from left to right and the value(s) of the last `<expression>` is(are) returned. If no `<expression>s` are present, then the value of the `do` expression is unspecified in standard Scheme. For example

```
(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
      (sum 0 (+ sum (car x))))
      ((null? x) sum)))
```

returns 25. We can reduce the problem of evaluating this `do` expression to the problem of evaluating the following expression involving `begin`, `define` and `if` expressions:

```
(let ((x '(1 3 5 7 9)))
  (begin (define (iter x sum)
          (if (null? x)
              sum
              (iter (cdr x) (+ sum (car x)))))
        (iter x 0)))
```

We include syntax procedures that extract the parts of a `do` expression, and a procedure `do->combination` that transform `do` expression into ordinary procedure calls.


```
(define (do? exp) (tagged-list? exp 'do))

(define (list-of-bindings exp) (cadr exp))

(define (do-termination exp) (caddr exp))

(define (do-command exp) (cdddr exp))
```

---

\*. Creative Commons  2014, Lawrence X. Amlord (颜世敏, aka 颜序).  
Email address: informlarry@gmail.com

1. A more specific description of `do` can be found in Kelsey 1998.

```

(define (list-of-variables bindings)
  (if (null? bindings)
      '()
      (cons (caar bindings)
            (list-of-variables (cdr bindings)))))

(define (list-of-inits bindings)
  (if (null? bindings)
      '()
      (cons (cadar bindings)
            (list-of-inits (cdr bindings)))))

(define (list-of-steps bindings)
  (if (null? bindings)
      '()
      (cons (caddar bindings)
            (list-of-steps (cdr bindings)))))

(define (termination-test termination)
  (car termination))

(define (termination-actions termination)
  (cdr termination))

(define (do->combination exp)
  (do-component->begin (do-bindings exp)
                      (do-termination exp)
                      (do-command exp)))

(define (do-component->begin bindings termination command)
  (if (or (null? bindings) (null? termination))
      'false
      (let ((vars (list-of-variables bindings))
            (inits (list-of-inits bindings))
            (steps (list-of-steps bindings))
            (test (termination-test termination))
            (exit-actions (termination-actions termination)))
        (let ((definition (list 'define
                                (cons 'iter vars)
                                (make-if test
                                        exit-actions
                                        (sequence->exp
                                         (cons command
                                               (cons 'iter steps)))))))
          (application (cons 'iter inits)))
          (sequence->exp (cons definition application))))))

```