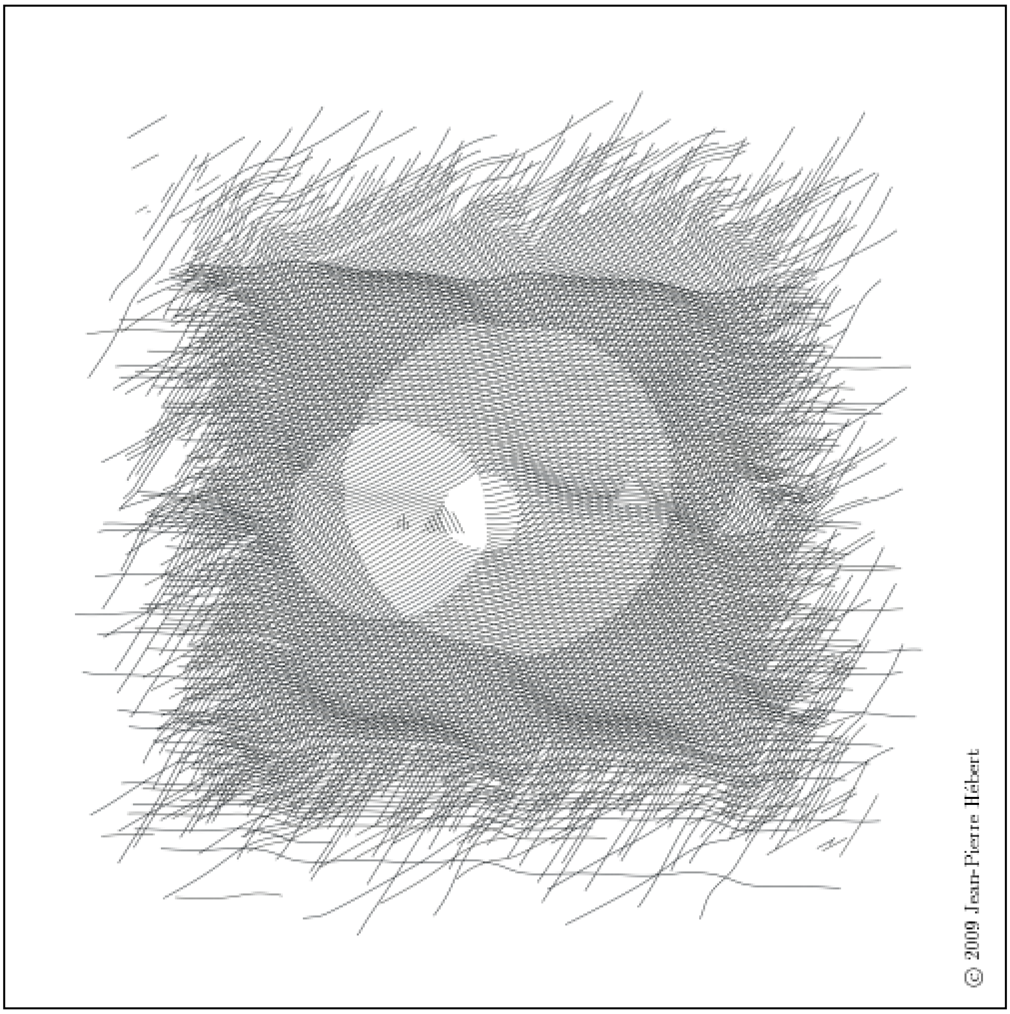

Chapter 4. Procedures and Variable Bindings

Procedures and variable bindings are the fundamental building blocks of Scheme programs. This chapter describes the small set of syntactic forms whose primary purpose is to create procedures and manipulate variable bindings. It begins with the two most fundamental building blocks of Scheme programs: variable references and `lambda` expressions, and continues with descriptions of the variable binding and assignment forms such as `define`, `letrec`, `let-values`, and `set!`.

Various other forms that bind or assign variables for which the binding or assignment is not the primary purpose (such as named `let`) are found in Chapter 5.

Section 4.1. Variable References

syntax: *`variable`*
returns: the value of *`variable`*

Any identifier appearing as an expression in a program is a variable if a visible variable binding for the identifier exists, e.g., the identifier appears within the scope of a binding created by `define`, `lambda`, `let`, or some other variable-binding construct.

```
list ⇒ #<procedure>
(define x 'a)
(list x x) ⇒ (a a)
(let ([x 'b])
  (list x x)) ⇒ (b b)
(let ([let 'let]) let) ⇒ let
```

It is a syntax violation for an identifier reference to appear within a `library` form or top-level program if it is not bound as a variable, keyword, record name, or other entity. Since the scope of the definitions in a `library`, top-level program, `lambda`, or other local body is the entire body, it is not necessary for the definition of a variable to appear before its first reference appears, as long as the reference is not actually evaluated until the definition has been completed. So, for example, the reference to `g` within the definition of `f` below

```
(define f
  (lambda (x)
    (g x)))
(define g
  (lambda (x)
    (+ x x)))
```

只要用到前准备好就行

is okay, but the reference to `g` in the definition of `q` below is not.

```
(define q (g 3))
(define g
  (lambda (x)
    (+ x x)))
```

Section 4.2. Lambda

syntax: `(lambda` *`formals`* *`body`*$_1$ *`body`*$_2$ `...)`
returns: a procedure
libraries: `(rnrs base)`, `(rnrs)`

The `lambda` syntactic form is used to create procedures. Any operation that creates a procedure or establishes local variable bindings is ultimately defined in terms of `lambda` or `case-lambda`.

The variables in *`formals`* are the formal parameters of the procedure, and the sequence of subforms *`body`*$_1$ *`body`*$_2$ `...` is its body.

The body may begin with a sequence of definitions, in which case the bindings created by the definitions are local to the body. If definitions are present, the keyword bindings are used and discarded while expanding the body, and the body is expanded into a `letrec*` expression formed from the variable definitions and the remaining expressions, as described on page 292. The remainder of this description of `lambda` assumes that this transformation has taken place, if necessary, so that the body is a sequence of expressions without definitions.

When the procedure is created, the bindings of all variables occurring free within the body, excluding the formal parameters, are retained with the procedure. Subsequently, whenever the procedure is applied to a sequence of actual parameters, the formal parameters are bound to the actual parameters, the retained bindings are restored, and the body is evaluated.

Upon application, the formal parameters defined by *`formals`* are bound to the actual parameters as follows.

- If *`formals`* is a proper list of variables, e.g., `(x y z)`, each variable is bound to the corresponding actual parameter. An exception with condition type `&assertion` is raised if too few or too many actual parameters are supplied.

  参数数量要一致哦

- If *`formals`* is a single variable (not in a list), e.g., `z`, it is bound to a list of the actual parameters.
- If *`formals`* is an improper list of variables terminated by a variable, e.g., `(x y . z)`, each variable but the last is bound to the corresponding actual parameter. The last variable is bound to a list of the remaining actual parameters. An exception with condition type `&assertion` is raised if too few actual parameters are supplied.

When the body is evaluated, the expressions in the body are evaluated in sequence, and the procedure returns the values of the last expression.

Procedures do not have a printed representation in the usual sense. Scheme systems print procedures in different ways; this book uses the notation `#<procedure>`.

```
(lambda (x) (+ x 3)) ⇒ #<procedure>
((lambda (x) (+ x 3)) 7) ⇒ 10
((lambda (x y) (* x (+ x y))) 7 13) ⇒ 140
((lambda (f x) (f x x)) + 11) ⇒ 22
((lambda () (+ 3 4))) ⇒ 7

((lambda (x . y) (list x y))
 28 37) ⇒ (28 (37))
((lambda (x . y) (list x y))
 28 37 47 28) ⇒ (28 (37 47 28))
((lambda (x y . z) (list x y z))
 1 2 3 4) ⇒ (1 2 (3 4))
((lambda x x) 7 13) ⇒ (7 13)
```

Section 4.3. Case-Lambda

A Scheme `lambda` expression always produces a procedure with a fixed number of arguments or with an indefinite number of arguments greater than or equal to a certain number. In particular,

```
(lambda (var1 ... varn) body1 body2 ...)
```

accepts exactly *n* arguments,

```
(lambda r body1 body2 ...)
```

accepts zero or more arguments, and

```
(lambda (var1 ... varn . r) body1 body2 ...)
```

accepts *n* or more arguments.

`lambda` cannot directly produce, however, a procedure that accepts, say, either two or three arguments. In particular, procedures that accept optional arguments are not supported directly by `lambda`. The latter form of `lambda` shown above can be used, in conjunction with length checks and compositions of `car` and `cdr`, to implement procedures with optional arguments, though at the cost of clarity and efficiency.

The `case-lambda` syntactic form directly supports procedures with optional arguments as well as procedures with fixed or indefinite numbers of arguments. `case-lambda` is based on the `lambda*` syntactic form introduced in the article "A New Approach to Procedures with Variable Arity" [11].

syntax: `(case-lambda` *`clause`* `...)`
returns: a procedure
libraries: `(rnrs control)`, `(rnrs)`

A `case-lambda` expression consists of a set of clauses, each resembling a `lambda` expression. Each *`clause`* has the form below.

```
[formals body1 body2 ...]
```

The formal parameters of a clause are defined by *`formals`* in the same manner as for a `lambda` expression. The number of arguments accepted by the procedure value of a `case-lambda` expression is determined by the numbers of arguments accepted by the individual clauses.

When a procedure created with `case-lambda` is invoked, the clauses are considered in order. The first clause that accepts the given number of actual parameters is selected, the formal parameters defined by its *`formals`* are bound to the corresponding actual parameters, and the body is evaluated as described for `lambda` above. If *`formals`* in a clause is a proper list of identifiers, then the clause accepts exactly as many actual parameters as there are formal parameters (identifiers) in *`formals`*. As with a `lambda` *`formals`*, a `case-lambda` clause *formals* may be a single identifier, in which case the clause accepts any number of arguments, or an improper list of identifiers terminated by an identifier, in which case the clause accepts any number of arguments greater than or equal to the number of formal parameters excluding the terminating identifier. If no clause accepts the number of actual parameters supplied, an exception with condition type `&assertion` is raised.

The following definition for `make-list` uses `case-lambda` to support an optional fill parameter.

```
(define make-list
  (case-lambda
    [(n) (make-list n #f)]
    [(n x)
     (do ([n n (- n 1)] [ls '() (cons x ls)])
         ((zero? n) ls))]))
```

想到：重载和模式匹配

The `substring` procedure may be extended with `case-lambda` to accept either no *end* index, in which case it defaults to the end of the string, or no *start* and *end* indices, in which case `substring` is equivalent to `string-copy`:

```
(define substring1
  (case-lambda
    [(s) (substring1 s 0 (string-length s))]
    [(s start) (substring1 s start (string-length s))]
    [(s start end) (substring s start end)]))
```

It is also possible to default the *start* index rather than the *end* index when only one index is supplied:

```
(define substring2
  (case-lambda
    [(s) (substring2 s 0 (string-length s))]
    [(s end) (substring2 s 0 end)]
    [(s start end) (substring s start end)]))
```

It is even possible to require that both or neither of the *start* and *end* indices be supplied, simply by leaving out the middle clause:

```
(define substring3
  (case-lambda
    [(s) (substring3 s 0 (string-length s))]
    [(s start end) (substring s start end)]))
```

Section 4.4. Local Binding

syntax: `(let ((`*var expr*`) ...)` *body*$_1$ *body*$_2$ `...)`
returns: the values of the final body expression
libraries: `(rnrs base)`, `(rnrs)`

`let` establishes local variable bindings. Each variable *var* is bound to the value of the corresponding expression *expr*. The body of the `let`, in which the variables are bound, is the sequence of subforms *body*$_1$ *body*$_2$ `...` and is processed and evaluated like a `lambda` body.

The forms `let`, `let*`, `letrec`, and `letrec*` (the others are described after `let`) are similar but serve slightly different purposes. With `let`, in contrast with `let*`, `letrec`, and `letrec*`, the expressions *expr* `...` are all outside the scope of the variables *var* `...`. Also, in contrast with `let*` and `letrec*`, no ordering is implied for the evaluation of the expressions *expr* `...`. They may be evaluated from left to right, from right to left, or in any other order at the discretion of the implementation. Use `let` whenever the values are independent of the variables and the order of evaluation is unimportant.

```
(let ([x (* 3.0 3.0)] [y (* 4.0 4.0)])
  (sqrt (+ x y))) ⇒ 5.0

(let ([x 'a] [y '(b c)])
```

let只是labmda调用的语法糖， 调用时参数求值顺序不定。

```
  (cons x y)) ⇒ (a b c)

(let ([x 0] [y 1])
  (let ([x y] [y x])
    (list x y))) ⇒ (1 0)
```

The following definition of `let` shows the typical derivation of `let` from `lambda`.

```
(define-syntax let
  (syntax-rules ()
    [(_ ((x e) ...) b1 b2 ...)
     ((lambda (x ...) b1 b2 ...) e ...)]))
```

Another form of `let`, *named* `let`, is described in Section 5.4, and a definition of the full `let` can be found on page 312.

syntax: `(let* ((`*`var expr`*`) ...)` *body*$_1$ *body*$_2$ `...)`
returns: the values of the final body expression
libraries: `(rnrs base)`, `(rnrs)`

`let*` is similar to `let` except that the expressions *`expr`* `...` are evaluated in sequence from left to right, and each of these expressions is within the scope of the variables to the left. Use `let*` when there is a linear dependency among the values or when the order of evaluation is important.

```
(let* ([x (* 5.0 5.0)]
       [y (- x (* 4.0 4.0))])
  (sqrt y)) ⇒ 3.0

(let ([x 0] [y 1])
  (let* ([x y] [y x])
    (list x y))) ⇒ (1 1)
```

Any `let*` expression may be converted to a set of nested `let` expressions. The following definition of `let*` demonstrates the typical transformation.

```
(define-syntax let*
  (syntax-rules ()
    [(_ () e1 e2 ...)
     (let () e1 e2 ...)]
    [(_ ((x1 v1) (x2 v2) ...) e1 e2 ...)
     (let ((x1 v1))
       (let* ((x2 v2) ...) e1 e2 ...))]))
```

syntax: `(letrec ((`*`var expr`*`) ...)` *body*$_1$ *body*$_2$ `...)`
returns: the values of the final body expression
libraries: `(rnrs base)`, `(rnrs)`

`letrec` is similar to `let` and `let*`, except that all of the expressions *`expr`* `...` are within the scope of all of the variables *`var`* `...`. `letrec` allows the definition of mutually recursive procedures.

```
(letrec ([sum (lambda (x)
                (if (zero? x)
                    0
                    (+ x (sum (- x 1)))))])
  (sum 5)) ⇒ 15
```

The order of evaluation of the expressions *`expr`* `...` is unspecified, so a program must not evaluate a reference to any of the variables bound by the `letrec` expression before all of the values have been computed. (Occurrence of a variable within a `lambda` expression does not count as a reference, unless the resulting procedure is applied before all of the values have been computed.) If this restriction is violated, an exception with condition type `&assertion` is raised.

An *`expr`* should not return more than once. That is, it should not return both normally and via the invocation of a continuation obtained during its evaluation, and it should not return twice via two invocations of such a continuation. Implementations are not required to detect a violation of this restriction, but if they do, an exception with condition type `&assertion` is raised.

Choose `letrec` over `let` or `let*` when there is a circular dependency among the variables and their values and when the order of evaluation is unimportant. Choose `letrec*` over `letrec` when there is a circular dependency and the bindings need to be evaluated from left to right.

A `letrec` expression of the form

```
(letrec ((var expr) ...) body1 body2 ...)
```

may be expressed in terms of `let` and `set!` as

```
(let ((var #f) ...)
  (let ((temp expr) ...)
    (set! var temp) ...
    (let () body1 body2 ...)))
```

where *temp* ... are fresh variables, i.e., ones that do not already appear in the `letrec` expression, one for each `(var expr)` pair. The outer `let` expression establishes the variable bindings. The initial value given each variable is unimportant, so any value suffices in place of `#f`. The bindings are established first so that *expr* ... may contain occurrences of the variables, i.e., so that the expressions are computed within the scope of the variables. The middle `let` evaluates the values and binds them to the temporary variables, and the `set!` expressions assign each variable to the corresponding value. The inner `let` is present in case the body contains internal definitions.

A definition of `letrec` that uses this transformation is shown on page 310.

This transformation does not enforce the restriction that the *expr* expressions must not evaluate any references of or assignments to the variables. More elaborate transformations that enforce this restriction and actually produce more efficient code are possible [31].

syntax: `(letrec* ((var expr) ...) body1 body2 ...)`
returns: the values of the final body expression
libraries: `(rnrs base)`, `(rnrs)`

`letrec*` is similar to `letrec`, except that `letrec*` evaluates *expr* ... in sequence from left to right. While programs must still not evaluate a reference to any *var* before the corresponding *expr* has been evaluated, references to *var* may be evaluated any time thereafter, including during the evaluation of the *expr* of any subsequent binding.

A `letrec*` expression of the form

```
(letrec* ((var expr) ...) body1 body2 ...)
```

may be expressed in terms of `let` and `set!` as

```
(let ((var #f) ...)
  (set! var expr) ...
  (let () body1 body2 ...))
```

The outer `let` expression creates the bindings, each assignment evaluates an expression and immediately sets the corresponding variable to its value, in sequence, and the inner let evaluates the body. `let` is used in the latter case rather than `begin` since the body may include internal definitions as well as expressions.

```
(letrec* ([sum (lambda (x)
                 (if (zero? x)
                     0
                     (+ x (sum (- x 1)))))]
          [f (lambda () (cons n n-sum))]
          [n 15]
```

```
          [n-sum (sum n)])
  (f)) ⇒ (15 . 120)

(letrec* ([f (lambda () (lambda () g))]
          [g (f)])
  (eq? (g) g)) ⇒ #t

(letrec* ([g (f)]
          [f (lambda () (lambda () g))])
  (eq? (g) g)) ⇒ exception: attempt to reference undefined variable f
```

Section 4.5. Multiple Values

syntax: `(let-values ((formals expr) ...) body1 body2 ...)`
syntax: `(let*-values ((formals expr) ...) body1 body2 ...)`
returns: the values of the final body expression
libraries: `(rnrs base)`, `(rnrs)`

`let-values` is a convenient way to receive multiple values and bind them to variables. It is structured like `let` but permits an arbitrary formals list (like `lambda`) on each left-hand side. `let*-values` is similar but performs the bindings in left-to-right order, as with `let*`. An exception with condition type `&assertion` is raised if the number of values returned by an *`expr`* is not appropriate for the corresponding *`formals`*, as described in the entry for `lambda` above. A definition of `let-values` is given on page 310.

```
(let-values ([(a b) (values 1 2)] [c (values 1 2 3)])
  (list a b c)) ⇒ (1 2 (1 2 3))

(let*-values ([(a b) (values 1 2)] [(a b) (values b a)])
  (list a b)) ⇒ (2 1)
```

Section 4.6. Variable Definitions

syntax: `(define var expr)`
syntax: `(define var)`
syntax: `(define (var0 var1 ...) body1 body2 ...)`
syntax: `(define (var0 . varr) body1 body2 ...)`
syntax: `(define (var0 var1 var2 ... . varr) body1 body2 ...)`
libraries: `(rnrs base)`, `(rnrs)`

In the first form, `define` creates a new binding of *`var`* to the value of *`expr`*. The *`expr`* should not return more than once. That is, it should not return both normally and via the invocation of a continuation obtained during its evaluation, and it should not return twice via two invocations of such a continuation. Implementations are not required to detect a violation of this restriction, but if they do, an exception with condition type `&assertion` is raised.

The second form is equivalent to `(define var unspecified)`, where *`unspecified`* is some unspecified value. The remaining are shorthand forms for binding variables to procedures; they are identical to the following definition in terms of `lambda`.

```
(define var
  (lambda formals
    body1 body2 ...))
```

where *`formals`* is `(var1 ...)`, var_r, or `(var1 var2 ... . varr)` for the third, fourth, and fifth `define` formats.

Definitions may appear at the front of a `library` body, anywhere among the forms of a top-level-program body, and at the front of a `lambda` or `case-lambda` body or the body of any form derived from `lambda`, e.g., `let`, or `letrec*`. Any body that begins with a sequence of definitions is transformed during macro expansion into a `letrec*` expression as described on page 292.

Syntax definitions may appear along with variable definitions wherever variable definitions may appear; see Chapter 8.

```
(define x 3)
x ⇒ 3

(define f
  (lambda (x y)
    (* (+ x y) 2)))
(f 5 4) ⇒ 18

(define (sum-of-squares x y)
  (+ (* x x) (* y y)))
(sum-of-squares 3 4) ⇒ 25

(define f
  (lambda (x)
    (+ x 1)))
(let ([x 2])
  (define f
    (lambda (y)
      (+ y x)))
  (f 3)) ⇒ 5
(f 3) ⇒ 4
```

A set of definitions may be grouped by enclosing them in a `begin` form. Definitions grouped in this manner may appear wherever ordinary variable and syntax definitions may appear. They are treated as if written separately, i.e., without the enclosing `begin` form. This feature allows syntactic extensions to expand into groups of definitions.

```
(define-syntax multi-define-syntax
  (syntax-rules ()
    [(_ (var expr) ...)
     (begin
       (define-syntax var expr)
       ...)]))
(let ()
  (define plus
    (lambda (x y)
        (if (zero? x)
            y
            (plus (sub1 x) (add1 y)))))
  (multi-define-syntax
    (add1 (syntax-rules () [(_ e) (+ e 1)]))
    (sub1 (syntax-rules () [(_ e) (- e 1)])))
  (plus 7 8)) ⇒ 15
```

Many implementations support an interactive "top level" in which variable and other definitions may be entered interactively or loaded from files. The behavior of these top-level definitions is outside the scope of the Revised^6 Report, but as long as top-level variables are defined before any references or assignments to them are evaluated, the behavior is consistent across most implementations. So, for example, the reference to `g` in the top-level definition of `f` below is okay if `g` is not already defined, and `g` is assumed to name a variable to be defined at some later point.

```
(define f
  (lambda (x)
    (g x)))
```

If this is then followed by a definition of `g` before `f` is evaluated, the assumption that `g` would be defined as a variable is proven correct, and a call to `f` works as expected.

```
(define g
  (lambda (x)
    (+ x x)))
(f 3) ⇒ 6
```

If `g` were defined instead as the keyword for a syntactic extension, the assumption that `g` would be bound as a variable is proven false, and if `f` is not redefined before it is invoked, the implementation is likely to raise an exception.

Section 4.7. Assignment

syntax: `(set! var expr)`
returns: unspecified
libraries: `(rnrs base)`, `(rnrs)`

赋值从一开始就是不好的，因此加了个叹号

`set!` does not establish a new binding for *var* but rather alters the value of an existing binding. It first evaluates *expr*, then assigns *var* to the value of *expr*. Any subsequent reference to *var* within the scope of the altered binding evaluates to the new value.

Assignments are not employed as frequently in Scheme as in most other languages, but they are useful for implementing state changes.

状态的改变，面向对象中用得比较多。

类似于cache的实现中用得比较多。

```
(define flip-flop
  (let ([state #f])
    (lambda ()
      (set! state (not state))
      state)))

(flip-flop) ⇒ #t
(flip-flop) ⇒ #f
(flip-flop) ⇒ #t
```

Assignments are also useful for caching values. The example below uses a technique called *memoization*, in which a procedure records the values associated with old input values so it need not recompute them, to implement a fast version of the otherwise exponential doubly recursive definition of the Fibonacci function (see page 69).

```
(define memoize
  (lambda (proc)
    (let ([cache '()])
      (lambda (x)
        (cond
          [(assq x cache) => cdr]
          [else
           (let ([ans (proc x)])
             (set! cache (cons (cons x ans) cache))
             ans)])))))

(define fibonacci
  (memoize
    (lambda (n)
      (if (< n 2)
          1
          (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))))

(fibonacci 100) ⇒ 573147844013817084101
```

R. Kent Dybvig / The Scheme Programming Language, Fourth Edition

ISBN 978-0-262-51298-5 / LOC QA76.73.S34D93