函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

http://blog.sina.com.cn/u/3266490431

第十一讲
元循环求值器

元循环求值器

●用 Scheme 做一个 Scheme 求值器，而后在已有的Scheme解释器的支持下运行它，接受一段scheme程序作为输入，输出该程序运行的结果

●用一种语言实现其自身的求值器，称为元循环（meta-circular）

●scheme程序由表达式构成，表达式求值也是一些符号操作，Scheme 和其他 Lisp 方言都特别 适合做这种操作

求值的环境模型

●求值过程的核心步骤(复习：gw_sicp_07.ppt 求值的环境模型)

1）求值组合式（非特殊形式的复合表达式）时，先求值组合式的各子表达式，而后把运算符子表达式的值作用于运算对象子表达式的值

2）把复合过程应用于实参，是在一个新环境里对该过程的过程体进行求值。新环境里包含形参到实参的约束，新环境的外围环境指针指向复合过程所对应的过程对象里的环境。

求值的环境模型

●求值过程的核心步骤(复习：gw_sicp_07.ppt 求值的环境模型)

1）求值组合式（非特殊形式的复合表达式）时，先求值组合式的各子表达式，而后把运算符子表达式的值作用于运算对象子表达式的值

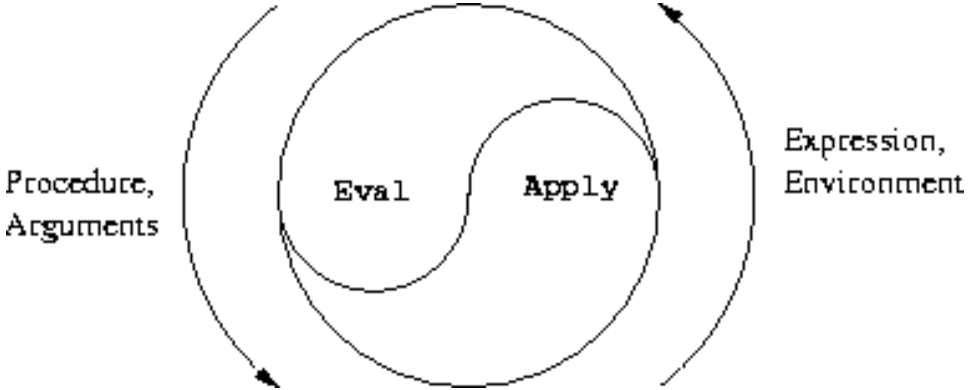
2）把复合过程应用于实参，是在一个新环境里对该过程的过程体进行求值。新环境里包含形参到实参的约束，新环境的外围环境指针指向复合过程所对应的过程对象里的环境。

●两个求值步骤都可能递归（自己递归或相互递归。求子表达式的值可能要应用复合过程，过程体本身通常又是组合式），直到遇到

1)符号（直接到环境里取值）
2)基本过程,如+,-，map（直接调用基本过程的代码）
3)本身就是值的表达式（如数，直接用其本身）

求值的核心过程eval和apply

- eval 负责对表达式分析和求值，apply 负责过程应用。二者相互递归调用，eval还递归调用自身



课本上apply的实现中调用了所谓基本过程“apply-in-underlying-scheme”来完成过程的应用。但由于scheme并无“apply-in-underlying-scheme”，scheme有基本过程apply，因此后文我们将自己编写的“apply”更名为“my-apply”。**my-apply必须调用sheme的基本过程apply方能实现**。

求值的核心过程eval

- eval 以一个表达式 exp 和一个环境 env 为参数，根据exp的不同情况分别求值：

1）基本表达式:
  - 自求值表达式（如数等）：直接返回其本身
  - 变量：从环境中找出它的当前值

2）特殊形式：
  - 单引号表达式（如 '(1 2 3)）：返回引号后面的表达式
  - 变量赋值或定义：递归调用eval去计算出需要关联于该变量的新值。然后需要修改环境，建立或修改该变量的约束
  - if 表达式：求值条件部分，而后根据情况求值相应子表达式
  - lambda 表达式：建立过程对象，包装过程的参数表、体、和环境
  - begin 表达式：按顺序求值其中的各个表达式
  - cond 表达式：将其变换为一系列 if 而后求值

3）组合式（过程应用）：

递归地求值运算符部分和运算对象部分，然后将得到的过程和参数交给my-apply,由其完成过程的执行。

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp);自求值表达式
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp));单引号表达式
        ((assignment? exp) (eval-assignment exp env));赋值语句
        ((definition? exp) (eval-definition exp env));特殊形式define
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                        env));生成过程对象
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env));cond转换为if
        ((application? exp);除了上面各种情况之外的，都认为是函数调用表达式
         (my-apply (eval (operator exp) env)
                   (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

求值的核心过程eval

eval的实现没有依赖于具体的语言形式。

比如，赋值语句是什么形式的，变量是什么样的，begin是什么样的，lambda表达式是什么样的，这些在 eval中都没有规定。只需更改 **variable?, assignment? , lambda?**等的实现，**eval**就能用于不同形式的语言。

如果使用“数据导向”的方法编写 eval，则更容易添加新的表达式形式。

被eval调用的函数 -- 分支判断函数

```
(define (self-evaluating? exp)
  (cond ((number? exp) true) ;number?是scheme基本过程
        ((string? exp) true) ;string?是scheme基本过程
        (else false)))

(define (variable? exp) (symbol? exp)) ;symbol?是scheme基本过程

(define (quoted? exp)
  (tagged-list? exp 'quote))
;单引号开头的表达式会被scheme自动转换成 (quote ...)列表形式

(define (text-of-quotation exp) (cadr exp))

(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
```

被eval调用的函数-- 分支判断函数

- 赋值表达式形如(set! x y)

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))
```

被eval调用的函数-- 分支判断函数

- define有两种形式：

```
(define <var> <value>)
(define (<var> <parameter1> ... <parametern>)    <body>)
```

第二种形式等价于:

```
(define <var>  (lambda (<parameter1> ... <parametern>)    <body>))
```

```
(define (definition? exp)
  (tagged-list? exp 'define));exp形如(define ....)
(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)  ;针对第一种形式
      (caadr exp)));针对第二种形式，此时变量名就是函数名
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp);针对第一种形式
      (make-lambda (cdadr exp)    ; formal parameters
                   (cddr exp))))  ; body
```

被eval调用的函数-- 分支判断函数

- lambda表达式形如 : (lambda (x y) (* x y) (+ x y))

```
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp)) ;body可能是个表达式序列
```

- definition-value中调用的 make-lambda:

```
(define (make-lambda parameters body) ;构造一个lambda表达式
  (cons 'lambda (cons parameters body)))
```

被eval调用的函数-- 分支判断函数

- if表达式形如 : (if (> a 2) (* a 3) (+ a 4))

```
(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))
```

被eval调用的函数-- 分支判断函数

- begin表达式形如 : (begin (* x 3) (+ x 6) ....)

```
(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions exp) (cdr exp))

;下面seq是一个列表，每个元素都是exp
(define (last-exp? seq) (null? (cdr seq)));判断seq里是否只有一个表达式
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))

(define (sequence->exp seq);把表达式列表变成一个表达式
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
(define (make-begin seq) (cons 'begin seq))
```

被eval调用的函数-- 分支判断函数

- 对 cond 的处理是将其转换成if

```
(cond ((> x 0) x)
      ((= x 0) (display 'zero) 0)
      (else (- x)))
```

➔

```
;嵌套if
(if (> x 0)
    x
    (if (= x 0)
        (begin (display 'zero)
               0)
        (- x)))
```

被eval调用的函数-- 分支判断函数

- 对 cond 的处理是将其转换成if

```
(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp)) ;返回所有分支的列表
;以下clause是一个条件分支，如 ((> x 3) (+ x 3) (* x 3))
(define (cond-predicate clause) (car clause))
(define (cond-actions clause) (cdr clause))
(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))


(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))
```

cond->if

```
;clauses是一个列表，每个元素是一个分支，元素形如: ((> x 3) (+ x 3) (* x 3))
(define (expand-clauses clauses)
  (if (null? clauses)
      'false                                ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))

(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))
```

被eval调用的函数-- 分支判断函数

- 非前述所有情况的表达式，被认为是函数调用表达式

```
(define (application? exp) (pair? exp))
;exp是函数调用表达式的前提下：
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
;下面ops是操作数的列表
(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))
```

被eval调用的函数-- 分支处理函数

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp);自求值表达式
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp));单引号表达式
        ((assignment? exp) (eval-assignment exp env));赋值语句
        ((definition? exp) (eval-definition exp env));特殊形式define
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                        env));生成过程对象
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env));cond转换为if
        ((application? exp);除了上面各种情况之外的，都认为是函数调用表达式
         (my-apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

被eval调用的函数-- 分支处理函数

- ((assignment? exp) (eval-assignment exp env));赋值语句

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)
```

- ((definition? exp) (eval-definition exp env));特殊形式define

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
```

- ((if? exp) (eval-if exp env))

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

被eval调用的函数-- 分支处理函数

- ((lambda? exp)
  (make-procedure (lambda-parameters exp)
  (lambda-body exp)
  env));生成过程对象

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
;过程对象是一个列表，包含参数和函数体。
;parameters是一个列表，元素就是参数的名字，形如(x y)。
;body是函数体，形如：(* x y)

(define (compound-procedure? p)
  (tagged-list? p 'procedure))
;过程对象形如： '(procedure (x y) (* x y) env) env是指向环境的指针
(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))
```

被eval调用的函数-- 分支处理函数

- ((begin? exp)
  (eval-sequence (begin-actions exp) env))

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

被eval调用的函数-- 分支处理函数

- ((variable? exp) (lookup-variable-value exp env))

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)));到外围环境继续找
            ((eq? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

框架和环境相关函数

```
(define (make-frame variables values)
  (cons variables values));框架形如 ((x y z) 1 2 3)

(define (frame-variables frame) (car frame))

(define (frame-values frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))

(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())
```

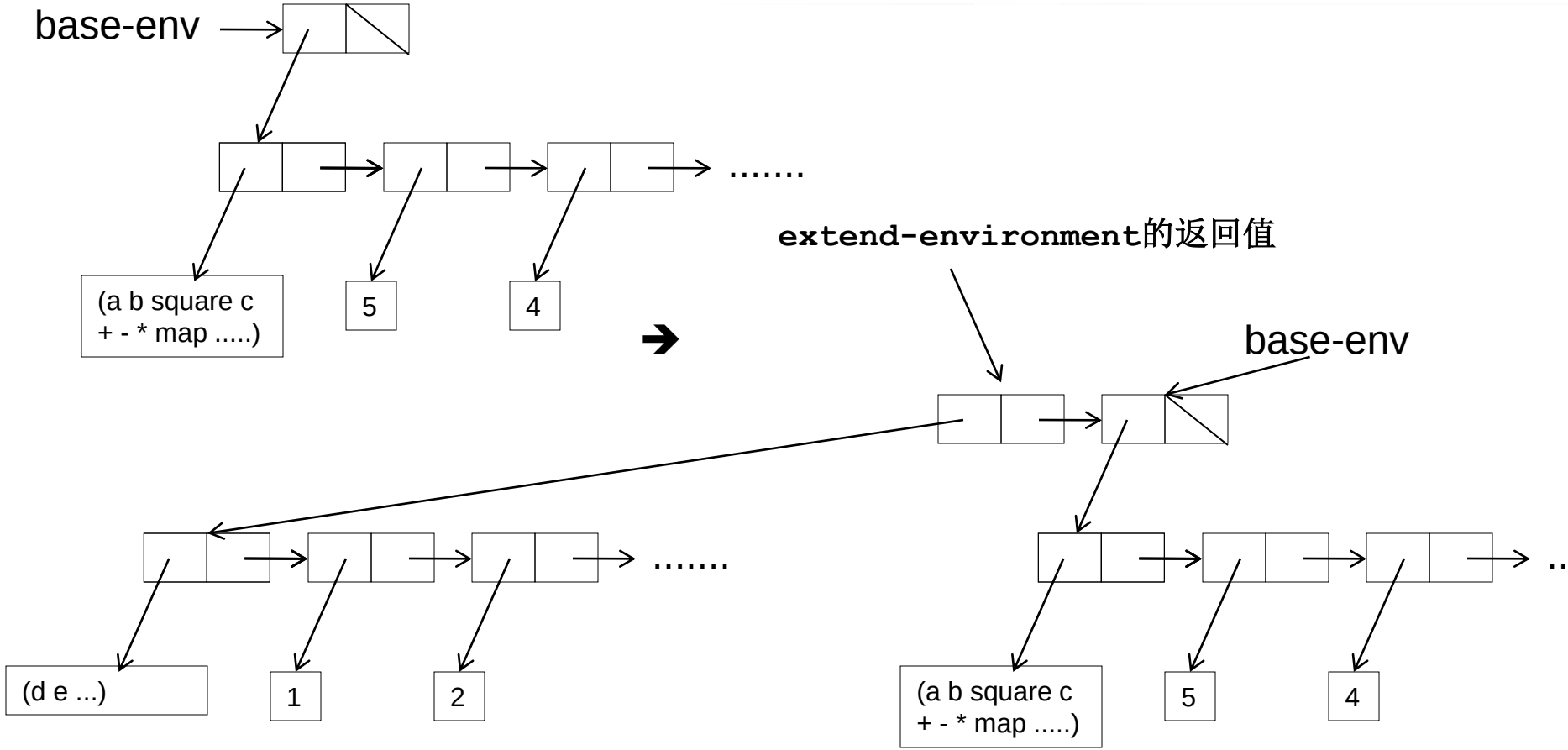
环境是框架的列表，形如((x y z) 1 2 3) ((a b c) 6 7 8))

框架和环境相关函数

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```

;在 base-env前面添加一个框架，形成一个新的环境。新的环境的cdr就是其外围环境指针，即base-env

框架和环境相关函数


base-env
.....
(a b square c
+ - * map .....)
5
4
extend-environment的返回值
base-env
.....
(d e ...)
1
2
(a b square c
+ - * map .....)
5
4

框架和环境相关函数

```
(define (set-variable-value! var val env);仅被eval-assignment调用
  (define (env-loop env)
    (define (scan vars vals) ;frame形如:((a b c) 1 2 3)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)
```

框架和环境相关函数

```
(define (define-variable! var val env);仅被eval-definition 调用
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars);如果变量不存在，就加到env的第一个 frame里面
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

```
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
```

框架和环境相关函数

```
define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))

(define glb-env (setup-environment)) ;初始的全局环境
```

在空的环境里，添加一个**frame**，里面包含预定义过程的约束，然后再加进去对 **true**和**false**的约束

过程相关函数

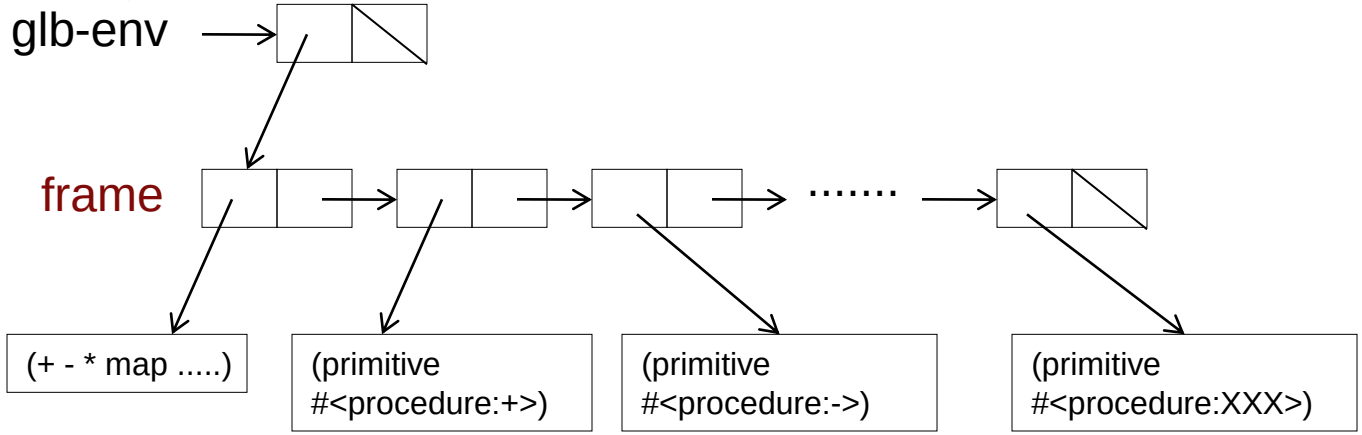
```
(define primitive-procedures ;预定义过程列表。预定义过程必须和scheme基本过程对应吗?
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        <more primitives>
        ))
(define (primitive-procedure-names) ;预定义过程名字列表
  (map car
       primitive-procedures))
(define (primitive-procedure-objects) ;生成预定义过程的函数对象列表
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))
;预定义过程的函数对象形如 (primitive  #<procedure:+>),不需要环境指针
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))
```

“环境”的结构详解

程序开始运行时的 glb-env：

((false true + - * map .....)
#f #t
(primitive #<procedure:+>) (primitive #<procedure:->)
(primitive #<procedure:*>) (primitive #<procedure:map>) .....))
;红色括号内部为 一个 frame



define-variable!

```
(define (define-variable! var val env);仅被eval-definition 调用
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars);如果变量不存在，就加到env的第一个 frame里面
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

```
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
```

核心函数 my-apply

```
(define (my-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply (primitive-implementation procedure) arguments))
;(primitive-implementation proc) 返回形如:  #<procedure:car> ,
#<procedure:my-square>之类的东西(如果my-square被定义成primitive的话)
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-enviroment procedure))))
;每执行一次函数调用，哪怕是递归的，都要新建一个环境,记录本次函数调用的参数的值
        (else
         (error "unkonwn procedure type -- APPLY" procedure))))
```

procedure 形如： (procedure (x y) (* x y) env) env是指向环境的指针
或 (primitive #<procedure:+>)

被eval调用的函数-- 分支处理函数

- ((lambda? exp)
  (make-procedure (lambda-parameters exp)
  (lambda-body exp)
  env));生成过程对象

- 过程对象相关函数：

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
;过程对象是一个列表，包含参数和函数体。
;parameters是一个列表，元素就是参数的名字，形如(x y)。
;body是函数体，形如：(* x y)

(define (compound-procedure? p)
  (tagged-list? p 'procedure))
;过程对象形如： '(procedure (x y) (* x y) env) env是指向环境的指针
(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))
```

过程相关函数

```
(define primitive-procedures ;预定义过程列表。预定义过程必须和scheme基本过程对应吗?
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        (list '+ +)
        (list 'my-square my-square)> ; 是否可行？如何可行？
        ))
(define (primitive-procedure-names) ;预定义过程名字列表
  (map car
       primitive-procedures))
(define (primitive-procedure-objects) ;生成预定义过程的函数对象列表
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))
;预定义过程的函数对象形如 (primitive  #<procedure:+>),不需要环境指针
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))
(define (primitive-implementation proc) (cadr proc))
```

测试元循环求值器

```
(define glb-env (setup-environment)) ;初始的全局环境
(display glb-env)
=>
{{{false true car cdr cons null? + * - / < > = my-square}
#f #t
{primitive #<procedure:car>} {primitive #<procedure:cdr>}
{primitive #<procedure:cons>} {primitive #<procedure:null?>}
{primitive #<procedure:+>} {primitive #<procedure:*>}
{primitive #<procedure:->} {primitive #<procedure:/>}
{primitive #<procedure:<>} {primitive #<procedure:>>}
{primitive #<procedure:=>}
{primitive #<procedure:my-square>}}}
```

测试元循环求值器

```
(eval '(define test1 (lambda (x y) (+ x y))) glb-env)
(display glb-env)
```

```
'(define test1 (lambda (x y) (+ x y)))
```

首先被scheme解释器替换成

```
(quote (define test1 (lambda (x y) (+ x y))))
```

```
=> ?
```

测试元循环求值器

```
(eval '(define test1 (lambda (x y) (+ x y))) glb-env)
(display glb-env)
=> ?
#0={{{test1 false true car cdr cons null? + * - / < > =
my-square}
(procedure (x y) ((+ x y)) #0#)
#f #t
{primitive #<procedure:car>} {primitive #<procedure:cdr>}
{primitive #<procedure:cons>} {primitive #<procedure:null?>}
{primitive #<procedure:+>} {primitive #<procedure:*>}
{primitive #<procedure:->} {primitive #<procedure:/>}
{primitive #<procedure:<>} {primitive #<procedure:>>}
{primitive #<procedure:=>}
{primitive #<procedure:my-square>}}}
```

测试元循环求值器

```
(define bank '(define (make-withdraw balance)
  (lambda (amount)
    (if (> balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))))
(eval bank glb-env)
(display glb-env)
=>?
```

测试元循环求值器

```
#0={{{make-withdraw test1 false true car cdr cons null? + * -
/ < > = my-square}
(procedure (balance) ((lambda (amount) (if (> balance amount)
(begin (set! balance (- balance amount)) balance)
Insufficient funds))) #0#)
(procedure (x y) ((+ x y)) #0#)
#f #t
{primitive #<procedure:car>} {primitive #<procedure:cdr>}
{primitive #<procedure:cons>} {primitive #<procedure:null?>}
{primitive #<procedure:+>} {primitive #<procedure:*>}
{primitive #<procedure:->} {primitive #<procedure:/>}
{primitive #<procedure:<>} {primitive #<procedure:>>}
{primitive #<procedure:=>}
{primitive #<procedure:my-square>}}}
```

测试元循环求值器

```
(eval '(define W1 (make-withdraw 100)) glb-env)
(display glb-env)
=> ?
```

测试元循环求值器

```
(eval '(define W1 (make-withdraw 100)) glb-env)
(display glb-env)
=>
#0={{{W1 make-withdraw test1 false true car cdr cons null? + * -
/ < > = my-square}
(procedure #1=(amount) #2=((if (> balance amount) (begin (set!
balance (- balance amount)) balance) Insufficient funds))
{{{balance} 100} . #0#})
(procedure (balance) ((lambda #1# . #2#)) #0#)
(procedure (x y) ((+ x y)) #0#)
#f #t {primitive #<procedure:car>}
{primitive #<procedure:cdr>} {primitive #<procedure:cons>}
{primitive #<procedure:null?>} {primitive #<procedure:+>}
{primitive #<procedure:*>} {primitive #<procedure:->}
.........
}}
```

测试元循环求值器

```
(eval '(W1 70) glb-env)
(display glb-env)
=> ?
```

测试元循环求值器

```
(eval '(define W1 (make-withdraw 100)) glb-env)
(display glb-env)
=>
#0={{{W1 make-withdraw test1 false true car cdr cons null? + * -
/ < > = my-square}
(procedure #1=(amount) #2=((if (> balance amount) (begin (set!
balance (- balance amount)) balance) Insufficient funds))
{{{balance} 30} . #0#})
(procedure (balance) ((lambda #1# . #2#)) #0#)
(procedure (x y) ((+ x y)) #0#)
#f #t {primitive #<procedure:car>}
{primitive #<procedure:cdr>} {primitive #<procedure:cons>}
{primitive #<procedure:null?>} {primitive #<procedure:+>}
{primitive #<procedure:*>} {primitive #<procedure:->}
.........
}}
```

元循环求值器执行过程分析

在关键函数中增加输出，得以下结果:

```
(define glb-env (setup-environment))
-------in extend-environment:vars and vals :{car cdr cons null? + * - / < > = my-square} vals: {{primitive #<procedure:car>} {primitive #<procedure:cdr>} {primitive #<procedure:cons>} {primitive #<procedure:null?>} {primitive #<procedure:+>} {primitive #<procedure:*>} {primitive #<procedure:->} {primitive #<procedure:/>} {primitive #<procedure:<>} {primitive #<procedure:>>} {primitive #<procedure:=>} {primitive #<procedure:my-square>}}
-------in define-variable! var =true  val=#t
-------in define-variable! var =false  val=#f
(eval '(define test1 (lambda (x y) (+ x y))) glb-env)
-------in eval, exp=:(define test1 (lambda (x y) (+ x y)))
-------in eval-definition,exp = (define test1 (lambda (x y) (+ x y)))
-------in eval, exp=:(lambda (x y) (+ x y))
-------in define-variable! var =test1  val=(procedure (x y) ((+ x y)) {{{false true car cdr cons null? + * - / < > = my-square} #f #t {primitive #<procedure:car>} {primitive #<procedure:cdr>} {primitive #<procedure:cons>} {primitive #<procedure:null?>} {primitive #<procedure:+>} {primitive #<procedure:*>} {primitive #<procedure:->} {primitive #<procedure:/>} {primitive #<procedure:<>} {primitive #<procedure:>>} {primitive #<procedure:=>} {primitive #<procedure:my-square>}}})
```

元循环求值器执行过程分析

```
(define bank '(define (make-withdraw balance)
  (lambda (amount)
    (if (> balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))))
(eval bank glb-env)
```

-------in eval, exp=:(define (make-withdraw balance) (lambda (amount) (if (> balance amount) (begin (set! balance (- balance amount)) balance) Insufficient funds)))

-------in eval-definition,exp = (define (make-withdraw balance) (lambda (amount) (if (> balance amount) (begin (set! balance (- balance amount)) balance) Insufficient funds)))

-------in eval, exp=:(lambda (balance) (lambda (amount) (if (> balance amount) (begin (set! balance (- balance amount)) balance) Insufficient funds)))

-------in define-variable! var =make-withdraw val=(procedure (balance) ((lambda (amount) (if (> balance amount) (begin (set! balance (- balance amount)) balance) Insufficient funds))) #0={{{test1 false true car cdr cons null? + * - / < > = my-square} (procedure (x y) ((+ x y)) #0#) #f #t {primitive #<procedure:car>} {primitive #<procedure:cdr>} {primitive #<procedure:cons>} {primitive #<procedure:null?>} {primitive #<procedure:+>} {primitive #<procedure:*>} {primitive #<procedure:->} {primitive #<procedure:/>} {primitive #<procedure:<>} {primitive #<procedure:>>} {primitive #<procedure:=>} {primitive #<procedure:my-square>}}})

元循环求值器执行过程分析

```
(eval '(define W1 (make-withdraw 100)) glb-env)
```

-------in eval, exp=:(define W1 (make-withdraw 100))
-------in eval-definition,exp = (define W1 (make-withdraw 100))
-------in eval, exp=:(make-withdraw 100)
-------in eval, exp=:make-withdraw
-------in lookup-variable-value:make-withdraw
-------in eval, exp=:100
-------in my-apply, procedure = (procedure (balance) ((lambda (amount) (if (> balance amount) (begin (set! balance (- balance amount)) balance) Insufficient funds))) glb-env) argumets= (100)
-------in extend-environment:vars and vals :{balance} vals: {100} ;建立E1
-------in eval-sequence, exps= ((lambda (amount) (if (> balance amount) (begin (set! balance (- balance amount)) balance) Insufficient funds)))
-------in eval, exp=:(lambda (amount) (if (> balance amount) (begin (set! balance (- balance amount)) balance) Insufficient funds))
-------in define-variable! var =W1 val=(procedure #0=(amount) #1=((if (> balance amount) (begin (set! balance (- balance amount)) balance) Insufficient funds)) {{{balance} 100} . #2={{{make-withdraw test1 false true car cdr cons null? + * - / < > = my-square} (procedure (balance) ((lambda #0# . #1#)) #2#) (procedure (x y) ((+ x y)) #2#) #f #t {primitive #<procedure:car>} {primitive #<procedure:cdr>} {primitive #<procedure:cons>} {primitive #<procedure:null?>} {primitive #<procedure:+>} {primitive #<procedure:*>} {primitive #<procedure:->} {primitive #<procedure:/>} {primitive #<procedure:<>} {primitive #<procedure:>>} {primitive #<procedure:=>} {primitive #<procedure:my-square>}}}}}

元循环求值器执行过程分析

```
(eval '(W1 70) glb-env)
-------in eval, exp=:(W1 70)
-------in eval, exp=:W1
-------in lookup-variable-value:W1
-------in eval, exp=:70
-------in my-apply, procedure = (procedure (amount) ((if (> balance amount) (begin (set! balance (- balance
amount)) balance) Insufficient funds)) E1)  argumets= (70)
-------in extend-environment:vars and vals :{amount} vals: {70}
-------in eval-sequence, exps= ((if (> balance amount) (begin (set! balance (- balance amount)) balance)
Insufficient funds))
-------in eval, exp=:(if (> balance amount) (begin (set! balance (- balance amount)) balance) Insufficient funds)
-------in eval, exp=:(> balance amount)
-------in eval, exp=:>
-------in lookup-variable-value:>
-------in eval, exp=:balance
-------in lookup-variable-value:balance
-------in eval, exp=:amount
-------in lookup-variable-value:amount
-------in my-apply,procedure = {primitive #<procedure:>>}  argumets= (100 70)
-------in eval, exp=:(begin (set! balance (- balance amount)) balance)
-------in eval-sequence, exps= ((set! balance (- balance amount)) balance)
```

元循环求值器执行过程分析

-------in eval, exp=:(set! balance (- balance amount))
-------in eval, exp=:(- balance amount)
-------in eval, exp=:-
-------in lookup-variable-value:-
-------in eval, exp=:balance
-------in lookup-variable-value:balance
-------in eval, exp=:amount
-------in lookup-variable-value:amount
-------in my-apply,procedure = {primitive #<procedure:->} argumets= (100 70)
-------in eval-sequence, exps= (balance)
-------in eval, exp=:balance
-------in lookup-variable-value:balance

30

用求值器处理键盘输入的sheme程序

```
(define input-prompt ";;;M-Eval input:")
(define output-prompt ";;;M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read))) (let ((output (eval input glb-env)))
;read 每次读取一个完整的表达式。输入 'x  , read返回的是 (quote x)
;eval 的返回值给output, input 就是本次read进来的表达式
       (announce-output output-prompt)
       (user-print output)))
  (driver-loop))

(define (prompt-for-input string)
   (newline) (newline) (display string) (newline))

(define (announce-output string)
   (newline) (display string) (newline))
```

用求值器处理键盘输入的sheme程序

```
(define (user-print object)
   (if (compound-procedure? object)
       (display (list 'compound-procedure
                      (procedure-parameters object)
                      (procedure-body object)
                      '<procedure-env>))
       (display object)))
```

```
(driver-loop)
```
;开始等待用户输入。输入一个表达式就求值它并输出其值，然后等待输入下一个表达式。各输入的表达式之间是有关联的。

用求值器处理键盘输入的sheme程序

```
输入:   (define x 5)
输出:   ;;;M-Eval value:  'ok
输入:    (define (test x ) (* x x))
输出:   ;;;M-Eval value:  'ok
输入:    (test 100)
输出:   ;;;M-Eval value:  10000
输入:    test
输出:   ;;;M-Eval value:
        (compound-procedure {x} ((* x x)) <procedure-env>)
输入:    (test 20) (test 30)
输出:   ;;;M-Eval value: 400
        ;;;M-Eval value: 900
```

用求值器处理键盘输入的sheme程序

处理(test 30)的过程：

```
-------in eval, exp=:(test 30)
-------in eval, exp=:test
-------in lookup-variable-value:test
-------in eval, exp=:30
-------in my-apply,exp = #<procedure:exp>
-------in extend-environment:vars and vals :{x} vals: {30}
-------in eval-sequence, exps= ((* x x))
-------in eval, exp=:(* x x)
-------in eval, exp=:*
-------in lookup-variable-value:*
-------in eval, exp=:x
-------in lookup-variable-value:x
-------in eval, exp=:x
-------in lookup-variable-value:x
-------in my-apply,exp = #<procedure:exp>
```

将数据作为程序

eval是scheme的基本过程。可以在程序中直接使用，用来对一段以数据形式存在的程序求值

```
(require r5rs)
(define env (scheme-report-environment  5))
; scheme-report-environment是包含基本过程的scheme环境

(eval '(* 5 5) env)  ;=> 25
(eval (cons '* (list 5 5)) env)  ;=>25
```

内部定义

- 分析下面包含内部定义的程序的解释执行过程:

```
(define inner-func '(define (f x)
  (define (g y)
    (k y))
  (define (k z)
    (+ z 1))
  (* (g x) x)))

(eval inner-func glb-env)
(eval '(f 5) glb-env)
```

```
(define inner-func '(define (f x)
  (define (g y)
    (k y))
  (define (k z)
    (+ z 1))
  (* (g x) x)))
(eval inner-func glb-env)
```

```
in eval, exp=:(define (f x) (define (g y) (k y)) (define (k z) (+ z 1)) (* (g x)
x))
in eval-definition,exp = (define (f x) (define (g y) (k y)) (define (k z) (+ z 1))
(* (g x) x))
in eval, exp=:(lambda (x) (define (g y) (k y)) (define (k z) (+ z 1)) (* (g x) x))
in define-variable! var =f  val=(procedure (x) ((define (g y) (k y)) (define (k z)
(+ z 1)) (* (g x) x)) glb-env)
```

把 f 及其值加到了 glb-env里面

```
(eval '(f 5) glb-env)
in eval, exp=:(f 5) env=glb-env
in eval, exp=:f env=glb-env
in lookup-variable-value:f
in eval, exp=:5
in my-apply,procedure = (procedure (x) ((define (g y) (k y)) (define (k z)
(+ z 1)) (* (g x) x)) glb-env) argumets= (5)
in extend-environment:vars and vals :{x} vals: {5} 新建一个环境E1，其外围环境
是 glb-env。E1:  (((x) 5 glb-env))
in eval-sequence, exps= ((define (g y) (k y)) (define (k z) (+ z 1)) (* (g
x) x))  env=E1
in eval, exp=:(define (g y) (k y)) env=E1
in eval-definition,exp = (define (g y) (k y)) env=E1
in eval, exp=:(lambda (y) (k y)) env=E1
in define-variable! var =g  val=(procedure (y) ((k y)) E1) 把g加到E1
in eval-sequence, exps= ((define (k z) (+ z 1)) (* (g x) x)) env=E1
in eval, exp=:(define (k z) (+ z 1)) env=E1
in eval-definition,exp = (define (k z) (+ z 1)) env=E1
in eval, exp=:(lambda (z) (+ z 1)) env=E1
in define-variable! var =k  val=(procedure (z) ((+ z 1)) E1) 把 k 加到E1
in eval-sequence, exps= ((* (g x) x))  env=E1
```

```
in eval, exp=:(* (g x) x) env=E1
in eval, exp=:* env=E1
in lookup-variable-value:* env=E1
in eval, exp=:(g x) env=E1
in eval, exp=:g env=E1
in lookup-variable-value:g env=E1
in eval, exp=:x env=E1
in lookup-variable-value:x env=E1, 找到 x=5
in my-apply,procedure = (procedure (y) ((k y)) E1) argumets= (5)
in extend-environment:vars and vals :{y} vals: {5}  新建一个环境E2，其外围环境
是E1 E2:  (((y) 5 E1))
in eval-sequence, exps= ((k y)) env=E2
in eval, exp=:(k y) env=E2
in eval, exp=:k env=E2
in lookup-variable-value:k env=E2
in eval, exp=:y env=E2
in lookup-variable-value:y env=E2 找到 y = 5
in my-apply,procedure = (procedure (z) #((+ z 1)) E1) argumets= (5)
in extend-environment:vars and vals :{z} vals: {5} 新建一个环境E3，其外围环境
是E1 E3:  (((z) 5 E1))
in eval-sequence, exps= ((+ z 1)) env=E3
in eval, exp=:(+ z 1) env=E3
```

```
in eval, exp=:+ env=E3
in lookup-variable-value:+ env=E3
in eval, exp=:z env=E3
in lookup-variable-value:z env=E3 找到 z=5
in eval, exp=:1
in my-apply,procedure = {primitive #<procedure:+>}  argumets= (5 1)
in eval, exp=:x
in lookup-variable-value:x
in my-apply,procedure = {primitive #<procedure:*>}  argumets= (6 5)
30
>
```

内部定义

```
(define inner-func '(define (f x)
  (define (g y)
    (k y))
  (define (k z)
    (+ z 1))
  (* (g x) x)))
```

g里的 k 是后面定义的过程（此时还没定义）。可见k 的作用域应是整个 f 体，不是它定义之后的部分。也就是说，块结构里的所有定义应该同时加入环境，具有相同作用域

这里的求值器并没有这样做。但它“恰好”能正确处理这种情况，因为它总在处理完所有的定义后才去用它们。只要所有内部定义都出现在使用所定义变量的表达式 的求值之前，顺序定义和同时定义产生的效果一样（练习4.19）

内部定义

可以修改定义让所有内部定义具有同样作用域。一个办法是做 lambda 表达式的变换，把内部定义取出来放入 let 表达式，然后再赋值。如：

```
(lambda <vars>
   (define u <e1>)
   (define v <e2>)
   <e3>)
```

➔

```
(lambda <vars>
     (let ((u '*unassigned*)
           (v '*unassigned*))
     (set! u <e1>)
     (set! v <e2>)
    <e3>))
```

也可以采用效果相同的其他变换。参看练习4.18

内部定义

```
(let ((a 1))
  (define (f x)
    (define  b (+ a x))
    (define a 5)
    (+ a b))
  (f 10))
```

根据不同的解释器设计方案，可能有三种结果