函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

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

第十二讲

语法和执行分离以及惰性求值器

语法分析和执行分离

语法分析和执行的分离

●考虑递归函数:

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

```
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)
                   (cddr exp))))
```

经过 eval-definition 求值后形成的函数对象是:

```
(procedure (n) ((if (= n 1) 1 (* (factorial (- n 1)) n))) glb-env)
```

函数对象中包含了函数的源代码。

变量**factorial** 和该函数对象的约束被加入环境 **glb-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))
```

原核心函数 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)  ;对函数体分析并求值
           (extend-environment
            (procedure-parameters procedure)
            arguments
            (procedure-enviroment procedure))))
;每执行一次函数调用，哪怕是递归的，都要重新分析函数体，非常浪费
        (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

```
(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))))
```

factorial的函数对象：

```
(procedure (n) ((if (= n 1) 1 (* (factorial (- n 1)) n))) glb-env)
```

每次递归调用factorial，都需要先对 factorial求值找到其对应的函数对象，然后再对函数对象中保存的对函数体的源代码进行分析和求值，十分浪费!!!!

语法分析和执行的分离

- 函数体只应该分析一次

- 函数体分析的结果是一个可执行函数(非源代码形式的函数)，类似于 primitive 函数对象里面的 #<procedure +>。可执行函数由Racket生成，其接收一个参数，就是环境。

  复合函数对象：(procedure (x y) ((* x y)) env)　;env是指向环境的指针

  primitive函数对象：(primitive #<procedure:+>)

核心函数analyze

●原eval将过程的分析和过程的执行混合在一起，现在需要将过程的分析和执行分开。分析只做一次。eval依然是执行求值功能，但变成了对分析结果的调用。

```
(define (eval exp env)
  ((analyze exp) env))
(define (analyze exp) ;返回值是一个可执行函数（不是源代码形式的也不是函数对象），以环境为参数。
  (cond ((self-evaluating? exp)
         (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp) (analyze-sequence (begin-actions exp)))
        ((cond? exp) (analyze (cond->if exp)))
        ((application? exp) (analyze-application exp))
        (else      (error "Unknown expression type -- ANALYZE" exp))))
```

对比：原核心函数eval

```
(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))))
```

处理analyze分支的过程

```
(define (analyze-self-evaluating exp)
  (lambda (env) exp))


(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env) qval)))
```

```
(define (eval exp env)
  ((analyze exp) env))
```

- 查找变量的值还是要在执行过程中做，因为此时才有合适的环境：

```
(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))
```

处理analyze分支的过程

●analyze-assignment必须到执行时才能设置变量的值，因为那时才有环境。但是对计算变量的值的表达式exp，只需分析一次。 analyze-definition类似。

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env) ;vproc是一个可执行函数,(vproc env)才是变量的值
      (set-variable-value! var (vproc env) env)
      'ok)))
(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env) ;vproc是一个可执行函数
      (define-variable! var (vproc env) env)
      'ok)))
```

如果(definition-value exp)是一个lambda表达式，则
(vproc env)就是一个函数对象，内部包含该lambda表达式对应的可执行函数。

处理analyze分支的过程

- **analyze-if** 要分别分析出条件，结果和替代部分。

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

处理analyze分支的过程

- ((lambda? exp) (analyze-lambda exp))

analyze-lambda 分析lambda表达式，返回值是一个可执行函数，这个函数以环境为参数，其执行结果是返回一个函数对象

```
(define (analyze-lambda exp) ;exp是lambda表达式
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
;bproc是可执行函数,需要以环境作为参数
    (lambda (env) (make-procedure vars bproc env))))

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
```

生成的函数对象形如：

(procedure (x y) #<procedure:XXX> env)

#<procedure:XXX>是可执行函数，它需要以环境作为参数

处理analyze分支的过程

- ((application? exp) (analyze-application exp))

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)));fproc是一个可执行函数
        (aprocs (map analyze (operands exp))));aprocs是可执行函数列表
    (lambda (env)
      (execute-application (fproc env);(fproc env)是函数对象
                           (map (lambda (aproc) (aproc env))
                                aprocs)))))
```

假设exp中需要执行的函数是f，则：

fproc是一个可执行函数，其以环境为参数，返回值是一个函数对象，对象里面包含可执行的f。可执行的f需要以环境作为参数。

aprocs中的元素是可执行函数p1,p2,p3...，Pi以环境为参数，Pi返回值是f需要的最终实参值Ai

核心过程 excute-application (对应原my-apply)

```
(define (execute-application proc args)
;proc形如: (procedure (x y) #<procedure:XXX> env)
;原来的my-apply中,  #<procedure:XXX>处是具体的源代码
  (cond ((primitive-procedure? proc)
         (apply (primitive-implementation proc) args))
        ((compound-procedure? proc)
         ((procedure-body proc) ;形如: #<procedure:XXX>
          (extend-environment (procedure-parameters proc)
                              args
                              (procedure-environment proc))))
        (else
         (error
          "Unknown procedure type -- EXECUTE-APPLICATION"
          proc))))
```

原核心函数 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))))
        (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:+>)

处理analyze分支的过程

- ((begin? exp) (analyze-sequence (begin-actions exp)))

analyze-sequence 在analyze分析begin表达式时被调用，以及在analyze-lambda中被调用，用以生成一个函数体的源代码所对应的可执行函数。该可执行函数需要以环境作为参数。

```
(define (analyze-sequence exps)
;exps是表达式列表,形如 ((* x y) (+ x 4)....)
  (define (sequentially proc1 proc2)
    (lambda (env) (proc1 env) (proc2 env)))
  (define (loop first-proc rest-procs);rest-procs是表达式列表
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)));procs是可执行函数列表
    (if (null? procs)
        (error "Empty sequence -- ANALYZE"))
    (loop (car procs) (cdr procs))))
;若 exps =(p1 p2) ,返回值是什么?
```

处理analyze分支的过程

```
(define (analyze-sequence exps)
;exps是表达式列表,形如 ((* x y) (+ x 4)....)
  (define (sequentially proc1 proc2)
    (lambda (env) (proc1 env) (proc2 env)))
  (define (loop first-proc rest-procs);rest-procs是表达式列表
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps));procs是可执行函数列表
    (if (null? procs)
        (error "Empty sequence -- ANALYZE"))
    (loop (car procs) (cdr procs))))
```

```
exps: (p1 p2)
procs: (#p1 #p2);#p1是p1的可执行函数
(loop #p1 (#p2))
(loop (sequentially #p1 #p2) ())
(sequentially #p1 #p2)
返回值:(lambda (env) (#p1 env) (#p2 env))
```

```
(define (analyze-sequence exps)
;exps是表达式列表,形如 ((* x y) (+ x 4)....)
  (define (sequentially proc1 proc2)
    (lambda (env) (proc1 env) (proc2 env)))
  (define (loop first-proc rest-procs);rest-procs是表达式列表
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)));procs是可执行函数列表
    (if (null? procs)
        (error "Empty sequence -- ANALYZE"))
    (loop (car procs) (cdr procs))))
```

```
exps:(p1 p2 p3)
procs : (#p1 #p2 #p3)
(loop #p1 (#p2 #p3))
(loop (sequentially #p1 #p2) (#p3))
(loop (sequantially (sequentially #p1 #p2) #p3) ())
(sequentially (sequentially #p1 #p2) #p3)
(lambda (env) ((sequentially #p1 #p2) env) (#p3 env))
返回值: (lambda (env) ((lambda (env) (#p1 env) (#p2 env)) env) (#p3 env))
```

analyze-sequence另一写法(习题4.23)

```
(define (analyze-sequence exps)
  (define (execute-sequence procs env)
    (cond ((null? (cdr procs)) ((car procs) env))
          (else ((car procs) env)
                (execute-sequence (cdr procs) env))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence -- ANALYZE"))
    (lambda (env) (execute-sequence procs env))))
```

是否可行?

analyze-sequence另一写法(习题4.23)

```
(define (analyze-sequence exps)
  (define (execute-sequence procs env)
    (cond ((null? (cdr procs)) ((car procs) env))
          (else ((car procs) env)
                (execute-sequence (cdr procs) env))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence -- ANALYZE"))
    (lambda (env) (execute-sequence procs env))))
```

```
exps:(p1 p2 p3)
procs : (#p1 #p2 #p3)
(lambda (env) (execute-sequence (#p1 #p2 #p3) env))))
```

可行但是效率低

惰性求值

正则序和应用序

- Scheme 采用的是应用序求值，即在调用过程之前，会对所有的参数进行求值

- 正则序求值则是将参数求值推迟到实际需要时，也称为惰性求值

- 考虑以下过程及其调用：

```
(define (try a b)
  (if (= a 0) 1 b))

(try 0 (/ 1 0))
```

用应用序求值，会先求值 (/ 1 0)，导致出错。
用正则序求值，则不会对 (/ 1 0)求值，不会导致出错，会得到结果1

正则序和应用序

●另一个需要正则序求值的例子：

```
(define (unless condition usual-value exceptional-value)
  (if condition exceptional-value usual-value))
```

使用其的代码：

```
(unless (= b 0)
        (/ a b)
        (begin (display "exception: returning 0")
               0))
```

应用序求值在b=0时导致出错，正则序求值会显示异常信息，并返回0

过程对参数的严格性

●一个过程及其某个参数，如果该过程要求在执行过程体之前，该参数值必须已经求出，则称该过程对于该参数是严格的。否则就是不严格的。一个过程，可以对有的参数严格，对有的参数不严格。

●纯应用序的语言（如**scheme**），所有过程对其所有参数都是严格的。此情况下，有些表达式不能被看作过程，必须被看作“特殊形式”。如 **if ， and， or ， define**

●纯正则序的语言，复合过程对所有参数不严格，基本过程的参数可以是严格的，也可以是不严格的。

●有的语言提供手段让程序员定义过程时，指定参数的严格性。

●允许有的过程的有的参数不严格，有时会很方便。比如用 **cons**构造一个数据结构，如果参数可以不严格，则可在该结构是什么样还没有完全确定的情况下就使用它。

采用惰性求值（正则序）的解释器

- 将前面的低效解释器改写成正则序。复合过程参数都不严格，基本过程参数仍然严格。（有无可能做到基本过程参数也不严格？）

- 如何修改解释器以实现惰性求值？

修改与过程调用有关的部分，不对所有参数都立即求值，而是先检查过程的参数是否需要求值。不是必须求值的参数就不求值，为这个参数建一个称为槽(thunk)的特殊对象。

槽里封装求值参数所需要的全部信息，包括参数表达式本身和相应的求值环境

- 对槽中表达式的求值称为强迫。需要用这个表达式的值时，才去强迫它的槽，求出其值

采用惰性求值（正则序）的解释器

- 需要用槽中的值的三种情况：
  - 某个基本过程需要用槽的值
  - 槽的值被作为条件表达式的谓词
  - 某个复合表达式以这个槽的值作为运算符，并且要求值该表达式

- □还可以考虑是否将槽定义为带记忆的，第一次求值时记录得到的值，以后再求值则直接返回记录的值。
  - 请考虑：这样的修改会不会改变语言的语义？
  - 但也会带来一些不好处理的问题
  - 有关情况参看书上的相关练习

修改求值器 eval

```
(define (eval exp env);(eval exp env)的结果可能是个延时求值对象
  (cond ((self-evaluating? exp ) exp)
        ((null? exp) (void))
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((and? exp) (eval-and (cdr exp) env))
        ((or? exp) (eval-or (cdr exp) env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp) env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
       ((application? exp)
         (my-apply (actual-value (operator exp) env) ;参数暂不求值，且要传递env
                   (operands exp)
                   env))
        (else
             (error "unknown expression type -- EVAL" exp))))
```

哪些函数要修改？

```
(define (actual-value exp env)
  (force-it (eval exp env)))
;(eval exp env)的结果可能是个
延时求值对象 --- 什么情况下？
```

```
(my-apply (eval (operator exp) env);原来的
          (list-of-values (operands exp) env)))
```

修改 eval-if

```
(define (eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
;原来: (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

eval-and,eval-or也要相应修改。考虑:

```
(define (func x y)
  (if (and x y)
      (display "good")
      (display "bad")))
(func (begin (display "p1") false) (begin (display "p2") true))
```

修改求值器 my-apply

```
(define (my-apply procedure arguments env)
;arguments中的参数都是未经求值的原始形式
  (cond ((primitive-procedure? procedure)
         (let ((tmp (list-of-arg-values arguments env)))
           (apply (primitive-implementation procedure) tmp)))
        ;tmp里面是已经求得最终值的参数的列表。原来tmp处就是arguments
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env);原来此处就是arguments
           ;生成一个延时求值对象的列表。每个对象对应于arguments中的一个参数
           (procedure-enviroment procedure))))
        (else
         (error "unkonwn procedure type -- APPLY" procedure))))
```

原核心函数 my-apply

```
(define (my-apply procedure arguments);arguments里是最终参数值
  (cond ((primitive-procedure? procedure)
         (apply (primitive-implementation procedure) arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-enviroment procedure))))
        (else
         (error "unkonwn procedure type -- APPLY" procedure))))
```

force-it和delay-it

```
(define (actual-value exp env)
      (force-it (eval exp env)));(eval exp env)的结果可能是个延时求值对象

(define (list-of-arg-values exps env)
;exps是表达式列表，每个表达式是一个参数。此处要求出参数的最终值
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps) env))))

(define (list-of-delayed-args exps env)
;exps是表达式列表，每个表达式是一个参数。此处要构造参数所对应的延时求值对象的列表
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps) env))))
```

force-it和delay-it的实现

●delay-it生成一个“槽”(thunk)，即延时求值对象，该对象内部包含要求值的表达式，以及对该表达式进行求值时需要用到的环境。

```
(define (delay-it exp env)
  (list 'thunk exp env))
```

exp本身是否可能也是个槽？

●force-it以一个“槽”作为参数，对槽中的表达式进行求值并返回该值。

```
(define (force-it obj) ;针对不带记忆的槽
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))
```

force-it的结果是否可能仍然是个槽？

```
(define (actual-value exp env)
      (force-it (eval exp env)));(eval exp env)的结果可能是个延时求值对象
```

force-it和delay-it的实现

●delay-it生成一个“槽”(thunk)，即延时求值对象，该对象内部包含要求值的表达式，以及对该表达式进行求值时需要用到的环境。

```
(define (delay-it exp env)
  (list 'thunk exp env))
```

exp本身是否可能也是个槽？不可能。exp总是源代码形式的表达式

●force-it以一个“槽”作为参数，对槽中的表达式进行求值并返回该值。

```
(define (force-it obj) ;针对不带记忆的槽
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))
```

force-it的结果是否可能仍然是个槽？不可能。

```
(define (actual-value exp env)
      (force-it (eval exp env)));(eval exp env)的结果可能是个延时求值对象
```

带记忆的槽(thunk)相关函数

```
(define (thunk? obj)
  (tagged-list? obj 'thunk))

(define (thunk-exp thunk) (cadr thunk))

(define (thunk-env thunk) (caddr thunk))

(define (evaluated-thunk? obj) ;判断槽是否已经求值过
  (tagged-list? obj 'evaluated-thunk))

(define (thunk-value evaluated-thunk) (cadr evaluated-thunk))
```

带记忆的槽(thunk)相关函数

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)  ; replace exp with its value
           (set-cdr! (cdr obj) '())     ; forget unneeded env
           result))
        ((evaluated-thunk? obj)
         (thunk-value obj))
        (else obj)))
```

修改读入循环

```
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output
           (actual-value input glb-env)))
;原来是 (eval input glb-env)
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

思考:下面做法是否可行?

```
(define (actual-value exp env)
  (force (eval exp env)))   ;(eval exp env)的结果可能是个延时求值对象

(define (list-of-arg-values exps env)
;exps是表达式列表，每个表达式是一个参数。此处要求出参数的最终值
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps)
                                env))))

(define (list-of-delayed-args exps env)
;exps是表达式列表，每个表达式是一个参数。此处要构造参数所对应的延时求值对象的列表
  (if (no-operands? exps)
      '()
      (cons (delay (actual-value (first-operand exps) env));或者,eval?
            (list-of-delayed-args (rest-operands exps)
                                  env))))
```

正则序解释器工作过程分析

```
(eval '(define x 3) glb-env)
(eval '(define y 4) glb-env)
(eval '(define (maxx a b) (if (> a b) a b)) glb-env)
;maxx所对应的值是一个函数对象，还是一个槽，里面包含函数对象?
(eval '(+ x y) glb-env) ;=> ?
(eval '(maxx x y) glb-env) ;=> ?
```

正则序解释器工作过程分析

```
(eval '(+ x y) glb-env) ;=> ?
```

进入 my-apply, + 是primitive-procedure于是:

```
(apply + (3 4))
```

```
=> 7
```

正则序解释器工作过程分析

```
(eval '(maxx x y) glb-env) ;=> ?
```

1) 进入 my-apply, 是compound-procedure于是:

正则序解释器工作过程分析

```
(eval '(maxx x y) glb-env) ;=> ?
```

1) 进入 my-apply, 是compound-procedure于是:

2)
```
(eval-sequence ((if (> a b) a b))
       (ext-env (a b) ((thunk x glb-env) (thunk y glb-env)) glb-env)
```

正则序解释器工作过程分析

```
(eval '(maxx x y) glb-env) ;=> ?
```

1) 进入 my-apply, 是compound-procedure于是:
2) (eval-sequence ((if (> a b) a b))
   (ext-env (a b) ((thunk x glb-env) (thunk y glb-env)) glb-env)
3) a => (thunk x glb-env)
   b => (thunk y glb-env)

正则序解释器工作过程分析

```
(eval '(maxx x y) glb-env) ;=> ?
```

1) 进入 my-apply, 是compound-procedure于是:
2) (eval-sequence ((if (> a b) a b))
   (ext-env (a b) ((thunk x glb-env) (thunk y glb-env)) glb-env)
3) a => (thunk x glb-env)
   b => (thunk y glb-env)
4) 执行 (> a b)时, a,b都被强制于是:
   a=> (evaluated-thunk 3)
   b=> (evaluated-thunk 4)

正则序解释器工作过程分析

```
(eval '(maxx x y) glb-env) ;=> ?
```

1) 进入 my-apply, 是compound-procedure于是:
2) (eval-sequence ((if (> a b) a b))
   (ext-env (a b) ((thunk x glb-env) (thunk y glb-env)) glb-env)
3) a => (thunk x glb-env)
   b => (thunk y glb-env)
4) 执行 (> a b)时, a,b都被强制于是:
   a=> (evaluated-thunk 3)
   b=> (evaluated-thunk 4)
5) if语句返回b,即 (evaluated-thunk 4)

正则序解释器工作过程分析

```
(eval '(maxx x y) glb-env) ;=> ?
```

1) 进入 my-apply, 是compound-procedure于是:
2) (eval-sequence ((if (> a b) a b))
   (ext-env (a b) ((thunk x glb-env) (thunk y glb-env)) glb-env)
3) a => (thunk x glb-env)
   b => (thunk y glb-env)
4) 执行 (> a b)时, a,b都被强制于是:
   a=> (evaluated-thunk 3)
   b=> (evaluated-thunk 4)
5) if语句返回b,即 (evaluated-thunk 4)

```
(actual-value '(maxx x y) glb-env) ;=> 4
```

对(eval '(maxx x y) glb-env)的返回结果，即 (evaluated-thunk 4)进行强制，结果就是4

某种形式的thunk嵌套(thunk可以形成一条链)

```
(eval '(define (f x) x) glb-env)
(eval '(define (g y) (f y)) glb-env)
(actual-value '(g 2) glb-env)
```

1) 将f和g约束到函数对象，将这两个约束加入glb-env
2) 进入 my-apply, g是compound-procedure于是:

```
(eval-sequence ((f y))
    (ext-env (y) ((thunk 2 glb-env)) glb-env)
```

3) 新环境E1中有约束: y => (thunk 2 glb-env)
4) eval-sequence在E1中求值(f y)，进入my-apply, f是compound-procedure,于是

```
(eval-sequence ((x))
    (ext-env (x) ((thunk y E1)) glb-env)
```

5) 新环境E2中有约束: x => (thunk y E1)
6) eval-sequence在E2中对 x进行求值，得到 (thunk y E1) 即(eval '(g 2) glb-env)的返回值
7) 强迫(thunk y E1)，在E1中对 y进行求值，得到 (thunk 2 glb-env)
8) 强迫(thunk 2 glb-env)，得到最终结果 2

思考:

能否修改`lookup-variable` 过程，找到变量的值，发现是个`thunk`，就强迫它，然后用求得的值替代`thunk`。这样能够提高效率。

思考:

能否修改lookup-variable 过程，找到变量的值，发现是个thunk，就强迫它，然后用求得的值替代thunk。这样能够提高效率。

解释器中有些地方只用 eval ，而不是用actual-value，说明有些地方需要返回值就是个thunk。因此不能在lookup-variable中将查到的thunk替换成值。

```
(define (func x) x)
(define w (func (/ 1 0))) ;此处(eval 'func (/ 1 0))的返回值是个槽
(define k w) ;此处需要lookup-variable去找w的值，但找到一个槽就OK，不应真的求值
(define (try a b)
          (if (= a 0)
              a
              b))
(try 0 k)   ;=> 0
```

思考:

在正则序解释器中，下面程序的结果是:

```
(define (p2 x)
  (define (p e)
    e
    x)
  (p (set! x (cons x '(2)))))

(p2 1)
```

思考:

在正则序解释器中，下面程序的结果是:

```
(define (p2 x)
  (define (p e)
    e
    x)
  (p (set! x (cons x '(2)))))

(p2 1)
```

eval-sequence中会 (eval e)，得到的结果是个槽。而不是 (acutal-value e)。
该槽是: (thunk e E2)
E2: e => (thunk (set! x (cons x '(2))))

流和表的统一(用惰性的表实现流)

流和表的统一(用惰性的表实现流)

●有了正则序解释器，解释器的用户可以编写程序,在程序中实现表，并且该表就是惰性的，即也是流。总之，在用户程序中，表和流是相同的。

●下面的程序都不是解释器的一部分，而是用户编写的实现流的程序，这些程序可以由正则序解释器解释执行。

流的构造函数和选择函数

```
(define (cons x y) ;一个流就是一个闭包，是一个过程
  (lambda (f) (f x y)))
(define (car z)
  (z (lambda (p q) p)))
(define (cdr z)
  (z (lambda (p q) q)))
```

在正则程序中，流不但 cdr 是延迟求值的，car都是延迟求值的!!

流相关函数

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items))
            (map proc (cdr items)))))
(define (scale-list items factor)
  (map (lambda (x) (* x factor))
       items))
(define (add-lists list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        (else (cons (+ (car list1) (car list2))
                    (add-lists (cdr list1) (cdr list2))))))
```

流的应用

```
(define ones (cons 1 ones))
(define integers (cons 1 (add-lists ones integers)))
(define (integral integrand initial-value dt);积分器
  (define int
    (cons initial-value
          (add-lists (scale-list integrand dt)
                     int)))
  int)
(define (solve f y0 dt) ;解微分方程 dy/dt = f(y), y(0) = y0
  (define y (integral dy y0 dt))
  (define dy (map f y))
  y)
(car (cdr (cdr integers)))
(list-ref (solve (lambda (x) x) 1 0.001) 1000)
```

解释器运行上面的程序，则输出结果为：

```
3
2.716924
```

正则序解释器处理流的工作过程分析

```
(define (cons x y) ;一个流就是一个闭包，是一个过程
  (lambda (f) (f x y)))

(eval '(define x1 (cons 1 '())))
```

将x1的约束加入了glb-env。 x1 的值是什么样的？环境有什么变化？

正则序解释器处理流的工作过程分析

```
(define (cons x y) ;一个流就是一个闭包，是一个过程
  (lambda (f) (f x y)))

(eval '(define x1 (cons 1 '())))
```

将x1的约束加入了glb-env。 x1 的值是什么样的？环境有什么变化？

```
x1: (procedure (f) ((f x y)) E1)
E1: (((x y) (thunk 1 glb-env) (thunk '() glb-env)) . glb-env)
```

正则序解释器处理流的工作过程分析

```
(define (car z)
  (z (lambda (p q) p)))

(eval '(car x1) glb-env)
```

的结果和执行过程？

正则序解释器处理流的工作过程分析

```
(define (car z)
  (z (lambda (p q) p)))
```

(eval '(car x1) glb-env) 的执行过程？

```
(my-apply ((actual-value car glb-env)
           (x1)
           glb-env))
```

正则序解释器处理流的工作过程分析

```
(define (car z)
  (z (lambda (p q) p)))
```

(eval '(car x1) glb-env) 的执行过程？

```
(my-apply ((actual-value car glb-env)
           (x1)
           glb-env))

(my-apply (procedure (z) ((z (lambda (p q) p))) glb-env)
           (x1)
           glb-env)
```

正则序解释器处理流的工作过程分析

```
(define (car z)
  (z (lambda (p q) p)))
```

(eval '(car x1) glb-env) 的执行过程？

```
(my-apply ((actual-value car glb-env)
           (x1)
           glb-env))

(my-apply (procedure (z) ((z (lambda (p q) p))) glb-env)
           (x1)
           glb-env)

(eval-squence ((z (lambda (p q) p)))
              (ext-env (z) ((thunk x1 glb-env)) glb-env))
```

正则序解释器处理流的工作过程分析

ext-env返回值是 E2: ((z) (thunk x1 glb-env) glb-env)

```
(my-apply (actual-value (thunk x1 glb-env) glb-env)
   ((lambda (p q) p))
   glb-env)
```

正则序解释器处理流的工作过程分析

ext-env返回值是 E2: ((z) (thunk x1 glb-env) glb-env)

```
(my-apply (actual-value (thunk x1 glb-env) glb-env)
   ((lambda (p q) p))
   glb-env)

(my-apply (procedure (f) ((f x y)) E1)
     ((lambda (p q) p))
     glb-env)
```

在E2里，z所对应的对象变为：

```
(evaluated-thunk (procedure (f) ((f x y)) E1))
```

正则序解释器处理流的工作过程分析

```
(eval-sequence ((f x y))
               (ext-env (f) ((thunk (lambda (p q) p) glb-env)) E1))
```

ext-env返回值是 E3 : ((f) (thunk (lambda (p q) p) glb-env) E1)

```
(my-apply (actual-value (thunk (lambda (p q) p) glb-env))
          (x y)
          E1)

(my-apply (procedure (p q) (p) glb-env)
          (x y)
          E1)
```

正则序解释器处理流的工作过程分析

```
(eval-sequence ((f x y))
               (ext-env (f) ((thunk (lambda (p q) p) glb-env)) E1))
```

ext-env返回值是 E3 : ((f) (thunk (lambda (p q) p) glb-env) E1)

```
(my-apply (actual-value (thunk (lambda (p q) p) glb-env))
          (x y)
          E1)

(my-apply (procedure (p q) (p) glb-env)
          (x y)
          E1)

(eval-sequence (p)
       (ext-env (p q) ((thunk x E1) (thunk y E1))) E1))
```

正则序解释器处理流的工作过程分析

ext-env返回的结果是E4 :

```
((p q) (thunk x E1) (thunk y E1))
```

上面:

```
(eval-sequence (p)
       (ext-env (p q) ((thunk x E1) (thunk y E1))) E1))
```

在E4中对p求值,

最终, `(eval '(car x1) glb-env)` 返回结果是:

```
(thunk x E1)
```

```
E1: (((x y) (thunk 1 glb-env) (thunk '() glb-env)) . glb-env)
```