4. 元语言抽象(2)

本节讨论 **Scheme** 语言实现的两个改造：

- 语法分析和执行的分离
  - 目的是改进求值器的效率
  - 分离语法分析和执行

    做法：尽可能避免重复工作
- 正则序求值器
  - 全部参数都采用延时求值
  - 改造求值器的核心操作 **eval** 和 **apply**

回顾：元循环求值器

- 基本部分：
  - **eval** 和 **apply** 核心过程
  - 各种表达式的数据抽象：接口和实现
  - 求值器数据结构
- 求值器数据结构：
  - 环境数据抽象的实现
  - 环境的结构
    - 一系列框架
    - 求值过程对象时，基于过程的环境建立新当前框架和新环境
  - 环境的查询和修改（**set!** 和 **define**）
- 初始环境的构造和主循环（读入-求值-打印）
- 回顾两个核心过程

核心过程 eval：在 env 里求值 exp

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((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))
        ((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)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```


核心过程 apply

- apply 以一个过程和一个实参表为参数，实现过程应用

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error "Unknown procedure type -- APPLY" procedure))))
```

有了这个求值器，就可以考虑其改造

首先是改造的理由：为什么要改造原来的求值器？

语法分析和执行的分离

- 前面做好的元循环求值器的特点
  - 实现非常简单
  - 但效率比较低
- 主要问题：
  - 表达式的语法分析和执行交织在一起
  - 如果在程序执行中一个表达式需要求值很多次，就需要对它做很多次语法（结构）分析
- 需要多次执行的情况：
  - 函数体，一个函数可能被多次调用
  - 特别是递归定义的函数
    - 每次递归都需要重新分析函数体
    - 函数体表达式可能被任意多次重复分析

语法分析和执行的分离

- 以求阶乘函数为例：

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

- 在求 **(factorial 4)** 时
  - 每次调用都要确定过程体是否为 **if**
    - 而后再提取谓词部分求值
    - 再基于条件的值继续向下求值，等等
  - 求值其中的子表达式时
    - **eval** 又需要一个个分情况处理
- 分析表达式结构的代价很高，而且完全没必要重复做

  在解释程序的执行过程中，函数体的代码没有变化

语法分析和执行分离

- 如何提高执行效率？
  - 渐少（或消除）重复的分析工作
  - 最理想状态：
    - 对每段程序代码的分析只做一次
    - 重复执行时不需要做任何分析工作
- 需要有办法记录和使用结构分析的结果
  - 在一次分析中，根据程序的情况生成相应的结果
  - 执行时直接使用分析的结果
- 这意味着
  - 分析的结果应该能直接执行
  - 也就是说：分析的结果应该是能直接执行的程序
- 下面工作：修改求值器，重新安排处理 **Scheme** 程序的方式

语法分析和执行分离

- 具体做法：把 **eval** 的工作分为两部分
  - 第一步是分析被处理的程序，生成与之对应的可执行程序
  - 第二步是直接执行生成的程序
- 考虑：定义一个过程 **analyze**
  - 专门做对被求值表达式的语法分析
  - 对每个被分析的表达式，**analyze**返回一个过程，称为执行过程
  - 把表达式分析的结果封装在这个执行过程（的体）里
- 求值工作的第二步：
  - 以环境作为参数，实际执行前一步分析得到的执行过程

    使它产生的效果等同于对原表达式求值的效果
  - 这样做，显然每个表达式的分析只需要做一次
- 可能比直接分析和执行效率高一些

语法分析和执行分离

- 根据新的求值方式，**eval** 变成了一个简单过程：

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

- 注意一些情况：
  - **analyze** 从 **exp** 出发生成一个过程
  - 生成的过程（表达式的执行过程）

    以 **env** 为参数执行

    执行的效果应等同于求值器在环境 **env** 里解释 **exp**
- 下面考虑 **analyze**
- **analyze** 的工作是原来的 **eval** 工作的前一部分，可以采用类似的结构
  - 基本结构是一个分情况分析
  - 根据被分析表达式的情况，分别生成相应的执行过程

analyze

- **analyze** 不做实际的表达式求值，只构造可以直接执行的程序

  各种执行过程的具体构造通过相应子程序完成

  analyze 只是根据表达式类型完成工作分配

```
(define (analyze exp)
  (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))))
```

基本表达式的分析

- 现在考虑各种表达式的分析，以及对应执行过程的构造

  对一个具体表达式的分析生成一个过程

  这种过程只能用一个 **lambda** 表达式描述

- 所有执行过程（**lambda** 表达式）都以一个环境为参数

  这种过程可以以一个环境为参数直接执行

  产生的效果与直接求值被分析的表达式时产生的效果相同

- 简单表达式的分析结果应该直接生成

  直接用一个 **lambda** 表达式描述

- 组合表达式产生的过程，应该

  是由成分表达式的执行过程组合而成

  其执行也就是相应成分表达式的过程执行的某种组合

- 下面首先考虑基本表达式的分析及其结果

基本表达式的分析

- 分析自求值表达式，生成相应过程

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

  在任何环境里，自求值表达式 **exp** 的值总是它自身

- 分析引号表达式时直接取出被引的表达式

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

  不把相应工作留到执行过程执行时，可以提高执行效率

- 取变量值，生成的过程仍在执行时到环境里查找变量的值

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

赋值的分析

- 赋值和定义表达式的工作都需要在求值的时候做
- 赋值表达式的分析

  需要设置变量（在实际环境里操作）

  先完成被赋值表达式的分析，可以提高执行时的工作效率
- 实现：

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env)
      (set-variable-value! var (vproc env) env)
      'ok)))
```

  一些辅助过程沿用了前面元循环求值器的定义

  如这里的语法过程 **assignment-variable, assignment-value** 等

定义的分析

- 定义表达式的分析结果与赋值表达式结构相同，只是调用的辅助过程（实际完成操作的过程）不同
- 相应分析过程的定义

```
(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env)
      (define-variable! var (vproc env) env)
      'ok)))
```

- 注意：

  这里也继续使用元循环求值器的语法过程

  definition-variable

  definition-value

  作为分析结果的过程（**lambda** 表达式的值），执行时完成实际定义

if 表达式的分析

- 对 if 表达式

  首先提取出谓词部分和两个分支表达式

  对这三个部分分别分析

  最后基于三个部分的分析结果构造 if 的分析结果

- 实现：

```
(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)))))
```

  if 的执行过程先调用谓词的求值过程，而后分情况执行

lambda 表达式的分析

- lambda 表达式体的分析同样只做一次

  其分析结果还是一个函数对象

  但函数对象的体已经是一个执行过程，在函数调用时直接执行

  多次执行可能大大提高效率

- 实现

```
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env) (make-procedure vars bproc env))))
```

  这里继续采用元循环求值器的语法过程

  构造函数 make-procedure 还是用前面的定义

表达式序列的分析

- 对表达式序列（**begin** 表达式或 **lambda** 体）需要深入分析

  其中每个表达式产生一个执行过程

  再把它们组合起来做成一个执行过程

```
(define (analyze-sequence exps)
  (define (sequentially proc1 proc2)
    (lambda (env) (proc1 env) (proc2 env)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs) (error "Empty sequence -- ANALYZE"))
    (loop (car procs) (cdr procs))) )
```

组合起序列里前两个子表达式的执行过程

分析各子表达式

过程应用表达式的分析

- 过程应用的分析：
  - 分别分析其中的运算符和运算对象
  - 对每个子表达式生成一个执行过程
  - 把分析结果组合为一个过程
  - 最后送给 **execute-application** 过程

    这个过程与 **apply** 对应

    它能实际地执行这个过程（过程调用的执行过程）

过程应用的分析

分析各运算对象

完成过程应用

执行各运算对象的执行过程，得到实参表

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp))) # 分析运算符表达式
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application (fproc env)
                           (map (lambda (aproc) (aproc env)) aprocs)))))

(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                              args
                              (procedure-environment proc))))
        (else
         (error "Unknown proc type -- EXEC-APPLICATION" proc))))
```


分析求值器：总结

- 至此分析求值器的构造完成。总结一下
- 具体做法是
  - 把 **eval** 的表达式求值工作分为两部分：表达式分析和实际执行
  - 定义一个完成分析的过程 **analyze**
    - 它完成对被求值表达式（及其中嵌套的各层子表达式）的语法分析，构造相应的执行过程
    - 这种过程的执行时行为等价于被分析表达式的求值
  - 执行过程都以环境为参数执行，产生表达式求值的效果
- 这样做，被求值表达式只需要做一次分析，可以多次执行
  - 特别是对各种可能重复执行的表达式，如包含递归的过程体等
  - 避免重复分析可能大大提高工作的效率

分析求值器：总结

- 这里的做法类似于高级语言程序的解释和编译
  - 直接解释，就是一遍遍分析程序代码，实现其语义

    例如最早的 **BASIC** 语言实现
  - 编译把实现程序功能的工作分为两步：
    - 通过一次分析生成一个可执行的程序
    - 在此之后可以任意地多次执行这个程序
- 这里做的是从 **Scheme** 源程序（元循环解释器/求值器处理的“数据”）到 **Scheme** 可执行程序的翻译

  这是一种 **Just In-time Translation，JIT**（即时翻译）
- 目前成熟的 **Java** 实现（和另外一些语言的实现）都做这种“即时翻译”，希望通过这种技术提高执行效率

  在实际中应用，还需进一步考虑整体效率问题

Scheme 的变形：惰性求值

- 做好一个求值器，还可以考虑修改它，试验语言设计的各种选择和变化
- 开发新语言的一般做法：
  - 基于某种现有的语言，实现新语言的求值器，而后研究评价各种设计想法和可能的设计选择
  - 在求值器里实现各种设计选择

    高层次的求值器更容易实现、测试和修改

    可以从基础语言借用大量有用的功能
  - 给相关人员使用和评价，根据评价修改求值器，再送出评价
  - 至新语言较成熟，再考虑另行实现完整的语言系统
- 下面研究 **Scheme** 的一些变形
  - 它们提供了一些基本 **Scheme** 没有的功能
  - 相应的求值器都能共享元循环求值器的许多设计和代码

正则序和应用序

- **Scheme** 采用的是应用序求值

  过程应用之前完成对所有参数的求值

  正则序把参数求值推迟到实际需要时

  正则序求值也被称为惰性求值（懒求值，消极求值）

- 考虑下面过程：

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

  在 **Scheme** 里求值 **(try 0 (/ 1 0))** 将会出错

  求值 **b** 的实参 **(/ 1 0)** 时出现除数为 **0** 的情况

  实际上，这个求值过程中并没有用到 **b** 的值

正则序和应用序

- 另一个需要正则序的例子

  希望实现一种控制：除非出现异常条件，否则就正常处理

- 过程定义：

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

  希望实现的效果：在 **condition** 不真时以 **usual-value** 实参的值作为值（而且只求值这个参数），否则以 **exceptional-value** 的值作为值（也只求值这个参数）

- 如果不采用正则序求值，这个过程完全没用：

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

正则序和应用序

- 过程对参数的严格性（概念）

  对于一个过程和它的任一个参数

  - 如果该过程要求在进入过程体之前，先行完成对该参数的求值，就称这一过程对于这个参数是严格的
  - 如果不要求其先行完成求值，则说这一过程对这个参数是非严格的
- 在纯的应用序语言里，每个过程对它的每个参数都是严格的

  需要有采用特殊求值方式的“特殊形式”，如 **Scheme**
- 在纯的正则序语言里
  - 每个复合过程对其每个参数都是非严格的
  - 基本过程对其参数可以是严格的或者非严格的
- 有的语言允许程序员控制所定义过程对各参数的严格性
  - 例如，提供描述参数严格性的机制

正则序和应用序

- 能将过程的某些参数定义为非严格，也很有用
  - 例：**cons** 或任何数据结构的构造函数

    如果一个数据结构的构造函数允许非严格参数，就可以在不知道数据结构某些部分的情况下使用该数据结构
  - 前面的流模型就是这样的结构
- 本节考虑实现一个正则序语言
  - 该语言的语法形式与 **Scheme** 语言完全一样
  - 但其中的所有复合过程的参数都采用惰性求值
  - 基本过程仍然采用应用序求值

    执行前先求值所有的参数
- 下面将考虑修改前面的元循环求值器，实现这个“新”语言

  显然，由于语法没变，可以继承元循环求值器的很多基本部分

正则序和应用序

- 考虑求值器的修改
  - 实际上，只需要修改与过程应用有关的结构
  - 其中不是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
- 需要延时的参数不求值，而是
  - 为这个表达式建一个称为槽 **(thunk)** 的特殊结构
  - 在建立的槽里封装求值该表达式所需要的信息

    包括表达式本身和相应的求值环境
  - 这种带着求值环境的表达式称为闭包
- 对入槽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，求出值

采用惰性求值的解释器

- 需要用值的几种情况：
  - 某个基本过程需要用这个表达式的值
  - 表达式的值被作为条件表达式的谓词
  - 某个复合表达式以这个表达式的值作为运算符

    下面需要应用它（应该是一个过程）
- 还可以考虑是否将槽定义为带记忆的，第一次求值记录得到的值

  这是一个设计选择

  请考虑：这样的修改会不会改变语言的语义？
- 下面采用带记忆的槽，这样实现可能更高效

  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

- 修改求值器的关键：

  修改过程 **eval** 和 **apply** 里对表达式的处理

- **eval** 里 **cond** 中的 **application?** 子句修改为（其他都不改）

```
((application? exp)
 (apply (actual-value (operator exp) env)
        (operands exp)
        env))
```

直接把未求值的运算对象表达式送给 **apply**

在前面的实现里，是把求值之后的实际参数送给 **apply**

还要把当前环境送给 **apply**

因为它可能需要构造参数槽

参数槽需要携带有关表达式的求值环境

修改求值器：eval

- **actual-value** 取得表达式的实际值

```
(define (actual-value exp env)
    (force-it (eval exp env)))
```

需要实际参数的值时，强迫求值

force-it 的定义后面考虑

修改求值器：apply

apply 也要做相应修改

现在来自 **eval** 的都是未求值的运算对象（表达式），送给基本过程之前，需要求出这些表达式的值：

```
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env)))  ; 修改了
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)  ; 修改了
           (procedure-environment procedure))))
        (else
         (error "Unknown procedure type -- APPLY" procedure))))
```

算出实际参数值

延时相关参数

辅助过程

- 两个辅助过程：

```
(define (list-of-arg-values exps env)
  (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)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps) env))))
```

强迫计算实参值

构造相应的槽

if 的改造

- if 需要修改：

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

- 还有槽数据结构和几个相关过程
  - force-it
  - delay-it

  后面考虑
- 下面先考虑执行循环

驱动循环

- 修改驱动循环

  用 actual-value 代替其中 eval，要求做表达式的实际求值：

```
(define input-prompt ";;; L-Eval input:")
(define output-prompt ";;; L-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output
           (actual-value input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop) )
```

实际使用

- 运行示例

```
(define the-global-environment (setup-environment))
(driver-loop)
;;; L-Eval input:
(define (try a b)
  (if (= a 0) 1 b))
;;; L-Eval value:
ok
;;; L-Eval input:
(try 0 (/ 1 0))
;;; L-Eval value:
1
```

- 做这个试验前，需要先完成槽的实现（下面完成）

槽的表示

- 槽也是一种数据抽象，它实现一种求值允诺

  其中封装一个表达式和一个环境，需要时可以求出表达式的值

- 实际求值时应该用 **actual-value** 而不是 **eval**（现在的 **eval** 是延时的）
- 强迫时求值到不是槽为止（**force-it** 和 **actual-value** 相互递归）：

```
(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))
```

- 槽的简单实现就是用一个表把表达式和环境包装起来：

```
(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))
```

带记忆的槽

- 实现带记忆的槽时，在槽求值后将其换成得到的值表达式

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

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

(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) ; 把 exp 换成它的值
           (set-cdr! (cdr obj) '())    ; 丢掉不再用的环境 env
           result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj)))
```

- 无论有没有记忆，这个求值器都能工作

流作为惰性的表

- 前面研究流计算时，把那里的流实现为一种延时的表，其中用了特殊形式 **delay** 和 **cons-stream**。该方式的缺点：
  - 需要用特殊形式，特殊形式不是一级对象，无法与高阶过程协作
  - 流与表类似，但又是另一类不同的对象，因此需要为流重新实现各种表操作，而且这些操作只能用于流

  采用惰性求值，就可以直接用表作为流，不再需要任何特殊形式
- 要实现流，还需要 **cons** 是非严格的。存在多种方式完成这件事：
  - 修改求值器允许非严格基本过程，并把 **cons** 实现为非严格过程
  - 把 **cons** 实现为复合过程（**2.1.3** 节，**61**页）
  - 重新定义 **cons**，用过程的方式表示序对（练习**2.4**）：

```
(define (cons x y) (lambda (m) (m x y)))
(define (car z) (z (lambda (p q) p)))
(define (cdr z) (z (lambda (p q) q)))
```

流作为惰性的表

- 基于这些基本操作，各种表操作的标准定义

  可以用于有穷的表

  也能自然地适用于无穷的表（流）

- 流操作都可以简单地实现为相应的表操作：

```
(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)))))
```


流作为惰性的表

- 更多是表操作（流操作）：

```
(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))))))
```

- 应用实例：

```
(define ones (cons 1 ones))

(define integers (cons 1 (add-lists ones integers)))

;;; L-Eval input:
(list-ref integers 17)
;;; L-Eval value:
18
```

流作为惰性表

- 这里的表比前面的流更惰性：
  - 现在表的 **car** 部分也是延时的

    同样是直到需要用时才真正求值
  - 取序对的 **car** 或 **cdr** 时都不求值

    对这两个部分的求值都延时到真正需要时
- 需要值的两种情况：
  - 用作基本过程的参数
  - 或者需要打印输出

流作为惰性表

- 惰性序对还能解决流引起的其他问题
  - 例如，前面讨论流的时候，看到了一些情况

    如果用流处理的情况里包含了信息反馈

    有些情况里需要显式地使用 **delay** 操作
  - 现在一切参数都是延时的

    上述情况也不需要特殊处理了
- 原来需要显式使用 **delay** 以保证求值能够进行
  - 现在不需要显式使用了
  - 看那个例子

流作为惰性表

- 例如，现在可以直接定义表积分。重新试验前面求解微分方程的例子

```
(define (integral integrand initial-value dt)
  (define int
    (cons initial-value (add-lists (scale-list integrand dt) int)))
  int)

(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (map f y))
  y)

;;; L-Eval input:
(list-ref (solve (lambda (x) x) 1 0.001) 1000)
;;; L-Eval value:
2.716924
```

- 至此有关正则序的元循环求值器的讨论结束

总结

- 元循环求值器的两个改造：
- 第一个是为了提高效率，并不改变语言和语义
  - 把求值器中分析表达式结构的部分独立出来
  - 避免多次分析，可以提高程序执行效率
- 第二个改造也不改变语言，但改变了语义
  - 求值器采用正则序
  - 所有复合过程的参数在调用时不求值，实际使用时再求值
  - 采用的技术是建立保存求值相关信息的槽

    一个槽包含一个带求值表达式和它的求值环境
- 虽然构造的求值器与前面元循环求值器不同
  - 它们都继承了元循环求值器的基本结构
  - 重用了元循环求值器的大量数据抽象