4. 元语言抽象(I)

本节讨论

- 语言抽象
- 元循环求值器
  - 基本求值过程
  - 求值器的核心操作 eval 和 apply
  - 表达式数据抽象和接口操作
  - 求值器数据结构
- 求值器的改进

程序设计和语言

- 前面讨论了多种程序设计技术，主要研究：
  - 如何组合基本程序设计元素，构成具有更复杂功能的结构
  - 如何将复杂结构抽象为高层构件，以方便地用于进一步组合
  - 如何采用一些更高层次的观点和组织方式，提高系统的模块性

  这些都是编程，我们一直用 Lisp (方言 Scheme) 作为编程语言
- 问题更复杂时，或需要解决某领域的大量问题，可能发现手头可用的语言（Lisp 或其他）都不满意，不方便，希望有一种能更有效地表述想法的新语言。如果没有现成的适用语言，就需要自己做一个
- 设计一个好的适用的语言很不容易，不应轻易去做。但另一方面，建立新语言是控制复杂性最有力的策略
- 针对问题 (领域) 设计的专门语言能提供最适用的原语、组合方式和抽象方式，使人能以最有效的方式描述问题，大大提高在一定范围内处理复杂情况的能力

程序设计和语言

- 有时也应该考虑做一个语言设计师（至少应注意这种可能）
- 程序设计工作中通常会涉及多层次的多种语言（应该看到这个问题）
  - 有的很简单，如 C 标准库 printf 的排版语言
  - 最复杂的是高级编程语言

  理解语言和语言解释器，也能帮助我们进一步理解程序设计
- 工作中不仅应考虑根据需要设计语言，还能通过构造解释器实现它
  - 语言解释器是一个过程，它应用于相应语言的一个表达式时，会按该表达式的要求执行相应动作
  - 实现语言应是程序设计最本质的思想：求值器定义了语言里各种表达式的意义，而它本身也就是一个程序
- 复杂的程序都有可能看作某种语言的求值器。如前面逻辑电路模拟器和约束传播系统是完整的语言，有基本原语、组合手段和抽象手段

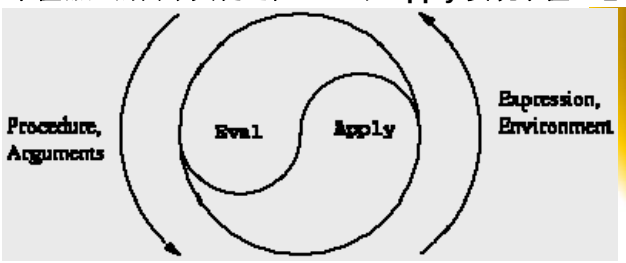
程序设计和语言

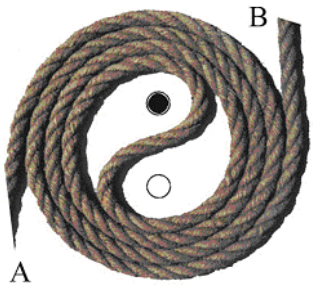
- 本章研究如何实现新语言。将求值器实现为 Lisp（Scheme）过程。Lisp 符号处理能力特别强大，很适合做这种工作。下面做的事情有：
- 实现一个 Lisp 求值器。它包含 Lisp 的主要功能，足以求值本书的大部分程序。该求值器有普遍意义，各种编程语言处理器里都包含一个类似的求值器
- 实现一个正则序求值的 Lisp 求值器。由于求值器是一个过程，改变求值方式的工作量不大。采用正则序，就可以用表实现流的功能
- 设计和实现一个非确定性计算语言，其中表达式可以有多个值，通过某种搜索求出它们的值。这种语言里的计算进程好像能分叉，维护多重轨迹的工作由求值器完成
- 实现一个逻辑程序设计语言，使人可以用关系的形式表达与与计算相关知识，而不是按函数观点写出计算过程。这个语言与 Lisp 很不一样，但其求值器还是可以享用 Lisp 求值器的结构

元循环求值器

- 用 Lisp 做 Lisp 求值器，让它在 Lisp 里运行，像是循环定义。这可行，求值就是一种计算，可以用任何足够强的语言实现，包括 Lisp
- 用一种语言实现其自身的求值器，称为元循环。这个求值器本质上就是实现了前面定义的 Scheme 求值模型
- 回忆一下求值的状态模型：
  - 求值组合式（非特殊形式）时，先求值其子表达式，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
  - 复合过程应用于一集实参就是在一个新环境里求值过程体。新环境是过程的环境加一个新框架，其中是所有形参与对应实参值的约束
- 这两步都可能递归应用。直到遇到
  - 符号(直接到环境里取值)
  - 基本过程 (直接调用代码)
  - 本身就是值的表达式（如数）

求值器里的两个关键过程 eval 和 apply 实现求值，它们的相互关系见图





元循环求值器

- 求值器的实现依赖于一些处理被求值表达式的语法过程
- 下面采用数据抽象技术，使求值器独立于语言的具体表示。如
  - assignment? 检查是否赋值表达式，而不直接判断是否 set!
  - assignment-variable 和 assignment-value 取其中的部分
- 还有一些与过程和环境有关的抽象接口过程，如
  - make-procedure 构造复合过程
  - lookup-variable-value 取变量的值
  - apply-primitive-procedure 应用基本过程
- 元循环求值器的核心是过程 eval 和 apply。它们都需要处理多种情况
  - 这里将始终采用数据抽象技术，用抽象谓词判断表达式等的类型
  - 保证两个核心过程的实现独立于表达式的具体形式

核心过程 eval

eval 以一个表达式和一个环境为参数，分情况求值各种表达式

- 基本表达式：
  - 各种自求值表达式（如数）：直接将其返回
  - 变量：从环境中找出它的当前值
- 特殊形式：
  - 引号表达式：返回被引表达式
  - 变量赋值或定义：修改环境，建立或修改相应约束
  - 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；等等
- 组合式（过程应用）：递归求值它的各子表达式，最后把得到的过程和所有实际参数送给 apply，要求执行过程应用

核心过程 eval

```
(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))))
```

这里用分情况处理的实现方式

完全可用数据导向的编程方式，使增加表达式类型更方便

许多实际系统采用后一方式

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。两种情况：
  - 基本过程：用 apply-primitive-procedure 直接处理
  - 复合过程：建立新环境（用形参和实参表建立一个框架），而后在新环境里顺序求值过程体里的表达式（允许多个表达式）

```
(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))))
```


过程参数，if 表达式

- 对过程调用，eval 需要求值各实参表达式，得到实参表：

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
```

  也可用 map。上面做法说明求值器可以在无高阶过程的语言里实现

- eval-if 求值条件表达式：

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

  true? 把 eval 条件表达式 的结果翻译到实现语言 (Scheme) 的逻辑值。这样做，元循环求值器的逻辑值就可以用任何表示形式，完全可以和 Scheme 里的表示形式不同

赋值和定义，序列

- 赋值和定义的实现方法类似，修改环境的工作留给下层过程：

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

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

- apply 求值过程体的表达式序列，就是在同一环境里逐个求值它们：

```
(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))))
```

表达式表示

- 符号表达式（Lisp 程序）具有递归结构，求值器根据表达式类型确定操作。表达式看作数据抽象，可松弛操作规则和表达式形式的联系
- 下面实现各种表达式的语法过程
- 自求值表达式。只需要一个谓词，数和字符串属于此类：

```
(define (self-evaluating? exp)
  (cond ((number? exp) true)
        ((string? exp) true)
        (else false)))
```

- 变量。直接用符号表示，只需要一个谓词

```
(define (variable? exp) (symbol? exp))
```

- 区分各种表达式需要判断类型标志，需要下面过程

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


表达式表示： quote，赋值

- 表达式形式 (quote *<text-of-quotation>*)，相应语法过程和谓词:

```
(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))
```

- 表达式形式 (set! *<var>* *<value>*)，语法过程:

```
(define (assignment? exp) (tagged-list? exp 'set!))
(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))
```


表达式表示：定义

- 定义有两种形式：(define *<var>* *<value>*) 和
  (define (*<var>* *<parameter$_1$>* ... *<parameter$_n$>*) *<body>*)

  后一形式是下面形式的语法包装：

  (define *<var>* (lambda (*<parameter$_1$>* ... *<parameter$_n$>*) *<body>*))

  语法过程：

```
(define (definition? exp) (tagged-list? exp 'define))
(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)   ; formal parameters
                   (cddr exp)))) ; body
```


表达式表示：lambda，if

- lambda 表达式在形式上是一个表，以 lambda 为第一个元素：

```
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))
```

  构造函数（实现 define 需要）：

```
(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

- if 表达式的语法过程：

```
(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))
```

  if 表达式的构造函数（实现 cond 时要用）

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

表达式表示：begin

- begin 包装一系列表达式，要求顺序地对它们求值：

```
(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions exp) (cdr exp))
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))
```

需要构造函数 sequence->exp (用于 cond->if)，它把多个表达式的序列包装为一个 begin 表达式：

```
(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
(define (make-begin seq) (cons 'begin seq))
```


表达式表示：过程应用

- 另一些语法过程：

```
(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))
```

- 至此基本表达式的语法过程全部定义完成
- 语言里还有一些派生表达式类型，包括 cond 等
  - 下面考虑它们的实现，实际上是把它们翻译为基本表达式

派生表达式：cond

- cond 总可以用嵌套的 if 表达式实现。对 cond 的求值变换为 if

```
(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))
(define (cond-else-clause? clause) (eq? (cond-predicate clause) 'else))
(define (cond-predicate clause) (car clause))
(define (cond-actions clause) (cdr clause))
(define (cond->if exp) (expand-clauses (cond-clauses exp)))

(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))))))
```

还可以定义其他的派生表达式

本小节的练习中提出了许多问题

求值器数据结构：谓词和过程

- 为完成表达式处理，还需要定义好过程和环境的表示形式，逻辑值等的表示方式，这些都属于求值器的内部数据结构
- 谓词检测。把所有非 false 对象作为逻辑真

```
(define (true? x) (not (eq? x false)))
(define (false? x) (eq? x false))
```

- 设 (apply-primitive-procedure *<proc>* *<args>*) 处理基本过程应用，(primitive-procedure? *<proc>*) 检查基本过程。复合过程的处理：

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? p)
  (tagged-list? p 'procedure))
(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))
```

求值器数据结构：环境操作

- 求值需要环境，现在考虑环境的实现
- 环境是框架的序列，每个框架是一个表格，其中的项就是变量与值的约束。将环境看着数据抽象，提供下面操作：
  - (lookup-variable-value <*var*> <*env*>) 取得符号 <*var*> 在环境 <*env*> 里的约束值，没有约束时报错
  - (extend-environment <*variables*> <*values*> <*base-env*>) 返回所构造的新环境，其第一个框架里包含 <*variables*> 与 <*values*> 的关联，外围环境是 <*base-env*>
  - (define-variable! <*var*> <*value*> <*env*>) 在环境 <*env*> 的第一个框架里加入 <*var*> 与 <*value*> 的关联
  - (set-variable-value! <*var*> <*value*> <*env*>) 修改环境 <*env*> 里变量 <*var*> 的约束，使其关联值变成 <*value*>。找不到 <*var*> 的约束时报错
- 在此基础上，考虑实现环境的数据结构

求值器数据结构：框架

- 环境用框架的表表示，其 cdr 是外围环境

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

- 框架是表的序对，其 car 是变量表，cdr 是关联值表

```
(define (make-frame variables values) (cons variables values))
(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))))
```

- 扩充环境时在前面增加一个框架。变量表元素与值表长度不同时报错

```
(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))))
```


求值器数据结构：变量检索

- 变量取值：顺序检查环境中的框架，找到第一个出现时返回值表中与变量对应的元素。遇到空环境时报错

```
(define (lookup-variable-value var env0)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((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 env0))
```


求值器数据结构：变量赋值

- 变量赋值。修改环境中变量第一个出现的关联值。找不到变量时报错：

```
(define (set-variable-value! var val env0)
  (define (env-loop env)
    (define (scan vars vals)
      (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 env0))
```

求值器数据结构：变量定义

- 定义变量。在第一个框架里加入该变量与值的关联。如果存在该变量的约束时修改与之关联的值

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (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))))
```

- 同样的功能可能用不同方法实现，采用不同数据结构和算法。上面是一种朴素的简单实现，许多问题没有慎重考虑，特别是效率
  - 程序执行中频繁检索变量（求值/赋值/定义等），检索效率很关键
  - 在框架里顺序比较，在框架序列中顺序检索的效率太低。实际求值器需要采用精心设计的结构和技术

求值器的运行

- 求值器给出了Lisp 求值模型的严格描述（用 Lisp 本身描述）
- 这样描述的求值器可以运行，可以帮助理解语言；还可以改造它去试验不同求值规则。下面考虑这些方面的问题
- 表达式求值最终归结到基本过程调用。运行求值器之前要解决调用基本 Lisp 功能的问题。每个基本过程要有一个约束，使 eval 求值中能找到相应过程对象并把它传给 apply。为此要建立一个初始环境，其中为每个基本过程名建立对象关联，还需包含 true 和 false 等的约束
- 创建初始环境：

```
(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 the-global-environment (setup-environment))
```


求值器的运行

- 基本过程的表示形式不重要，但 apply 需要识别它们，并能借助相应过程去应用它们。这里让它们都以符号 primitive 开头

```
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

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

(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
       <more primitives>
       ))
(define (primitive-procedure-names)
  (map car primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))
```


求值器的运行

- 应用基本过程时，可以直接利用基础 Lisp 系统将它们应用于参数

```
(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))
```

- 注意：apply-in-underlying-scheme 也就是 Scheme 系统的 apply

  元循环求值器重新定义了 apply，掩盖了系统原有的定义

  应该在重新定义前为原来的 apply 建立另一个引用

```
(define apply-in-underlying-scheme apply)
```

  这就使过程 apply-primitive-procedure 可以调用基础系统的 apply 基本过程

求值器的运行

- 为方便使用，可仿照 Scheme 系统，给元循环求值器定义一个基本循环，输出提示符后读入-求值-打印。输出前加一个特殊标志：

```
(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 the-global-environment)))
       (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))
```


求值器的运行

- 定义 user-print 过程是为了避免打印复合过程的环境

  这种环境可能有复杂的结构，而且可能包含循环结构

  打印它可能出问题

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


运行实例

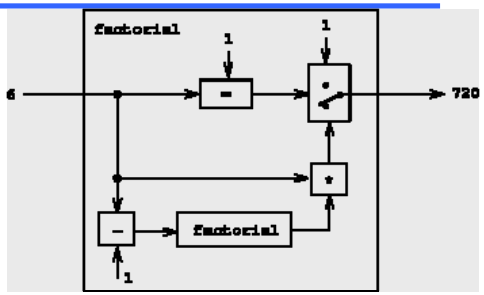
- 一段运行实例：

```
(define the-global-environment (setup-environment))
  (driver-loop)
  ;;; M-Eval input:
  (define (append x y)
    (if (null? x)
        y
        (cons (car x) (append (cdr x) y))))
  ;;; M-Eval value:
  ok
  ;;; M-Eval input:
  (append '(a b c) '(d e f))
  ;;; M-Eval value:
  (a b c d e f)
```


以数据作为程序

- 考虑用 Lisp 程序求值 Lisp 表达式的问题
- 按操作的观点，程序是一部（可能为无穷大的）抽象机器的一个描述。例如：

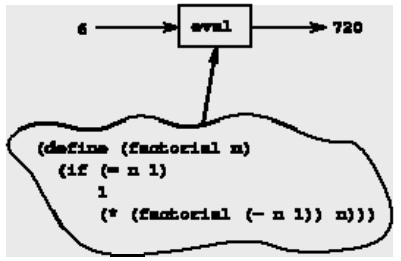
```
(define (factorial n)
  (if (= n 1)
     1
     (* (factorial (- n 1)) n)))
```



阶乘机器是一部递归机器，由一些基本部件和开关等构成

同样，求值器是一部特殊机器。对给定输入（一部机器的描述），它能模拟该输入所描述的机器的行为

求值器的功能是“表演”任意机器的行为，它是一部“万能表演机器”



以数据作为程序

- 也就是说，求值器是一部通用机器，它能模拟任何用 Lisp 程序描述的机器的行为。第一个清晰地提出这一思想的人是图灵
- 想想：有没有可能在其他领域中实现这种具有通用功能的东西？例如：电子线路？汽车？机床？催化剂？书？药？有可能吗？
- 一个事实：求值器也可以模拟它自己！(当然要付出效率的代价)
- 求值器是联系数据对象和编程语言的桥梁。在其运行过程中，人输入的表达式是被求值程序，求值器把这种表达式当作数据（当作一个表），按特定规则去操作这个表
- 把用户程序当作求值器的数据，不仅没带来混乱，还可能带来方便

  Lisp 系统通常都把 eval 作为基本函数，允许直接调用自己的求值器

  (eval '(* 5 5) user-initial-environment)

  (eval (cons '* (list 5 5)) user-initial-environment)

  都会返回 25。这就使程序可以在运行中构造程序，然后去求值它们

内部定义

- 元循环求值器顺序执行收到的一个个定义，把定义加入环境框架。这符合交互式开发的需要，因为程序员的定义和使用工作常常交替进行
- 但这可能不是处理块结构的内部定义的最好方式。例如

```
(define (f x)
  (define (even? n)
    (if (= n 0)
        true
        (odd? (- n 1))))
  (define (odd? n)
    (if (= n 0)
        false
        (even? (- n 1))))
  <rest of body of f>)
```

even? 里的 odd? 是后面定义的过程（此时还没定义）。可见 odd? 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

内部定义

- 我们的求值器“恰好”能正确处理这种情况，因为它总在处理了所有定义后才用它们。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）
- 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。如：

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

变换为：

```
(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
- 本节练习讨论了许多与定义有关的语义和其他问题。请自己看看

分离语法分析和执行

- 前面求值器很简单，但效率很低。主要是表达式的语法分析和执行交织在一起，如果一个表达式执行多次，就要做多次语法分析。如：

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

求 (factorial 4) 时，每次调用都要确定过程体是否为 if，而后提取谓词部分并基于其值继续求值，等等。求值子表达式时 eval 又要做分情况处理。这种分析的代价很高，没必要重复做

- 提高效率的一种方式：修改求值器，重新安排处理过程，使表达式分析只做一次。具体做法是把 eval 的工作分为两部分：
  - 过程 analyze 对被求值表达式进行语法分析，返回一个执行过程，将分析结果封装在其中
  - 执行过程以环境作为参数实际执行，产生表达式求值的效果。这样就只需要做一次分析，可以多次执行

分离语法分析和执行

- 分析和执行分离后，eval 变成：

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

- 过程 analyze 像 eval 一样做分情况分析，但不做完全的求值，只是构造一个可以直接执行的程序（具体构造调用子程序完成）：

```
(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))))
```


分离语法分析和执行

- 注意：现在要把分析各种表达式，做出相应的执行过程

  这种过程以环境作为参数，产生表达式执行的效果

- 生成自求值表达式对应的过程

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

- 在分析引号表达式时直接取出被引表达式，就不必每次求值时做了：

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

- 取变量值，仍是在执行时到环境里查找：

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


分离语法分析和执行

- 赋值和定义都需要到求值时做，需要设置变量（需要环境）。先做了被赋值表达式的分析，可以大大提高执行效率：

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

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


分离语法分析和执行

- if 表达式，分析中提取谓词和两个分支表达式

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

- lambda 表达式，体的分析只做一次，多次执行可能大大提高效率

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

make-procedure 继续使用前面的定义

分离语法分析和执行

- 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))))
```

把前两个子表达式的执行过程组合起来

分析各子表达式

- 过程应用的分析最复杂，需要分析其中运算符和运算对象，每一个生成一个执行过程，而后把它们送给 execute-application 过程（与 apply 对应），要求它执行这个过程

分离语法分析和执行

分析各运算对象

- 过程应用的分析

```
(define (analyze-application exp)
    (let ((fproc (analyze (operator exp)))
          (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))))
```

完成过程应用

执行各运算对象的执行过程，得到实参值表

总结：元循环求值器

- 有时需要自己设计和实现专用的语言
- 基于语言的封装是封装的最高级形式，这样做可能
  - 提供最合适的基本操作，组合方式和抽象手段
  - 为解决特定（领域的）问题提供最大方便
- 用一个语言写出的本语言的解释器称为元循环解释器
- Scheme 解释器（系统的，自己写的），核心是两个过程
  - eval 在一个环境里的求值一个表达式
  - apply 将一个过程对象应用于一组实际参数
- 实现元循环解释器，可以用分情况分析技术，或者数据导向技术
- 提高求值器效率的一种重要想法是把分析和执行分离
  - 构造执行过程，也就是基于原程序构造新的更高效的程序
  - 这是一种优化