

4. 元语言抽象(I)

本章研究语言的设计和实现

- 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
- 一个非确定性计算语言，一个逻辑编程语言

程序设计技术和方法

裘宗燕, 2011-2012 / 1

程序设计和语言

- 前面讨论了一些程序设计技术，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，前面一直用 **Lisp** (方言 **Scheme**) 作为编程语言
- 如果问题变得更复杂，或需要解决某领域的大量问题，就可能发现手头可用的语言（**Lisp** 或其他）都不够满意，不够方便
 - 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，就需要自己设计和实现一个语言
- 设计一个好的适用的语言很不容易，不应该轻易去做
 - 但建立新语言也是控制复杂性最有力的策略
 - 在适当（或者不得已）的情况下应该考虑

程序设计技术和方法

裘宗燕, 2011-2012 / 2

程序设计和语言

- 针对具体问题 (问题领域) 设计的专门语言
 - 有可能提供一套最适用的原语、组合方式和抽象方式
 - 使人能以最有效的方式描述要处理的问题
 - 大大提高在一定范围内处理复杂情况的能力
 - 这些情况说明, 做语言设计师也是软件工作者的一项可能工作 (至少应注意这种可能性)
 - 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, 有的很复杂
 - 最简单的如 **C** 标准库 **printf** 的格式描述串 (一种简单排版语言)
 - 操作系统命令语言, **HTML** 语言 (一种排版语言)
 - 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

程序设计和语言

- 我们不仅应该考虑根据需要设计语言, 还能通过构造解释器实现它
 - 语言解释器也是一个过程。将它应用于相应语言的一个表达式, 它就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: 语言的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, 都是提供某种完整语言
 - 完整语言都有基本原语、组合手段和抽象手段
- 本章研究如何实现新语言
 - 将所需的语言求值器实现为一个 (一组) **Lisp (Scheme)** 过程
 - **Lisp** 语言有强大的符号处理能力, 特别适合用于做这种工作
- 学习求值器的实现, 可以了解语言实现中的许多问题。求值器是复杂的程序, 构造求值器中开发的技术可能用到许多其他地方

本章将研究的问题包括：

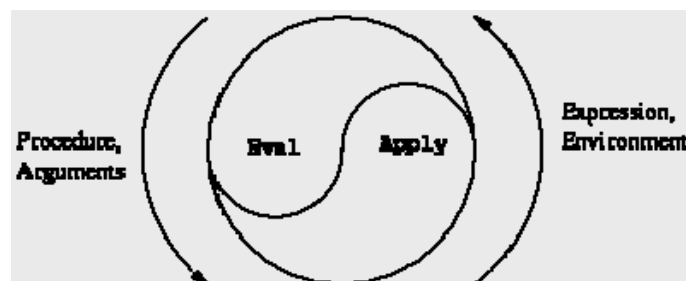
- 实现一个 **Lisp** 求值器
 - 它包含 **Lisp** 的主要功能，足以求值本书里的大部分程序
 - 该求值器有普遍意义，各种编程语言处理器都包含类似的求值器
- 实现一个正则序求值的 **Lisp** 求值器
 - 由于求值器是一个过程，改变求值方式的工作量不大
 - 采用正则序求值，就可以用表实现流的功能
- 设计和实现一个支持非确定性计算的语言，其中表达式可以有多个值，通过某种特定的搜索过程求出它们的值。这种语言里的计算进程好像能分叉，维护多重轨迹的工作由求值器自动完成
- 实现一个逻辑程序设计语言，使人可以用关系的形式表达与计算相关的知识，而不是按函数观点写出计算过程。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

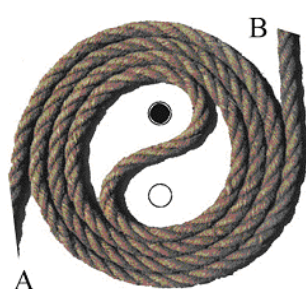
- 用 **Lisp** 做 **Lisp** 求值器后让它在 **Lisp** 里运行，像是循环定义。这件事可行：求值也是计算，可以用任何足够强的语言实现，包括 **Lisp**
- 用一种语言实现其自身的求值器，称为元循环。下面做的这个求值器实际上就是实现了前面定义的 **Scheme** 求值的状态模型
- 回忆一下求值的状态模型：
 - 求值组合式（非特殊形式）时，先求值其子表达式，而后把运算符子表达式的值作用于运算对象子表达式的值
 - 把复合过程应用于实参就是在一个新环境里求值过程体。新环境是过程的环境加一个新框架，其中是所有形参与对应实参的约束
- 在这两个求值步骤里都可能出现递归应用。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）

关键过程 eval 和 apply

- 实现求值器两个核心步骤的是两个关键过程 **eval** 和 **apply**
 - **eval** 负责表达式求值，**apply** 负责过程的应用
 - 两个过程相互递归调用，**eval** 工作中也可能递归调用自身



程序设计技术和方法



袁宗燕, 2011-2012 / 7

元循环求值器

- **eval** 和 **apply** 的工作中都需要处理多种不同情况
 - 下面实现中将始终采用数据抽象技术
 - 求值器的工作基于一些处理被求值表达式的语法过程来实现
 - 基于抽象接口过程，使求值器的实现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
- 下面考虑 **eval** 和 **apply** 的实现

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）：递归地求值组合式的各个子表达式，而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
 - 把表达式看作数据抽象，松弛了操作规则和表达式形式的联系

现在考虑各种表达式的语法过程的实现

- 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)))
```

可以根据所需常量的种类进一步扩充

- 变量。直接用符号表示，只需要一个谓词

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

表达式表示: quote, 赋值

- 区分各种表达式需要判断类型标志, 用下面过程

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

- 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> <parameter₁> ... <parameter_n>) <body>)

后一形式也就是下面形式的语法包装:

```
(define <var> (lambda (<parameter1> ... <parametern>) <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? (cddddr exp)))  
      (cddddr 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
- 在这些设计的基础上，下面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）
 - 运行构造好的求值器
 - 可以帮助理解语言
 - 还可以进一步考虑去改造它以试验不同的求值方式和规则
- 下面考虑这些方面的问题
- 表达式求值，最终将归结到对基本过程的调用
 - 运行求值器之前要解决调用基本 **Lisp** 功能的问题
 - 需要为每个基本过程建立一个约束，使 **eval** 在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** 等符号（名字）的约束

求值器的运行

- 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))
```
- 基本过程的具体表示形式并不重要，关键是使 **apply** 能够识别这一类别，而后通过相应的过程去完成工作
 - 这里为基本过程建立专门的抽象形式
 - 用一个基本过程表 **primitive-procedures** 记录基本过程名到其定义的约束，借助它建立初始环境

求值器的运行

- 基本过程对象以符号 **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** 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

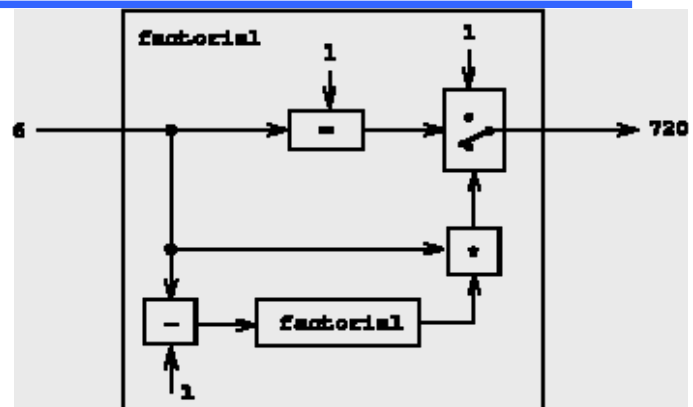
■ 一段运行实例:

```
(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:

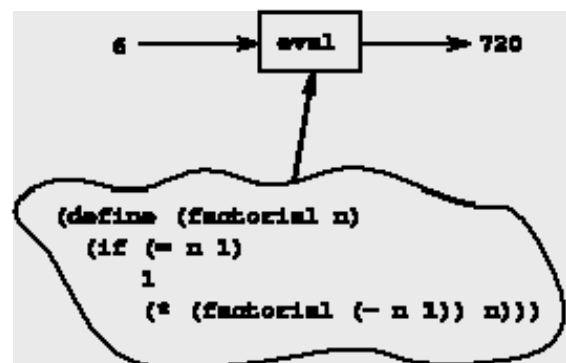
```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```



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

- 这里的求值器“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

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