4. 元语言抽象(1)

本章研究语言的设计和实现

- 语言抽象
  - 程序设计（编程）和语言
  - 语言抽象的意义和方法
- 元循环求值器（用 **Scheme** 写的 **Scheme** 解释器）
  - 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

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，构造功能更复杂的结构
  - 如何将复杂结构抽象为高层构件，方便进一步组合
  - 如何采用一些高层次的观点和组织方式，提高系统的模块性

  都是编程的问题（编程：用一种语言写程序）

  一直用 **Scheme**（一种**Lisp**方言）作为编程语言
- 如果遇到的问题更复杂，或者需要解决某领域的大量问题，有可能发现现实可用的语言（**Lisp**，或其他）都不够满意或不够方便
  - 可能看到需要解决的问题有一些共性和特殊需要
  - 最好是有能最有效表述与问题相关的想法的语言
  - 如果现成语言都不太适用，就需要自己设计和实现新语言
  - 通过语言设计，提供语言层抽象，可能获得更好的模块化

程序设计和语言

- 设计一个好的适用语言，很不容易
  - 必须对面临问题有比较深入的认识和总结
  - 设计和实现语言的工作量很大，不应该轻易去做
  - 但是，建立适用的新语言是控制系统复杂性的最有力的策略

    在适当（或者不得已）的情况下应该考虑
- 针对具体问题 (或问题领域) 设计的专门语言
  - 可能提供一套专用的最适用的原语、组合方式和抽象方式
  - 使开发者能以最有效的方式描述要处理的问题
  - 使应用领域的专家有可能直接参与应用系统开发
  - 可能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，软件开发者有时也可能需要做语言的设计和实现

  至少说，应该注意到这种可能性

程序设计和语言

- 从事程序设计和系统开发的工作，会涉及到许多不同层次的语言（应该注意这个问题），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）
  - 服务于各种专门或一般用途的脚本语言
  - 最复杂的是高级编程语言
- 还有许许多多面向专门应用领域的语言
  - 操作系统命令语言
  - HTML，XML，网页描述和“排版”
  - Mathematica，Maple，数学计算（符号计算）
  - Matlab，数值计算和模拟应用
  - Coq，PVS， 形式化模型和定理证明（计算机科学，数学）
  - R，统计计算和应用；等等

程序设计和语言

- 如果设计了语言，就要考虑语言的“实现”问题
  - 计算机工作者设计语言，主要是为了让计算机能根据用语言描述的指示完成一些工作，为了用这些语言构造应用系统
  - “实现”一个语言，就是让这个语言可以实际地在计算机上可用。常见方法是开发一个能处理该语言的解释器
- 语言的解释器自身也是一个过程
  - 送给它相应语言的一个表达式（一段完整描述，一个“程序”），它就会完成该表达式要求做的相应动作
  - 解释器“理解”这个语言，理解语言中各种结构的意义
- 实现语言，也是程序设计本质思想的典型体现：
  - 语言的求值器（在某种意义下）定义了语言里各种表达式的意义
  - 它自身也就是一个程序
- 理解语言和语言的解释器，能帮助进一步理解程序设计本身

程序设计和语言

- 许多复杂程序可以看作某种语言的解释器（求值器）
  - 前面逻辑电路模拟器和约束传播系统，都提供了某种完整语言
  - 完整的语言有自己的基本原语、组合手段和抽象手段
- 本章研究语言的实现问题
  - 要考虑（几个）语言的求值器，将其分别实现为 **Scheme（Lisp）** 过程（实现为一组相互关联的过程）
  - **Scheme** 语言有强大的符号处理能力，特别适合用于做这种工作
- 学习求值器实现，有助于理解语言本身和语言实现中的问题
  - 求值器是很复杂的程序，实现复杂的功能。有关工作
    - 需要正确地实现这些功能
    - 需要有良好的程序组织
  - 在构造求值器中开发的技术和方法可能用到许多其他地方

程序设计和语言

- 本次课研究如何实现一个 **Scheme** 求值器
  - 它包含 **Scheme** 的主要功能，能求值本书里的大部分程序
  - 这里要研究
    - 基本求值过程的实现，**apply** 和 **eval**
    - 求值器数据结构
    - 内部表示的处理，等等
  - 这一求值器有普遍意义

    很多语言处理器都包含类似的求值器

本章后面还要实现几个语言解释器

- 实现一个采用正则序求值 **Scheme** 表达式的求值器
  - 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

程序设计和语言

- 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
  - 通过某种特殊搜索过程求出它们的值
  - 用这种语言描述的计算进程好像能分叉
    - 在一个分叉点，计算进程可以进入其任何一个分支。在任何一个分支完成计算都是这个程序完成计算
    - 维护多条计算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
  - 这个语言的结构和语义与 **Scheme** 差别极大

    但其求值器仍采用了 **Scheme** 求值器的基本结构

元循环求值器

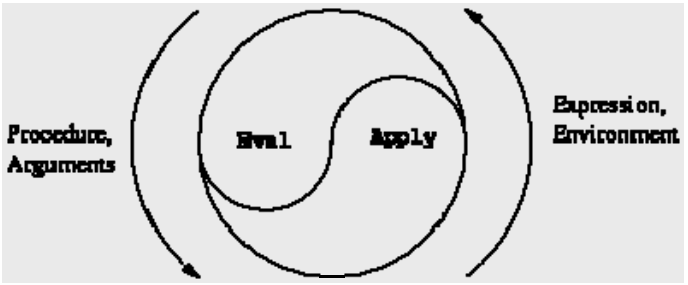
- 用 **Scheme** 做一个 **Scheme** 求值器，而后在 **Scheme** 系统（的支持）下运行它。这一做法像是循环定义
  - 但实际可行：**Scheme** 表达式的求值也是一种计算。如果一个语言功能足够强，就可以用于实现这种计算
  - 各种编程语言的能力都足够强，都能用于实现 **Scheme** 解释器。当然，由于不同语言的特点，做这件事的方便程度可能不同
  - **Scheme** 语言能力足够强，可以用它实现 **Scheme** 语言解释器
  - 表达式求值也是一些符号操作，**Scheme** 和其他 **Lisp** 方言都特别适合做这种操作
- 用一种语言实现其自身的求值器，称为元循环（**meta-circular**）
  - 下面将要做的求值器基于前面讨论过的 **Scheme** 程序求值的状态模型，这个求值器可以看作该模型的一个实现
  - 通过这个求值器，可以看清 **Scheme** 程序求值的更多细节

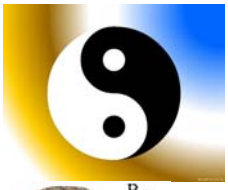
元循环求值器

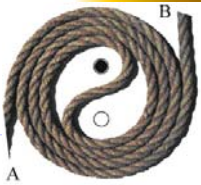
- 复习求值的状态模型：
  - 求值组合式（非特殊形式）时
    - 先求值组合式的各子表达式
    - 而后把运算符子表达式的值作用于运算对象子表达式的值
  - 把复合过程应用于实参，是在一个新环境里求值过程体
    - 新环境：过程对象（里环境指针指向）的环境加一个新框架
    - 新框架里是过程的形参与对应实参的约束
- 两个求值步骤都可能递归（自己递归或相互递归。求值的子表达式可能要应用复合过程，过程体本身通常又是组合式），直到遇到
  - 符号（直接到环境里取值）
  - 基本过程（直接调用基本过程的代码）
  - 本身就是值的表达式（如数，直接用其本身）

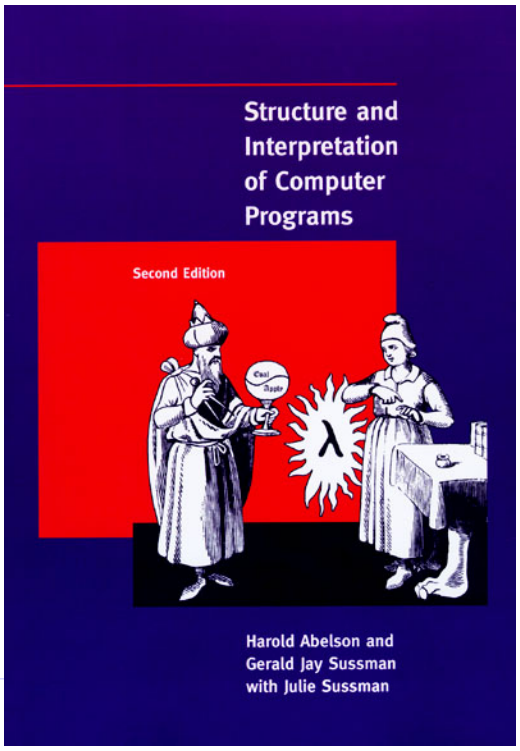
关键过程 eval 和 apply

- 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** 负责过程应用
  - 相互递归调用，**eval** 还有自递归









元循环求值器

- 过程 **eval** 和 **apply** 在工作中都要处理许多不同情况
  - 实现中始终采用数据抽象技术
  - 求值过程基于一些针对不同类型的表达式的抽象语法过程
  - 定义都基于接口过程，求值器实现与语言的具体表示无关
- 例如，对赋值表达式
  - 用 **assignment?** 检查是否赋值表达式，而不直接判断是否 **set!**
  - 用 **assignment-variable** 和 **assignment-value** 取其中的部分
- 有几个与过程和环境有关的抽象接口过程，如
  - **make-procedure** 构造复合过程（构造过程对象）
  - **lookup-variable-value** 取变量的值（在环境中检索）
  - **apply-primitive-procedure** 应用基本过程
- 先考虑核心过程 **eval** 和 **apply** 的实现

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** 分情况求值

- 基本表达式：
  - 各种自求值表达式（如数等）：直接将其返回作为结果
  - 变量：从环境中找出它的当前值
- 特殊形式：
  - 引号表达式：返回被引的表达式
  - 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子表达式，然后把得到的过程和所有实参送给 **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))))
```

这里采用分情况处理的方式

完全可以用数据导向技术，使增加表达式类型的工作更方便

许多实际系统采用数据导向技术

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

- 对于过程调用，eval 先求出各实参表达式的值，得到实参表：

  eval 在当前环境里求值这些表达式

  语言没有规定表达式的求值顺序，可以选择最方便的方式，但作为求值结果的表必须保证正确顺序

  顺序实现给这个求值强加了一种顺序。也可以考虑并行求值

- 实现很简单：

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

  也可以用 map 完成这里的工作

  这里用的方法说明，没有高阶过程的语言也足以开发求值器

if 表达式

- **eval-if** 求值条件表达式：
  - 首先求值条件部分
  - 而后根据条件部分的值选择被求值的子表达式
- 实现：

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

  - 过程 **true?** 把条件表达式求值的结果翻译到实现语言 **(Scheme)** 里的逻辑值。这样做之后
    - 元循环求值器的逻辑值就可以用任何表示形式
    - 完全可以采用与 **Scheme** 不同的表示形式

赋值和定义

- 赋值和定义的实现方法类似，把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)
```

- 这两个过程都先求出用于赋值/定义的表达式的值

  取得被操作的变量名和被求值表达式的方法都是抽象的

  赋值/定义的实际动作（如何修改环境）分别由 **set-variable-value!** 和 **define-variable!** 描述，也是抽象的

序列（表达式）

- 在一些地方需要求值一系列表达式（表达式序列），并以最后一个表达式的值作为整个表达式序列的值。包括
  - **lambda** 体（过程体）
  - **begin** 表达式
  - **cond** 条件之后的表达式
- 求值表达式序列，就是在同一个环境里逐个求值序列里的表达式：

```
(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))))
```

  - 前面子表达式的值，在求出之后都直接舍弃
  - 这个实现不允许空序列。如果允许空序列，可以修改过程定义，还需要为空序列确定一个默认值

表达式表示

- 符号表达式（**Scheme** 程序）具有递归的结构
  - 求值器根据遇到的表达式的类型确定操作
  - 把表达式作为数据抽象，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。只需要定义一个谓词，数和字符串属于此类：

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

  可以根据语言中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 和赋值

- 其他表达式通过判断类型标志加以区分，定义判断标志的通用过程

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

- 引号表达式 (quote <*text-of-quotation*>) 的语法过程:

```
(define (quoted? exp) (tagged-list? exp 'quote))

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

- 赋值表达式 (set! <*var*> <*value*>) 的语法过程:

```
(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

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

  后一形式实际上是下面形式的定义表达式的语法包装：

  (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

- lambda 表达式在形式上是一个表，以 lambda 为第一个元素
- 语法过程

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))
```

- 定义一个构造函数（下面实现 define 时需要用）：

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

构造出形式正确的 lambda 表达式

表达式表示：if

- 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，要求对它们顺序求值：

```
(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))
```

- 需要一个构造函数 sequence->exp（用在下面的 cond->if 里），它把包含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

  完全可以直接实现，与前面做法类似

  下面采用的实际做法是把它们翻译成基本表达式

派生表达式：cond

- cond 总可以用嵌套的 if 表达式实现。把对 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

- 为了实现对各种表达式的处理，还需要定义好过程和环境的表示形式，逻辑值等的表示形式，这些都属于求值器的内部数据结构
- 谓词检测。把所有非 false 对象都当作逻辑真

```
(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

- 解释器求值时需要有一个环境，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。把环境定义为数据抽象，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

求值器数据结构：框架

- 下面考虑实现环境的数据结构
- 环境用框架的表表示，其 **cdr** 是外围环境

```
(define (first-frame env) (car env))

(define (enclosing-environment env) (cdr 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))))
```

求值器数据结构：框架

- 环境扩充
  - 建立一个框架
  - 将其放在给定环境的前面
  - 返回扩充后的环境
- 实现：

  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：顺序检查环境中的各框架
  - 找到变量的第一个出现时，返回值表中与变量对应的元素（值）
  - 遇到空环境时报错（已经无环境可查，变量无定义）
- 实现：

```
(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
  - 修改环境里变量的第一个出现的关联值
  - 找不到变量时报错
- 实现：

```
(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的方法实现，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。在做实际的求值器时，需要采用精心设计的数据结构和技术

求值器的运行

- 这个求值器给出了 **Scheme** 求值模型的一个严格描述（用 **Scheme** 语言本身描述）。请仔细对比前面讨论的求值的状态模型
- 运行构造好的求值器
  - 可以帮助理解语言
  - 还可以进一步考虑改造它，试验不同的求值方式和规则

  后面会考虑第二个问题
- 对表达式求值，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 **Scheme** 功能的问题
  - 必须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**
  - 为此需要建立一个初始环境，在其中为每个基本过程名建立对象关联，还需要包含 **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不重要，关键是 **apply** 能识别这一类别，而后通过调用相应的过程去完成工作
  - 这里为基本过程建立一种专门的抽象形式
  - 建立基本过程表 **primitive-procedures**，记录基本过程名到其定义的约束，用它建立初始环境

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

- 需要应用基本过程时，直接通过基础 **Scheme** 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：

  上面的 **apply-in-underlying-scheme** 就是 **Scheme** 系统的 **apply**

  在元循环求值器里重新定义了 **apply**

  这样做会掩盖系统里 **apply** 的原有定义

  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)

  这样，通过 **apply-primitive-procedure** 就可以调用到基础系统里原来的 **apply** 基本过程

求值器的运行

- 为了使用方便，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

  打印环境的内容可能出问题

```
(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

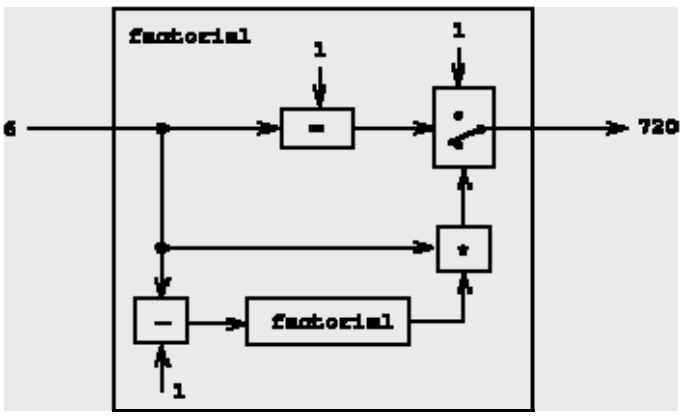
- 一段运行实例：

```
(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

- 现在考虑用 Scheme 程序求值 Scheme 表达式的问题
- 按操作的观点，程序就是一部（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)))
```

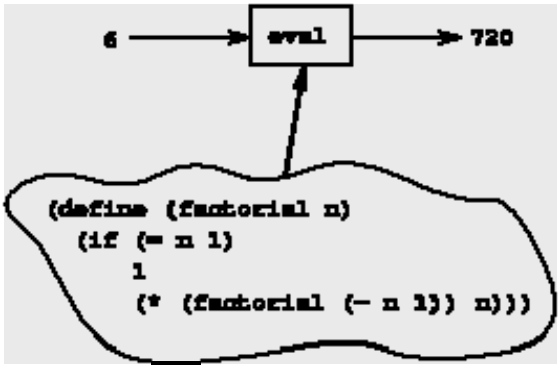


阶乘机器是一部递归机器，由一些基本部件和开关等构成

实际上是无穷大的机器，因为其结构里还包含自己

以数据作为程序

- 同样，求值器是一部特殊机器
  - 其输入应该是一部机器的描述
  - 其功能是模拟该输入所描述的那部机器的行为
- 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”
- 就是说，求值器是一部通用机器，它能模拟用 **Scheme** 程序描述的任何机器的行为。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？



- 电子书
- 音乐播放器，...
- 都属于“专用的通用设备”
- 计算机（一个编程语言就是一种抽象计算机）比它们更进了一步：它能模拟自己

以数据作为程序

- 事实：求值器也可以模拟它自己！

  可以让前面的 **Scheme** 解释器求值这个 **Scheme** 解释器自身

  当然，运行中可能付出效率的代价
- 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），按照一组特定的规则去操作这个表
- 把用户程序当作求值器的数据，不仅不会带来混乱，还可能带来方便
- **Scheme** 系统通常都把 **eval** 作为基本函数，允许直接调用求值器

  (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得到的一个个定义，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）
- 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

- 软件开发工作有可能要设计和实现特殊的语言
- 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，以及自己写的），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