5. 寄存器机器里的计算(2)

本节讨论如何管理计算机的内存以支持表结构的处理，做出能解释 Scheme 程序的寄存器机器：

- 存储管理和废料收集，可看作是在真实计算机的有穷存储器上制造出一种无穷存储的假象
- 用寄存器语言实现求值器是低层次的工作，能揭示 Scheme 程序解释中的许多前面无法涉及的控制细节，包括：
  - 过程调用时参数值的传递和结果返回
  - 尾递归的实现
- 继续用前面求值器的基本数据结构和语法过程

  完全可以用更基本的操作实现它们，做出可以对应到常规高级语言或常规机器语言的求值器

存储分配和废料收集

- 下面讨论怎样做实现 Scheme 求值器的机器
- 为简化讨论，可以假定寄存器机器有一个表结构存储器，表操作都是基本操作。这种抽象使人能集中精力考虑求值器的关键特征
- 但表存储是 Scheme 的基础，不理解它，对系统的理解有缺陷。为完整起见，现讨论怎样在常规计算机内存上实现表存储结构
- 表结构的实现要考虑两个问题：
  - 表示。如何只用典型计算机的存储单元和寻址功能，把序对的“指针盒子”结构映射到常规计算机的连续内存
  - 实现。把管理存储的工作实现为一个过程
- Scheme 程序的执行高度依赖于能随时创建对象，包括
  - 程序里用的序对和其他对象
  - 支持程序执行而隐式创建的对象，如环境、框架和参数表等

存储分配和废料收集

- 如果计算机的存储无穷大，就可以创建任意多的对象
- 但实际计算机的存储总有限，因此系统需要一种自动机制
  - 用有限存储制造一种无穷假象
  - 当已分配的对象不再需要时自动将其回收。这就是自动废料收集
- 常规计算机的内存是一串很小的单元
  - 每个单元里可保存一点信息
  - 有一个唯一的名字称为地址
  - 典型操作：取特定单元的内容和给单元赋新值
  - 通过地址增量操作可以顺序地访问一批单元
- 有些操作要求把地址作为数据
  - 将其存入内存单元，或在寄存器里对地址做各种运算
  - 表处理是地址运算的典型实例

存储作为向量

- 为了模拟计算机内存，下面用一种称为向量的新数据结构
- 向量是一种复合数据对象，其元素可通过整数下标访问，访问所需时间与元素位置无关。用两个过程描述向量操作：

  (vector-ref <*vector*> <*n*>) 返回向量里的第*n*个元素

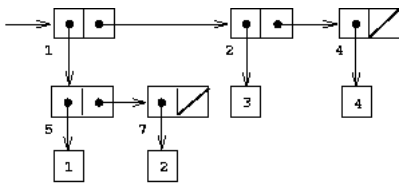
  (vector-set! <*vector*> <*n*> <*v*>) 将向量里第*n*个元素设置为 <*v*>

  - 对计算机内存单元的访问可以通过地址算术实现（用向量基址加特定元素的偏移量）
- 向量是计算机内存的抽象。但新型计算机内存的许多性质已很难用简单向量表现了
  - 它们有复杂的缓存系统，复杂的缓存一致性算法
  - 多核的加入使情况进一步复杂化，理解其细节行为变得更加困难，需要通过复杂的模拟
  - 但这种抽象模型仍反映了它的一部分情况和性质

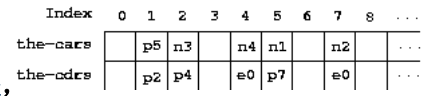
Scheme 数据的表示

- 不难用向量实现表存储器所需的序对结构：1) 设想两个向量 the-cars 和 the-cdrs；2) 指向序对的指针用向量的下标表示，序对的 car 就是向量 the-cars 里特定元素的内容，cdr 类似
- 非序对数据可用带类型指针的方式表示。为此要扩充指针增加类型信息
  - 类型信息可以设法表示，如在指针里加标志位（有带标志位的硬件机器，也可利用地址中不用的位等）

((1 2) 3 4)



- eq? 就是指针相同
- 符号用带类型指针表示。实际 Scheme 系统有符号表，对象表，读入遇到新符号时创建表项，取得符号指针



基本表操作的实现

- 基于序对表示，各种基本表操作都可以“代换”为几个或几个向量操作。下面假定有向量访问和赋值，指针算术运算
- 寄存器机器支持的赋值指令

  (assign <*reg*$_1$> (op car) (reg <*reg*$_2$>))
  (assign <*reg*$_1$> (op cdr) (reg <*reg*$_2$>))

  可实现为

  (assign <*reg*$_1$> (op vector-ref) (reg the-cars) (reg <*reg*$_2$>))
  (assign <*reg*$_1$> (op vector-ref) (reg the-cdrs) (reg <*reg*$_2$>))
- 寄存器机器的执行指令

  (perform (op set-car!) (reg <*reg*$_1$>) (reg <*reg*$_2$>))
  (perform (op set-cdr!) (reg <*reg*$_1$>) (reg <*reg*$_2$>))

  实现为

  (perform (op vector-set!) (reg the-cars) (reg <*reg*$_1$>) (reg <*reg*$_2$>))
  (perform (op vector-set!) (reg the-cdrs) (reg <*reg*$_1$>) (reg <*reg*$_2$>))

基本表操作的实现

- 执行 cons 时创建新序对单元，并将相关 car 和 cdr 分别存入。假定有一个特殊寄存器 free 总指向一个空闲下标，增加它的值可以得到下一可用下标（要求空闲位置连续）。这时，cons 指令

  (assign <*reg*$_1$> (op cons) (reg <*reg*$_2$>) (reg <*reg*$_3$>))

  可实现为下面指令序列：

  (perform (op vector-set!) (reg the-cars) (reg free) (reg <*reg*$_2$>))
  (perform (op vector-set!) (reg the-cdrs) (reg free) (reg <*reg*$_3$>))
  (assign <*reg*$_1$> (reg free))
  (assign free (op +) (reg free) (const 1))
- 操作 eq?

  (op eq?) (reg <*reg*$_1$>) (reg <*reg*$_2$>)

  检查寄存器内容是否相等

  pair?, null?, symbol?, number? 检查指针的类型域

栈的实现

- 寄存器机器里需要一个栈，可以用表模拟，用特殊寄存器 the-stack

  操作 (save <*reg*>)

  实现为

  (assign the-stack (op cons) (reg <*reg*>) (reg the-stack))

  操作 (restore <*reg*>)

  实现为

  (assign <*reg*> (op car) (reg the-stack))
  (assign the-stack (op cdr) (reg the-stack))

  操作 (perform (op initialize-stack))

  实现为

  (assign the-stack (const ()))
- 这些操作都可以基于前面的向量解释。在实际系统里，常另用一个向量来实现栈，主要是考虑实现的效率

造成无穷存储的假象

- 表结构的实现问题已经解决了，但有前提
  - 保证执行 cons 时总有可用的自由空间
  - 为此需要无穷大的存储
- 实际计算机里不断执行 cons，终将用尽整个序对空间
- 注意：建立的序对中许多用于保存临时数据
  - 中间结果
  - 临时建立的环境框架等
  - 用后数据丢弃，其存储已无必要保留
- 例如：

  (accumulate + 0 (filter odd? (enumerate-interval 0 n)))

  执行中构造两个表：枚举表和奇数表，求和完成后都没用了

造成无穷存储的假象

- 需要做出安排，周期性地回收已分配但不再用的单元
  - 如果回收与分配的速度相当
  - 而且程序每个时刻实际使用的单元不多于可供应的单元

  系统就可以永远运转。这就造成了一种无穷大存储的假象
- 要想回收不用的序对，需要确定那些确实不需要，即，其存在与否（其内容）对后面的计算不产生影响
- 下面提出的方法称为废料收集，其基本思想是
  - 有用单元，都是从当前所有寄存器的内容出发，通过一系列的 car/cdr 操作可以到达的单元
  - 不可达的单元都可以回收
- 典型的基本废料收集方法有两类，后来有许多发展

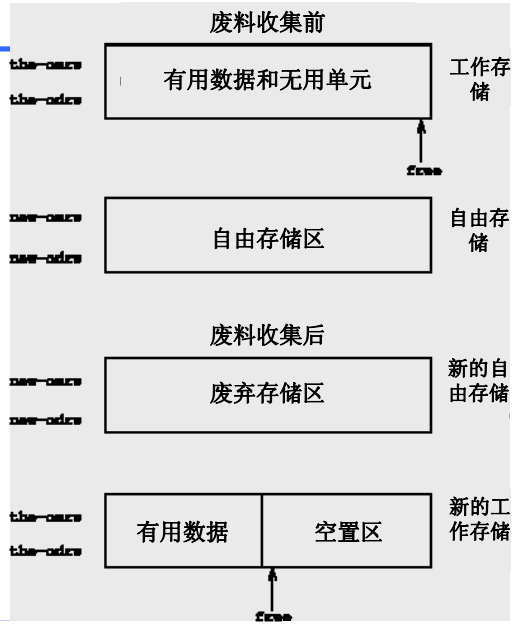
  包括分代式废料收集，并行废料收集，以及各种更复杂的环境里的废料收集等。废料收集已经成为支持软件系统运行的基本技术

废料收集

- 简单的废料收集工作周期性地进行：
  - 当时工作存储区满时中断计算，启动新一轮废料收集
  - 收集完成后重启暂停的计算工作
- 最早的技术称为“标记和清扫”，工作方式是
  - 从寄存器出发沿 car 和 cdr 指针周游单元存储区，给单元加标记
  - 而后扫描整个存储器，回收无标记单元
- 下面考虑的是另一种技术（stop-and-move），基于复制有用对象
- 基本想法：
  - 把一片存储区里的有用对象都搬走
  - 整个存储区都可以重用了
- 具体技术见下页

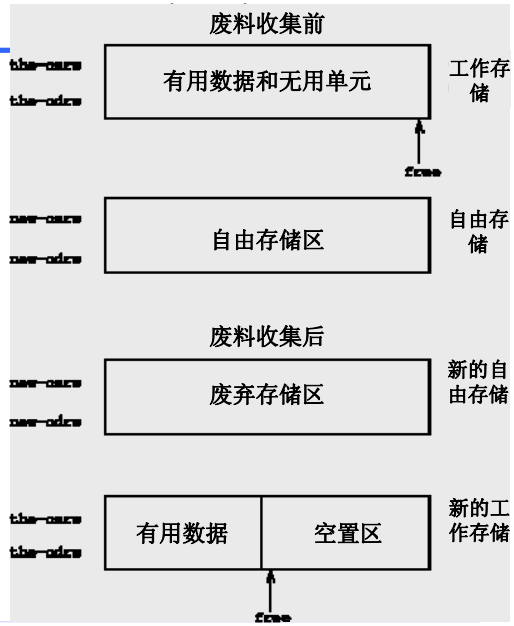
废料收集

- 将存储区分为大小相等的两个半区：工作存储区和自由存储区
- cons 总在工作存储区里顺序分配，下次分配下一位置
- 工作存储区满时做废料收集，把工作区中有用的序对都搬到自由存储区。具体技术是从所有寄存器出发，追踪 car 和 cdr 指针
- 若工作区有无用单元，最后自由区应剩下可用于分配的空闲单元
- 完成搬移后交换工作区和自由区的地位



收集工作的现场

- 下面假定寄存器 root 的值为指针，所指结构可以到达所有在用单元
- the-cars 和 the-cdrs 指向的两个向量是工作区，new-cars 和 new-cdrs 指的两个向量是自由区
- free 指向工作区里第一个空闲单元，它随分配移动，到达存储器右端表示空闲单元已用完了
- 废料收集前后的情况如右图所示。收集完成后交换两对向量指针，实现存储区的“切换”



stop-and-move 的实现

- 收集过程维护两个指针 free 和 scan，收集的初始化操作
  - 把 free 和 scan 设置为指向自由区起点
  - 把 root 所指单元复制到自由区的第一个单元
  - root 所指单元的 car 设一个特殊标志，cdr 设为 free (单元新位置)
  - root 指向新位置，更新 free 使之指向下一空单元
- 注意：scan 指向已移入新区 (收集前的自由区) 的单元，但其 car 和 cdr 所指单元可能还在老区。收集过程是反复做 (若 scan < free)：
  - 若 scan 所指单元的 car 还在老区就将它搬到新区 free 处。设置原单元的 car 为特殊标志，cdr 为 free，将 free 推进一个单元
  - 若 scan 所指单元的 cdr 还在老区就将它搬到新区 free 处。置原单元的 car 为特殊标志，cdr 为 free，将 free 推进一个单元
  - 若发现被 scan 所指单元的 car/cdr 所指的老区单元有特殊标志，则更新这个 car/cdr，使之正确指向该单元的新位置
- 反复上述操作至 scan 和 free 相等时收集完成（请考虑算法正确性）

stop-and-move 的实现

- 考虑用寄存器机器语言描述本算法，给表单元确定新位置的基本操作由子程序 relocate-old-result-in-new 完成，其参数是指向被移对象的指针，取自寄存器 old，新位置由寄存器 free 的当时值确定，新位置存入寄存器 new （并需要增大 free 值）。最后用转跳指令按寄存器 relocate-continue 的值返回
- 启动废料收集后首先设置 scan 和 free，而后进入上述子程序。子程序里首先为 root 所指单元重新分配位置：

```
begin-garbage-collection
  (assign free (const 0))
  (assign scan (const 0))
  (assign old (reg root))    ;; 让 old 指向老区里被处理的单元
  (assign relocate-continue (label reassign-root))
  (goto (label relocate-old-result-in-new)) ; 将 root 单元搬到新区
reassign-root
  (assign root (reg new)) ;; 让 root 指向结点的新位置
  (goto (label gc-loop))    ;; 转到基本 gc 循环（下页）
```

搬移单元的主要工作由位于 relocate-old-result-in-new 标号的子程序处理。转去前先在 relocate-continue 里设置好返回地址

stop-and-move 废料收集

- 主循环检查是否还有没扫描的单元（scan 不等于 free）

```
gc-loop
  (test (op =) (reg scan) (reg free))
  (branch (label gc-flip))                ; 基本收集循环结束，最后清理
  (assign old (op vector-ref) (reg new-cars) (reg scan))
  (assign relocate-continue (label update-car))
  (goto (label relocate-old-result-in-new))
```

如果 scan = free，搬移工作结束，转去切换存储区

否则就是搬移还没结束

先将 old 设置为新区中被 scan 所指的单元的 car 所指的单元（即需要处理的下一个老区单元）

然后转去执行 relocate-old-result-in-new 位置的代码，实际处理下一个单元的搬移工作

- 最主要的处理是位于 relocate-old-result-in-new 标号处的代码

stop-and-move 废料收集

- 处理了 scan 所指单元的 car 之后（保证它已到新区），将 scan 所指的单元的 car 设置为 new（相应单元的新位置）后处理其 cdr，转到 relocate-old-result-in-new 之前先设 old 和 relocate-continue

```
update-car ; 寄存器 new 里是 car 所指单元的新位置
  (perform (op vector-set!) (reg new-cars) (reg scan) (reg new))
  (assign old (op vector-ref) (reg new-cdrs) (reg scan))
  (assign relocate-continue (label update-cdr))
  (goto (label relocate-old-result-in-new))
```

```
update-cdr ;寄存器 new 里是 cdr 所指单元的新位置
  (perform (op vector-set!) (reg new-cdrs) (reg scan) (reg new))
  (assign scan (op +) (reg scan) (const 1))
  (goto (label gc-loop))
```

处理完 scan 所指单元的 cdr 后做的第一件事情和处理 car 后一样

处理完 scan 所指的单元后，将 scan 更新到下一位置


```
relocate-old-result-in-new                          核心代码段，完成一个单元的搬迁
  (test (op pointer-to-pair?) (reg old))
  (branch (label pair))
  (assign new (reg old))                            不是序对直接返回。
  (goto (reg relocate-continue))                    非序对对象不收集
pair
  (assign oldcr (op vector-ref) (reg the-cars) (reg old))
  (test (op broken-heart?) (reg oldcr))             这个单元是
  (branch (label already-moved))                    否已搬迁？
  (assign new (reg free)) ; 未搬，新位置是 free
  ;; 更新 free 指针
  (assign free (op +) (reg free) (const 1))
  ;; 将这个单元的 car 和 cdr 拷贝到新位置
  (perform (op vector-set!) (reg new-cars) (reg new) (reg oldcr))
  (assign oldcr (op vector-ref) (reg the-cdrs) (reg old))
  (perform (op vector-set!) (reg new-cdrs) (reg new) (reg oldcr))
  ;; 在原位置设置的 car 设标志，cdr 设索引指针
  (perform (op vector-set!) (reg the-cars) (reg old) (const broken-heart))
  (perform (op vector-set!) (reg the-cdrs) (reg old) (reg new))
  (goto (reg relocate-continue))
already-moved                                       单元已在新区，直
  (assign new (op vector-ref) (reg the-cdrs) (reg old))   接设置 new 后返回
  (goto (reg relocate-continue))
```

stop-and-move 废料收集

- 最后一步是交换两个（半）存储区的地位

```
gc-flip
  (assign temp (reg the-cdrs))
  (assign the-cdrs (reg new-cdrs))
  (assign new-cdrs (reg temp))
  (assign temp (reg the-cars))
  (assign the-cars (reg new-cars))
  (assign new-cars (reg temp))
```

至此废料收集结束，应转到操作结束的位置

显式控制求值器

- 前面考虑过如何把简单 Scheme 程序变换到寄存器机器描述，下面考虑一个复杂的 Scheme 程序，要变换前面的元循环求值器（用 eval 和 apply 实现的 Scheme 解释器）
  - 这一工作的结果是一个显式控制的求值器，求值中过程调用的参数传递都基于寄存器和栈描述
  - 这个寄存器语言描述接近常规机器语言，可以反映实际 Scheme 实现的许多情况，可用寄存器机器模拟器运行（可能较慢）
  - 它反映了用常规机器语言实现 Scheme 解释器的基本结构，可以从它出发实现真正能用的 Scheme 解释器，或者从它出发做出能解释 Scheme 程序的硬件处理器
- 书上有一个图，是一个 Scheme 处理器芯片
- Java 虚拟机（JVM）或其他脚本语言虚拟机，具有类似结构和功能
- 下面考虑求值器的具体实现

寄存器和操作

- 设计求值器对应的寄存器机器时，需要先设计其中各种操作
- 元循环求值器用了一些语法过程，如 quoted?, make-procedure 等。要实现完成求值的寄存器机器，按说要把这些过程都展开为表操作。可是那样做代码会变得很长
- 为简化描述，下面仍以元循环求值器的语法过程和表示环境的过程作为寄存器机器的基本过程。要真正实现这个求值器，还需基于更基本的操作将这些操作展开，并使用前面讨论的表结构表示
- 下面求值器里用一个栈和 7 个寄存器：
  - exp (表达式), val (在指定环境里求值表达式得到的结果)
  - env (当时环境), continue (用于实现递归)
  - 三个寄存器用于组合式的实现: proc (过程对象), argl (实参值表), unev (辅助寄存器，意为未求值的表达式)
- 具体操作隐含在控制器代码里，不专门列出（不明确写数据通路）

核心代码

求值器核心部分从 eval-dispatch 开始，基于 env 环境对 exp 求值。求值完成后按 continue 寄存器转移，求出的值存在 val。这里就是分情况处理：

```
eval-dispatch
  (test (op self-evaluating?) (reg exp)) ;; 自求值表达式
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))       ;; 变量
  (branch (label ev-variable))
  (test (op quoted?) (reg exp))         ;; quote 表达式
  (branch (label ev-quoted))
  (test (op assignment?) (reg exp)) ;; 赋值表达式
  (branch (label ev-assignment))
  (test (op definition?) (reg exp))     ;; 定义表达式
  (branch (label ev-definition))
  (test (op if?) (reg exp))             ;; if 表达式
  (branch (label ev-if))
  (test (op lambda?) (reg exp))         ;; lambda 表达式
  (branch (label ev-lambda))
  (test (op begin?) (reg exp))          ;; begin 表达式
  (branch (label ev-begin))
  (test (op application?) (reg exp))   ;; 过程应用
  (branch (label ev-application))
  (goto (label unknown-expression-type))
```

简单表达式的求值

下面几段代码处理各种简单表达式：

```
ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))
ev-variable
  (assign val (op lookup-variable-value) (reg exp) (reg env))
  (goto (reg continue))
ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))
ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  (assign val (op make-procedure) (reg unev) (reg exp) (reg env))
  (goto (reg continue))
```

注意：处理 lambda 时先把形式参数表和过程体分别存入 unev 和 exp，而后调用 make-procedure 构造过程对象

过程应用的求值

过程应用是组合式，其成分是运算符和运算对象。实际应用前需要先求值运算符和运算对象，而后调用 apply 实现函数应用

显式求值时也要做同样工作。递归调用同样是利用栈实现，调用前保存一些寄存器，需要仔细考虑要保存哪些信息，怎样保存

```
ev-application
  (save continue)
  (save env)          ;; 保存继续点和环境
  (assign unev (op operands) (reg exp))  ;; 使用临时寄存器
  (save unev)                            ;; 保存运算对象（表）
  (assign exp (op operator) (reg exp))   ;; 先求值运算符
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
```

最后，在设置了求出运算符值的继续点之后转去做实际求值

求值运算符后转到 ev-appl-did-operator，去求值各运算对象。这时 val 里是求出的运算符，栈里第一项是运算对象表，第二项是环境

过程应用的求值

求值运算对象：

```
ev-appl-did-operator
  (restore unev)              ; 弹出运算对象表
  (restore env)               ; 弹出环境
  (assign argl (op empty-arglist)) ; 实参（运算对象的值）表置空
  (assign proc (reg val))     ; 将运算符过程存入 proc
  (test (op no-operands?) (reg unev)) ; 如运算对象表空就去做实际应用
  (branch (label apply-dispatch))
  (save proc)                 ; 保存求出的运算符过程，而后向下求值运算对象
ev-appl-operand-loop
  (save argl)                 ; 保存实参表
  (assign exp (op first-operand) (reg unev)) ; 取出第一个运算对象
  (test (op last-operand?) (reg unev)) ; 检查是否最后一个运算对象
  (branch (label ev-appl-last-arg))    ; 最后一个运算对象特殊处理
  (save env)
  (save unev)                 ; 保存环境和运算对象表
  (assign continue (label ev-appl-accumulate-arg)) ; 设置继续点 (累积实参值)
  (goto (label eval-dispatch)) ; 求值第一个运算对象
```


过程应用的求值

每次求值完一个运算对象后都转到这里：

```
ev-appl-accumulate-arg
  (restore unev) ; 恢复运算对象表
  (restore env)  ; 恢复环境
  (restore argl) ; 恢复实参表
  (assign argl (op adjoin-arg) (reg val) (reg argl)) ; 新值加入实参表
  (assign unev (op rest-operands) (reg unev))    ; 丢掉一个运算对象 (已计算)
  (goto (label ev-appl-operand-loop))        ; 继续去求值下一运算对象
```

对最后一个运算对象的处理

```
ev-appl-last-arg
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))
ev-appl-accum-last-arg  ; 求值完最后一个运算对象后转到这里
  (restore argl)              ; 取出实参表
  (assign argl (op adjoin-arg) (reg val) (reg argl)); 新值加入实参表
  (restore proc)              ; 取出运算对象过程
  (goto (label apply-dispatch)) ; 转去做实际应用
```


过程应用

实际应用过程的工作，根据是基本过程还是复合过程分别处理

这时：proc 和 argl 分别为运算符过程对象和实际参数表，栈里第一项是求值完成后应该转去的继续点

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknown-procedure-type))
```

对基本过程，直接用 apply-primitive-procedure 完成过程应用：

```
primitive-apply
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (restore continue)
  (goto (reg continue))
```


复合过程应用

确定为复合过程时，proc 和 argl 分别为运算符过程对象和实际参数表

1, 先从 proc 里取出过程的形参表和环境

2, 将 env 设置为扩充后的环境

3, 取出过程体，转到序列求值代码的入口 ev-sequence

```
compound-apply
  (assign unev (op procedure-parameters) (reg proc))
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment)
              (reg unev) (reg argl) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))
```

序列表达式求值（元循环求值器的 eval-sequence）有两种情况：

1) 要求值的是个表达式序列，如过程体；2) 要求值的是 begin 表达式，去掉 begin 后，可以统一到前一情况

序列求值

对 begin 表达式的处理由 ev-begin 入口，其他地方来的由 ev-sequence 入口

```
ev-begin
  (assign unev (op begin-actions) (reg exp)) ; 取出 begin 的实际序列
  (save continue) ; 保存求值完的继续点，与其他入口一致
  (goto (label ev-sequence))

ev-sequence ; 此时 unev 里是待求值的表达式序列
  (assign exp (op first-exp) (reg unev)) ; 取出序列中第一个表达式
  (test (op last-exp?) (reg unev))          ; 是否为序列里最后一个表达式
  (branch (label ev-sequence-last-exp)) ; 最后的表达式特殊处理
  (save unev) ; 保存表达式序列
  (save env)  ; 保存环境
  (assign continue (label ev-sequence-continue)) ; 设完成求值后的继续点
  (goto (label eval-dispatch)) ; 求值 exp 里的表达式
ev-sequence-continue ; 完成了一个子表达式的求值
  (restore env)             ; 恢复
  (restore unev)
  (assign unev (op rest-exps) (reg unev)) ; 丢掉第一个子表达式
  (goto (label ev-sequence))                ; 转回去继续
ev-sequence-last-exp  ; 做序列中最后一个表达式的求值
  (restore continue)     ; 恢复继续点寄存器
  (goto (label eval-dispatch)) ; 直接转去求值最后一个表达式（在 exp 里）
```

尾递归

- 前面说下面过程形式是递归，但产生线性迭代计算，常量存储

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

  原因是最后调用自身时不用保存任何信息

- 如果一个求值器在递归执行这种过程时可以不分配存储，就称该求值器是尾递归的。元循环求值器是否尾递归的情况看不清楚，因为求值细节依赖于基础系统。现在这个求值器可以看清楚
- 前面求值器确实为尾递归的，因为它直接转去求值序列里的最后一个表达式，没在栈里保存任何信息，没使用新的空间
- 如果不想做尾递归（优化），可以修改代码统一处理子表达式（包括最后一个子表达式）。得到的代码简单些，但丧失尾递归性质

下面是改动后的代码（非伪递归实现）

尾递归

```
ev-sequence
  (test (op no-more-exps?) (reg unev)) ; 检查序列是否为空
  (branch (label ev-sequence-end))
  (assign exp (op first-exp) (reg unev)) ; 处理当前序列里第一个子表达式
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue ; 恢复环境，丢掉序列里第一个子表达式（已求过值）
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence)) ; 继续去处理剩下的序列
ev-sequence-end ; 所有子表达式都已完成求值
  (restore continue)
  (goto (reg continue))
```

改动很少，语义不变，但已不是尾递归。计算前面的过程需要线性空间。对于非尾递归求值器，语言必须为迭代提供专门的结构

条件

对 if 表达式，先求值其谓词部分，基于其值确定随后的求值。求值谓词之前保存整个 if 以便后面使用，也要保存环境和继续点

```
ev-if
  (save exp)                ; 保存整个 if 表达式供后面使用
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg exp))
  (goto (label eval-dispatch))  ; 转去求值谓词

ev-if-decide
  (restore continue)
  (restore env)
  (restore exp)
  (test (op true?) (reg val))
  (branch (label ev-if-consequent)) ; 检测结果为真时转
  (assign exp (op if-alternative) (reg exp))
  (goto (label eval-dispatch)) ; 转去求值第二个分支
ev-if-consequent
  (assign exp (op if-consequent) (reg exp))
  (goto (label eval-dispatch)) ; 转去求值第一个分支
```

赋值

赋值表达式用下面代码段处理：

```
ev-assignment
  (assign unev (op assignment-variable) (reg exp))
  (save unev)                ；保存变量供后面使用
  (assign exp (op assignment-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-assignment-1))
  (goto (label eval-dispatch))  ；求出被赋的值
ev-assignment-1                 ；恢复环境后完成实际赋值
  (restore continue)
  (restore env)
  (restore unev)            ；恢复变量
  (perform                  ；实际赋值
   (op set-variable-value!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
```


定义

定义的处理与赋值类似：

```
ev-definition
  (assign unev (op definition-variable) (reg exp))
  (save unev)                ；变量保存供后面使用
  (assign exp (op definition-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-definition-1))
  (goto (label eval-dispatch))  ；求出需要赋的值
ev-definition-1                 ；恢复环境等并完成定义
  (restore continue)
  (restore env)
  (restore unev)            ；恢复变量
  (perform                  ；实际定义
   (op define-variable!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
```


求值器的运行

- 至此显式控制求值器完成，从第一章开始对求值模型的讨论结束。一系列模型：代换模型，环境模型，元循环模型（环境模型的 Scheme 实现），到这个模型。一个比一个精确，加入更多细节
- 要理解求值器的行为，需要执行它，监视其行为。先做一个驱动循环：

```
read-eval-print-loop
  (perform (op initialize-stack))
  (perform (op prompt-for-input) (const ";;; EC-Eval input:"))
  (assign exp (op read))
  (assign env (op get-global-environment))
  (assign continue (label print-result))
  (goto (label eval-dispatch))
print-result
  (perform (op announce-output) (const ";;; EC-Eval value:"))
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))
```


求值器的运行

- 需要处理遇到的错误，出现错误时打印信息并回到驱动循环：

```
unknown-expression-type
  (assign val (const unknown-expression-type-error))
  (goto (label signal-error))
unknown-procedure-type
  (restore continue)   ; clean up stack (from apply-dispatch)
  (assign val (const unknown-procedure-type-error))
  (goto (label signal-error))
signal-error
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))
```

回到基本求值循环时将栈置空，重新开始新一次循环

- 为完成这一机器，需要把本节的所有代码收集起来，调用前面寄存器机器模拟器的操作，构造一个机器模型。还需加入所有所需的操作（由前面元循环求值器的代码得到）

求值器的运行

创建机器的操作列表，把所有要用的操作加入（取自元循环求值器）：

```
(define eceval-operations
  (list (list 'self-evaluating? self-evaluating)
        <机器的完整操作表>))
```

构造机器模型的代码框架，其中应填入前面所有代码：

```
(define eceval
  (make-machine
   '(exp env val proc argl continue unev)
   eceval-operations
  '(
    read-eval-print-loop
      <上面给出的求值器机器代码>
  )))
```


求值器的运行

运行前创建环境，而后启动这个求值器

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

(start eceval)
;;; EC-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
;;; EC-Eval value:
ok
;;; EC-Eval input:
(append '(a b c) '(d e f))
;;; EC-Eval value:
(a b c d e f)
```


监视求值器的执行

同样可以考虑监视求值器的运行。在驱动循环里增加一段代码：

```
print-result
  (perform (op print-stack-statistics)); 在操作表里加入统计操作
  (perform (op announce-output) (const ";;; EC-Eval value:"))
  ... ; 从这里开始的代码和原来一样
```

与求值器的一些交互（需要基本算术运算操作）：

```
;;; EC-Eval input:
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
(total-pushes = 3 maximum-depth = 3)
;;; EC-Eval value:
ok
;;; EC-Eval input:
(factorial 5)
(total-pushes = 144 maximum-depth = 28)
;;; EC-Eval value:
120
```


总结

- 表存储的连续存储实现，向量是常规计算机连续存储的抽象
  - 为支持在任何时刻自由地创建表结构，需要自动废料收集功能的支持，其实质就是用有穷存储维持一种无穷存储的假象
  - 一种有效的废料收集方法是将有用结点搬出在用的存储块。这种技术有许多变形
  - 为实现表和其他对象，可考虑带标志的指针（或带标志的存储）
- 在基本表结构实现基础上，做 Scheme 解释器的寄存器机器实现。在这一实现里可以看清 Scheme 求值中的一些细节
  - 解释器基本存储结构和数据结构（寄存器/栈/表和其他对象空间）
  - 通过栈支持递归，支持复杂的组合式求值过程
  - 过程调用的约定，通过栈和寄存器传递参数和返回值
  - 实现尾递归（优化）
  - 等等