

4. 元语言抽象(2)

本节讨论

- 惰性求值的 Scheme 求值器
 - 流作为惰性表
- 非确定性计算
 - amb 语言和搜索 (amb 取自 ambiguous)
 - 非确定性计算的实例
 - amb 求值器的实现

Scheme 的变形: 惰性求值

- 做好求值器后还可以修改, 试验语言设计的选择和变化
- 开发一个新语言, 通常
 - 先用现有语言实现新语言的求值器
 - 研究各种设计想法和设计选择, 在求值器里实现, 提供相关人员使用和评价。根据评价结果修改求值器, 并进一步送出评价
 - 在高层次上更容易实现、测试和修改求值器。可以从基础语言借用大量功能
 - 直至新语言比较成熟时再考虑另行实现完整的语言系统
- 下面研究 Scheme 的变形, 它们提供了基本 Scheme 没有的功能
- 本节第一个实例是做一个延时求值的 Scheme 语言求值器, 其中
 - 用“槽”(trunk)封装尚未求值的表达式
 - 很容易实现带记忆的槽, 是否带记忆对求值器完全透明

正则序和应用序

- Scheme 采用应用序, 过程应用前完成所有参数的求值
- 正则序到实际需要时才求值参数, 也称为惰性求值 (懒求值)
- 考虑下面过程:

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

在 Scheme 里求 (try 0 (/ 1 0)) 会出错, 实际上 b 的值没用
- 需要正则序的另一个例子 (“除非出现异常条件, 否则就正常处理”):

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

如不采用正则序, 这个例子完全没用

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

正则序和应用序

- 如果在进入过程体前它的一个参数已完成求值, 称这个过程对该参数为严格的; 如果未完成求值, 则说这一过程对该参数为非严格的
 - 在纯应用序语言里, 每个过程对其每个参数都是严格的
 - 在纯正则序语言里, 每个复合过程对其每个参数都是非严格的, 而基本过程对其参数可以是严格的或非严格的
- 有这样的语言, 其中程序员可以控制所定义过程对各参数的严格性
- 能将过程的某些参数定义为非严格, 很有用
 - 例如: 对 cons (或任何数据结构的构造函数), 非严格允许在不知道数据结构的某些部分的情况下使用数据结构
 - “流模型”就利用了非严格参数

采用惰性求值的解释器

- 本节将实现一个正则序求值的语言，它在形式上与 Scheme 语言一样，但所有复合过程的参数都采用惰性求值，基本过程仍采用应用序
- 实现这一语言的方式是修改已有求值器。所有修改都围绕过程应用：
 - 先确定参数是否立即求值。要求延时的对象不求值，而是建一个槽 (trunk)，存入产生相关值所需的信息 (表达式和环境)
- 对入槽表达式的求值称为**强迫**，需要其值时去强迫它。几种情况：
 - 基本过程需要用这个值
 - 这个值被作为条件表达式的谓词
 - 复合表达式以这个值为运算符，要求应用它
- 设计选择：是否用带记忆的槽，第一次求值时记录值？下面用带记忆的槽，这样可能效率高，也带来一些难处理的问题 (书上相关练习)

修改求值器

- 关键是 eval 和 apply 里的处理
- 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
把当前环境送给 apply，它需要把环境放入参数槽
- 需要实际参数的值时，强迫求出它的值：

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

修改求值器

- 修改后的 apply，改动不大。注意：eval 送来的是未求值的运算对象，送给基本过程之前需要求值这些表达式：

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

修改求值器

- 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))))
```
- 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)))
```

修改求值器

- 修改驱动循环，用 `actual-value` 代替其中 `eval`，要求求值：

```
(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

- 运行例子

```
(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
```

- 实际上，做这种试验前需要先完成槽的实现（下面做）

槽的表示

- 槽作为数据抽象，实现一种安排，其中封装一个表达式和一个环境。如果需要，就可以求出其中表达式的值
- 实际求值时应该用 `actual-value` 而不是 `eval`（现在的 `eval` 是延时的）
- 强追求值时算到不是槽为止（`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))
```

槽的表示

- 实现带记忆的槽，要修改定义，求值后将槽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) ; 用值取代表达式
       (set-cdr! (cdr obj) '()) ; 丢掉不再需要的环境
       result))
    ((evaluated-thunk? obj) (thunk-value obj))
    (else obj))))
```

- 无论有没有记忆，前面求值器都能工作

流作为惰性表

- 前面讨论流的时候将它实现为一种延时的表，其中用特殊形式 **delay** 和 **cons-stream**。该方式的缺点：

- 需要特殊形式，特殊形式不是一级对象，无法与高阶过程协作
- 需要把流做为另一种对象，与表很类似但又不同。因此需要对流重新实现各种表操作，而且这些操作只能用于流

现在采用惰性求值，可以把表作为流，也不再需要任何特殊形式

- 要按流的方式工作，**cons** 必须是非严格的。可以有多种办法：

- 修改求值器支持非严格基本过程，并把 **cons** 实现为非严格过程
- 或者把 **cons** 实现为一个复合过程（2.1.3 节，61页）
- 或者采用序对的过程性表示（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)))
```

流作为惰性表

- 现在各种表操作的标准定义都适用于无穷表（流），流操作就是表操作：

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

- 这种表比前面的流更惰性：其 **car** 也延时，需要时才求值。求序对的 **car** 或 **cdr** 时也不求值，其求值延时到用作基本过程的参数或打印时
- 惰性序对还解决了流的其他问题。前面包含反馈的流有时要显式用 **delay**。现在一切参数都是延时的，表积分可以直接定义，求解微分方程的流也没问题了：

```
(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
```

非确定性计算

- 现在考虑一种支持非确定性计算的 **Scheme** 变形。非确定性计算通过求值器的自动搜索实现。这一修改意义深远
- 非确定性计算与流有些类似，都适合“生成与检查”的应用问题。如：
有两个整数表，要求从中找出一对整数，它们分属不同的表，两个数之和是素数（推广：从两集数据中找出符合需要的一对数据）
- 前面做过类似工作：对有限序列做过，也对无穷流做过。采用的方法都是生成一些数对，而后从中过滤出和为素数的数对
- 非确定性计算的描述方式不同，用它可以直接按问题写程序：

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))
```

好像只是把问题重说了一遍。但这就是一个合法的非确定性程序

非确定性计算：关键思想

■ 关键思想：

- 一个表达式可以有多个可能值。如 **an-element-of** 可能返回一个表的任何元素。求值器自动选出一个值并维持轨迹
- 如果后面的要求不满足，求值器会重新选择，直至求值成功；或者所有选择都已用完时求值失败

选择和重新选择隐藏在求值器里，程序员不必关心和处理

■ 非确定性求值和流处理中时间的表现形式（比较）：

- 流处理用惰性求值松弛潜在的流和流元素的实际产生时间之间的紧密联系，貌似整个流都存在，元素的产生没有时间顺序
- 非确定性计算的表达式表示对一批可能世界的探索，每个世界由一串选择决定。求值器造成的假相：时间能分叉。程序保存所有可能的执行历史，计算遇到死路时退回前面选择点转到另一分支

非确定性计算：amb

■ 下面的非确定性语言基于一种称为 **amb** 的特殊形式

■ 定义了前面非确定性过程后送给 **amb** 求值器

```
;;; Amb-Eval input:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
;;; Starting a new problem
;;; Amb-Eval value:
(3 20)
```

求值器从两个表里反复选择元素，直至做出一次成功选择

amb 和搜索

■ Scheme 的非确定性扩充引进一种称为 **amb** 的特殊形式

(amb <e₁> <e₂> ... <e_n>) 返回几个参数表达式之一的值

■ (list (amb 1 2 3) (amb 'a 'b)) 可能返回下面任何一个表达式：

(1 a) (1 b) (2 a) (2 b) (3 a) (3 b)

■ 如果一个 **amb** 表达式只有一个选择，就（确定地）返回该元素的值。没有选择的 (**amb**) 没有值，它的求值导致计算失败且不产生值

■ 例：要求谓词 **p** 必须为真：

```
(define (require p)
  (if (not p) (amb)))
```

■ 例：an-element-of 可实现为：

```
(define (an-element-of items)
  (require (not (null? items)))
  (amb (car items) (an-element-of (cdr items))))
```

表为空时计算失败，
否则它总返回表中的
某个元素

amb 和搜索

■ 可以描述无穷选择。如要求得到一个大于或等于给定 **n** 的值：

```
(define (an-integer-starting-from n)
  (amb n (an-integer-starting-from (+ n 1))))
```

这个过程就像是在构造一个流，但它只返回一个整数

■ 非确定性地返回一个选择与返回所有选择不同。用户看到 **amb** 表达式返回一个选择；实现则看到它能逐个地返回所有选择，都可以用

■ **amb** 表达式导致计算进程分裂为多个分支

- 如果有多个处理器，可把它们分派到不同处理器，同时搜索

- 只有一个处理器时每次选一个分支，保留其他选择权，如：

- 随机选择，失败了退回重新选择

- 系统地探查这些分支。如总选第一个未检查分支；失败时退回最近选择点，探查下一未选过的分支（LIFO）

amb 语言：驱动循环

- amb 求值器读入表达式，输出第一个成功得到的值。允许人工要求回溯：输入 try-again，求值器将设法找下一结果：

```
;;; Amb-Eval input:
(prime-sum-pair '(1 3 5 8) '(20 35 110))
;;; Starting a new problem
;;; Amb-Eval value:
(3 20)
;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
(3 110)
;;; Amb-Eval input:
try-again
;;; Amb-Eval value:
(8 35)
;;; Amb-Eval input:
try-again
;;; There are no more values of
(prime-sum-pair (quote (1 3 5 8)) (quote (20 35 110)))
;;; Amb-Eval input:
(prime-sum-pair '(19 27 30) '(11 36 58))
;;; Starting a new problem
;;; Amb-Eval value:
(30 11)
```

遇到 try-again 之外的其他表达式，
都认为是重新开始一个新任务

非确定性计算实例：逻辑谜题

- 深入考虑 amb 求值器的实现之前，先看两个应用

1，求解逻辑谜题。考虑：

Baker、Cooper、Fletcher、Miller 和 Smith 住在五层公寓的不同层，
Baker 没住顶层，Cooper 没住底层，Fletcher 没住顶层和底层
Miller 比 Cooper 高一层，Smith 没有住与 Fletcher 相邻的层
Fletcher 没有住与 Cooper 相邻的层
问：各住哪层？

- 用 amb 只需简单列举各种可能性，就可以得到这个问题的解
- 对后面定义的过程，求值 (multiple-dwelling) 将得到一个解：
(baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))

逻辑谜题

- 用 amb 写的求解程序：

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5)) (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5)) (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith)))))
```

非确定性计算实例：自然语言的语法分析

- 要用处理自然语言，程序首先要做语法分析，识别输入句子的结构。例如，被处理的句子是一个冠词后跟一个名词和一个动词
- 首先要能辨别词的类属。假定有下面几个词类表：
(define nouns '(noun student professor cat class))
(define verbs '(verb studies lectures eats sleeps))
(define articles '(article the a))
- 还需要语法（描述由简单元素构造语法元素的规则。如
 - 句子由一个名词短语和一个动词构成
 - 名词短语由一个冠词和一个名词构成
- The cat eats 可以分析为：
(sentence (noun-phrase (article the) (noun cat)) (verb eats))
- 下面开发一个完成这种分析的简单程序，对一个输入句子，返回作为分析结果的表（上面这样的表），其中表示了句子的结构和成分

自然语言的语法分析

- 下面是一个 **amb** 程序
- 对句子，要辨认出它的两个部分，并用符号 **sentence** 标记：

```
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-word verbs)))
```

- 名词短语，要找出其中的冠词和名词：

```
(define (parse-noun-phrase)
  (list 'noun-phrase
        (parse-word articles)
        (parse-word nouns)))
```

自然语言的语法分析

- 最底层检查下一个未分析的单词，看是否属于期望的单词类表。这里用全局变量 ***unparsed*** 保存未分析的输入。它不空且其中第一个单词属于给定单词表时，删除单词并返回其词类（单词表第一个元素）

```
(define (parse-word word-list)
  (require (not (null? *unparsed*)))
  (require (memq (car *unparsed*) (cdr word-list)))
  (let ((found-word (car *unparsed*)))
    (set! *unparsed* (cdr *unparsed*))
    (list (car word-list) found-word)))
```

- 开始时将 ***unparsed*** 设置为整个输入，然后设法分析它：

```
(define *unparsed* '())

(define (parse input)
  (set! *unparsed* input)
  (let ((sent (parse-sentence))) ; 分析出一个句子
    (require (null? *unparsed*)) ; 输入用完才是分析成功
    sent))
```

自然语言的语法分析

- 分析实例：

```
;;; Amb-Eval input:
(parse '(the cat eats))
;;; Starting a new problem
;;; Amb-Eval value:
(sentence (noun-phrase (article the) (noun cat)) (verb eats))
```

- 在这里 **amb** 很有用。分析中的约束条件很容易用 **require** 描述，也容易扩充更复杂的语法。这里可以看到搜索和回溯的作用

- 加一个介词表：

```
(define prepositions '(prep for to in by with))
```

加入介词短语的定义：

```
(define (parse-prepositional-phrase)
  (list 'prep-phrase
        (parse-word prepositions)
        (parse-noun-phrase)))
```

自然语言的语法分析

- 修改句子的定义，其中动词短语可以是一个动词，或一个动词加一个介词短语（下面定义允许加上任意多个介词短语）

```
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-verb-phrase)))

(define (parse-verb-phrase)
  (define (maybe-extend verb-phrase)
    (amb verb-phrase
          (maybe-extend (list 'verb-phrase
                                verb-phrase
                                (parse-prepositional-phrase))))))
  (maybe-extend (parse-word verbs)))
```

“maybe-extend” 说可以扩充介词短语，得到的动词短语还可扩充

自然语言的语法分析

- 扩充名词短语，允许“a cat in the class”形式（一个名词短语后跟一个介词短语，实际上允许任意多个）

```
(define (parse-simple-noun-phrase)
  (list 'simple-noun-phrase
        (parse-word articles)
        (parse-word nouns)))

(define (parse-noun-phrase)
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
          (maybe-extend (list 'noun-phrase
                                noun-phrase
                                (parse-prepositional-phrase))))))
  (maybe-extend (parse-simple-noun-phrase)))
```

自然语言的语法分析

- 实例。分析：

(parse '(the student with the cat sleeps in the class))

可得到分析：

```
(sentence
 (noun-phrase
  (simple-noun-phrase (article the) (noun student))
  (prep-phrase (prep with)
                (simple-noun-phrase (article the) (noun cat)))))

(verb-phrase
 (verb sleeps)
 (prep-phrase (prep in)
               (simple-noun-phrase (article the) (noun class)))))
```

- 有的句子存在多种分析结果。例如“The professor lectures to the student with the cat”，就有两种分析（有歧义）

自然语言的语法分析

```
(sentence
 (simple-noun-phrase (article the) (noun professor))
 (verb-phrase
  (verb-phrase
   (verb lectures)
   (prep-phrase (prep to)
                 (simple-noun-phrase (article the) (noun student)))))
 (prep-phrase (prep with)
               (simple-noun-phrase (article the) (noun cat)))))
```

输入 try-again 将得到：

```
(sentence
 (simple-noun-phrase (article the) (noun professor))
 (verb-phrase
  (verb lectures)
  (prep-phrase (prep to)
                (noun-phrase
                 (simple-noun-phrase (article the) (noun student))
                 (prep-phrase (prep with)
                               (simple-noun-phrase (article the) (noun cat)))))))
```

amb 求值器的实现：基本设计

- 常规的 Scheme 表达式可能求出一个值，或不终止，或产生错误。非确定性的 Scheme 表达式还可能走入死胡同，使求值过程回溯，因此这种表达式的解释更复杂一些
- 下面要修改已有的求值器，基于分析求值器实现 amb 求值器。新求值器的不同就在于它生成不同的执行过程
- 重要概念：继续（continuation，延续）。“继续”是一种过程参数，将在过程的最后一步调用（不准返回，有尾递归优化可以处理）
- 常规 Scheme 求值器生成的执行过程要求一个环境参数，而 amb 分析器产生的执行过程要求三个参数，一个环境和两个继续过程（一个成功继续和一个失败继续），表达式求值结束时将调用这两个过程之一

□ 如果求值正常并得到结果，就调用“成功继续”

□ 如果求值进入死胡同，就调用“失败继续”

实际回溯是通过构造适当的成功继续和失败继续而实现的

amb 求值器的实现：基本设计

- 成功继续得到一个值（参数）并将计算进行下去。它还得到一个失败继续，如果用得到的值做计算进入死胡同，就调用这个失败继续
- 失败继续（过程）的作用是探查另一个非确定性分支
- 非确定性计算必须能处理多种可能性。在遇到无法确定取哪个值能得到最后结果时先取定一个值，同时构造一个失败继续，并将它们一起送给成功继续过程，以便将来遇到失败时回溯
- 求值无法进行时（如遇到 (amb)）失败，调用当时的失败继续，使执行回到前一选择点去考虑其他分支。如果这里已无更多选择，执行就会回到更前面的选择点（那里有以前的失败继续）
- try-again 导致驱动循环直接调用当时的失败继续
- 如果被选分支做了有副作用的操作（例如变量赋值），后来遇到死胡同回溯时，需要在进入其他选择前撤销该副作用
处理方法：让产生副作用的操作生成一个能撤销副作用的失败继续过程，该过程撤销所做修改之后再回溯到前面选择点

amb 求值器的实现

- 总结失败继续（过程）的构造，几种情况：
 - amb 表达式：提供一种机制，使当前选择失败时可以换一个选择
 - 最高层驱动循环：在用尽所有选择的情况下报告失败
 - 赋值：拦截出现的失败并在回溯前消除赋值的效果
- 失败的原因是求值遇到死胡同，两种情况下出现：
 - 用户程序执行 (amb) 时
 - 用户输入 try-again 时
- 一个执行过程失败，它就调用自己的失败继续：
 - 由赋值构造出的失败继续先消除自己的副作用，然后调用该赋值拦截的那个失败继续，将失败进一步回传
 - 如果某 amb 的失败继续发现所有选择已用完时，就调用这个 amb 最早得到的那个失败继续，把失败传到更早的选择点

求值器结构

- amb 求值器的语法过程和数据结构表示、基本的 analyze 过程都和分析求值器一样。只需增加识别 amb 表达式的语法过程

```
(define (amb? exp) (tagged-list? exp 'amb))
(define (amb-choices exp) (cdr exp))
```
- 在 analyze 里增加处理 amb 表达式的分支：

```
((amb? exp) (analyze-amb exp))
```
- 最高层的 ambeval 分析给定的表达式，应用得到的执行过程：

```
(define (ambeval exp env succeed fail)
  ((analyze exp) env succeed fail))
```
- 所有成功继续都有两个参数：一个值参数和一个失败继续。失败继续是无参过程。执行过程的形式都是（三个参数）：

```
(lambda (env succeed fail)
  ;; succeed is (lambda (value fail) ...)
  ;; fail is (lambda () ...)
  ...)
```

求值器结构

- 例，在最上层的 ambeval 调用：

```
(ambeval <exp>
  the-global-environment
  (lambda (value fail) value) ; 两参数的过程，直接给出 value
  (lambda () 'failed))
```

其执行求值 <exp>，最后可能返回求出的值（如果得到值），或返回符号 failed 表示求值失败
后面实现的驱动循环里用了一个更复杂的继续过程，以支持用户输入的 try-again 请求
- amb 求值器实现中，最复杂的东西就是继续过程的构造和传递
阅读下面代码时，请特别注意这方面情况
可以对比这里的代码和前面分析求值器的代码

简单表达式

- 简单表达式的分析和前面一样，但需要传递继续过程

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

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

(define (analyze-variable exp)
  (lambda (env succeed fail)
    (succeed (lookup-variable-value exp env) fail)))

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

这些表达式的求值总成功，所以都调用自己的成功继续。其中查找变量值可能出错，但程序错误并不导致回溯和重新选择

条件表达式

- 条件表达式的处理与前面类似：

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env succeed fail)
      (pproc env
        ;; 求值谓词的成功继续就是得到谓词的值
        (lambda (pred-value fail2)
          (if (true? pred-value)
              (cproc env succeed fail2)
              (aproc env succeed fail2)))
        ;; 谓词求值的失败继续，就是 if 的失败继续
        fail))))
```

谓词执行过程 pproc 的成功继续过程。pproc 成功时会把求出的真假值传给 pred-value

生成的执行过程调用由谓词生成的执行过程 pproc，送给 pproc 的成功继续检查谓词的值，根据其真假调用 cproc 或 aproc

pproc 执行失败时调用 if 表达式的失败继续过程 fail

序列

- 将序列中各表达式的执行过程组合起来

```
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail)
      (a env
        ;; success continuation for calling a
        (lambda (a-value fail2) (b env succeed fail2))
        ;; failure continuation for calling a
        fail)))
  (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)))))
```

执行实际的组合工作，把 b 作为 a 的成功继续

a-value 是序列中第一个表达式的值，丢掉

执行过程的体，调用 loop 把序列中各表达式生成的执行过程组合起来

定义

- 定义变量时先求值其值表达式（调用值表达式的执行过程 vproc），以当时环境、完成实际定义的成功继续和调用时的失败继续 fail 为参数：

```
(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env succeed fail)
      (vproc env
        (lambda (val fail2)
          (define-variable! var val env)
          (succeed 'ok fail2))
        fail))))
```

vproc 的成功继续完成实际的变量定义并成功返回（调用定义表达式的成功继续 succeed）。**这里没有考虑覆盖原有定义的问题**

- 赋值的情况更复杂。其前一部分与处理定义类似，先做值表达式的执行过程，其失败也是整个赋值表达式失败

赋值

- 值表达式求值成功后赋值。为将来失败时可以撤销赋值效果，求值表达式的成功继续把变量原值存在 `old-value` 后再赋值，并把恢复值的动作插入它传给赋值的成功继续 `succeed` 的失败继续过程里，该失败继续过程最后调用 `fail2`，把失败回传

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (vproc env
              (lambda (val fail2)
                (let ((old-value (lookup-variable-value var env)))
                  (set-variable-value! var val env)
                  (succeed 'ok
                          (lambda ()
                            (set-variable-value! var old-value env)
                            (fail2))))))
              fail))))
```

如果 `succeed` 失败，调用这个失败继续将恢复变量原值

赋值式里值表达式的执行过程的成功继续。它保存被赋值变量原值并完成赋值后调用 `succeed`

过程应用

- 过程应用的执行过程较繁琐，但没有特别有趣的问题

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env succeed fail)
      (fproc env
              (lambda (proc fail2)
                (get-args
                 aprocs
                 env
                 (lambda (args fail3)
                   (execute-application proc args succeed fail3))
                 fail2))
              fail))))
```

求出各运算对象的执行过程

`fproc` 的成功继续，它执行各运算对象的执行过程，其成功继续做实际过程调用

`get-args` 调用各运算对象的执行过程

`execute-application` 执行实际的过程调用

过程应用

- `get-args` 顺序执行各运算对象的执行过程

```
(define (get-args aprocs env succeed fail)
  (if (null? aprocs)
      (succeed '() fail)
      ((car aprocs) env
          ;; success continuation for this aproc
          (lambda (arg fail2)
            (get-args
             (cdr aprocs)
             env
             ;; success continuation for recursive
             ;; call to get-args
             (lambda (args fail3)
               (succeed (cons arg args) fail3))
             fail2))
          fail)))
```

第一个运算对象的成功继续，它去求值其余运算对象

注意，成功继续收集求值结果，最终把它们送给实际过程调用

求值第一个运算对象的执行过程，其成功继续求值其余对象

过程应用

- 实现实际的过程调用的执行过程：

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

这个过程较长，实际上比较简单

amb 表达式

■ amb 表达式的执行过程

```
(define (analyze-amb exp)
  (let ((cprocs (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
            ((car choices) env
              succeed
              (lambda () (try-next (cdr choices))))))
        (try-next cprocs))))
```

做出各子表达式的执行过程

试第一个选择分支
其成功是 amb 成功

第一个选择分支的失败继续将去试探其余选择分支

驱动循环

■ amb 的驱动循环的特点是用户可以输入 try-again 要求找下一个成功选择。这一特性使驱动循环比较复杂

■ 循环中用了一个 internal-loop, 它以一个 try-again 过程为参数

- 如果用户的输入为 try-again, 就调用由参数 try-again 得到的过程, 否则就重新启动 ambeval 去求值下一表达式
- ambeval 的失败继续通知用户没有下一个值并继续循环
- ambeval 的成功继续过程输出得到的值, 并用得到的失败继续过程作为 try-again 过程

■ 下面是两个提示串:

```
(define input-prompt ";;; Amb-Eval input:")
(define output-prompt ";;; Amb-Eval value:")
```

```
(define (driver-loop)
  (define (internal-loop try-again)
    (prompt-for-input input-prompt)
    (let ((input (read)))
      (if (eq? input 'try-again)
          (try-again)
          (begin
             (newline) (display ";;; Starting a new problem ")
             (ambeval input
                       the-global-environment
                       ;; ambeval 的成功继续
                       (lambda (val next-alternative)
                         (announce-output output-prompt)
                         (user-print val)
                         (internal-loop next-alternative))
                       ;; ambeval 的失败继续
                       (lambda ()
                         (announce-output ";;; There are no more values of")
                         (user-print input)
                         (driver-loop)))))))
    (internal-loop
     (lambda ()
       (newline)
       (display ";;; There is no current problem")
       (driver-loop))))
```

开始一次新求值

成功继续: 输出并以得到的失败继续过程作为 try-again 过程

失败表示已没有更多的值, 输出信息后再次进入循环

internal-loop 初始的 try-again 过程说明已“无事可做”

总结

■ 惰性求值器对组合式的求值规则与常规的 Scheme 求值器不同。但只需对 eval 和 apply 做不多修改, 就能实现新的求值规则。

可修改 cons 的求值规则, 使表可作为流使用, 支持无穷数据结构

■ 确定性和非确定性计算, 通过搜索和回溯实现非确定性计算

■ 继续是一种过程参数, 它被过程作为最后的动作调用 (且不返回)

问题: 过程的最后调用另一过程, 它的框架会不会永远存在?

请考虑在 C/C++/Java 等语言里的情况和 Scheme 里的情况 (考虑前面讨论的环境模型和求值器), 看看情况怎么样

■ amb 实现技术: 分析被求值表达式生成的执行过程采用一种标准接口 (成功继续和失败继续), 把复杂的控制流隐含在巧妙编织的结构中

■ 要恢复破坏性操作 (如赋值等), 必须设法保存恢复信息。注意: 面向对象技术的设计模式中为这种需要设计了专门的模式