4. 元语言抽象(3)

本节讨论 Scheme 语言的一个改造及其实现

- 在 Scheme 里扩充非确定性计算功能
- amb 语言的概念和基本想法

  amb 的名字来自 ambiguous（歧义，多义）
- amb 程序实例
- amb 语言和搜索
- 非确定性计算的实例
- amb 求值器的实现

Scheme 的扩充：非确定性计算

- 现在考虑一种支持非确定性计算的 Scheme 变形
- 非确定性语言的基本想法：
  - 非确定性的语言里，表达式可以有多个值
  - 可以根据使用者的需要求出一个值或更多的值
- 非确定性计算与流有类似之处，都适合“生成与检查”方面的应用问题

  例如：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：从任意的两集数据中找出符合需要的一对或多对数据
- 前面做过类似工作：对有限序列做过，也对无穷流做过
- 前面采用的方法都是：
  - 先生成一些数对（很多候选）
  - 而后从中过滤出满足要求的数对（和为素数的数对）

非确定性计算：情况

- 在非确定性语言里，计算的描述方式不同
  - 有可能直接按照实际问题的要求写程序
  - 用下面将要开发的语言可以写出更容易理解的程序
- 例如：

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

- 这里好像只是把问题重说了一遍

  而这就是一个合法的非确定性程序

  下面要考虑语言的设计，以及它的实现

非确定性计算：关键思想

- 非确定性计算里最关键的思想：
  - 允许一个表达式有多个可能的值。例如，**an-element-of** 可能返回作为其参数的表里的任何一个元素
  - 在求值这种表达式时，求值器可以自动选出一个值

    可能从可以选的值中任意选出一个

    还需要维持与选择相关的轨迹（知道哪些元素已经选过，哪些没选过。在后续计算中要保证不出现重选的情况）
  - 如果已做选择不能满足后面的要求，求值器就会回到有关的表里再次选择，直至求值成功；或者所有选择都已用完时求值失败
- 非确定性计算的过程将通过求值器自动进行的搜索实现
  - 选择和重新选择的方法和实际过程都隐藏在求值器的实现里，程序员不需要关心，不需要做任何与之相关的事情
  - 这一修改的意义深远，语言扩充了，语义有重要改变

非确定性计算与流处理

- 非确定性求值和流处理有相似的地方

  现在比较一下非确定性求值和流处理中时间的表现形式

- 流处理中，通过惰性求值，松解潜在的（有可能是无穷的）流和流元素的实际产生时间之间的紧密联系
  - 造成的假象是整个流似乎都存在
  - 元素的产生并没有严格的时间顺序
- 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确定
  - 求值器造成的假相：时间好像能分叉
  - 求值器保存着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，换一个探索空间

非确定性计算：amb

- 下面做的非确定性语言基于一种称为 **amb** 的特殊形式
  - **amb** 语言的名字和设计思想来自 **John McCarthy**（**September 4, 1927 – October 24, 2011**）。 **McCarthy** 还提出了“人工智能”这个术语，被人们称为“人工智能之父”。**2011** 年去世
  - **amb** 的名字取自 **ambiguous**（歧义，多种意义）
- 如果定义了前面的非确定性过程后将其送给 **amb** 求值器，会看到：

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

在求值这里的表达式时，**amb** 求值器将从作为实参的两个表里反复选取元素，直至做出一次成功选择

人还可以要求它做进一步的选择，求出更多可能的值

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 <e_1> <e_2> ... <e_n>) 返回几个参数表达式之一的值

- **(list (amb 1 2 3) (amb 'a 'b))** 可能返回下面几个表达式之一：

  (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。无选择的表达式 **(amb)** 没有值，其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，但调用它只返回一个整数

- 非确定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；实现看到它能逐个返回所有选择，这些选择都可能使用

- **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，可以把各分支分派到不同处理器，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，可以：
    - 随机选择，失败了退回重新选择
    - 按某种系统化的方式探查可能的分支。例如，每次总选第一个尚未检查过的分支；失败时退回最近选择点，探查那里的下一个尚未探查过的分支（**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

实例：逻辑迷题

- 深入考虑 amb 求值器的实现之前，先看两个应用

1，求解逻辑迷题。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：这些人各住在哪一层？

- 可以在任何语言里写程序解决这个问题，例如用 Scheme 写

  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))

逻辑迷题

- 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) )  ))
```


实例：自然语言的语法分析

- 用计算机处理自然语言，首先要做语法分析，识别输入句子的结构

  例如，句子在结构上是一个冠词后跟一个名词和一个动词

- 做语法分析前需要辨别词的类属

  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))

- 还需要描述语法（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
- 例如，句子 **the cat eats** 可以分析为：

  (sentence (noun-phrase (article the) (noun cat)) (verb eats))

自然语言的语法分析

- 下面开发一个完成这种分析的简单 **amb** 程序，调用时
  - 处理一个句子，返回作为分析结果的表（上面这样的表）
  - 表中元素表示句子的结构和成分
- 对一个句子，需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)))
```

- 处理名词短语，需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

  这里用一个全局变量 ***unparsed*** 保存被分析句子中尚未分析的部分

  在 ***unparsed*** 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）
- 过程 **parse-word** 描述对一个词的处理和归类

  返回二元表 (词类 单词)

```
(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)))
```

自然语言的语法分析

- 处理开始时，先把 ***unparsed*** 设置为整个句子输入

  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))
```

- 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))
```


自然语言的语法分析

- 在这里 **amb** 的非确定计算功能很有价值
  - 分析中的约束条件很容易用 **require** 描述
  - 很容易扩充更复杂的语法
  - 由于总存在多种选择，可以看到搜索和回溯的作用
- 简单扩充：加入介词和对介词的处理

  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

- 另一扩充（修改）：修改句子的定义

  其中动词短语可以是一个动词，或一个动词后跟一个介词短语（下面定义实际上允许后跟任意多个介词短语）

```
(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

- 另一扩充（修改）：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，下面定义实际上允许任意多个介词短语）

```
(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：基本问题

- 现在考虑 amb 求值器的实现问题
- 常规的 Scheme 表达式可能
  - 求出一个值
  - 或求值不终止
  - 或求值中出错，报告错误
- 非确定性的 Scheme 表达式还可能
  - 求值走入死胡同，失败
  - 求值过程回溯

  显然，这种表达式的解释更复杂
- 下面的工作是修改已有的求值器
  - 准备基于前面的分析求值器实现 amb 求值器
  - 新求值器的特点在于它将生成与前面不同的执行过程

重要概念：“继续”

- 一个重要概念：继续（continuation，延续）
  - “继续”是一种过程参数

    它总在过程的最后一步调用
  - 带有“继续”参数的过程不准备返回

    过程的最后一步是调用某个“继续”过程
  - 是“尾调用”，调用过程的代码已经全部执行完毕

    最后一步就是使用继续过程

    这时调用过程内部的信息已不再有任何价值，可以抛弃
- 有尾递归优化特性的语言实现有可能处理好继续参数，问题：
  - 能不能抛弃或重用调用过程的局部空间？
  - 或以其他方式自动优化运行所需的空间

“继续”和尾递归优化

- 如果语言没有实现尾调用优化
  - 采用继续参数的方式工作，栈空间就会越来越大
  - 因为调用继续参数时，函数并不返回
- 例如（考虑 C 语言的情况）

  typedef int (*Fun)(int)

  int f (..., Fun p) { ...; p(...); return ...; }

  int f (..., Fun p) { ...; return p(...); }

  在函数 f 实际返回前，它占用的栈空间会不会释放

  问题：C 系统能不能在调用 p 时重新使用 f 的运行栈帧？
- 研究课题：具体 C 系统实现了哪些尾调用优化？

  与自递归不同，这里在最后一步调用其他函数

  相互递归，或更复杂的递归中也出现这类情况

实现技术

- 在常规 **Scheme** 语言的分析求值器里
  - **eval** 生成的执行过程要求一个环境参数
  - 这种过程的执行实现原表达式在环境中求值的效果
- 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
    - 一个失败继续
  - 执行过程的体求值结束前的最后一步总是调用这两个过程之一
    - 如果求值工作正常完成并得到结果，就调用由“成功继续”参数得到的那个过程
    - 如果求值进入死胡同，就调用“失败继续”参数过程
- 统一设计，生成的所有执行过程都采用同样的工作模式

实现技术

- 在求值过程中回溯
  - 也通过构造适当的成功继续和失败继续实现
  - 成功继续过程总是有两个参数：
    - 一个是为下步计算所用的参数值，随后的计算将基于它进行
    - 另一个是一个失败继续（过程），如果用得到的值做计算遇到死胡同，就调用这个失败继续
  - 失败继续（过程）的行为是探查另一个非确定性分支

amb 求值器的实现：基本设计

- 在非确定性计算的实现中，需要处理一些新的可能情况
- 遇到非确定性选择点，这时存在多种可取的值，而且不知道选哪个值去继续工作能得到所需结果。处理方法：
  - 从多种可能中选取一个值，同时构造一个失败继续

    失败继续描述了将来求值失败退回这一点时应该做的工作
  - 两者一起送给当时的成功继续过程（因本操作成功得到结果）
  - 传过去的失败继续过程用于在将来出现求值失败时的回溯
- 求值无法进行时（主要是遇到 **(amb)**），失败的处理：
  - 调用当时的失败继续，使执行回到前一选择点去考虑其他可能性
  - 如果回溯一步还是没有更多选择，求值就再次失败，使执行回到更前面的选择点（每步都保存着回到更前面选择点的失败继续）
- **try-again** 导致驱动循环直接调用当时的失败继续

amb 求值器的实现：基本设计

- 有些操作有副作用，需要特殊处理（主要是变量赋值）
  - 如果在一个有副作用的操作之后，求值遇到了死胡同并回溯，就
    - 要求回退到这个操作之前的选择点
    - 回退中经过这种操作时，需要撤销其副作用，恢复原来的状态
  - 处理方法：
    - 每个有副作用的操作都生成一个能撤销副作用的失败继续过程
    - 该过程先撤销本操作做的修改，而后再回溯到前面选择点
- 定义应该怎么处理？
  - 下面实现中并没有很好考虑
  - 定义是新增加的东西，也可能是修改已有定义
  - 请自己考虑合理的处理办法，也可以考虑与赋值类似的方式

amb 求值器的实现

- 构造失败继续（过程）的几种情况：
  - **amb** 表达式：提供某种机制，使当前选择失败时可以换一个选择
  - 最高层驱动循环：在用尽了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：
  - 一个赋值构造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** 早先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 求值器共享分析求值器的一些结构成分：
  - 语法过程
  - 数据结构表示
  - 基本的 analyze 过程
- 只需要增加识别 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))
```


求值器结构

- 求值器实现中几个方面的统一设计
  - 所有成功继续过程都有两个参数
    - 一个值参数
    - 一个失败继续
  - 失败继续是无参过程
  - 执行过程的都有三个参数，基本模式是：

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

- 统一设计带来了良好的系统结构

  把非常复杂的执行流程规范化，很值得学习

求值器结构

- 例，在最上层的 **ambeval** 调用（简单实现）：

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

  它的执行求值 **<*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，以便能支持用户输入的 **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

  在阅读下面代码时，需要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。这些表达式的求值总成功，所以它们都调用自己的成功继续，也都需要传递 **fail** 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)) ))
```

  查找变量的值可能出错，这种程序错误与回溯和重新选择无关

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

- 把序列中各个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**）。这里没考虑覆盖原有定义的问题，也没考虑后来失败了撤销这个定义的问题

- 赋值的情况更复杂一点。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。为了以后出现失败能撤销赋值的效果，求值表达式的成功继续把变量原值存在 **old-value** 后再实际赋值，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))))
```

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**

如果**succeed**失败，调用这个失败继续将恢复变量原值

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)))
```

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

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

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))))
```

这个过程看着比较长，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

- 现在考虑 amb 的驱动循环，特点：
  - 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，整体结构还是比较自然
- 循环中用了一个 internal-loop，它以一个 try-again 过程为参数
  - 如果用户输入 try-again，就调用由参数 try-again 得到的过程，否则就重新启动 ambeval，等待求值下一表达式
  - 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))))
```

基本过程

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

- 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
  - 求值器需要在内部维护相关的信息
- 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？

  比较在 **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），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