函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

http://blog.sina.com.cn/u/3266490431

第十三讲
非确定性计算

用非确定性计算处理整数对问题

- 给定一对正整数的表L1, L2，求所有满足以下条件的整数对:
  1） 一个整数a取自L1, 另一个整数b取自L2　　2） a+b是素数

- 传统方法：生成所有整数对，从中过滤出和为素数的整数对。

- 用非确定性语言来解决，关键程序如下：

```
(define (prime-sum-pair list1 list2) ;求出一对和为素数的整数对
  (let ((a (an-element-of list1)) ;an-element-of 的取值可以有多种选择
        (b (an-element-of list2)))
    (require (prime? (+ a b))) ;require指明了对a,b的限制条件
    (list a b)))

(prime-sum-pair '(1 3 5 8) '(20 35 110))    => (3 20)
```

非确定语言中,表达式可以有多于一个可能的值。一个不合适，就换另一个。

特殊形式amb

- 上面的非确定性程序，需要非确定性求值器，即amb求值器的支持才能运行
- amb求值器支持特殊形式 amb
  - `(amb <e1> <e2> ... <en>)` 的返回值可能是 `<e1> <e2> ... <en>` 中任何一个
  - `(list (amb 1 2 3) (amb 'a 'b))` 的返回值可能是以下之一：
    `(1 a) (1 b) (2 a) (2 b) (3 a) (3 b)`
  - 表达式 `(amb)` 不返回任何值，并导致“计算失败”

amb和require

- 如果要求表达式 exp 必须为真，可以写：

```
(require exp)
```

- require的实现：

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

- 用amb和require实现 an-element-of

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

amb和require

- 下面过程可能返回任何一个大于或等于n的整数：

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

amb实现的关键是深度优先搜索。搜索所有的可能性，结果失败时就回溯到上一个分支点。

驱动循环

- amb求值器读入一个表达式exp，输出第一个满足exp中限制条件的exp的值。

- amb求值器读入符号"try again"，则会回溯，试图输出第二个满足exp中限制条件的exp的值。再接受"try again"，则试图输出第三个值...直到再也找不到合适的值。

- 如果输入除了"try again"之外的表达式，则之前的表达式作废，重新开始一个新问题。

驱动循环

```
(prime-sum-pair '(1 3 5 8) '(20 35 110))
=> (3 20)
try-again
=> (3 110)
try-again
=> (8 35)
try-again
=> There are no more values
(prime-sum-pair '(19 27 30) '(11 36 58))
=> (30 11)
```

非确定性程序实例 --- 逻辑谜题

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？

传统解法：枚举所有可能。

非确定性程序实例 --- 逻辑谜题

- 非确定性程序解法

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5));baker的楼层可能是五个数之一
        (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 ;要求五人楼层各不相同
      (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)))
```

非确定性程序实例 --- 逻辑谜题

```
    (require (not (= (abs (- fletcher cooper)) 1)))
  (list (list 'baker baker)
        (list 'cooper cooper)
        (list 'fletcher fletcher)
        (list 'miller miller)
        (list 'smith smith))))
```

在amb求值器中对 (multiple-dwelling)求值，得到：

```
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
```

自然语言的语法分析

- 分析"The cat eats",需要单词分类表:

名词表:

```
(define nouns '(noun student professor cat class))
```

动词表:

```
(define verbs '(verb studies lectures eats sleeps))
```

冠词表:

```
(define articles '(article the a))
```

- 还需要语法，即描述如何从简单元素组合出复杂元素的规则。规定好语法规则后，程序对"The cat eats"分析的结果可以是:

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

表明"the..."是一个句子,该句子由一个名词短语和一个动词组成。名词短语又由一个冠词和一个名词组成。

语法规则的描述

- 从自然语言的输入流中取走一个句子，返回描述该句子的对象(列表)：

```
define (parse-sentence)
  (list 'sentence  ;规定句子由名词短语加动词组成
        (parse-noun-phrase)
        (parse-word verbs)))
```

- 从自然语言的输入流中取走一个名词短语，返回描述该短语的对象(列表)：

```
(define (parse-noun-phrase)
  (list 'noun-phrase   ;规定名字短语由冠词加名词组成
        (parse-word articles)
        (parse-word nouns)))
```

```
(sentence (noun-phrase (article the) (noun cat))
          (verb eats))  ;句子对象，短语对象和单词对象
```

语法规则的描述

●从自然语言的输入流中取走一个单词，返回描述该单词的对象(列表)：

```
(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)))
```

➢返回值是单词对象。一个单词对象由词性和单词组成，形如： (verb eats)

➢*upparsed* 是剩余的尚未被分析过的自然语言输入流

➢word-list 是某类单词的词汇总表，形如：

➢(noun student professor cat class ...)或

(verb studies lectures eats sleeps ...)

➢该过程被调用时，语法分析器认定*upparsed*中的下一个单词必须在word-list中，否则就认为输入不符合语法规则(类似于编译器)。

语法规则的描述

- 进行语法分析：

```
(define *unparsed* '()) ;*unparsed*用于存放输入流

(define (parse input) ;input是输入流
  (set! *unparsed* input)
  (let ((sent (parse-sentence)))
    (require (null? *unparsed*))
    sent))
```

本过程要求input是一个句子，且该句子后面没有多余东西，才能成功执行。

```
(parse '(the cat eats))
```

=> *(sentence (noun-phrase (article the) (noun cat)) (verb eats))*

语法规则的描述

- 增加介词表和介词短语：

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

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

介词短语由介词和名字组成，形如 ：

```
(prep-phrase (prep for) (noun-phrase (article the) (noun cat)))

"for the cat"
```

语法规则的描述

- 增加动词短语，修改句子定义：

  - 句子是一个名词加一个动词短语
  - 动词短语可以是一个动词，也可以是一个动词短语加上一个介词短语，动词短语可以无限长，即在一个动词后跟无限个介词短语

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

(define (parse-verb-phrase)
  (define (maybe-extend verb-phrase)
    ;第一次调用maybe-extend时ver-phrase形如(verb eats)
    (amb verb-phrase
         (maybe-extend (list 'verb-phrase
                             verb-phrase
                             (parse-prepositional-phrase)))))
  (maybe-extend (parse-word verbs)))
```

语法规则的描述

- 动词短语示例：

```
(define (parse-verb-phrase)
  (define (maybe-extend verb-phrase)
    ;第一次调用maybe-extend时ver-phrase形如(verb eats)
    (amb verb-phrase
         (maybe-extend (list 'verb-phrase
                             verb-phrase
                             (parse-prepositional-phrase)))))
  (maybe-extend (parse-word verbs)))
```

```
(verb-phrase
   (verb-phrase (verb eats)
         (prep-phrase (prep with) (noun-phrase (article the) (noun cat))))
   (prep-phrase (prep for) (noun-phrase (article a) (noun dog))))
```

"eats with the cat for a dog"

语法规则的描述

- 增强名词短语(前面的名词短语以后称为简单名词短语)：

  ➢名词短语可以是一个简单名词短语，也可以是一个名词短语加一个介词短语。名词短语可以无限长。

```
(define (parse-simple-noun-phrase);简单名词短语 = 冠词+名词
  (list 'simple-noun-phrase
        (parse-word articles)
        (parse-word nouns)))

(define (parse-noun-phrase)   ;名词短语
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
         (maybe-extend (list 'noun-phrase
                             noun-phrase
                             (parse-prepositional-phrase)))))
  (maybe-extend (parse-simple-noun-phrase)))
```

语法规则的描述

- 名词短语示例：

```
(define (parse-noun-phrase)   ;名词短语
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
         (maybe-extend (list 'noun-phrase
                             noun-phrase
                             (parse-prepositional-phrase)))))
  (maybe-extend (parse-simple-noun-phrase)))
```

```
(noun-phrase
  (noun-phrase (simple-noun-phrase (article the) (noun horse))
       (prep-phrase (prep with) (simple-noun-phrase (article the) (noun cat))))
 (prep-phrase (prep on) (simple-noun-phrase (article a) (noun desk))))
```

"the horse with the cat on a desk"

语法规则的描述

- 句子分析示例1：

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

结果：

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

```
(define (parse input)
  (set! *unparsed* input)
  (let ((sent (parse-sentence)))
    (require (null? *unparsed*))
    sent))
```

语法规则的描述

- 句子分析示例2：

```
(parse '(the professor lectures to the student with the cat))
```

amb求值器可以返回所有可能结果。结果一(猫是跟着教授的）：

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

语法规则的描述

- 句子分析示例2：

```
(parse '(the professor lectures to the student with the cat))
```

输入 try again 得到结果二(猫是跟着学生的) ：

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

```
(parse '(the professor lectures to the student with the cat))
```

```
(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)))

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

```
(parse '(the professor lectures to the student with the cat))

(define (parse-noun-phrase)   ;名词短语
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
         (maybe-extend (list 'noun-phrase
                             noun-phrase
                             (parse-prepositional-phrase)))))
  (maybe-extend (parse-simple-noun-phrase)))
```

amb求值器的基本思路

●本质上是解释器及其生成的代码执行了深度优先搜索的工作，在发现规定条件不满足时，会回溯，做不同的选择后再继续。

●对于每个**amb**表达式，都先取第一个选择作为其值，然后继续执行。用户程序执行到失败的结果时，自动回溯到最近的一个**amb**表达式，在该**amb**表达式中取下一个选择，作为其值，然后继续。

●此后讨论的都是用 **ambeval** 对一个表达式进行求值的经过。

```
(ambeval <exp> glb-env glb-succeed glb-fail)
```

amb求值器的工作流程

- amb求值器对表达式exp求值时，执行以下过程：

```
(ambeval <exp> glb-env glb-succeed glb-fail)
```

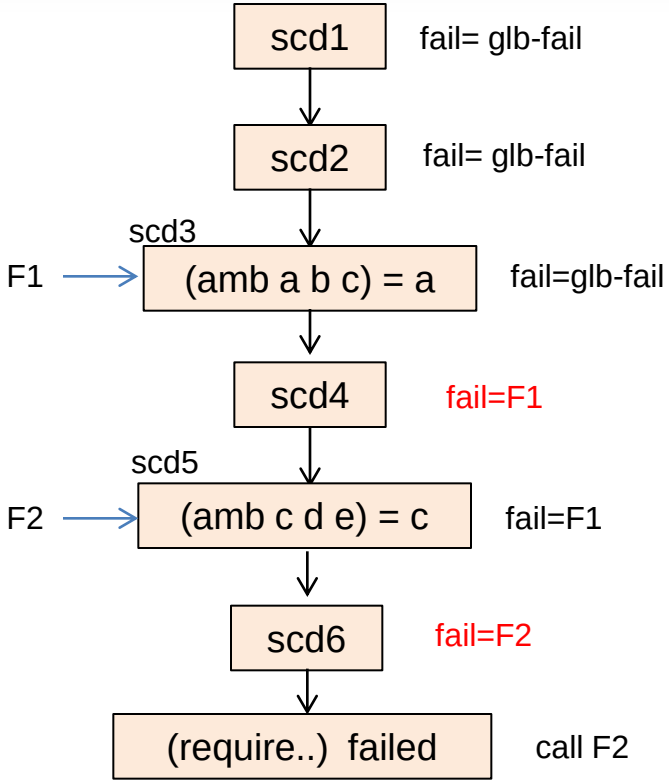
glb-succeed glb-fail由求值器编写者定义：

```
(define glb-succeed
  (lambda (val fail)
    (display "succeed,val = " )(display val) (newline)))

(define glb-fail
  (lambda ()
    (display "glb-failed") (newline)))
```

- exp求值成功，则会执行(glb-succeed val XXX)
  val是exp的值,XXX是失败函数。如果glb-succeed调用XXX，则会导致求exp的下一个可能值。
- exp求值不成功，则会执行 (glb-fail)

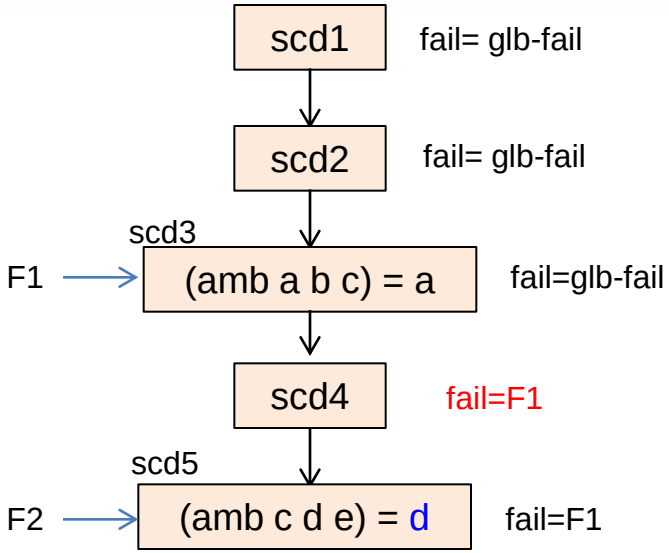
amb求值器的工作流程

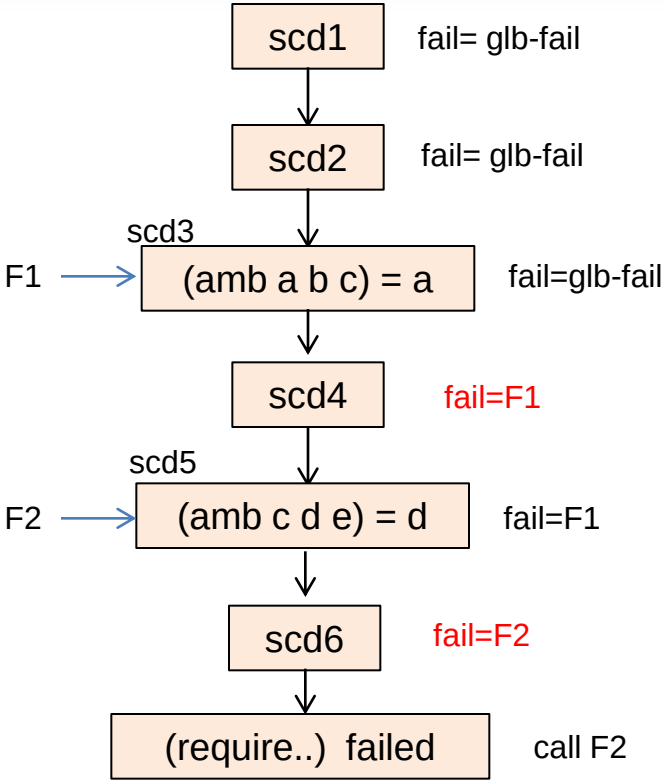


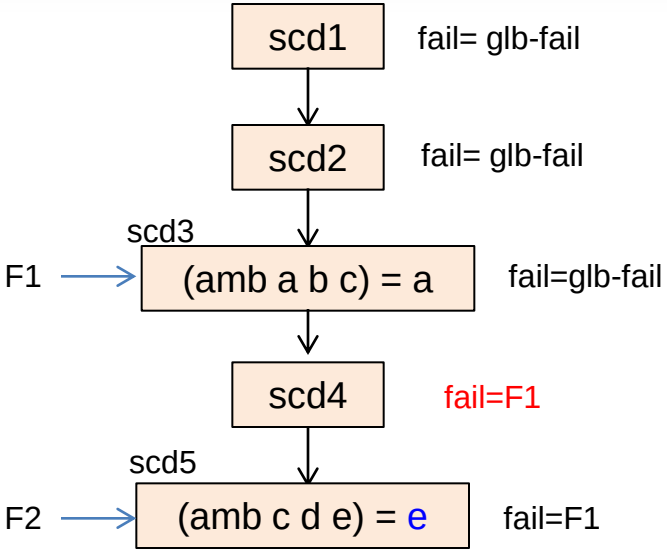
图中scd代表步骤，即成功函数

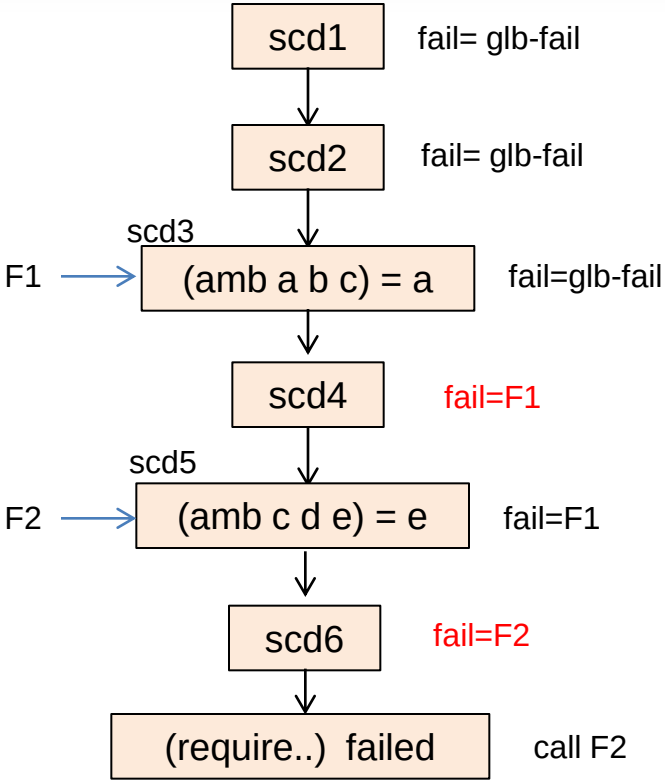
- amb求值器实现回溯的关键在于，用户程序执行的每个步骤（每个表达式的求值）中，都会带有一个“失败函数”fail。当require 失效时，require表达式对应的fail就会被调用，引起回溯。

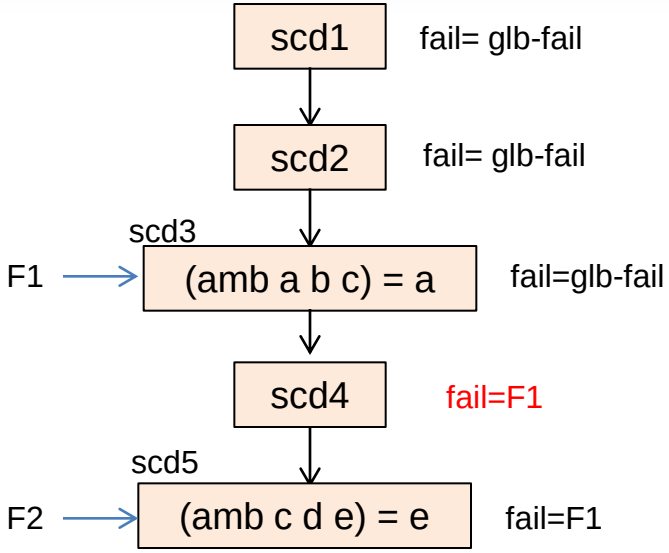
- 最初的失败函数是 glb-fail。失败函数在表达式求值过程中作为参数向下传递。每经过一个 amb表达式，向下传递的fail就被修改。


scd1
fail= glb-fail
scd2
fail= glb-fail
scd3
F1
(amb a b c) = a
fail=glb-fail
scd4
fail=F1
scd5
F2
(amb c d e) = d
fail=F1


scd1
fail= glb-fail
scd2
fail= glb-fail
scd3
F1
(amb a b c) = a
fail=glb-fail
scd4
fail=F1
scd5
F2
(amb c d e) = d
fail=F1
scd6
fail=F2
(require..) failed
call F2


scd1
fail= glb-fail
scd2
fail= glb-fail
scd3
F1
(amb a b c) = a
fail=glb-fail
scd4
fail=F1
scd5
F2
(amb c d e) = e
fail=F1


scd1
fail= glb-fail
scd2
fail= glb-fail
scd3
F1
(amb a b c) = a
fail=glb-fail
scd4
fail=F1
scd5
F2
(amb c d e) = e
fail=F1
scd6
fail=F2
(require..) failed
call F2



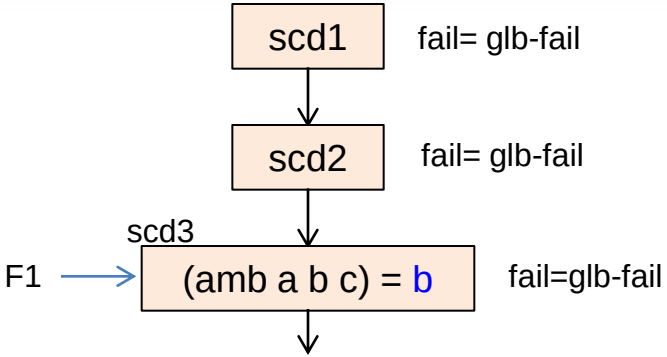
本 amb 表达式失败，call F1

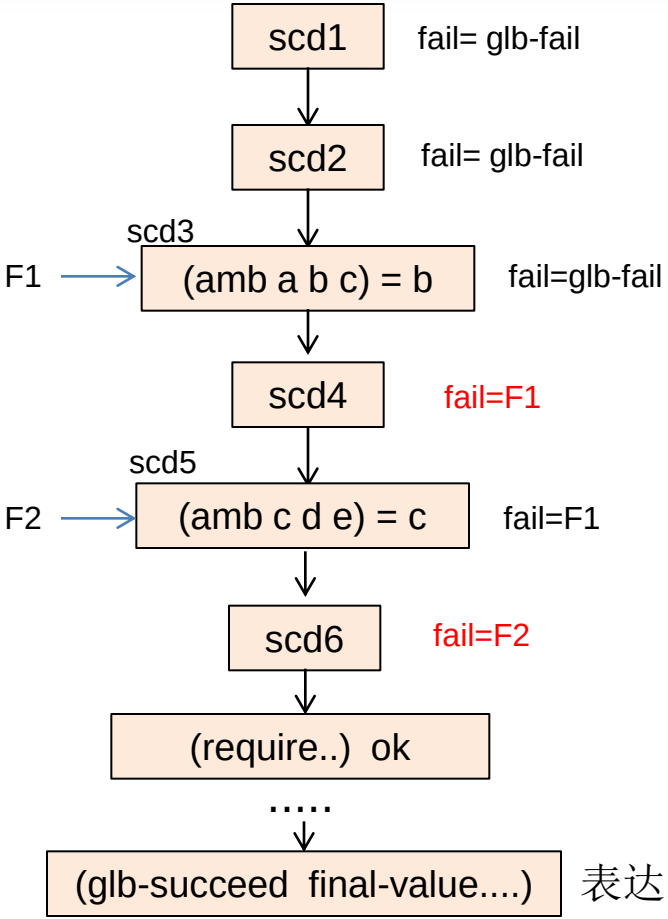
- 用户程序中**require**的实现：

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

●解释器执行 **(amb)** 会引发对 **fail**的调用，导致回溯到最晚的一个 **amb** 表达式求值处，选择一个新值。如果没有新值可选，则又会引发对另一个**fail**的调用，回溯到上一个**amb**表达式...

●如果最早的**amb**表达式所有选择都用完，则这个表达式**exp**求值失败，**glb-fail**被调用


scd1
fail= glb-fail
scd2
fail= glb-fail
scd3
F1
(amb a b c) = b
fail=glb-fail


scd1
fail= glb-fail
scd2
fail= glb-fail
scd3
F1
(amb a b c) = b
fail=glb-fail
scd4
fail=F1
scd5
F2
(amb c d e) = c
fail=F1
scd6
fail=F2
(require..) ok
.....
(glb-succeed final-value....)

表达式求值结束!!

amb求值器的实现

- 基于旧的分析和执行分离的求值器来实现amb求值器
- 添加amb相关过程

```
(define (amb? exp) (tagged-list? exp 'amb))
(define (amb-choices exp) (cdr exp))
```

amb求值器的实现

- 在 (analyze exp)中增加一个分支：

```
(define (analyze exp)
    (cond ((self-evaluating? exp)
           (analyze-self-evaluating exp))
          ((quoted? exp) (analyze-quoted exp))
          ((variable? exp) (analyze-variable exp))
          ((assignment? exp) (analyze-assignment exp))
          ((definition? exp) (analyze-definition exp))
          ((if? exp) (analyze-if exp))
          ((lambda? exp) (analyze-lambda exp))
          ((begin? exp) (analyze-sequence (begin-actions exp)))
          ((cond? exp) (analyze (cond->if exp)))
          ((let? exp) (analyze (let->combination exp)))
          ((amb? exp) (analyze-amb exp))
          ((application? exp) (analyze-application exp))
          (else
           (error "Unknown expression type -- ANALYZE" exp))))
```

- 顶层求值函数：

```
(define (ambeval exp env succeed fail)
  ((analyze exp) env succeed fail))
```

```
(ambeval '(+ 3 4) exp glb-env
         glb-succeed glb-fail)
```

amb求值器的实现

- 解释器中大量用到“成功函数”和“失败函数”

- 成功函数形如：

```
(lambda (value fail) ...)
```

value是刚刚求得的一个表达式的值，fail是失败函数。成功函数利用 value 进行下一个表达式的求值，并把 fail传递下去。

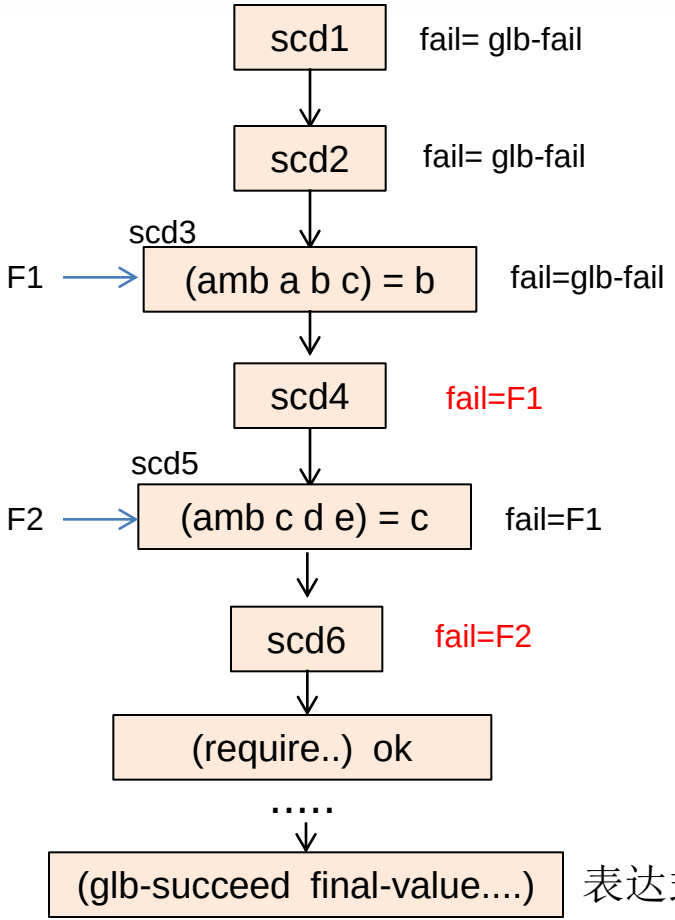
- 失败函数形如：

```
(lambda () ...)
```

- 示例：

```
(ambeval <exp>   ;若成功则输出 <exp>的值，失败则输出 fail
        glb-env
        (lambda (value fail) value)
        (lambda () 'failed))
```

amb求值器的实现



表达式求值结束!!

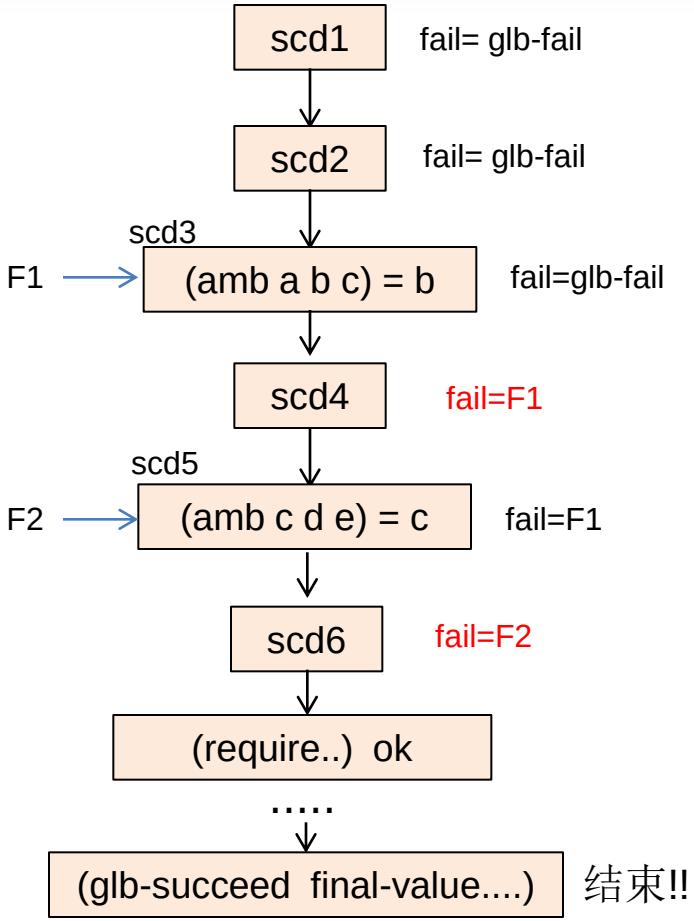
```
(ambeval <exp> glb-env
         glb-succeed glb-fail)
```

对表达式**exp**的求值就是通过程序运行时由一系列解释器自动生成的“成功函数” 的嵌套调用实现的。**fail**也沿着“成功函数”被传递。最里层的“成功函数”就是**glb-succeed**。

每个成功函数**F**都是一个闭包,闭包内有一个变量**succeed**，表示下一个成功函数。**F**可以直接调用**succeed**，也可以用**succeed**作为参数调用一个分派函数,借此实现**succeed**的调用(还有别的方式调用**succeed**)

在分析阶段，解释器生成一系列的 “成功函数”内部也会包含对解释器中的各个函数的调用 --- 用于对子表达式求值。

amb求值器的实现




```
(ambeval <exp> glb-env
         glb-succeed glb-fail)
```

对表达式exp的求值总是要通过按顺序求值一系列子表达式p1,p2,p3...pn (pn就是整个表达式<exp>的值)

p_i对应的成功过程scd_i 。p_i的值,记为 (val p_i)求出后，会执行:

$(scd_i$ (val p_i) $fail_i)$;$fail_1$ = glb-fail

scd_i是个闭包，内部包含scd_{i+1} 。在scd_i执行的过程中，会执行 $(scd_{i+1}$ (val p_{i+1}) $fail_{i+1})$

如果<exp>成功求值，最后的 scd_n会执行glb-succeed，然后各scd_i一层层返回。

在p_i是amb表达式的情况下,$fail_i$会变化。

amb求值器的实现

假设已经有过：

```
(ambeval '(define (inc x) (+ x 1)) glb-env glb-succeed glb-fail)
```

分析下面的求值过程：

```
(ambeval '(if (> 2 3) (inc 3) (* 2 (inc 8)))
         glb-env glb-succeed glb-fail)
```

整个表达式的求值被分解成p1,p2...p13依次求值，p13的值就是整个表达式的值。

```
p1 >
p2 2
p3 3
p4 (> 2 3) val = #f
p5 *
p6 2
p7 inc
p8 8
p9 +
p10 x
p11 1
p12 (inc 8) val = 9
p13 18
```

amb求值器的实现

- 分派函数

解释器中出现的形如

```
(lambda (env succeed fail) ....)
```

的函数。每个分派函数对应于一个表达式p。解释器对p进行分析的结果，就是得到一个分派函数。有的情况下，分派函数执行过程中，会用env求得 p的值(val p)，然后直接调用

```
(succeed (val p) fail)   ;(p是amb时fail会变化)
```

如果没有直接调用 succeed，也会间接调用 succeed

通过调用 succeed,去求下一个表达式的值

amb求值器的实现

- 解释器对每个表达式进行分析的结果，都是分派函数！

```
(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))))
```

amb求值器的实现

- 解释器对每个表达式进行分析的结果，都是分派函数！

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp))) ;pproc是分派函数
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env succeed fail) ;分派函数 L7
      (pproc env ;pproc 以 (if-predicate exp)的值为参数去调用S3
             (lambda (pred-value fail2) ;成功函数，S3
               (if (true? pred-value)
                   (cproc env succeed fail2)
                   (aproc env succeed fail2)))
             fail))))
```

amb求值器的实现

- 解释器对每个表达式进行分析的结果，都是分派函数！

```
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail) ;分派函数L6
      (a env
         (lambda (a-value fail2)   ;成功函数S2
           (b env succeed fail2))
         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))))
```

amb求值器的实现

- analyze-sequence工作过程：

```
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail) ;分派函数L6
      (a env
         (lambda (a-value fail2)   ;成功函数S2
           (b env succeed fail2))
         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))))
```

(analyze-sequence (begin p1 p2))
(loop #p1 (#p2))
(loop (seq #p1 #p2) ())
(seq #p1 #p2)
=>(lbd (env succeed fail)
    (#p1 env
        (lbd (a-value fail2)
            (#p2 env succeed fail2))
        fail)))
#p1是 (analyze p1)的结果，是个分派函数

amb求值器的实现

- analyze-sequence工作过程：

```
(analyze-sequence (begin p1 p2 p3))
```

返回值是下面这个分派函数：

```
(lbd (env succeed fail)
     (#p1 env
          (lbd (a-value fail2) ;成功函数SCD1
               (#p2 env
                    (lbd (a-value fail3) ;成功函数SCD2
                         (#p3 env succeed fail3))
                    fail2))
          fail))

(ambeval '(begin 1 2 3) glb-env glb-succeed glb-fail)
```

amb求值器的实现

- analyze-sequence工作过程：

```
(ambeval '(begin 1 2 3) glb-env glb-succeed glb-fail)
((analyze '(begin 1 2 3)) glb-env glb-succeed glb-fail)
((analyze-sequence '(begin 1 2 3)) glb-env glb-succeed glb-fail)
(SCD1 (val #1) glb-fail)
(SCD2 (val #2) glb-fail)
(succeed (val #3) fail)
(glb-succeed (val #3) glb-fail)
```

```
(lbd (env succeed fail)
     (#1  env
          (lbd (a-value fail2) ;成功函数SCD1
               (#2 env
                    (lbd (a-value fail3) ;SCD2
                         (#3 env succeed fail3))
                    fail2))
          fail))
```

amb求值器的实现

- 解释器对每个表达式进行分析的结果，都是分派函数！

```
(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) ;做好定义，然后再成功继续
               (succeed (void) fail2)) ;define表达式的返回值是 (void)
             fail))))
```

amb求值器的实现

- 处理过程应用：

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env succeed fail)
      (fproc env
             (lambda (proc fail2) ;成功函数 L
               (get-args aprocs ;proc是fproc在env中的值，是个函数对象
                         env
                         (lambda (args fail3) ;成功函数 S
                           (execute-application
                            proc args succeed fail3))
                         fail2))
             fail))))
```

对比老的解释器的:

```
(lambda (env)
    (execute-application (fproc env)
    (map (lambda (aproc) (aproc env)
```

aprocs里的每个元素#pi,对应于(但不是）exp中的一个实参。#pi基本上形如:

```
(lambda (env succeed fail) (succeed (val #pi) fail))
```

(val #pi)是 #pi所对应的那个实参值。(fproc env ...)的目的，就是要做到

(excute-application proc args succeed fail),其中, args = ((val #p1) (val #p2) .....)

关键在于提取 (val #pi)

amb求值器的实现

```
(define (get-args aprocs env succeed fail)
  (if (null? aprocs)
      (succeed '() fail)
      ((car aprocs) env
                    (lambda (arg fail2) ;成功函数K1
                      (get-args (cdr aprocs)
                                env
                                (lambda (args fail3) ;成功函数 K2
                                  (succeed (cons arg args)
                                           fail3))
                                fail2))
                    fail)))
```

amb求值器的实现

- (analyze-application exp)执行序列分析：

```
((analyze-application exp) succeed fail) ;假设exp中实参是 p1,p2
(fproc env L[aprocs,succeed,env] fail);L是闭包，中括号表示闭包里的变量及其值
(L (val fproc) fail) ;(val fproc)是个函数对象 (procedure
                                                    (..) ....),
(get-args aprocs env  S[proc:(val fproc),succeed] fail)
(get-args (#p1 #p2) env S fail)

(#p1 env K1[scd:S,aprocs:(#p1 #p2)] fail) ;scd指的是K1中的 succeed
(K1[scd:S,aprocs:(#p1 #p2)] (val #p1) fail)
(get-args (#p2) env K2[scd:S,arg:(val #p1)] fail)
(#p2 env K1[scd:K2[scd:S,arg:(val #p1)],aprocs:(#p2)] fail)
(K1[scd:K2[scd:S,arg:(val #p1)],aprocs:(#p2)]  (val #p2) fail)
(get-args () env
            K2[scd:K2[scd:S,arg:(val #p1)],arg:(val #p2)]
            fail)
(K2[scd:K2[scd:S,arg:(val #p1)],arg:(val #p2)] () fail)
(K2[scd:S,arg:(val #p1)] (cons (val #p2) ()) fail)
(S (cons (val #p1) (cons (val #p2) ())) fail)
(S ((val #p1) (val #p2)) fail)
(exectute-application (val fproc ) ((val #p1) (val #p2)) succeed fail)
```

amb求值器的实现

```
(define (execute-application proc args succeed fail)
;proc是函数对象,args是实际参数列表
  (cond ((primitive-procedure? proc)
         (succeed (apply-primitive-procedure proc args)
                  fail))
        ((compound-procedure? proc)
         ((procedure-body proc) ;是个分派函数
          (extend-environment (procedure-parameters proc)
                              args
                              (procedure-environment proc))
          succeed
          fail))
        (else
         (error
          "Unknown procedure type -- EXECUTE-APPLICATION"
          proc))))
```

amb求值器的实现

- 求值 amb 表达式

```
(define (analyze-amb exp) ;require中的(amb)也会经由这里来执行，其结果就是调用 fail
  (let ((cprocs (map analyze (amb-choices exp))))
    (lambda (env succeed fail) ;分派函数
      (define (try-next choices)
        (if (null? choices)
            (fail);回溯。选择用完了也会走这里
            ((car choices) env ;(car choice)是分派函数
                           succeed
                           (lambda () ;新的失败函数 F1，是个闭包,可以选amb的下一个值
                             (try-next (cdr choices))))))
      (try-next cprocs))))
```

选取一个amb的值(car choice)后，执行：

```
(succeed (val (car choice)  F1)
;继续剩下的任务，且从此往下传的失败函数改变为F1
```

amb求值器的实现

- 测试求值器

开始求值前必须做的：

```
(define rq '(define (require p)
  (if (not p)  (amb)(void))))

(define glb-succeed
  (lambda (val next)
    (display val) (newline)))

(define glb-fail
  (lambda ()
    (display "glb-failed") (newline)))

(define glb-env (setup-environment))
(ambeval rq glb-env glb-succeed glb-fail) ;让require可用
```

amb求值器的实现

- 测试求值器

```
(ambeval '(amb 1 2 3 4) glb-env glb-succeed glb-fail)
=>1

(define t2 '(define (test2);amb做运算符
  (let ((op (amb - +))
        (k (amb 1 2 3)))
    (let ((r (op 4 k)))
      (require (< r 2))
      r))))
(ambeval t2 glb-env glb-succeed glb-fail)
(ambeval '(test2) glb-env glb-succeed glb-fail)
=>?
```

amb求值器的实现

- 测试求值器

```
(ambeval '(amb 1 2 3 4) glb-env glb-succeed glb-fail)
=>1

(define t2 '(define (test2);amb做运算符
  (let ((op (amb - +))
        (k (amb 1 2 3)))
    (let ((r (op 4 k)))
      (require (< r 2))
      r))))
(ambeval t2 glb-env glb-succeed glb-fail)
(ambeval '(test2) glb-env glb-succeed glb-fail)
=>1
```

amb求值器的实现

- 测试求值器

```
(define t3 '(define (test3) ;amb做if的条件
    (let ((m (if (amb true false)
                 3
                 4)))
      (require (= m 4))
      m)))
(ambeval t3 glb-env glb-succeed glb-fail)
(ambeval '(test3) glb-env glb-succeed glb-fail)
```

```
=>4
```

amb求值器的实现

- 测试求值器

xxxx处写什么,可以让

```
(ambeval '(amb 1 2 3) glb-env XXXX glb-fail)
```

输出结果是:

```
1
2
3
glb-failed
```

amb求值器的驱动循环

```
(define (driver-loop)
  (define (internal-loop try-again)
    (let ((input (read)))
      (if (eq? input 'try-again)
          (try-again) ;输入为try-again时，会回溯到最后的amb表达式求另一个解
          (begin (newline)(display ";;; Starting a new problem ")
            (ambeval input
                     glb-env
                     (lambda (val next-alternative);成功函数
                                    ;next-alternative是失败函数
                       (user-print val)
                       (internal-loop next-alternative))
                                    ;第一次成功以后，try-again就变成失败函数
                     (lambda () ;失败函数
                       (display ";;; There are no more values of")
                       (user-print input)
                       (driver-loop)))))))
  (internal-loop
   (lambda ()
     (newline) (display ";;; There is no current problem")
     (driver-loop))))
```

amb求值器的驱动循环

输入输出:

```
(amb 1 2 3)
1
try-again
2
try-again
3
try-again
;;; There are no more values of(amb 1
2 3)
(let ((m (+ (amb 1 2 3) (amb 4 5 6))))
  (require (> m 7))
  m)
8
try-again
8
```