编程语言的邀请

前言

编程语言是对于编程语言的研究, 其核心是语义学. 任何对于编程语言的学习都应该从刻画编程语言的语义开始. 我们以Scheme程序呈现了编程语言的基本想法, 它来源于John McCarthy (注: Lisp语言的设计者, 被认为是人工智能之父) 为了展现Lisp的表达力而编写的元解释器.

1 句法

本章我们呈现了一个简单的编程语言, 以作为之后章节的基础.

1.1 抽象句法树

我们的程序常以程序文本作为输入或输出, 实际上是程序文本的表示. 至于如何表示程序, 让我们回忆句法的定义. 句法描述了由较小的语言成分构造较大的语言成分之规则. 这暗示着, 在最理想的情况下, 程序文本具有上下文无关的树结构, 即抽象句法树. 将程序文本从字符序列的表面形式转换为抽象句法树的过程被称为句法分析. 我们没有涵盖句法分析, 鉴于它不是编程语言的最核心内容.

我们遵循Lisp界的惯例 (Scheme是Lisp的现代方言), 将抽象句法树表示为符号表达式 (S-exp, symbolic expression). 实际上, 我们将Scheme当作描述抽象句法树的元语言. 例如, 表达式`'(+ 1 (* 2 3))`描述了句法树`(+ 1 (* 2 3))`, 表达式`(+ 1 (* 2 3))`描述了句法树7. 我们也将不加限制地使用`quasiquote`形式, 它来源于逻辑学家Quine的准引用. 例如, 表达式`` `(= (+ 1 (* 2 3)) ,(+ 1 (* 2 3))) ``描述了句法树`(= (+ 1 (* 2 3)) 7)`. (注记: 鉴于这种记号与数学实践的冲突和元层次的不明晰性, 我们只能部分地追求对于语言的运用的一致性, 以在严格性与可读性之间达成平衡. 实际上, 我们寄希望于读者通过已有的经验和上下文来精确地推断符号的含义.)

1.2 BNF文法

BNF文法是描述句法的标准工具.

鉴于BNF文法总以变种形式出现, 我们以一个例子刻画其用法.

```
<exp> ::= <int>
       |  <bool>
       |  (if <exp> <exp> <exp>)
       |  (<op> <exp> <exp>)
<op> ::= + | - | * | =
```

以尖括号括起来的符号代表句法范畴, 即句法树的集合. 其余的符号代表字面本身. `::=`的含义是定义, 而`|`的含义近于或. 每个出现于右侧的句法范畴都意图被其一个元素替换以产生左侧的一个实例. 我们预先定义了一些范畴, `<int>`代表Scheme整数的集合, `<bool>`代表Scheme布尔的集合, `<var>`代表Scheme符号的集合.

1.3 解释器的风味

解释是赋予句法语义的过程. 一个解释器是一个程序, 其以程序文本为输入, 以值为输出. 于是, 解释器可以被视为对于其所解释的语言的语义的描述.

本节我们呈现了一个简单的解释器, 其为句法导向的结构递归. 为了方便地解构句法树, 我们引入了用于模式匹配的宏`match`, 其定义见附录.

```
(define (interp exp)
  (match exp
```

```
(,int (guard (integer? int)) int)
(,bool (guard (boolean? bool)) bool)
((if ,e1 ,e2 ,e3)
 (let ((v1 (interp e1)))
   (if (true? v1)
       (interp e2)
       (interp e3))))
((,op ,e1 ,e2)
 (guard (memq op '(+ - * =)))
 (let* ((v1 (interp e1))
        (v2 (interp e2)))
   (case op
     ((+) (+ v1 v2))
     ((-) (- v1 v2))
     ((*) (* v1 v2))
     ((=) (= v1 v2)))))))
```

鉴于对于真的解释在某种意义是开放的, 所以我们将其抽象成了过程`true?`.

```
(define (true? val)
  (if (boolean? val)
      val
      (error 'interp "the predicate of if is not of type bool")))
```

以上的解释器相当简单直接, 只有一点值得说明, 即我们使用了`let*`而不是`let`以确保求值顺序是自左向右的. 之后的语言也将采用自左向右求值的约定, 这免除了不少微妙的麻烦.

2 绑定

前一章的语言是无趣的, 因而本章我们为语言添加了绑定机制, 探索了绑定的行为.

2.1 环境

至于如何实现绑定机制, 替换是一种直接的选择. 然而, 正确的替换的定义是微妙的, 并不如人们所想象的那样简单. 实际上, 历史上著名的逻辑学家, 比如Frege, Hilbert, Russell等, 都在这个问题上栽过跟头. 因此, 我们另辟蹊径, 为解释器添加了一个新的参数—环境.

环境是求值的数据上下文, 其确定了表达式的每个变量的意义. 一个环境是一个从变量的集合到值的集合的映射. 如果采用抽象数据类型的观点, 我们需要确定环境的接口以及其所需要满足的约束条件.

1. `empty-env`: `(empty-env)`产生一个空的环境;

2. `extend-env`: 对于变量`var`, 值`val`, 环境`env`, `(extend-env var val env)`返回一个扩展了的环境, 其中变量`var`的值是`val`, 而其余情况与`env`一致;

3. `apply-env`: 对于环境`env`, 变量`var`, `(apply-env env var)`将环境`env`应用于变量`var`, 返回环境`env`里变量`var`所关联至的值.

以上三个接口过程需要满足如下约束条件.

1. 对于变量`var`, `var^`, 值`val`, 环境`env`, `(apply-env (extend-env var val env) var^)`的值, 在`var`等于`var^`的情况下为`val`, 否则的话等于`(apply-env env var^)`;

2. 对于变量`var`, `(apply-env (empty-env) var)`没有意义, 于是我们选择引起一个异常.

现在我们给出环境的接口的两种不同实现.

```
(define (empty-env) '())
(define (extend-env var val env)
  (cons (cons var val) env))
(define (apply-env env var)
  (cond ((assq var env) => cdr)
        (else (error 'apply-env "unbound variable ~s" var))))
```

```
(define (empty-env)
  (lambda (var)
    (error 'apply-env "unbound variable ~s" var)))
(define (extend-env var val env)
  (lambda (var^)
    (if (eq? var var^)
        val
        (apply-env env var^))))
(define (apply-env env var)
  (env var))
```

第1种实现将环境表示为关联列表, 而第2种实现将环境表示为过程, 但这两种实现的确都满足接口.

2.2 一个带有绑定的语言

本节我们呈现一个带有绑定的语言.

首先我们给出语言的句法.

```
<exp> ::= <int>
       |  <bool>
       |  <var>
       |  (if <exp> <exp> <exp>)
       |  (let <var> <exp> <exp>)
       |  (<op> <exp> <exp>)
<op> ::= + | - | * | =
```

相较于前一章的语言的句法, <exp>的定义增加了两个产生式. 这两个产生式显然是对偶的, 一个引入变量绑定, 另一个则是变量引用.

从此刻起, 我们做出一个句法上的约定, 这可以免除许多微妙的麻烦, 即任何绑定结构都不应该引入与"关键词"相同的名字, 例如这里的关键词是`if`, `let`, `+`, `-`, `*`, `=`.

现在我们给出解释器, 注意它多了一个代表环境的参数. 实际上, 只有解释新增产生式的部分是有趣的.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (apply-env env var))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((let ,x ,e ,body)
     (let* ((v (interp e env))
            (env^ (extend-env x v env)))
       (interp body env^)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
```

```
      (let* ((v1 (interp e1 env))
             (v2 (interp e2 env)))
        (case op
          ((+) (+ v1 v2))
          ((-) (- v1 v2))
          ((*) (* v1 v2))
          ((=) (= v1 v2)))))))
```

让我们以一个简单的例子结束本节.

```
> (interp '(let x 0
              (let x (+ x 1)
                (let x (+ x 2)
                  (+ x 3))))
           (empty-env))
6
```

这个例子呈现了变量遮盖的现象. 每个变量引用以最内层的绑定为准, 而外层的绑定被遮住了.

2.3 一个带有过程的语言

什么是过程? 这个问题并不像看上去那么简单, 而且John McCarthy在设计Lisp时就掉进了坑里. 实际上, 在很长一段时间内, 动态作用域 (之后我们将实现动态作用域的语言) 一直被视为Lisp的本质特征之一, 而这招致了不少批评.

请看以下Scheme的读取求值输出循环交互.

```
> (let ((two 2))
    (let ((double (lambda (x) (* x two))))
      (let ((two 3))
        (double 4))))
8
```

这有什么可希奇的呢! 不过, 对于一个使用远古Lisp方言的程序员而言, 他很可能认为结果应该是`12`. 这是由于他会认为`double`的值是`(lambda (x) (* x two))`, 一个列表而已.

现代编程语言学家十分清楚这是完全错误的设计和看法, 但这对于当时的人们而言尚不明了. 实际上, 一个过程不仅是定义过程的代码本身, 还包括过程所在的环境. 这个环境确定了过程的自由变量的意义, 使其成为封闭形式. 因此, 人们将过程的代码 (形式参数和过程体) 和过程所在的环境合称为闭包 (closure).

如果一个语言的过程的自由变量的意义由创建过程的环境决定, 那么称这个语言采用了词法作用域. "词法"应该理解为"可以从程序文本推断出来", 这在动态的求值过程之前, 因而其也被称为静态作用域.

既然明白了什么是过程, 现在我们呈现一个带有过程的(词法作用域的)语言.

首先仍然是给出语言的句法.

```
<exp> ::= <int>
       |  <bool>
       |  <var>
       |  (if <exp> <exp> <exp>)
       |  (lambda <var> <exp>)
       |  (let <var> <exp> <exp>)
       |  (<op> <exp> <exp>)
       |  (<exp> <exp>)
<op> ::= + | - | * | =
```

这个句法相较于前一节的语言, `<exp>`新增了两个产生式, 它们的确也是对偶的. 一个用于创建过程, 另一个用于应用过程.

现在我们给出解释器, 同样只有对于新增的产生式的解释是有趣的.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (apply-env env var))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x ,body)
     (lambda (arg)
       (interp body (extend-env x arg env))))
    ((let ,x ,e ,body)
     (let* ((v (interp e env))
            (env^ (extend-env x v env)))
       (interp body env^)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env)))
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))
    ((,rator ,rand)
     (let* ((closure (interp rator env))
            (arg (interp rand env)))
       (closure arg)))))
```

我们将被解释的语言的闭包表示为Scheme的闭包, 这是一种过程性的表示. 如果我们采取数据抽象的观点, 将闭包视为抽象数据类型, 那么我们将闭包的创建和应用抽象为`make-closure`和`apply-closure`.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (apply-env env var))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x ,body)
     (make-closure x body env))
    ((let ,x ,e ,body)
     (let* ((v (interp e env))
            (env^ (extend-env x v env)))
       (interp body env^)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env)))
       (case op
         ((+) (+ v1 v2))
```

```
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))
    ((,rator ,rand)
     (let* ((closure (interp rator env))
            (arg (interp rand env)))
       (apply-closure closure arg)))))
(define (make-closure formal body env)
  (lambda (arg)
    (interp body (extend-env formal arg env))))
(define (apply-closure closure arg)
  (closure arg))
```

我们也可以更换闭包的具体表示, 例如以下的表示更加具体直接.

```
(define (make-closure formal body env)
  (vector 'closure formal body env))
(define (closure-formal closure)
  (vector-ref closure 1))
(define (closure-body closure)
  (vector-ref closure 2))
(define (closure-env closure)
  (vector-ref closure 3))
(define (apply-closure closure arg)
  (let* ((formal (closure-formal closure))
         (body (closure-body closure))
         (env (closure-env closure)))
    (interp body (extend-env formal arg env))))
```

我们想说的是, 没有真正的魔法! 用闭包表示闭包似乎是一种魔法, 但其也可以表示为寻常的数据结构.

注记: 这里的`apply-closure`使用`let*`的确没有什么必要, 但鉴于`let`并不规定求值顺序, 为了获得确定的控制流 (其一个好处在于保证不同的实现产生相同的行为), 我们还是使用了`let*`. 以后我们也将常常这么做.

让我们以一个有趣的例子结束本节.

```
> (interp
   '(let make-fact (lambda make-fact
                     (lambda n
                       (if (= n 0)
                           1
                           (* n ((make-fact make-fact) (- n 1))))))
      (let fact (make-fact make-fact)
        (fact 10)))
   (empty-env))
3628800
```

它展示了不使用`letrec`编写递归过程的方法. (显然, 其中隐藏着一个不动点组合子.)

2.4 de Bruijn索引

实际上, 足够敏锐的读者可能会发现我们可以在对于程序求值之前预测变量绑定于环境中的位置. 例如, 对于过程`(lambda x (lambda y (+ x y)))`而言, 当它依次被应用于两个具体的参数上去后, 将对于表达式`(+ x y)`求值, 不论在怎样的环境里被应用, 不论具体的参数如何, `y`的变量绑定一定出现于环境的最顶层, 而`x`的变量绑定一定出现于次顶层. 于是, 我们称`y`具有词法深度`0`, `x`具有词法深度`1`. 正如我们之前所言, “词法”应该被理解为“可在动态的求值过程之前由程序文本静态地进行推断”.

de Bruijn索引是程序的一种匿名表示, 它将变量替换为其词法深度. 并且, 显然此时绑定结构无需引入变量的名字了. 现在我们考虑编写一个转换过程, 其将普通的具名表示转换为de Bruijn索引形式的匿名表示. 在编程语言的上下文中, 句法转换过程一般被称为编译. 并且, 编译一般指的是保持语义的句法转换过程. 何谓保持语义? 若给定语言L_1和L_2, 设其相应的定义解释器 (注: 定义解释器即定义语言的语义的解释器) 为I_1和I_2, 对于从L_1到L_2的(保持语义的)编译器C (此时称L_1为源语言, L_2为目标语言), 对于每个语言L_1的程序P_1, 以及P_1经过C的转换得到的语言L_2的程序P_2, P_1在I_1下的解释应该等价于P_2在I_2下的解释 (这可以被画成一个交换图). “等价”的定义往往是微妙的, 但至少这两个计算应该同时收敛或发散 (注: 一个计算如果不终止, 或终止于一个运行时异常, 则称该计算发散), 且在收敛时得到的值相等. 一般情况下, 更多的性质希望被保持, 例如迭代控制行为 (相较于递归控制行为), 这个概念将在“延续”一章得到仔细检视.

既然我们知道编译是保持语义的句法变换, 我们需要明白源语言和目标语言的句法和语义. 我们选择的源语言是前一节的语言, 而目标语言的句法和语义现在给出.

```
<exp> ::= <int>
       |  <bool>
       |  <var>
       |  (if <exp> <exp> <exp>)
       |  (lambda <exp>)
       |  (let <exp> <exp>)
       |  (<op> <exp> <exp>)
       |  (<exp> <exp>)
<var> ::= (var <int>)
<op> ::= + | - | * | =
```

```
(define (interp exp denv)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    ((var ,depth) (apply-denv denv depth))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 denv)))
       (if (true? v1)
           (interp e2 denv)
           (interp e3 denv))))
    ((lambda ,body)
     (make-closure body denv))
    ((let ,e ,body)
     (let* ((v (interp e denv))
            (denv^ (extend-denv v denv)))
       (interp body denv^)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 denv))
            (v2 (interp e2 denv)))
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))
    ((,rator ,rand)
     (let* ((closure (interp rator denv))
            (arg (interp rand denv)))
       (apply-closure closure arg)))))
(define (empty-denv) '())
(define (extend-denv val denv)
  (cons val denv))
(define (apply-denv denv depth)
```

```
  (list-ref denv depth))
(define (make-closure body denv) (vector 'closure body denv))
(define (closure-body closure) (vector-ref closure 1))
(define (closure-denv closure) (vector-ref closure 2))
(define (apply-closure closure arg)
  (let* ((body (closure-body closure))
         (denv (closure-denv closure)))
    (interp body (extend-denv arg denv))))
```

这个解释器的结构与之前的仍然保持一致, 但一些细节发生了改变. 首先, 环境变成了"动态环境", 它无需维护名字和值的序对, 而只需要压入值即可, 因为词法深度预测了变量的值的位置. 鉴于环境变成了动态环境, 因而`make-closure`和`apply-closure`也发生了相应的改变, 但那是很显然的.

在编写编译器之前, 我们或许还需要回答一个问题, 即de Bruijn索引有何好处呢? 在数学上, 我们知道一致的换名不改变函数的意义. 例如, `(lambda x (* x x))`和`(lambda y (* y y))`应该是相等的. 在逻辑上, 换名一般被称为α变换. 许多时候, 我们需要考虑α变换导出的等价关系 (α等价) 带来的等价类. 虽然我们不会给出证明, 但读者应该能够料想到, α等价的项的de Bruijn索引应该在字面上是相等的, 例如`(lambda x (* x x))`和`(lambda y (* y y))`的de Bruijn索引皆为`(lambda (* (var 0) (var 0)))`, 这可以简化许多论证.

那么, 该如何编写这个编译器呢? 或许的确需要一点真正的聪明. 但是, 正如Alan Perlis (注: 第1届Turing奖获得者) 所言, "我希望我们不要成为传教士. 不要觉得自己好像是圣经兜售者. 这个世界已经有太多那样的人了. 你对于计算的理解, 别人也能学会. 不要觉得好像成功计算的钥匙只掌握在你的手里."

现在我们给出编译器的程序, 然后再解释其原理.

```
(define (compil exp senv)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var
     (guard (symbol? var))
     (let ((depth (apply-senv senv var)))
       `(var ,depth)))
    ((if ,e1 ,e2 ,e3)
     (let* ((e1^ (compil e1 senv))
            (e2^ (compil e2 senv))
            (e3^ (compil e3 senv)))
       `(if ,e1^ ,e2^ ,e3^)))
    ((lambda ,x ,body)
     (let ((body^ (compil body (extend-senv x senv))))
       `(lambda ,body^)))
    ((let ,x ,e ,body)
     (let* ((e^ (compil e senv))
            (senv^ (extend-senv x senv))
            (body^ (compil body senv^)))
       `(let ,e^ ,body^)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((e1^ (compil e1 senv))
            (e2^ (compil e2 senv)))
       `(,op ,e1^ ,e2^)))
    ((,rator ,rand)
     (let* ((rator^ (compil rator senv))
            (rand^ (compil rand senv)))
       `(,rator^ ,rand^)))))
(define (empty-senv) '())
(define (extend-senv var senv)
  (cons var senv))
```

```
(define (apply-senv senv var)
  (let iter ((rest senv) (depth 0))
    (cond ((null? rest)
           (error 'apply-senv "unbound variable ~s" var))
          ((eq? (car rest) var) depth)
          (else (iter (cdr rest) (+ depth 1))))))
```

实际上, 这不仅应该理解为编译, 更应该理解为“程序分析”, 而且是“静态分析” (即求值前进行的分析). 只不过, 我们将分析的结果包裹在了一个语言里. 我们所想要分析的是变量的词法深度, 正如之前所说, 这其实是一种预测, 即对于变量绑定于动态的求值过程在环境中出现的位置的预测. 其正确与否的准则, 自然是预测和现实是否一致.

正是“预测和现实是否一致”启发了我们, 因为若我们直接运行程序, 就可以直接看到我们想要的正确答案, 那为什么不这么做呢? 的确没有任何理由阻止我们这么做, 而我们的编译器正进行着这样的操作. 只不过, 求值过程中的某些信息我们是不需要的, 比如变量被绑定至的值. 因此, 我们的编译器就没有计算这种信息. 实际上, 应该将这个编译器视为“抽象的解释器”, 它不过就是在“虚拟地运行程序”, 然后收集所需的信息. “抽象解释”几乎是最重要的静态分析的方法, 但它最原初的想法并不复杂.

在这个编译器里, 环境被“静态环境”所代替. 静态环境只追踪变量的词法深度, 但不包含实际计算的值. 这个编译器也是一个句法导向的结构递归, 并且它的结构与解释器的结构如出一辙, 只是它的结果是程序而不是计算的值, 但这不是本质性的区别. 扩展环境的方式在编译器和解释器里是一致的, 这就是正确性的保证.

让我们以一个例子结束本节.

```
> (compil
   '(let make-fact (lambda make-fact
                     (lambda n
                       (if (= n 0)
                           1
                           (* n ((make-fact make-fact) (- n 1))))))
      (let fact (make-fact make-fact)
        (fact 10)))
   (empty-senv))
(let (lambda
       (lambda
         (if (= (var 0) 0)
             1
             (* (var 0) (((var 1) (var 1)) (- (var 0) 1))))))
  (let ((var 0) (var 0))
    ((var 0) 10)))
> (interp
   '(let (lambda
           (lambda
             (if (= (var 0) 0)
                 1
                 (* (var 0) (((var 1) (var 1)) (- (var 0) 1))))))
      (let ((var 0) (var 0))
        ((var 0) 10)))
   (empty-denv))
3628800
```

这个例子来源于前一节, 正说明何谓保持语义.

2.5 递归过程

在2.3节的例子里, 我们见识了一个编写递归过程的技巧. 如果将这个技巧抽象出来, 就会得到一个不动点组合子, 请看以下例子 (这里的`interp`来自于2.3节). 我们姑且将该不动点称为“按值调用的Y组合子”, Y组合子是由Haskell Curry发现的一个不动点组合子, 但若在按值调用的语言中试图直接运用其以得到递归过程则会发散, “按值调用的Y组合子”可以看作是对于Y组合子的修饰以将其用于按值调用的语言.

```
> (interp
   '(let y (lambda h
             ((lambda f (f f))
              (lambda g
                (h (lambda x ((g g) x))))))
      (let fact (y (lambda fact
                     (lambda n
                       (if (= n 0)
                           1
                           (* n (fact (- n 1))))))))
        (fact 10)))
   (empty-env))
3628800
```

本节的目的不在于建立递归过程的坚实语义, 那可能需要引入格论和论域论 (domain theory). 转而, 我们诉诸于递归的直觉, 也就是自引用 (self-reference), 实现了一个带有引入递归过程的构造的语言.

首先我们给出语言的句法.

```
<exp> ::= <int>
       |  <bool>
       |  <var>
       |  (if <exp> <exp> <exp>)
       |  (lambda <var> <exp>)
       |  (let <var> <exp> <exp>)
       |  (letrec <var> (lambda <var> <exp>) <exp>)
       |  (<op> <exp> <exp>)
       |  (<exp> <exp>)
<op> ::= + | - | * | =
```

现在我们呈现其解释器.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (apply-env env var))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x ,body)
     (make-closure x body env))
    ((let ,x ,e ,body)
     (let* ((v (interp e env))
            (env^ (extend-env x v env)))
       (interp body env^)))
    ((letrec ,f (lambda ,x ,fbody) ,body)
     (interp body (extend-env-rec f x fbody env)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env)))
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
```

```
         ((=) (= v1 v2)))))
    ((,rator ,rand)
     (let* ((closure (interp rator env))
            (arg (interp rand env)))
       (apply-closure closure arg)))))
(define (make-closure formal body env)
  (vector 'closure formal body env))
(define (closure-formal closure)
  (vector-ref closure 1))
(define (closure-body closure)
  (vector-ref closure 2))
(define (closure-env closure)
  (vector-ref closure 3))
(define (set-closure-env! closure env)
  (vector-set! closure 3 env))
(define (apply-closure closure arg)
  (let* ((formal (closure-formal closure))
         (body (closure-body closure))
         (env (closure-env closure)))
    (interp body (extend-env formal arg env))))
(define (extend-env-rec f x fbody env)
  (let* ((closure (make-closure x fbody 'foo))
         (env^ (extend-env f closure env)))
    (set-closure-env! closure env^)
    env^))
```

extend-env-rec是一个有趣的过程, 因为它需要构造一个含有"环路"的结构. extend-env-rec依赖于一个新的接口过程set-closure-env!, 并且该过程是具有副作用的. 不过, 也可以在不使用具有副作用的过程的情况下编写extend-env-rec.

```
(define (empty-env)
  (lambda (var)
    (error 'apply-env "unbound variable ~s" var)))
(define (extend-env var val env)
  (lambda (var^)
    (if (eq? var var^)
        val
        (apply-env env var^))))
(define (apply-env env var)
  (env var))
(define (extend-env-rec f x fbody env)
  (lambda (var)
    (if (eq? f var)
        (make-closure x fbody (extend-env-rec f x fbody env))
        (apply-env env var))))
```

以上依赖于环境的过程性表示, 但是将环境表示为更为传统的数据结构也是可能的, 这留给读者思考.

让我们以一个朴实无华的例子结束本节.

```
> (interp
   '(letrec fact (lambda n
                   (if (= n 0)
                       1
                       (* n (fact (- n 1)))))
      (fact 10))
   (empty-env))
3628800
```

2.6 动态作用域

本节更多地出于历史趣味, 而读者应该将动态作用域视为设计错误.

如果一个语言的过程的自由变量的意义不由创建过程的环境所决定, 而被推迟至应用过程时才决定, 那么就称这个语言采用了动态作用域. 这种语言的过程的意义在每次应用时均(可能)会改变, 因而和静态作用域的语言的过程相比是完全不同的对象.

本节我们呈现一个动态作用域的语言, 其句法与2.3节的语言完全一致, 但语义不同.

我们将创建和应用过程的过程称为`make-lambda`和`apply-lambda`, 这是相当合理的, 因为本节的过程不是闭包, 自然不该使用`make-closure`和`apply-closure`.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (apply-env env var))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x ,body)
     (make-lambda x body))
    ((let ,x ,e ,body)
     (let* ((v (interp e env))
            (env^ (extend-env x v env)))
       (interp body env^)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env)))
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))
    ((,rator ,rand)
     (let* ((lam (interp rator env))
            (arg (interp rand env)))
       (apply-lambda lam arg env)))))
(define (make-lambda formal body)
  (lambda (arg env)
    (interp body (extend-env formal arg env))))
(define (apply-lambda lam arg env)
  (lam arg env))
```

这里的过程的资料不再包括环境, 不过我们仍可以将其表示为Scheme过程. 这个表示法似乎比静态作用域的语言还要复杂一些, 那么为何早期编程语言设计者会落入动态作用域的陷阱中呢? 或许读者阅读以下程序就明白了, 我们将过程表示为过程的表达式.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (apply-env env var))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
```

```
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x ,body) exp)
    ((let ,x ,e ,body)
     (let* ((v (interp e env))
            (env^ (extend-env x v env)))
       (interp body env^)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env)))
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))
    ((,rator ,rand)
     (let* ((lam (interp rator env))
            (arg (interp rand env)))
       (apply-lambda lam arg env)))))
(define (apply-lambda lam arg env)
  (match lam
    ((lambda ,x ,body)
     (interp body (extend-env x arg env)))
    (,else
     (error 'apply-lambda "try to apply a non-lambda ~s" lam))))
```

这似乎非常合理, 但却是错的.

让我们以一个将错就错的例子结束本节.

```
> (interp
   '(let fact (lambda n
                (if (= n 0)
                    1
                    (* n (fact (- n 1)))))
      (fact 10))
   (empty-env))
3628800
```

这展示了在早期Lisp中人们编写递归过程的方式, 而在词法作用域的语言中, 它会提示`fact`是一个未绑定的变量.

3 状态

之前章节的语言都是*state-free*的, 也就是说, 没有副作用. 本节我们主要考虑最典型的副作用, 即赋值和变动, 这需要我们引入抽象内存的概念.

3.1 抽象内存

抽象内存是对于实际计算机器的具体内存的抽象化. 同样地, 我们将抽象内存视为抽象数据类型. 那么, 现在给出抽象内存的接口.

1. `newref`: 对于值`val`, `(newref val)`在抽象内存中分配一块未被分配的区域, 并返回对于该区域的引用, 如果抽象内存空间已尽, 则引起一个错误;

2. `deref`: 对于引用`ref`, `(deref ref)`返回抽象内存中引用`ref`指向的区域的内容, 如果该区域还未被分配值, 则返回值是未刻画的;

3. `setref`: 对于引用`ref`和值`val`, `(setref ref val)`将抽象内存中引用`ref`指向的区域的内容修改为`val`, 其返回值是未刻画的.

以下是抽象内存的一个实现. 读者应该注意到, 其运用了*naive object-oriented style à la SICP*.

```
(define (make-store size)
  (define store (make-vector size))
  (define next-loc 0)
  (define (newref val)
    (if (= next-loc size)
        (error 'newref "no space for allocation")
        (let ((loc next-loc))
          (vector-set! store loc val)
          (set! next-loc (+ next-loc 1))
          `(ref ,loc))))
  (define (deref ref)
    (let ((loc (match ref
                 ((ref ,loc) loc)
                 (,else
                  (error 'deref
                         "~s is not a reference."
                         ref)))))
      (vector-ref store loc)))
  (define (setref ref val)
    (let ((loc (match ref
                 ((ref ,loc) loc)
                 (,else
                  (error 'deref
                         "~s is not a reference."
                         ref)))))
      (vector-set! store loc val)))
  (lambda (msg . arg*)
    (let ((proc (case msg
                  ((newref) newref)
                  ((deref) deref)
                  ((setref) setref))))
      (apply proc arg*))))
```

如果想要使用这个实现, 以下是一个例子.

```
(define store0 (make-store 10000))
(define (newref val)
  (store0 'newref val))
(define (deref ref)
  (store0 'deref ref))
(define (setref ref val)
  (store0 'setref ref val))
```

当然, 保持原本的*naive object-oriented style à la SICP*也可以.

3.2 两种不同的设计

本节我们将赋值机制添加到了语言之中, 但是呈现了两种不同的设计.

3.2.1 第一种设计

第一种设计类似于Standard ML编程语言作出的决断, 它不直接对于变量进行赋值, 而是通过引用间接进行. 这也类似于许多Scheme实现提供的`box`, `unbox`, 和`set-box!`过程.

以下是语言的句法.

```
<exp> ::= <int>
       |  <bool>
       |  <var>
       |  (if <exp> <exp> <exp>)
       |  (lambda <var> <exp>)
       |  (let <var> <exp> <exp>)
       |  (begin <exp>+)
       |  (newref <exp>)
       |  (deref <exp>)
       |  (setref <exp> <exp>)
       |  (<op> <exp> <exp>)
       |  (<exp> <exp>)
<op> ::= + | - | * | =
```

`<exp>+`中的+被称为Kleene加号, 它表示一个至任意有限多个.

“Kleene加号”这个名字显然来源于“Kleene星号”, Kleene星号表示零个至任意有限多个.

然后是解释器.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (apply-env env var))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x ,body)
     (make-closure x body env))
    ((let ,x ,e ,body)
     (let* ((v (interp e env))
            (env^ (extend-env x v env)))
       (interp body env^)))
    ((begin . ,e+)
     (interp-e+ e+ env))
    ((newref ,e)
     (let ((val (interp e env)))
       (newref val)))
    ((deref ,e)
     (let ((ref (interp e env)))
       (deref ref)))
    ((setref ,e1 ,e2)
     (let* ((ref (interp e1 env))
            (val (interp e2 env)))
       (setref ref val)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env)))
```

```
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))
    ((,rator ,rand)
     (let* ((closure (interp rator env))
            (arg (interp rand env)))
       (apply-closure closure arg)))))
(define (interp-e+ e+ env)
  (if (null? e+)
      (error 'interp-e+ "the body of a begin should not be empty")
      (let iter ((e (car e+)) (e* (cdr e+)))
        (if (null? e*)
            (interp e env)
            (begin (interp e env)
                   (iter (car e*) (cdr e*)))))))
```

注意到新的构造begin, 其语义已经在过程interp-e+的定义中体现得很明了了.

读者应该明白一件事情, 虽然抽象内存没有被传递给解释器, 但是它也是求值的上下文之一. 只不过, 在整个求值过程中, 本质上只有一个抽象内存而已. 为了反映这个事实, 抽象内存是作为全局变量存在的.

让我们以几个例子结束本小节.

```
> (interp
   '(let fact-ref (newref 0)
      (begin
        (setref fact-ref
                (lambda n
                  (if (= n 0)
                      1
                      (* n ((deref fact-ref) (- n 1))))))
        ((deref fact-ref) 10)))
   (empty-env))
3628800
```

第一个例子以赋值实现了递归.

```
> (interp '(let counter (let x (newref 0)
                          (lambda _
                            (begin
                              (setref x (+ (deref x) 1))
                              (deref x))))
             (begin (counter 0)
                    (counter 0)
                    (counter 0)))
          (empty-env))
3
```

第二个例子是一个计数器.

```
> (interp
   '(let x (newref (newref 0))
      (begin
        (setref (deref x) 42)
        (deref (deref x))))
   (empty-env))
42
```

第三个例子是一个双重引用.

3.2.2 第二种设计

在第二种设计里, 用户不能显式地操作引用了, 但是却可以对于变量直接进行赋值.

以下是语言的句法.

```
<exp> ::= <int>
       |  <bool>
       |  <var>
       |  (set! <var> <exp>)
       |  (if <exp> <exp> <exp>)
       |  (lambda <var> <exp>)
       |  (let <var> <exp> <exp>)
       |  (begin <exp>+)
       |  (<op> <exp> <exp>)
       |  (<exp> <exp>)
<op> ::= + | - | * | =
```

读者应该注意到, `set!`表达式的第一个句法参数并不是任意的表达式, 而是变量.

然后是解释器.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (deref (apply-env env var)))
    ((set! ,x ,e)
     (guard (symbol? x))
     (let* ((v (interp e env))
            (r (apply-env env x)))
       (setref r v)))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x ,body)
     (make-closure x body env))
    ((let ,x ,e ,body)
     (let* ((v (interp e env))
            (r (newref v))
            (env^ (extend-env x r env)))
       (interp body env^)))
    ((begin . ,e+)
     (interp-e+ e+ env))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env)))
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))
    ((,rator ,rand)
     (let* ((closure (interp rator env))
            (arg (interp rand env))
            (ref (newref arg)))
```

```
       (apply-closure closure ref)))))
```

现在我们拥有一个双层的结构, 环境里变量和引用绑定, 然后通过引用可以在抽象内存中找到变量的值.

让我们以几个例子结束本节.

```
> (interp '(let fact 0
             (begin (set! fact
                          (lambda n
                            (if (= n 0)
                                1
                                (* n (fact (- n 1))))))
                    (fact 10)))
          (empty-env))
3628800
```

第一个例子仍以赋值实现了递归.

```
> (interp '(let counter (let x 0
                          (lambda _
                            (begin (set! x (+ x 1))
                                   x)))
             (begin (counter 0)
                    (counter 0)
                    (counter 0)))
          (empty-env))
3
```

第二个例子仍是一个计数器.

```
> (interp
   '(let swap (lambda a
                (lambda b
                  (let t a
                    (begin
                      (set! a b)
                      (set! b t)))))
      (let x 0
        (let y 1
          (begin
            ((swap x) y)
            x))))
   (empty-env))
0
```

第三个例子是“交换”, 但正如读者预料到的, 这是注定要失败的.

3.3 序对

本节之中我们要为3.2.2小节的语言添加序对机制, 序对是最基本的数据黏合剂.

以下是语言的句法.

```
<exp> ::= <int>
       |  <bool>
       |  <var>
       |  (set! <var> <exp>)
       |  (if <exp> <exp> <exp>)
       |  (lambda <var> <exp>)
       |  (let <var> <exp> <exp>)
```

```
       |  (begin <exp>+)
       |  (cons <exp> <exp>)
       |  (car <exp>)
       |  (cdr <exp>)
       |  (set-car! <exp> <exp>)
       |  (set-cdr! <exp> <exp>)
       |  (<op> <exp> <exp>)
       |  (<exp> <exp>)
<op> ::= + | - | * | =
```

实际上, 也就是添加了五个和序对相关的原始过程而已.

接着让我们观察解释器.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) `(int ,int))
    (,bool (guard (boolean? bool)) `(bool ,bool))
    (,var (guard (symbol? var)) (deref (apply-env env var)))
    ((set! ,x ,e)
     (guard (symbol? x))
     (let* ((v (interp e env))
            (r (apply-env env x)))
       (setref r v)))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x ,body)
     (make-closure x body env))
    ((let ,x ,e ,body)
     (let* ((v (interp e env))
            (r (newref v))
            (env^ (extend-env x r env)))
       (interp body env^)))
    ((begin . ,e+)
     (interp-e+ e+ env))
    ((cons ,e1 ,e2)
     (let* ((a (interp e1 env))
            (d (interp e2 env))
            (ra (newref a))
            (rd (newref d))
            (la (match ra
                  ((ref ,loc) loc))))
       `(pair ,la)))
    ((car ,e)
     (let ((v (interp e env)))
       (match v
         ((pair ,la) (deref `(ref ,la)))
         (,else (error 'interp:car "~s is not a pair" v)))))
    ((cdr ,e)
     (let ((v (interp e env)))
       (match v
         ((pair ,la) (let ((ld (+ la 1)))
                       (deref `(ref ,ld))))
         (,else (error 'interp:cdr "~s is not a pair" v)))))
    ((set-car! ,e1 ,e2)
```

```
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env))
            (la (match v1
                  ((pair ,la) la)
                  (,else
                   (error 'interp:set-car! "~s is not a pair" v1))))
            (ra `(ref ,la)))
       (setref ra v2)))
    ((set-cdr! ,e1 ,e2)
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env))
            (la (match v1
                  ((pair ,la) la)
                  (,else
                   (error 'interp:set-car! "~s is not a pair" v1))))
            (ld (+ la 1))
            (rd `(ref ,ld)))
       (setref rd v2)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env))
            (i1 (match v1
                  ((int ,i) i)
                  (,else (error 'interp:op:v1 "~s is not an integer" v1))))
            (i2 (match v2
                  ((int ,i) i)
                  (,else (error 'interp:op:v2 "~s is not an integer" v2)))))
       (case op
         ((+) `(int ,(+ i1 i2)))
         ((-) `(int ,(- i1 i2)))
         ((*) `(int ,(* i1 i2)))
         ((=) `(bool ,(= i1 i2))))))
    ((,rator ,rand)
     (let* ((closure (interp rator env))
            (arg (interp rand env))
            (ref (newref arg)))
       (apply-closure closure ref)))))
(define (true? val)
  (match val
    ((bool ,b) b)
    (,else (error 'interp "the predicate of if is not of type bool"))))
```

的确, 这里有些趣味. 为了区分序对和其他值, 我们给整数和布尔打上了标签, 这就像一个真实的Scheme实现. (因此, `true?`的定义也要随之改动.) 而且, 实现序对的方式也像一个真实的Scheme实现, 即我们只记录下`car`部分分配至的位置, 而`cdr`部分是连着分配空间的, 所以其位置可由`car`部分的位置计算得到. 同样真实地, 我们需要编写自己的对象输出例程, 因为例如当前的序对仅仅是一个类型标记加上一个地址而已. 以下的`value-of`将解释器的输出转换为可读的Scheme值.

```
(define (value-of obj)
  (match obj
    ((int ,i) i)
    ((bool ,b) b)
    ((pair ,la)
     (let* ((ld (+ la 1))
            (ra `(ref ,la))
            (rd `(ref ,ld))
```

```
            (oa (deref ra))
            (od (deref rd)))
       (cons (value-of oa) (value-of od))))
    (,closure closure)))
```

让我们以例子结束本节.

```
> (value-of
   (interp
    '(let < 0
       (let quotient 0
         (let remainder 0
           (let ext-gcd 0
             (begin
               (set! < (lambda a
                         (lambda b
                           (if (= b 0)
                               #f
                               (if (= a 0)
                                   #t
                                   ((< (- a 1)) (- b 1)))))))
               (set! quotient
                     (lambda a
                       (lambda b
                         (if ((< a) b)
                             0
                             (+ ((quotient (- a b)) b) 1)))))
               (set! remainder
                     (lambda a
                       (lambda b
                         (if ((< a) b)
                             a
                             ((remainder (- a b)) b)))))
               (set! ext-gcd
                     (lambda a
                       (lambda b
                         (if (= b 0)
                             (cons 1 0)
                             (let q ((quotient a) b)
                               (let r ((remainder a) b)
                                 (let p ((ext-gcd b) r)
                                   (let m (car p)
                                     (let n (cdr p)
                                       (cons n (- m (* q n))))))))))))
               ((ext-gcd 123) 321))))))
    (empty-env)))
(47 . -18)
```

第一个例子是扩展Euclid算法, 作为利用序对返回多个值的演示. (有趣的是这里我们递归地定义了<, quotient和remainder.)

```
> (value-of
   (interp
    '(let p (cons 0 1)
       (let q (cons p p)
         (begin (set-car! p (+ (car p) 2))
                (set-cdr! p (+ (cdr p) 3))
                q)))
```

```
    (empty-env)))
((2 . 4) 2 . 4)
```

第二个例子牵扯到结构的共享, 也应该是读者熟悉的.

当然, 关于序对的典型例子还可以举上许多, 但读者大概心里有数, 不再赘言了.

3.4 按值调用和按引用调用

在3.2.2小节的第三个例子里, 我们展示了一个"交换"的失败例子. 那是自然的, 因为3.2.2小节的语言是按值调用的, 意即参数的值被传递给过程. 本节我们要考虑另一种参数传递的方式, 即按引用调用. Fortran是一个采用按引用调用的语言的例子.

语言的句法仍然和3.2.2小节的语言一致, 但是语义并不相同, 以下是解释器.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (deref (apply-env env var)))
    ((set! ,x ,e)
     (guard (symbol? x))
     (let* ((v (interp e env))
            (r (apply-env env x)))
       (setref r v)))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x ,body)
     (make-closure x body env))
    ((let ,x ,e ,body)
     (let* ((v (interp e env))
            (r (newref v))
            (env^ (extend-env x r env)))
       (interp body env^)))
    ((begin . ,e+)
     (interp-e+ e+ env))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env)))
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))
    ((,rator ,var)
     (guard (symbol? var))
     (let* ((closure (interp rator env))
            (ref (apply-env env var)))
       (apply-closure closure ref)))
    ((,rator ,rand)
     (let* ((closure (interp rator env))
            (arg (interp rand env))
            (ref (newref arg)))
       (apply-closure closure ref)))))
```

实际上, 和3.2.2小节相比, 我们的解释器只添加了以下的部分.

```
    ((,rator ,var)
     (guard (symbol? var))
     (let* ((closure (interp rator env))
            (ref (apply-env env var)))
       (apply-closure closure ref)))
```

也就是说, 当参数是变量的时候, 我们不再“扒开”变量的值, 而是直接将它的引用传递给闭包.

让我们重新观察之前的例子.

```
> (interp
   '(let swap (lambda a
                (lambda b
                  (let t a
                    (begin
                      (set! a b)
                      (set! b t)))))
      (let x 0
        (let y 1
          (begin
            ((swap x) y)
            x))))
   (empty-env))
1
```

这是意料之中的结果.

3.5 按名调用和按需调用

按名调用是Algol 60界的一个术语 (或许它是导致Algol 60没有流行的重要原因之一), 它指的是这样的想法, 仅在用到参数时才对其求值, 而不是在应用过程之前对于每个参数求值. 因此, 按名调用可能带来计算代价的节约. 按需调用是按名调用的变体, 它的想法在于一旦参数被计算过了, 就应该记住其结果, 而无需再次计算. 按名调用和按需调用的语言也被称为“惰性求值”的, 原因是很显见的. Haskell可能是惰性求值语言的典型.

本节我们的语言的句法和2.3节是一致的 (鉴于赋值和惰性求值之间交互的复杂性, 我们放弃了赋值), 但是语义不同.

为了实现按名调用和按需调用, 我们使用了*thunk*. thunk包含被推迟求值的表达式, 其只有在被*force*时才会求值. 不同之处在于, 按需调用的thunk在第一次进行求值后就会记住结果.

首先让我们观察按名调用的语言的解释器.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (force-thunk (apply-env env var)))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x ,body)
     (make-closure x body env))
    ((let ,x ,e ,body)
     (let* ((t (make-thunk e env))
            (env^ (extend-env x t env)))
```

```
       (interp body env^)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env)))
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))
    ((,rator ,rand)
     (let* ((closure (interp rator env))
            (arg (make-thunk rand env)))
       (apply-closure closure arg)))))
(define (make-thunk exp env)
  (vector 'thunk exp env))
(define (thunk-exp thunk)
  (vector-ref thunk 1))
(define (thunk-env thunk)
  (vector-ref thunk 2))
(define (force-thunk thunk)
  (let* ((exp (thunk-exp thunk))
         (env (thunk-env thunk)))
    (interp exp env)))
```

然后是一个典型的例子.

```
> (interp '(let y (lambda f
                     ((lambda x (f (x x)))
                      (lambda x (f (x x)))))
             (let fact (y (lambda fact
                            (lambda n
                              (if (= n 0)
                                  1
                                  (* n (fact (- n 1)))))))
               (fact 10)))
          (empty-env))
3628800
```

这里的y正是Haskell Curry发现的Y组合子. 在一个按值调用的语言里, 求值的结果将会是发散的.

接着, 让我们观察按需调用的语言的解释器.

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (force-thunk (apply-env env var)))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x ,body)
     (make-closure x body env))
    ((let ,x ,e ,body)
     (let* ((t (build-thunk e env))
            (env^ (extend-env x t env)))
```

```
       (interp body env^)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env)))
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))
    ((,rator ,rand)
     (let* ((closure (interp rator env))
            (arg (build-thunk rand env)))
       (apply-closure closure arg)))))
(define (make-thunk exp env flag val)
  (vector 'thunk exp env flag val))
(define (build-thunk exp env)
  (make-thunk exp env #f 'uninitialized))
(define (thunk-exp thunk)
  (vector-ref thunk 1))
(define (thunk-env thunk)
  (vector-ref thunk 2))
(define (thunk-flag thunk)
  (vector-ref thunk 3))
(define (thunk-val thunk)
  (vector-ref thunk 4))
(define (set-thunk-flag! thunk flag)
  (vector-set! thunk 3 flag))
(define (set-thunk-val! thunk val)
  (vector-set! thunk 4 val))
(define (force-thunk thunk)
  (if (thunk-flag thunk)
      (thunk-val thunk)
      (let* ((exp (thunk-exp thunk))
             (env (thunk-env thunk))
             (val (interp exp env)))
        (set-thunk-flag! thunk #t)
        (set-thunk-val! thunk val)
        val)))
```

实际上, 我们只是修改了thunk的实现而已, 其余的地方几乎未动.

似乎按名调用和按需调用的语言应该是差不多的, 让我们看一个有趣的例子, 这个例子需要给语言添加函数`random` (不过那是平凡的, 因为我们可以借用Scheme本身提供的过程).

```
(interp '((lambda n
            (if (= n 0)
                (if (= n 0)
                    (if (= n 0)
                        (if (= n 0)
                            (if (= n 0)
                                (if (= n 0)
                                    (if (= n 0) #t #f)
                                    #f)
                                #f)
                            #f)
                        #f)
                    #f)
```

```
                (if (= n 0)
                    #f
                    (if (= n 0)
                        #f
                        (if (= n 0)
                            #f
                            (if (= n 0)
                                #f
                                (if (= n 0)
                                    #f
                                    (if (= n 0)
                                        #f
                                        #t)))))))
          (random 2))
        (empty-env))
```

如果是按需调用的语言, 对其求值的结果一定是`#t`. 但是如果是按名调用的语言, 对其求值的结果很有可能是`#f`, 只有很低的概率是`#t`. (读者应该想一想为什么.)

在本节的结尾, 我们想要指出关于thunk的一些微妙而棘手的地方, 鉴于其与主线关系不大, 所以我们将它置于附录之中, 供感兴趣的读者阅读. 另外, 本节我们也省略了关于惰性数据结构"流"的讨论, 读者首先可以参考*CONS should not Evaluate its Arguments*以及*SICP*第3.5节, 其次可以参考Oleg Kiselyov关于流的研究, 附录里我们也提供了一个简要的参考.

4 延续

在第2章里, 我们运用环境这一概念对于绑定和过程的行为进行了建模. 在第3章里, 我们运用抽象内存这一概念对于变动和参数传递进行了建模. 在本章中, 我们将运用延续这一概念对于编程语言的控制行为进行建模.

4.1 延续和延续传递风格

延续是对于控制的抽象, 包裹了剩余的计算. 这个定义并不容易凭空理解, 让我们慢慢解释.

以下是两个典型的Scheme过程, `fact`和`gcd`.

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

几乎每个Schemer都知道`gcd`是尾递归的, 而`fact`不是, 但不一定每个Schemer都能解释清楚原因.

让我们观察这两个过程的调用实例.

```
   (fact 3)
-> (* 3 (fact 2))
-> (* 3 (* 2 (fact 1)))
-> (* 3 (* 2 (* 1 (fact 0))))
-> (* 3 (* 2 (* 1 1)))
-> (* 3 (* 2 1))
-> (* 3 2)
-> 6
   (gcd 9 6)
```

```
-> (gcd 6 3)
-> (gcd 3 0)
-> 3
```

可以看到, 在对于(fact 3)求值的过程中, 先是膨胀, 后是收缩. 这与对于(gcd 9 6)的求值过程很不一样, 因为不论怎样调用gcd, 其计算过程只会像一条直线. 而且, 可以想见, 随着自然数n的增长, (fact n)的最大膨胀程度也会线性地增长.

很容易解释fact的行为, 因为对于非零的自然数n, (fact n)就等价于(* n (fact (- n 1))), 在对于(fact (- n 1))求值的过程中, 系统必须"记住"要给最终的结果乘上n, 这种记忆, 或者说上下文, 不断地累积, 直到对于(fact 0)求值后开始收缩.

我们将这样的控制上下文称为延续, 它代表了当得到一个结果的时候, 之后该做什么以完成整个计算. 如果在求值过程中延续是有界的, 那么就称其呈现了迭代计算行为, 否则的话就称其呈现了递归计算行为. 计算行为和实际的形式无关, 例如虽然表面上fact和gcd都是递归的, 但是fact呈现了递归计算行为, 而gcd呈现了迭代计算行为.

所谓的延续传递风格变换 (CPS conversion/transformation) 可以使我们的讨论变得更加显然. 延续传递风格 (continuation-passing style) 是一种显式传递延续的风格, 也就是说, 将控制暴露出来.

```
(define (fact-cps n k)
  (if (= n 0)
      (k 1)
      (fact-cps (- n 1) (lambda (v) (k (* n v))))))
(define (gcd-cps a b k)
  (if (= b 0)
      (k a)
      (gcd-cps b (remainder a b) k)))
```

这里的k均代表延续. 可以看到, gcd-cps不用修饰其延续, 而fact-cps需要记住给结果乘上n才行. 现在fact-cps也是尾递归的了, 但是本质上计算没有得到任何的简化, 因为它只是将原本在台面下发挥作用的延续搬到台面上而已.

接着, 请读者阅读fact-aps.

```
(define (fact-aps n a)
  (if (= n 0)
      a
      (fact-aps (- n 1) (* n a))))
```

这里的a代表accumulator (累积器), 因此aps的意思是accumulator-passing style (累积器传递风格). 读者应该明白对于每个自然数n, (fact-aps n 1)就等于(fact n). 实际上, 读者可以将a视为对于延续的一种"表示", 因为fact-cps的延续只是在累积需要乘上的一连串数字, 而这些数字之积就是a (如果最初的a是1的话). 这个想法不仅限于fact-cps和fact-aps.

关于对于每个自然数n和过程k, (fact-cps n k)等价于(k (fact n)), 以及关于对于每个自然数n和整数a, (fact-aps n a)等于(* (fact n) a)的证明, 见附录.

让我们再看一个例子, 比上面两个更复杂一些.

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
(define (fib-cps n k)
  (if (< n 2)
      (k n)
      (fib-cps (- n 1)
               (lambda (v1)
                 (fib-cps (- n 2)
```

```
                    (lambda (v2)
                      (k (+ v1 v2))))))))
```

实际上, 按照语义, 先对于`(fib (- n 1))`求值还是先对于`(fib (- n 2))`求值都是可以的, 这也正是高级编程语言的优美之处, 它让我们从显式指定计算顺序中解放出来. 不过, 如果我们要将`fib`变换为延续传递风格的话, 我们就必须考虑安排顺序, 且非得给每个中间结果命名. 这听起来很像汇编, 不是吗? 的确, 在控制方面, 延续传递风格更接近于汇编而不是一般的高级编程语言. 关于以延续传递风格作为中间表示的编译已经有了很长的研究历史, 读者首先可以参考*RABBIT: A Compiler for SCHEME*以及*Compiling with Continuations*.

4.2 延续传递风格解释器

为了简单起见, 本节我们从(按值调用的)lambda演算开始, 以下是句法.

```
<exp> ::= <var>
       |  (lambda <var> <exp>)
       |  (<exp> <exp>)
```

然后是解释器, 现在延续也成为其参数.

```
(define (interp exp env k)
  (match exp
    (,var
     (guard (symbol? var))
     (k (apply-env env var)))
    ((lambda ,x ,body)
     (k (lambda (arg k)
          (interp body (extend-env x arg env) k))))
    ((,rator ,rand)
     (interp rator env
             (lambda (closure)
               (interp rand env
                       (lambda (arg)
                         (closure arg k))))))))
```

实际上, 这就相当于对于原本的直接的解释器进行延续传递风格变换.

这个解释器的闭包和延续均以函数表示, 现在我们考虑将其转换为更加传统的数据结构, 这将使原本模糊的地方变得清晰起来.

```
(define (interp exp env k)
  (match exp
    (,var
     (guard (symbol? var))
     (apply-k k (apply-env env var)))
    ((lambda ,x ,body)
     (apply-k k (make-closure x body env)))
    ((,rator ,rand)
     (interp rator env (rator-k rand env k)))))
(define (make-closure formal body env)
  (vector 'closure formal body env))
(define (closure-formal closure)
  (vector-ref closure 1))
(define (closure-body closure)
  (vector-ref closure 2))
(define (closure-env closure)
  (vector-ref closure 3))
(define (apply-closure closure arg k)
  (let* ((formal (closure-formal closure))
```

```
         (body (closure-body closure))
         (env (closure-env closure)))
    (interp body (extend-env formal arg env) k)))
(define (empty-k) '(empty-k))
(define (rator-k rand env k)
  `(rator-k ,rand ,env ,k))
(define (rand-k closure k)
  `(rand-k ,closure ,k))
(define (apply-k k v)
  (match k
    ((rator-k ,rand ,env ,k)
     (interp rand env (rand-k v k)))
    ((rand-k ,closure ,k)
     (apply-closure closure v k))
    ((empty-k) v)))
```

从现在起, 我们开始考虑稍微复杂一些的语言, 即2.5节带有定义递归过程结构的语言.

不过, 我们将对于程序的结构作出一些改变. 首先我们将每个参数都变为全局的, 并使用赋值以在过程之间交流信息. 而且, 对于主要过程的调用都将出现在"尾位置", 这等价于goto, 即控制的直接移交. (到本节末, 读者应该能够理解这句陈述的含义.) 于是, 这个程序的控制结构将非常类似于在一个汇编语言中编写解释器. 还有一个小小的观察, 实际上读者应该注意到延续可以安排成栈的结构, 而且其可以被视为编程语言运行时的堆栈的抽象化. (对于低层次行为的理解不必停留于低层次, 而可以通过更高层次的概念进行演绎.)

鉴于程序的长度, 我们逐块阅读.

```
(define EXP (void))
(define ENV (void))
(define VAL (void))
(define CLO (void))
(define CTX (void))
(define depth (void))
(define max-depth (void))
(define (PUSH x)
  (set! CTX (cons x CTX))
  (set! depth (+ depth 1))
  (when (> depth max-depth)
    (set! max-depth depth)))
(define (POP)
  (if (null? CTX)
      (error 'POP "can't pop an empty stack")
      (let ((x (car CTX)))
        (set! CTX (cdr CTX))
        (set! depth (- depth 1))
        x)))
```

这是所有全局变量的定义, 其中`CLO`即closure, `CTX`即context (即continuation), 以及栈的实现.

```
(define (initialize! exp)
  (set! EXP exp)
  (set! ENV (empty-env))
  (set! VAL (void))
  (set! CLO (void))
  (set! CTX '())
  (set! depth 0)
  (set! max-depth 0))
```

`initialize!`为求值做准备.

```
(define (INTERP)
  (match EXP
    (,int
     (guard (integer? int))
     (set! VAL int)
     (APPLY_CTX))
    (,bool
     (guard (boolean? bool))
     (set! VAL bool)
     (APPLY_CTX))
    (,var
     (guard (symbol? var))
     (set! VAL (apply-env ENV var))
     (APPLY_CTX))
    ((if ,e1 ,e2 ,e3)
     (set! EXP e1)
     (PUSH (if-ctx e2 e3 ENV))
     (INTERP))
    ((lambda ,x ,body)
     (set! VAL (make-closure x body ENV))
     (APPLY_CTX))
    ((let ,x ,e ,body)
     (set! EXP e)
     (PUSH (let-ctx x body ENV)))
    ((letrec ,f (lambda ,x ,fbody) ,body)
     (set! EXP body)
     (set! ENV (extend-env-rec f x fbody ENV))
     (INTERP))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (set! EXP e1)
     (PUSH (op-ctx-1 op e2 ENV))
     (INTERP))
    ((,rator ,rand)
     (set! EXP rator)
     (PUSH (rator-ctx rand ENV))
     (INTERP))))
```

INTERP是解释器的主体. 可以看到, 有的时候我们立即得到了所需要的值, 将其置于VAL之中, 接着我们APPLY_CTX以考虑接下来对于这个值做些什么; 另外一些时候, 我们需要考虑先对于某个子表达式求值, 并且给CTX添加之后要做什么的信息.

```
(define (APPLY_CTX)
  (unless (null? CTX)
    (let ((top (POP)))
      (match top
        ((if-ctx ,e2 ,e3 ,env)
         (if (true? VAL)
             (set! EXP e2)
             (set! EXP e3))
         (set! ENV env)
         (INTERP))
        ((let-ctx ,x ,body ,env)
         (set! EXP body)
         (set! ENV (extend-env x VAL env))
         (INTERP))
        ((op-ctx-1 ,op ,e2 ,env)
```

```
         (set! EXP e2)
         (set! ENV env)
         (PUSH (op-ctx-2 op VAL))
         (INTERP))
        ((op-ctx-2 ,op ,v1)
         (case op
           ((+) (set! VAL (+ v1 VAL)))
           ((-) (set! VAL (- v1 VAL)))
           ((*) (set! VAL (* v1 VAL)))
           ((=) (set! VAL (= v1 VAL))))
         (APPLY_CTX))
        ((rator-ctx ,rand ,env)
         (set! EXP rand)
         (set! ENV env)
         (PUSH (rand-ctx VAL))
         (INTERP))
        ((rand-ctx ,closure)
         (set! CLO closure)
         (APPLY_CLO))))))
```

APPLY_CTX考虑对于值进行怎样的加工. 和INTERP的情况类似, 有时我们得到了想要的值, 于是需要进一步加工; 另外的时候, 我们需要考虑对于什么表达式进行求值, 并给CTX添加控制信息.

```
(define (APPLY_CLO)
  (set! EXP (closure-formal CLO))
  (set! ENV (closure-env CLO))
  (set! ENV (extend-env EXP VAL ENV))
  (set! EXP (closure-body CLO))
  (INTERP))
```

APPLY_CLO不过是应用闭包CLO于值VAL.

```
(define (if-ctx e2 e3 env)
  `(if-ctx ,e2 ,e3 ,env))
(define (let-ctx x body env)
  `(let-ctx ,x ,body ,env))
(define (op-ctx-1 op e2 env)
  `(op-ctx-1 ,op ,e2 ,env))
(define (op-ctx-2 op v1)
  `(op-ctx-2 ,op ,v1))
(define (rator-ctx rand env)
  `(rator-ctx ,rand ,env))
(define (rand-ctx closure)
  `(rand-ctx ,closure))
```

剩下的是一些简单的关于延续的数据结构的定义.

现在让我们看一个对比的例子.

```
> (initialize! '(letrec fact (lambda n
                               (if (= n 0)
                                   1
                                   (* n (fact (- n 1)))))
                  (fact 10)))
> (INTERP)
> VAL
3628800
> max-depth
12
```

```
> (initialize! '(letrec fact (lambda n
                               (if (= n 0)
                                   1
                                   (* n (fact (- n 1)))))
                  (fact 20)))
> (INTERP)
> VAL
2432902008176640000
> max-depth
22
> (initialize! '(letrec fact (lambda n
                               (if (= n 0)
                                   1
                                   (* n (fact (- n 1)))))
                  (fact 30)))
> (INTERP)
> VAL
265252859812191058636308480000000
> max-depth
32
> (initialize! '(letrec < (lambda a
                            (lambda b
                              (if (= b 0)
                                  #f
                                  (if (= a 0)
                                      #t
                                      ((< (- a 1)) (- b 1))))))
                  ((< 9) 10)))
> (INTERP)
> VAL
#t
> max-depth
3
> (initialize! '(letrec < (lambda a
                            (lambda b
                              (if (= b 0)
                                  #f
                                  (if (= a 0)
                                      #t
                                      ((< (- a 1)) (- b 1))))))
                  ((< 90) 100)))
> (INTERP)
> VAL
#t
> max-depth
3
> (initialize! '(letrec < (lambda a
                            (lambda b
                              (if (= b 0)
                                  #f
                                  (if (= a 0)
                                      #t
                                      ((< (- a 1)) (- b 1))))))
                  ((< 900) 1000)))
> (INTERP)
> VAL
```

```
#t
> max-depth
3
```

从这个例子可以看到递归过程fact和<的不同性质, fact呈现了递归计算行为, 因为最大的栈深度随着输入的增长而线性地增长. 反观<, 最大的栈深度一直是3.

若我们追踪对于表达式(fact (- n 1))求值时的延续.

```
> (initialize! '(letrec fact (lambda n
                               (if (= n 0)
                                   1
                                   (* n (fact (- n 1)))))
                 (fact 5)))
> (INTERP)
((op-ctx-2 * 5))
((op-ctx-2 * 4) (op-ctx-2 * 5))
((op-ctx-2 * 3) (op-ctx-2 * 4) (op-ctx-2 * 5))
((op-ctx-2 * 2) (op-ctx-2 * 3) (op-ctx-2 * 4) (op-ctx-2 * 5))
((op-ctx-2 * 1) (op-ctx-2 * 2) (op-ctx-2 * 3) (op-ctx-2 * 4) (op-ctx-2 * 5))
```

真正重要的原则是, 对于一个表达式的尾位置的表达式求值的延续, 应该与对于该表达式求值的延续是相等的, 这是因为尾位置的表达式的值, 就是整个表达式的值. 这也就是为什么说, 尾位置的过程调用就像goto一样, 因为无需添加任何需要做些什么的控制信息. 正是在于(fact (- n 1))没有出现在尾位置, 因而它必须记住还要做些什么.

4.3 控制行为

4.3.1 call/cc

call/cc是最典型的控制运算符, 它允许用户获得本来只在台面下发挥作用的延续.

以下是语言的句法.

```
<exp> ::= <int>
       |  <bool>
       |  <var>
       |  (if <exp> <exp> <exp>)
       |  (lambda <var> <exp>)
       |  (let <var> <exp> <exp>)
       |  (call/cc <exp>)
       |  (throw <exp> <exp>)
       |  (<op> <exp> <exp>)
       |  (<exp> <exp>)
<op> ::= + | - | * | =
```

相较于2.3节的语言, 我们添加了call/cc和throw. call/cc是为了引入延续, throw是为了应用延续.

现在给出解释器.

```
(define (interp exp env k)
  (match exp
    (,int
     (guard (integer? int))
     (apply-k k int))
    (,bool
     (guard (boolean? bool))
     (apply-k k bool))
    (,var
     (guard (symbol? var))
```

```
     (apply-k k (apply-env env var)))
    ((if ,e1 ,e2 ,e3)
     (interp e1 env (if-k e2 e3 env k)))
    ((lambda ,x ,body)
     (apply-k k (make-closure x body env)))
    ((let ,x ,e ,body)
     (interp e env (let-k x env body k)))
    ((call/cc ,e)
     (interp e env (call/cc-k k)))
    ((throw ,e1 ,e2)
     (interp e1 env (throw-k-1 e2 env k)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (interp e1 env (op-k-1 op e2 env k)))
    ((,rator ,rand)
     (interp rator env (rator-k rand env k)))))
(define (apply-k k v)
  (match k
    ((if-k ,e2 ,e3 ,env ,k)
     (if (true? v)
         (interp e2 env k)
         (interp e3 env k)))
    ((let-k ,x ,env ,body ,k)
     (interp body (extend-env x v env) k))
    ((call/cc-k ,k)
     (apply-closure v k k))
    ((throw-k-1 ,e2 ,env ,k)
     (interp e2 env (throw-k-2 v)))
    ((throw-k-2 ,k)
     (apply-k k v))
    ((op-k-1 ,op ,e2 ,env ,k)
     (interp e2 env (op-k-2 op v k)))
    ((op-k-2 ,op ,v1 ,k)
     (case op
       ((+) (apply-k k (+ v1 v)))
       ((-) (apply-k k (- v1 v)))
       ((*) (apply-k k (* v1 v)))
       ((=) (apply-k k (= v1 v)))))
    ((rator-k ,rand ,env ,k)
     (interp rand env (rand-k v k)))
    ((rand-k ,closure ,k)
     (apply-closure closure v k))
    ((empty-k) v)
    (,else (error 'apply-k "~s is not a continuation" k))))
(define (empty-k) '(empty-k))
(define (if-k e2 e3 env k)
  `(if-k ,e2 ,e3 ,env ,k))
(define (let-k x env body k)
  `(let-k ,x ,env ,body ,k))
(define (call/cc-k k)
  `(call/cc-k ,k))
(define (throw-k-1 e2 env k)
  `(throw-k-1 ,e2 ,env ,k))
(define (throw-k-2 k)
  `(throw-k-2 ,k))
(define (op-k-1 op e2 env k)
```

```
  `(op-k-1 ,op ,e2 ,env ,k))
(define (op-k-2 op v1 k)
  `(op-k-2 ,op ,v1 ,k))
(define (rator-k rand env k)
  `(rator-k ,rand ,env ,k))
(define (rand-k closure k)
  `(rand-k ,closure ,k))
```

以下是最简单的例子.

```
> (interp
   '(+ 2 (call/cc
          (lambda k
            (+ 3 (throw k 1)))))
   (empty-env)
   (empty-k))
3
> (interp
   '(+ 2 (call/cc
          (lambda k
            (+ 3 1))))
   (empty-env)
   (empty-k))
6
```

call/cc对于其参数传递了一个延续. 如果该延续没有被调用, 那么就会正常返回; 否则的话, 就会抛弃当前的延续, 转而使用call/cc提供的延续.

对于call/cc感兴趣的读者可以去了解一下David Madore的"阴阳谜题". 并且, 也不应该错过对于定界延续 (delimited continuation) 的研究.

4.3.2 异常处理

许多编程语言提供异常处理机制, 本节我们为2.3节的语言添加了try-catch.

以下是语言的句法.

```
<exp> ::= <int>
       |  <bool>
       |  <str>
       |  <var>
       |  (if <exp> <exp> <exp>)
       |  (lambda <var> <exp>)
       |  (let <var> <exp> <exp>)
       |  (raise <exp>)
       |  (try <exp> catch <var> <exp>)
       |  (<op> <exp> <exp>)
       |  (<exp> <exp>)
<op> ::= + | - | * | = | string=?
```

<exp>新增的<str>和<op>新增的string=?主要是为了表达和识别异常信息, 其中<str>代表Scheme字符串的句法范畴. 对于e1, e2∈<exp>, 和x∈<var>, 表达式`(try ,e1 catch ,x ,e2)的值, 在e1不通过raise表达式引起异常的情况下, 即e1的值, 否则的话, 变量x将会被绑定至异常信息, 然后对于e2求值. e2可能返回一个值, 但也可以引起一个新的异常.

现在我们给出解释器.

```
(define (interp exp env k)
  (match exp
    (,int
```

```
     (guard (integer? int))
     (apply-k k int))
    (,bool
     (guard (boolean? bool))
     (apply-k k bool))
    (,str
     (guard (string? str))
     (apply-k k str))
    (,var
     (guard (symbol? var))
     (apply-k k (apply-env env var)))
    ((if ,e1 ,e2 ,e3)
     (interp e1 env (if-k e2 e3 env k)))
    ((lambda ,x ,body)
     (apply-k k (make-closure x body env)))
    ((let ,x ,e ,body)
     (interp e env (let-k x env body k)))
    ((raise ,e)
     (interp e env (raise-k k)))
    ((try ,e1 catch ,x ,e2)
     (interp e1 env (try-k x e2 env k)))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * = string=?)))
     (interp e1 env (op-k-1 op e2 env k)))
    ((,rator ,rand)
     (interp rator env (rator-k rand env k)))))
(define (apply-k k v)
  (match k
    ((if-k ,e2 ,e3 ,env ,k)
     (if (true? v)
         (interp e2 env k)
         (interp e3 env k)))
    ((let-k ,x ,env ,body ,k)
     (interp body (extend-env x v env) k))
    ((raise-k ,k)
     (apply-handler k v))
    ((try-k ,x ,e2 ,env ,k)
     (apply-k k v))
    ((op-k-1 ,op ,e2 ,env ,k)
     (interp e2 env (op-k-2 op v k)))
    ((op-k-2 ,op ,v1 ,k)
     (case op
       ((+) (apply-k k (+ v1 v)))
       ((-) (apply-k k (- v1 v)))
       ((*) (apply-k k (* v1 v)))
       ((=) (apply-k k (= v1 v)))
       ((string=?) (apply-k k (string=? v1 v)))))
    ((rator-k ,rand ,env ,k)
     (interp rand env (rand-k v k)))
    ((rand-k ,closure ,k)
     (apply-closure closure v k))
    ((empty-k) v)))
(define (apply-handler k v)
  (match k
    ((if-k ,e2 ,e3 ,env ,k)
     (apply-handler k v))
```

```
    ((let-k ,x ,env ,body ,k)
     (apply-handler k v))
    ((raise-k ,k)
     (apply-handler k v))
    ((try-k ,x ,e2 ,env ,k)
     (interp e2 (extend-env x v env) k))
    ((op-k-1 ,op ,e2 ,env ,k)
     (apply-handler k v))
    ((op-k-2 ,op ,v1 ,k)
     (apply-handler k v))
    ((rator-k ,rand ,env ,k)
     (apply-handler k v))
    ((rand-k ,closure ,k)
     (apply-handler k v))
    ((empty-k)
     (error 'apply-handler "unhandled exception ~s" v))))
(define (empty-k) '(empty-k))
(define (if-k e2 e3 env k)
  `(if-k ,e2 ,e3 ,env ,k))
(define (let-k x env body k)
  `(let-k ,x ,env ,body ,k))
(define (raise-k k)
  `(raise-k ,k))
(define (try-k x e2 env k)
  `(try-k ,x ,e2 ,env ,k))
(define (op-k-1 op e2 env k)
  `(op-k-1 ,op ,e2 ,env ,k))
(define (op-k-2 op v1 k)
  `(op-k-2 ,op ,v1 ,k))
(define (rator-k rand env k)
  `(rator-k ,rand ,env ,k))
(define (rand-k closure k)
  `(rand-k ,closure ,k))
```

在正常情况下, 延续`try-k`是被忽略的. 但是, 当引起异常的时候, 它就会在延续中寻找"最顶层"的`try-k`. 如果找不到的话, 说明异常未被捕获, 这是一个错误.

以下是最简单的例子.

```
> (interp '(try (try (try (raise "foo")
                          catch x
                          (raise "bar"))
                     catch y
                     (raise "baz"))
                catch z z)
          (empty-env)
          (empty-k))
"baz"
> (interp '(let foobar (lambda str
                         (if (string=? str "foo")
                             (raise "foo")
                             (if (string=? str "bar")
                                 (raise "bar")
                                 (raise "baz"))))
             (try (foobar "foobar")
                  catch x
                  (if (string=? x "foo")
```

```
                      1
                      (if (string=? x "bar")
                          2
                          (if (string=? x "baz")
                              3
                              0)))))
          (empty-env)
          (empty-k))
3
```

4.4 延续传递风格变换

本节我们考虑编写一个进行延续传递风格变换的过程, 当然首先我们应该明确源语言和目标语言是什么.

以下是源语言的句法和语义, 它几乎就是2.5节的语言, 但是允许过程具有多个参数, 以及letrec绑定多个过程.

```
<exp> ::= <int>
       |  <bool>
       |  <var>
       |  (if <exp> <exp> <exp>)
       |  (lambda (<var>*) <exp>)
       |  (let <var> <exp> <exp>)
       |  (letrec ((<var> (lambda (<var>*) <exp>))*) <exp>)
       |  (<op> <exp> <exp>)
       |  (<exp> <exp>*)
<op> ::= + | - | * | =
```

```
(define (interp exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (apply-env env var))
    ((if ,e1 ,e2 ,e3)
     (let ((v1 (interp e1 env)))
       (if (true? v1)
           (interp e2 env)
           (interp e3 env))))
    ((lambda ,x* ,body)
     (make-closure x* body env))
    ((let ,x ,e ,body)
     (let* ((v (interp e env))
            (env^ (extend-env x v env)))
       (interp body env^)))
    ((letrec ,binding* ,body)
     (let-values (((f* x** fbody*) (decompose binding*)))
       (interp body (extend-env-rec f* x** fbody* env))))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (interp e1 env))
            (v2 (interp e2 env)))
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))
    ((,rator . ,rand*)
```

```
     (let* ((closure.arg* (interp* exp env))
            (closure (car closure.arg*))
            (arg* (cdr closure.arg*)))
       (apply-closure closure arg*)))))
(define (interp* exp* env)
  (if (null? exp*)
      '()
      (let* ((exp (car exp*))
             (exp* (cdr exp*))
             (val (interp exp env))
             (val* (interp* exp* env)))
        (cons val val*))))
(define (decompose binding*)
  (if (null? binding*)
      (values '() '() '())
      (let ((binding (car binding*))
            (binding* (cdr binding*)))
        (let-values (((f* x** fbody*) (decompose binding*)))
          (match binding
            ((,f (lambda ,x* ,fbody))
             (values (cons f f*)
                     (cons x* x**)
                     (cons fbody fbody*)))))))))
(define (extend-env* var* val* env)
  (append (map cons var* val*) env))
(define (extend-env-rec f* x** fbody* env)
  (let* ((closure*
          (map (lambda (x* fbody) (make-closure x* fbody 'foo)) x** fbody*))
         (env^ (extend-env* f* closure* env)))
    (for-each (lambda (closure)
                (set-closure-env! closure env^))
              closure*)
    env^))
(define (make-closure x* body env)
  (vector 'closure x* body env))
(define (closure-x* closure)
  (vector-ref closure 1))
(define (closure-body closure)
  (vector-ref closure 2))
(define (closure-env closure)
  (vector-ref closure 3))
(define (set-closure-env! closure env)
  (vector-set! closure 3 env))
(define (apply-closure closure arg*)
  (let* ((x* (closure-x* closure))
         (body (closure-body closure))
         (env (closure-env closure)))
    (interp body (extend-env* x* arg* env))))
```

目标语言实际上是源语言的一个子集.

```
<exp> ::= <tail>
<simple> ::= <int>
           |  <bool>
           |  <var>
           |  (lambda (<var>*) <tail>)
           |  (<op> <simple> <simple>)
```

```
<tail> ::= <simple>
        |  (if <simple> <tail> <tail>)
        |  (let <var> <simple> <tail>)
        |  (letrec ((<var> (lambda (<var>*) <tail>))*) <tail>)
        |  (<simple> <simple>*)
<op> ::= + | - | * | =
```

<simple>被认为是不会引起计算的表达式, 而可以引起计算的<tail>只能出现在尾位置.

```
(define (interp exp env)
  (Tail exp env))
(define (Simple simple env)
  (match simple
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (apply-env env var))
    ((lambda ,x* ,tail) (make-closure x* tail env))
    ((,op ,simple1 ,simple2)
     (guard (memq op '(+ - * =)))
     (let* ((v1 (Simple simple1 env))
            (v2 (Simple simple2 env)))
       (case op
         ((+) (+ v1 v2))
         ((-) (- v1 v2))
         ((*) (* v1 v2))
         ((=) (= v1 v2)))))))
(define (Tail tail env)
  (match tail
    (,simple (guard (simple? simple)) (Simple simple env))
    ((if ,simple ,tail1 ,tail2)
     (let ((v1 (Simple simple env)))
       (if (true? v1)
           (Tail tail1 env)
           (Tail tail2 env))))
    ((let ,x ,simple ,tail)
     (let* ((v (Simple simple env))
            (env^ (extend-env x v env)))
       (Tail tail env^)))
    ((letrec ,binding* ,tail)
     (let-values (((f* x** fbody*) (decompose binding*)))
       (Tail tail (extend-env-rec f* x** fbody* env))))
    ((,rator . ,rand*)
     (let* ((closure.arg* (Simple* tail env))
            (closure (car closure.arg*))
            (arg* (cdr closure.arg*)))
       (apply-closure closure arg*)))))
(define (simple? tail)
  (match tail
    (,int (guard (integer? int)) #t)
    (,bool (guard (boolean? bool)) #t)
    (,var (guard (symbol? var)) #t)
    ((lambda ,x* ,tail) #t)
    ((,op ,simple1 ,simple2) (guard (memq op '(+ - * =))) #t)
    (,else #f)))
(define (Simple* simple* env)
  (if (null? simple*)
      '()
```

```
      (let* ((simple (car simple*))
             (simple* (cdr simple*))
             (val (Simple simple env))
             (val* (Simple* simple* env)))
        (cons val val*))))
(define (apply-closure closure arg*)
  (let* ((x* (closure-x* closure))
         (body (closure-body closure))
         (env (closure-env closure)))
    (Tail body (extend-env* x* arg* env))))
```

接着我们考虑如何将源语言变换为目标语言.

或许首先应该进行α变换, 这将避免不必要的麻烦.

```
(define counter
  (let ((x -1))
    (lambda ()
      (set! x (+ x 1))
      x)))
(define (fresh-id x)
  (string->symbol
   (format "~s.~s" x (counter))))
(define (set? lst)
  (cond ((null? lst) #t)
        ((memq (car lst) (cdr lst)) #f)
        (else (set? (cdr lst)))))
(define (map0 f l)
  (if (null? l)
      '()
      (let* ((a (car l))
             (d (cdr l))
             (fa (f a))
             (fd (map0 f d)))
        (cons fa fd))))
(define (alpha exp env)
  (match exp
    (,int (guard (integer? int)) int)
    (,bool (guard (boolean? bool)) bool)
    (,var (guard (symbol? var)) (apply-env env var))
    ((if ,e1 ,e2 ,e3)
     (let* ((e1^ (alpha e1 env))
            (e2^ (alpha e2 env))
            (e3^ (alpha e3 env)))
       `(if ,e1^ ,e2^ ,e3^)))
    ((lambda ,x* ,body)
     (unless (set? x*)
       (error 'alpha "lambda duplicated variables ~s" x*))
     (let* ((x^* (map0 fresh-id x*))
            (env^ (extend-env* x* x^* env))
            (body^ (alpha body env^)))
       `(lambda ,x^* ,body^)))
    ((let ,x ,e ,body)
     (let* ((x^ (fresh-id x))
            (env^ (extend-env x x^ env))
            (e^ (alpha e env))
            (body^ (alpha body env^)))
```

```
      `(let ,x^ ,e^ ,body^)))
    ((letrec ,binding* ,body)
     (let-values (((f* x** fbody*) (decompose binding*)))
       (unless (set? f*)
         (error 'alpha "letrec duplicated variables ~s" f*))
       (let* ((f^* (map0 fresh-id f*))
              (env^ (extend-env* f* f^* env))
              (e^* (map0 (lambda (e) (alpha e env^))
                         (map (lambda (x* fbody)
                                `(lambda ,x* ,fbody))
                              x** fbody*)))
              (body^ (alpha body env^)))
         (Letrec f^* e^* body^))))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (let* ((e1^ (alpha e1 env))
            (e2^ (alpha e2 env)))
       `(,op ,e1^ ,e2^)))
    ((,rator . ,rand*)
     (map0 (lambda (e) (alpha e env)) exp))))
(define (Letrec x* e* body)
  `(letrec ,(map list x* e*) ,body))
```

其中map0类似于map, 但确保自左向由应用过程.

既然准备工作就绪, 现在是呈现延续传递风格变换的时候了.

```
(define (cps exp k)
  (match exp
    (,int (guard (integer? int)) (apply-k k int))
    (,bool (guard (boolean? bool)) (apply-k k bool))
    (,var (guard (symbol? var)) (apply-k k var))
    ((if ,e1 ,e2 ,e3)
     (cps e1 (lambda (s)
               (if (or (eq? k id) (symbol? k))
                   (let* ((t1 (cps e2 k))
                          (t2 (cps e3 k)))
                     `(if ,s ,t1 ,t2))
                   (let* ((k-id (fresh-id 'k))
                          (k-exp (build-k k))
                          (t1 (cps e2 k-id))
                          (t2 (cps e3 k-id)))
                     `(let ,k-id ,k-exp
                        (if ,s ,t1 ,t2)))))))
    ((lambda ,x* ,body) (apply-k k (cps-lambda x* body)))
    ((let ,x ,e ,body)
     (cps e (lambda (s)
              (let ((tail (cps body k)))
                `(let ,x ,s ,tail)))))
    ((letrec ,binding* ,body)
     (let-values (((f* x** fbody*) (decompose binding*)))
       (let* ((e* (cps-lambda* x** fbody*))
              (tail (cps body k)))
         (Letrec f* e* tail))))
    ((,op ,e1 ,e2)
     (guard (memq op '(+ - * =)))
     (cps e1 (lambda (s1)
               (cps e2 (lambda (s2)
```

```
                        (apply-k k `(,op ,s1 ,s2)))))))
    ((,rator . ,rand*)
     (cps* exp (lambda (s*)
                 (attach s* (build-k k)))))))
```

cps是延续传递风格变换过程的主体. 实际上, 这个代表延续的k可能是一个函数, 也可能只是一个标识符. 在某种意义上, 这是较为紧凑的写法 (且更容易阅读), 但若考虑到效率的话我们应该将cps拆分成两个互递归的过程. 对于if表达式, 为了避免重复延续 (使得程序膨胀), 我们做了特殊的安排.

```
(define (cps* exp* k)
  (if (null? exp*)
      (k '())
      (cps (car exp*)
           (lambda (s)
             (cps* (cdr exp*)
                   (lambda (s*)
                     (k (cons s s*)))))))))
(define (cps-lambda x* body)
  (let* ((k-id (fresh-id 'k))
         (y* (attach x* k-id))
         (tail (cps body k-id)))
    `(lambda ,y* ,tail)))
(define (cps-lambda* x** fbody*)
  (let iter ((e* '()) (x** x**) (fbody* fbody*))
    (if (null? x**)
        (reverse e*)
        (let ((x* (car x**))
              (x** (cdr x**))
              (fbody (car fbody*))
              (fbody* (cdr fbody*)))
          (iter (cons (cps-lambda x* fbody) e*)
                x** fbody*)))))
```

cps*是无奇的, 但应该注意顺序, 毕竟我们从一开始就约定求值自左向右进行. cps-lambda为lambda表达式添加了代表延续的参数, 其余是自然的.

```
(define id (lambda (x) x))
(define (apply-k k s)
  (if (symbol? k) `(,k ,s) (k s)))
(define (build-k k)
  (if (symbol? k)
      k
      (let* ((v-id (fresh-id 'v))
             (tail (k v-id)))
        `(lambda (,v-id) ,tail))))
(define (attach lst x)
  (append lst (list x)))
```

以上只是辅助过程, 没有什么可说的.

```
(define (CPS exp)
  (cps (alpha exp (empty-env)) id))
```

最后, 我们定义了一个方便的过程CPS.

以下是一些例子, 读者应该在α等价下阅读这些结果.

```
> (CPS '(letrec ((fact (lambda (n)
                         (if (= n 0)
                             1
```

```
                            (* n (fact (- n 1)))))))
          (fact 10)))
(letrec ((fact.0
          (lambda (n.1 k.2)
            (if (= n.1 0)
                (k.2 1)
                (fact.0 (- n.1 1) (lambda (v.3) (k.2 (* n.1 v.3))))))))
  (fact.0 10 (lambda (v.4) v.4)))
> (CPS '(letrec ((fib (lambda (n)
                        (if (= n 0)
                            0
                            (if (= n 1)
                                1
                                (+ (fib (- n 1))
                                   (fib (- n 2))))))))
          (fib 10)))
(letrec ((fib.0
          (lambda (n.1 k.2)
            (if (= n.1 0)
                (k.2 0)
                (if (= n.1 1)
                    (k.2 1)
                    (fib.0 (- n.1 1)
                           (lambda (v.3)
                             (fib.0 (- n.1 2)
                                    (lambda (v.4) (k.2 (+ v.3 v.4))))))))))))
  (fib.0 10 (lambda (v.5) v.5)))
> (CPS '(letrec ((even? (lambda (n)
                          (if (= n 0)
                              #t
                              (odd? (- n 1)))))
                 (odd? (lambda (n)
                         (if (= n 0)
                             #f
                             (even? (- n 1))))))
          (even? 88)))
(letrec ((even?.0 (lambda (n.2 k.4)
                    (if (= n.2 0) (k.4 #t) (odd?.1 (- n.2 1) k.4))))
         (odd?.1 (lambda (n.3 k.5)
                   (if (= n.3 0) (k.5 #f) (even?.0 (- n.3 1) k.5)))))
  (even?.0 88 (lambda (v.6) v.6)))
> (CPS '(lambda (f)
          (lambda (g)
            (lambda (h)
              (lambda (x)
                (f (g x) (h x)))))))
(lambda (f.0 k.4)
  (k.4
   (lambda (g.1 k.5)
     (k.5
      (lambda (h.2 k.6)
        (k.6 (lambda (x.3 k.7)
               (g.1 x.3 (lambda (v.8)
                          (h.2 x.3 (lambda (v.9)
                                     (f.0 v.8 v.9 k.7)))))))))))))
```

```
> (CPS '((lambda (x) (x x)) (lambda (x) (x x))))
((lambda (x.0 k.2) (x.0 x.0 k.2))
 (lambda (x.1 k.3) (x.1 x.1 k.3))
 (lambda (v.4) v.4))
> (CPS '(let square (lambda (x) (* x x))
          (+ (square 3) (square 4))))
(let square.0 (lambda (x.1 k.2) (k.2 (* x.1 x.1)))
  (square.0 3 (lambda (v.3) (square.0 4 (lambda (v.4) (+ v.3 v.4))))))
```

附录: 模式匹配宏`match`

```
(define-syntax match
  (syntax-rules (guard)
    ((_ v) (error 'match "~s" v))
    ((_ v (pat (guard g ...) e ...) cs ...)
     (let ((fk (lambda () (match v cs ...))))
       (ppat v pat (if (and g ...) (let () e ...) (fk)) (fk))))
    ((_ v (pat e ...) cs ...)
     (let ((fk (lambda () (match v cs ...))))
       (ppat v pat (let () e ...) (fk))))))
(define-syntax ppat
  (syntax-rules (unquote)
    ((_ v () kt kf) (if (null? v) kt kf))
    ((_ v (unquote var) kt kf) (let ((var v)) kt))
    ((_ v (x . y) kt kf)
     (if (pair? v)
         (let ((vx (car v)) (vy (cdr v)))
           (ppat vx x (ppat vy y kt kf) kf))
         kf))
    ((_ v lit kt kf) (if (eqv? v (quote lit)) kt kf))))
```

附录: 关于thunk的注记

以下内容摘取自R^5RS (Scheme语言第5次修订报告), 与惰性求值有关.

> where `make-promise` is defined as follows:
>
> ```
> (define make-promise
>   (lambda (proc)
>     (let ((result-ready? #f)
>           (result #f))
>       (lambda ()
>         (if result-ready?
>             result
>             (let ((x (proc)))
>               (if result-ready?
>                   result
>                   (begin (set! result-ready? #t)
>                          (set! result x)
>                          result)))))))))
> ```
>
> *Rationale*: A promise may refer to its own value, as in the last example above. Forcing such a promise may cause the promise to be forced a second time before the value of the first force has been computed. This complicates the definition of `make-promise`.

至于promise是什么, 读者将其理解为带备忘的thunk即可, 即与按需调用一致.

此部分的作者意识到, 在某个promise被force的过程中, 该promise可能会被再次force. 从原则上说, 一个promise的值一经确定则不能再次改变, 故有上文. 不过, 依照本论文作者的看法, 此时应该引起一个异常. 更加微妙的是, 若在promise被force的过程中出现异常, 那么这个异常该被记住吗? 最后还有一点微妙的地方, 虽然与之前的问题无关, 即promise应不应该是单独的数据类型. 对于以上问题的回答反映了实现者对于语言设计的理解, 但有一点可以肯定的是, 编写依赖于这些细节的代码是愚蠢的.

附录: 流

流是一种惰性数据结构, 以下是三个流的原语的定义.

```
(define-syntax $cons
  (syntax-rules ()
    ((_ a d) (cons a (delay d)))))
(define ($car $)
  (car $))
(define ($cdr $)
  (force (cdr $)))
```

至于`delay`和`force`, 以下是可能的定义. (让我们忽略"附录: 关于thunk的注记"里讨论的微妙之处.)

```
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
        (if result-ready?
            result
            (let ((x (proc)))
              (if result-ready?
                  result
                  (begin (set! result-ready? #t)
                         (set! result x)
                         result)))))))))
(define-syntax delay
  (syntax-rules ()
    ((_ exp) (make-promise (lambda () exp)))))
(define (force promise) (promise))
```

根据流的原语, 我们可以仿照列表处理定义流处理的过程.

```
(define ($map proc $)
  (if (null? $)
      '()
      ($cons (proc ($car $))
             ($map proc ($cdr $)))))
(define ($filter pred $)
  (cond ((null? $) '())
        ((pred ($car $))
         ($cons ($car $) ($filter pred ($cdr $))))
        (else ($filter pred ($cdr $)))))
```

我们也可以定义从流中取出元素的过程.

```
(define ($ref $ n)
  (if (= n 0)
      ($car $)
      ($ref ($cdr $) (- n 1))))
```

```
(define ($take $ n)
  (if (= n 0)
      '()
      (cons ($car $)
            ($take ($cdr $) (- n 1)))))
```

让我们欣赏一个优美的例子—Eratosthenes筛法.

```
(define (divides? a b)
  (= (remainder b a) 0))
(define (make-ints n)
  ($cons n (make-ints (+ n 1))))
(define (sieve $)
  ($cons ($car $)
         (sieve ($filter (lambda (x) (not (divides? ($car $) x)))
                         ($cdr $)))))
(define primes
  (sieve (make-ints 2)))
```

无限的流primes里的每个数字都是素数.

让我们看看primes的前20个元素分别是什么.

```
> ($take primes 20)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71)
```

附录: 关于**fact-cps**和**fact-aps**的证明

1 对于每个自然数n和每个过程k, (fact-cps n k)等价于(k (fact n)).

如果n等于0, 那么(fact-cps n k)就等价于(k 1), 而(fact 0)就等于1. 对于正整数n, 假定(fact-cps (- n 1) k)等价于(k (fact (- n 1))), 那么(fact-cps n k)就等价于(fact-cps (- n 1) (lambda (v) (k (* n v)))), 而根据归纳假设, 这又等价于((lambda (v) (k (* n v))) (fact (- n 1))), 即(k (* n (fact (- n 1)))), 而(* n (fact (- n 1)))就等于(fact n).

2 对于每个自然数n和整数a, (fact-aps n a)等于(* (fact n) a).

如果n等于0, 那么(fact-aps n a)就等于a, 而(fact 0)就等于1. 对于正整数n, 假定(fact-aps (- n 1) a)等于(* (fact (- n 1)) a), 那么(fact-aps n a)等于(fact-aps (- n 1) (* n a)), 根据归纳假设, (fact-aps (- n 1) (* n a))等于(* (fact (- n 1)) (* n a)), 也就等于(* (* (fact (- n 1)) n) a) (Scheme整数乘法的确满足结合律), 即(* (fact n) a).