# 采用代换模型的元循环求值器

张弛, 00848231

December 4, 2010

## 1 题目描述

考虑下面语言解释器的实现问题：

请利用第四章元循环求值器的语法过程实现一个采用代换模型的求值器。送给它一个组合式，它能一步步打印出采用代换模型求值的过程。请写一个报告，说明你怎样完成这一工作，其中采用了哪些主要的技术和设计。注意：

- 这里应该支持define 定义用户过程，但不支持set! 类操作
- 你可以考虑选择性地支持一些Scheme 内部过程。请在报告里说明你的求值器能处理哪些基本过程，并分析元循环解释器中实现了但你没有实现的基本过程，说明要实现它们有什么困难，需要对你完成的求值器做哪些修改。
- 可以不考虑局部定义。也可以考虑，作为一个选做问题

## 2 题目分析

为了对"代换模型"求值器有一个更具体的认识, 我们引用教材1.1.4的一个例子来说明其求值过程, 假设有Scheme定义:

Listing 1: 样例程序

```
1 (define (square x) (* x x))
2
3 (define (sum-of-squares x y)
4   (+ (square x) (square y)))
5
6 (define (f a)
7   (sum-of-squares (+ a 1) (* a 2)))
```

那么如果调用`(f 5)`, 那么代换模型的求值过程为:

```
(f 5)

would proceed according to the sequence of expansions

(sum-of-squares (+ 5 1) (* 5 2))

(+    (square (+ 5 1))      (square (* 5 2))  )

(+    (* (+ 5 1) (+ 5 1))   (* (* 5 2) (* 5 2)))

followed by the reductions

(+         (* 6 6)             (* 10 10))

(+           36                  100)

                   136
```

通过对上述过程的观察，我们可以总结出代换模型求值的几个基本特点：

1. 有着类似惰性求职的性质，一个参数只有在最后用到的时候才进行计算
2. 求值过程分为两部，第一部分为展开(expand)，第二部分为规约(reduce)
3. 展开过程时，以从外到里的顺序，逐渐展开表达式中的复合过程调用，将过程调用替换为其过程体
4. 规约过程时，以从里到外的顺序，每次求值最内部的基本过程，将基本过程调用替换为相应结果值

我们将紧紧抓住这个几个特点来有针对性的设计解释器。

# 3 设计构思

首先我们回顾一下书上4.1节最简单的元循环求值器的基本思路，其最重要的组件是两个过程eval和apply。eval负责对表达式进行语法分析并返回相应的结果，apply对其中的过程调用进行针对性的处理。其代码如下:

```
1  (define (eval exp env)
2    (cond ((self-evaluating? exp) exp)
3          ((variable? exp) (lookup-variable-value exp env))
4          ((quoted? exp) (text-of-quotation exp))
5          ((assignment? exp) (eval-assignment exp env))
6          ((definition? exp) (eval-definition exp env))
7          ((if? exp) (eval-if exp env))
8          ((lambda? exp)
9           (make-procedure (lambda-parameters exp)
10                          (lambda-body exp)
11                         env))
```

```
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))


(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
            arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

eval的分析过程是一种分治递归的过程, 比如拿if型语句为例, 它是一个由四个部分组成的表: (list 'if (if-predicate) (if-consequent) (if-alternative)), 那么如果对if表达式exp求值的话, 过程为

1. eval过程检测到exp是一个if
2. 分别将(if-predicate), (if-consequent), (if-alternative)送给eval求值
3. 根据(if-predicate)求值结果来选择返回(if-consequent)的结果还是(if-alternative)的.

但是对于"代换模型"所要求的求值器来说, 我们的求值过程稍稍有点不同. 不像元求值那样求值是一气呵成,一路求值到底的方式, 代换模型需要的是一种渐进式的求值. 比如以展开阶段为例, 如果我们给出的是前边的sum-of-squares的样例程序, 那么对

```
(f 5)
```

的求值求值返回的是一个展开的表达式

```
(sum-of-squares (+ 5 1) (* 5 2))
```

因为在求值阶段是不可能打印出全部的计算过程的, 所以我们只能是以求值器递进计算,然后外部不断打印的方式来实现逐步展开的方式来完成代换求值.

## 3.1 递进计算带来的问题

递进计算的方式会带来一个严重的问题，描述这个问题之前，我们先详细阐述一下过程调用的原理. 请注意元求值器里关于过程调用部分的实现:

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

如果在eval中语法分析器发现exp是一个过程调用，那么语法分析器会

1. 将exp中的过程送入eval，无论是基本过程调用还是复杂过程调用还是lambda表达式，结果应该返回一个procedure或者primitive
2. 将exp中的参数部分送入eval求值，放入arguments列表
3. 调用apply求值

从上面的apply代码可以看出，如果是基本过程的话，apply将直接返回计算结果，如果是复合过程，那么apply的做法是**新建一个扩展环境**，将argument的值绑定到参数列表中的相应变量上，然后再**在扩展环境中对复合过程的过程体进行求值**.

那么如果使用渐进计算的方式，apply会发生什么样的问题呢？还是以**(f 5)**为例，它返回的结果**(sum-of-squares (+ 5 1) (* 5 2))**. 这个时候发生的一个微妙变化就是，我在对**(sum-of-squares (+ 5 1) (* 5 2))**进行下一步求值的时候，开始求值的**初始环境应该是扩展之后的环境**而不是the-global-environment

这个问题在处理内部定义的时候非常的关键，因为如果一个过程在定义在另一个过程内部的时候，在外部环境是看不到的. 只能在实际调用到外部过程之后，将其内部定义加入调用环境，然后继续求值.

如果要在代换模型求值器里支持内部定义的话，那么在求值停下的那一步里，不仅要返回表达式的值，还需要返回求值到当前状态所能看到的环境，这个问题在多参数展开的时候会非常严重，比如对于下列程序

```
(define (calc1 n)
   (define (inter p) p)
   (inter n))
```

```
(define (calc2 n)
   (define (inter p) p)
   (inter n))
(define (calc a b)
   (+ (calc1 a) (calc2 b)))
```

那么如果求值**(calc 1 2)**的话,得到的结果是

```
(+ (calc1 1) (calc2 2))
```

再求值一步

```
(+ (inter 1) (inter 2))
```

这个时候,我们就需要为第一个(inter 1)保存环境为(calc1 n)的内部环境,然后为第二个(inter 2)保存(calc2 n)的内部环境.

这样的话,求值器需要修改为:无论认识时候对表达式求值,返回都是一个中间结果类型,其中包含表达式的具体值和求值结束时的环境.那么在处理上面的例子时,对于每个(inter x)都会自带其求值所需的环境.每次求值时,先判断当前表达式是否是中间结果,如果是的话,那么抛弃当前环境,解开中间结果,使用其表达式和自带的环境继续求值.具体的话即定义一个中间表达式类型intermediary:

```
(define (make-intermediary exp env)
    (list 'intermediary exp env))

(define (intermediary? exp)
    (tagged-list? exp 'intermediary))

(define (intermediary-exp exp)
    (cadr exp))

(define (intermediary-env exp)
    (caddr exp))
```

## 3.2 展开阶段

可以想见,其实对表达式的展开,其分析到最后,都是对复合过程的展开,其他表达式的展开,其最终到底层以后都是划归为复合过程调用→基本过程调用的转化.引用前面例子的展开阶段过程:

```
(f 5)

(sum-of-squares (+ 5 1) (* 5 2))

(+    (square (+ 5 1))      (square (* 5 2))  )
```

```
(+    (* (+ 5 1) (+ 5 1))   (* (* 5 2) (* 5 2)))
```

通过对上面的例子的观察, 对其展开过程可以总结为

1. 检查该过程是基本过程还是复合过程
2. 如果是简单过程, 那么过程体无法展开, 那么返回一个过程调用, 过程为当前基本过程, 参数为所有参数各自展开后的结果
3. 如果是复合过程, 那么将过程调用替换为过程体表达式返回即可. 但是注意, 这个时候过程体表达式中可能有对当前内部环境变量的引用, 因此需要将所有内部环境变量替换为其实际值以后才能返回, 比如在上述例子中, 对**(f 5)** 调用可以发现**(f a)**的函数体是**(sum-of-squares (+ a 1) (* a 2))**, 但是我们显然不能直接返回它, 因为当前a的值已经绑定为5了, 我们需要将表达式**(sum-of-squares (+ a 1) (* a 2))**中所有出现的本地变量替换为相应值后**(sum-of-squares (+ 5 1) (* 5 2))** 才能返回.

在第二步过程中, 是一个简单的问题递归, 在第三步中, 问题发生了变化, 因为递进求值的原则, 在展开一步过程之后, 就需要将过程体原样返回. 因此我们不仅需要一个"展开求值过程", 还需要一个"保持求值过程", 前者执行我们需要的递进求值, 而后者只是简单的替换表达式中出现的变量, 然后原样返回, 不作任何的展开操作. 在后面的实现过程中, 我们将实现一个**(expand exp env)** 和一个**(unexpand exp env)** 过程与之对应.

分治原则在大多数情形下都是适用的. 比如在展开if 表达式时, 只需要分别将(if-predicate), (if-consequent)以及(if-alternative)展开然后组合成新的if表达式返回即可.

这个方法在没有递归的情况下是完全正确的, 但是如果出现递归了以后呢? 比如求阶乘程序:

```
1 (define (fact n)
2    (if (= n 0)
3        1
4        (* n (fact (- n 1)))))
```

假如调用**(fact 1)**, 按照前面的规则, 我们会得到这样的结果:

```
(fact 1)
(if (= 1 0) 1 (* 1 (fact (- 1 1))))
(if (= 1 0) 1 (* 1 (if (= (- 1 1) 0) 1 (* (- 1 1) (fact (- (- 1 1) 1)))
   )))
```

到这里我们已经发现了问题, 如果按照前面的方法, 对if表达式的展开将没有止境, 因为在某个时刻递归边界中的条件已经成真, 递归出口中的某一半已经完全没有意义, 我们不能毫无节制的展开下去.

所以在if表达式的展开模式中, 我们需要另一种方法, 即在(if-consequent)以及(if-alternative)展开之前, 我们需要知道(if-predicate) 的值! 所以我们对if表达式的展开过程为:

1. 如果(if-predicate) 可以展开, 那么返回新的if表达式, 其(if-predicate)为展开一层以后的值, (if-consequent)和(if-alternative)保持不变
2. 如果(if-predicate) 不可展开, 但可以规约, 则那么返回新的if表达式, 其(if-predicate)为规约一层以后的值, (if-consequent)和(if-alternative)保持不变
3. 如果(if-predicate)既不可展开也不可规约, 那么它肯定是一个#t或者#f, 那么根据其值直接返回相应的(if-consequent)和(if-alternative) 之一.

在这个求值过程里, 我们需要三个过程的支持: 判断一个表达式可否展开, 判断一个表达式可否规约, 和规约表达式过程. 在下面的规约阶段分析中, 我们将详细阐述规约表达式分析. 在实现部分中, 我们将提供三个函数**(expandable? exp env)**, **(reducible? exp env)**和**(reduce exp env)**三个函数分别对应前面过程.

除了过程调用表达式和条件表达式外, 对于其他类型的表达式, 则其展开过程就和元循环求值器无二了.

**cond表达式** 转换为if处理

**define表达式** 直接添加入环境, 返回'ok

**lambda表达式** 转换为procedure返回, 不作展开

**begin表达式** 取出最后一个表达式返回即可, 但是我没有写.

具体处理可以参见实现部分的代码.

## 3.3 规约阶段

规约阶段的情况则比展开要简单了许多, 因为要对一个表达式进行规约之时, 其前提肯定是它已经展开完毕了, 即该表达式全部由基本过程调用和常数组成. 和展开的唯一不同, 是规约是由内到外, 自底向上的规约过程, 其过程概述如下:

1. 如果当前基本过程调用的所有参数中有任意一个是可规约的, 那么返回一个该过程调用, 参数为所有参数规约一次后的值(如果不能规约, 则规约后的值等于原值)
2. 如果当前基本过程调用的所有参数都不可规约, 则直接计算其基本过程, 返回结果

# 4 具体实现

## 4.1 求值器实现

前面提到了五个过程

1. 展开求值过程**(expand exp env)**
2. 保持求值过程**(unexpand exp env)**
3. 判断是否能展开**(expandable? exp env)**,
4. 判断是否能规约**(reducible? exp env)**
5. 规约求值过程**(reduce exp env)**

以下给出这五个过程的实现,代码有点长:

```
; ######################################################################
;                        expand and apply for expand
; ######################################################################
;
(define (expandable? exp env)
  (cond ((intermediary? exp)
             (expandable? (intermediary-exp exp) (intermediary-env exp)))
        ((self-evaluating? exp) #f)
        ((variable? exp) #f)
        ((quoted? exp) #f)
        ((if? exp)
            (or (expandable? (if-predicate exp) env)
                (expandable? (if-consequent exp) env)
                (expandable? (if-alternative exp) env)))
        ((lambda? exp) #f)

        ((cond? exp) (expandable? (cond->if exp) env))

        ((application? exp)
            (let ((procedure (expand (operator exp) env)))
              (cond ((primitive-procedure? procedure)
                     (apply or (map
                                (lambda (op) (expandable? op env))
                                (operands exp))))
                    ((compound-procedure? procedure) #t))))
        (else
            (error "Unknow expression type -- EVAL" exp))))



(define (expand exp env)
  (cond ((intermediary? exp)
             (expand (intermediary-exp exp) (intermediary-env exp)))
        ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))

        ; only definition correspond to old version of meta circular
        ((definition? exp) (eval-definition exp env))

        ((if? exp) (expand-if exp env))

        ((lambda? exp)
```

```
            (make-procedure (lambda-parameters exp)
                            (lambda-body exp)
                            env))

        ((cond? exp) (unexpand (cond->if exp) env))

        ((application? exp)
            (expand-apply
                    (expand (operator exp) env)
                    (map (lambda (arg) (unexpand arg env)) (operands exp))
                    env))
        (else
            (error "Unknow expression type -- EVAL" exp))))


(define (unexpand exp env)
  (cond ((intermediary? exp)
            (unexpand (intermediary-exp exp) (intermediary-env exp)))
        ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))

        ; only definition correspond to old version of meta circular
        ((definition? exp) (eval-definition exp env))

        ((if? exp)
            (make-intermediary
                (list 'if
                    (unexpand (if-predicate exp) env)
                    (unexpand (if-consequent exp) env)
                    (unexpand (if-alternative exp) env))
                env))

        ((lambda? exp)
            (make-procedure (lambda-parameters exp)
                            (lambda-body exp)
                            env))

        ((cond? exp) (unexpand (cond->if exp) env))

        ((application? exp)
            (make-intermediary
                (append-head (operator exp)
                    (map (lambda (arg) (unexpand arg env)) (operands exp)))
                env))
        (else
            (error "Unknow expression type -- EVAL" exp))))


(define (expand-apply procedure arguments env)
  (cond    ((primitive-procedure? procedure)

            ; if procedure is primitive, we need to expand its argument
            (append-head (primitive-procedure-name procedure)
                    (map (lambda (arg) (expand arg env)) arguments)))
        ((compound-procedure? procedure)

            ; if procedure is compound, then procedure body is expanded
            ; no need to expand its argument, saved for next time
            (unexpand-sequence

                ; does not support internal definition
                ; assuming that the procedure body has only one expression
                ;
                (procedure-body procedure)
                (extend-environment
                    (procedure-parameters procedure)
                    arguments
```

```
                    (procedure-environment procedure))))
        (else
            (error "Unknow procedure type -- APPLY" procedure))))


(define (unexpand-sequence exps env)
  (cond ((last-exp? exps) (unexpand (first-exp exps) env))
        (else (unexpand (first-exp exps) env)
              (unexpand-sequence (rest-exps exps) env))))



;#########################################################################
;                         expand various exp type
;#########################################################################

; if
(define (expand-if exp env)
    (cond
        ((expandable? (if-predicate exp) env)
        ; predicate is expandable, then expand it
        ; and keep consequent and alternative unchanged
        (list
            'if
            (expand (if-predicate exp) env)
            (unexpand (if-consequent exp) env)
            (unexpand (if-alternative exp) env)))
        ((reducible? (if-predicate exp) env)
        (list
            'if
            (reduce (if-predicate exp) env)
            (if-consequent exp)
            (if-alternative exp)))
        (else
            (if (if-predicate exp)
                (expand (if-consequent exp) env)
                (expand (if-alternative exp) env)))))



(define (make-intermediary exp env)
    (list 'intermediary exp env))

(define (intermediary? exp)
    (tagged-list? exp 'intermediary))

(define (intermediary-exp exp)
    (cadr exp))

(define (intermediary-env exp)
    (caddr exp))




;#########################################################################
;                           reduce and apply for it
;#########################################################################

(define (reducible? exp env)
  (cond ((tagged-list? exp 'intermediary) (reducible? (cadr exp) (caddr exp)))
        ((self-evaluating? exp) #f)
        ((variable? exp) #f)
        ((quoted? exp) #f)
        ((if? exp) #t)

        ;((cond? exp) (meval (cond->if exp) env))
```

```
        ((application? exp) #t)
        (else
            (error "Unknow expression type -- EVAL" exp))))


(define (reduce exp env)
  (cond    ((tagged-list? exp 'intermediary) (reduce (cadr exp) (caddr exp)))
        ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((if? exp)
            (if (or (reducible? (if-predicate exp) env)
                    (reducible? (if-consequent exp) env)
                    (reducible? (if-alternative exp) env))
                (list 'if
                    (reduce (if-predicate exp) env)
                    (reduce (if-consequent exp) env)
                    (reduce (if-alternative exp) env))

                ; predicate, consequent, alternative are
                ; simple value now, can be used directly
                (if (if-predicate exp)
                    (if-consequent exp)
                    (if-alternative exp))))

        ; in reduction stage the cond had already been
        ; transfered to if since cond  statement is expandable
        ; so we can ignore cond statement
        ;((cond? exp) (meval (cond->if exp) env))

        ; in reduction stage the procedure can only be primitive procedure
        ((application? exp)
            (let (  (proc (reduce (operator exp) env))
                    (args (operands exp)))

                ; if argument is reducible, then postpone
                ; actual calculation of primitive procedure
                (if (apply or (map (lambda (arg) (reducible? arg env)) args))
                    (append-head (operator exp) (map (lambda (arg) (reduce arg env
                        )) args))
                    (apply-in-underlying-scheme (primitive-procedure-body proc)
                        args))))

        (else
            (error "Unknow expression type -- EVAL" exp))))
```

其中有一个过程被多次用到**(append-head head .rears)**, 其功能是将rears代表的list集合按顺序append到一起, 然后以head为头元素, 这个过程在构造表达式特别有用, 因为表达式都是以一个字符串开头, 然后紧接着一串参数. 其实现为

```
(define (append-head head . rears)
    (define (loop answer rears)
        (if (null? rears)
            answer
            (loop (append answer (car rears)) (cdr rears))))
    (loop (list head) rears))
```

## 4.2 循环打印

因为前面阐述过的中间结果问题, 所以现在求值器返回的表达式不是一个可

以直接打印的表达式,它包含很多内部环境信息,所以在打印一个表达式之前,我们需要一个过程对中间结果进行过滤,返回一个可打印的表达式:

```
(define (printable exp)
  (cond    ((tagged-list? exp 'intermediary) (printable (cadr exp)))
       ((self-evaluating? exp) exp)
       ((variable? exp) exp)
       ((quoted? exp) exp)
       ((if? exp)
           (list
               'if
               (printable (if-predicate exp))
               (printable (if-consequent exp))
               (printable (if-alternative exp))))
       ((lambda? exp) 'wrong)

       ((cond? exp) exp)

       ((application? exp)
           (append-head (operator exp)
                   (map (lambda (arg) (printable arg)) (operands exp))))
       (else
           (error "Unknow expression type -- EVAL" exp))))
```

在驱动循环阶段,在得到输入的展开结果以后,需要不断对该值进行测试,如果可以展开则一直展开下去,否则进入规约阶段,直到求值完毕为止.

```
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (expand input the-global-environment)))
      (announce-output output-prompt)
        (if (tagged-list? input 'define)
            (user-print output)
            (expand-print output))))
  (driver-loop))

; expand phase
(define (expand-print object)
    (define (loop output)
        (newline)
        (display "expand:  ")
        (display (printable output))
        (if (expandable? output the-global-environment)
            (loop (expand output the-global-environment))
            (reduce-print output)))
    (loop object))

; reduce phase
(define (reduce-print object)
    (define (loop output)
        (newline)
        (display "reduce:  ")
        (display (printable output))
        (if (reducible? output the-global-environment)
            (loop (reduce output the-global-environment))
            (begin
                (newline)
                (display 'done))))
    (loop object))
```

# 5 程序测试

## 5.1 基本过程调用

```
;;; M-Eval input:
(define (square x) (* x x))

;;; M-Eval value:
ok

;;; M-Eval input:
(define (sum-of-squares a b)
    (+ (square a) (square b)))

;;; M-Eval value:
ok

;;; M-Eval input:
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))

;;; M-Eval value:
ok

;;; M-Eval input:
(f 5)

;;; M-Eval value:

expand:  (sum-of-squares (+ 5 1) (* 5 2))
expand:  (+ (square (+ 5 1)) (square (* 5 2)))
expand:  (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
reduce:  (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
reduce:  (+ (* 6 6) (* 10 10))
reduce:  (+ 36 100)
reduce:  136
done
```

## 5.2 内部定义

```
;;; M-Eval input:
(define (calc1 n)
   (define (inter p) (+ p 1))
   (inter n))

;;; M-Eval value:
ok

;;; M-Eval input:
(define (calc2 n)
   (define (inter p) p)
   (inter n))

;;; M-Eval value:
ok

;;; M-Eval input:
(define (calc a b)
   (+ (calc1 a) (calc2 b)))

;;; M-Eval value:
```

```
ok

;;; M-Eval input:
(calc 1 2)

;;; M-Eval value:

expand:  (+ (calc1 1) (calc2 2))
expand:  (+ (inter 1) (inter 2))
expand:  (+ (+ 1 1) 2)
reduce:  (+ (+ 1 1) 2)
reduce:  (+ 2 2)
reduce:  4
done
```

## 5.3 lambda语句

```
;;; M-Eval input:
((lambda (x) (+ x 1)) 1)

;;; M-Eval value:

expand:  (+ 1 1)
reduce:  (+ 1 1)
reduce:  2
done

;;; M-Eval input:
(((lambda (x) (lambda (p) (+ x p))) 1) 2)

;;; M-Eval value:

expand:  (+ 1 2)
reduce:  (+ 1 2)
reduce:  3
done
```

## 5.4 if语句

```
;;; M-Eval input:
(define (inc n) (+ n 1))

;;; M-Eval value:
ok

;;; M-Eval input:
(if (= 0 1)
    1
    (inc 1))

;;; M-Eval value:

expand:  (if #f 1 (inc 1))
expand:  (+ 1 1)
reduce:  (+ 1 1)
reduce:  2
done
```

## 5.5 递归和if的结合

```
;;; M-Eval input:
(define (fact n)
   (if (= n 0)
       1
       (* n (fact (- n 1)))))

;;; M-Eval value:
ok

;;; M-Eval input:
(fact 2)

;;; M-Eval value:

expand:  (if (= 2 0) 1 (* 2 (fact (- 2 1))))
expand:  (if #f 1 (* 2 (fact (- 2 1))))
expand:  (* 2 (if (= (- 2 1) 0) 1 (* (- 2 1) (fact (- (- 2 1) 1)))))
expand:  (* 2 (if (= 1 0) 1 (* (- 2 1) (fact (- (- 2 1) 1)))))
expand:  (* 2 (if #f 1 (* (- 2 1) (fact (- (- 2 1) 1)))))
expand:  (* 2 (* (- 2 1) (if (= (- (- 2 1) 1) 0) 1 (* (- (- 2 1) 1) (fact (- (- (-
      2 1) 1) 1))))))
expand:  (* 2 (* (- 2 1) (if (= (- 1 1) 0) 1 (* (- (- 2 1) 1) (fact (- (- (- 2 1)
     1) 1))))))
expand:  (* 2 (* (- 2 1) (if (= 0 0) 1 (* (- (- 2 1) 1) (fact (- (- (- 2 1) 1) 1))
     ))))
expand:  (* 2 (* (- 2 1) (if #t 1 (* (- (- 2 1) 1) (fact (- (- (- 2 1) 1) 1))))))
expand:  (* 2 (* (- 2 1) 1))
reduce:  (* 2 (* (- 2 1) 1))
reduce:  (* 2 (* 1 1))
reduce:  (* 2 1)
reduce:  2
done
```

## 5.6 cond语句

```
;;; M-Eval input:
(define (fact n)
   (cond ((= n 0) 1)
         (else (* n (fact (- n 1))))))

;;; M-Eval value:
ok

;;; M-Eval input:
(fact 2)

;;; M-Eval value:

expand:  (if (= 2 0) 1 (* 2 (fact (- 2 1))))
expand:  (if #f 1 (* 2 (fact (- 2 1))))
expand:  (* 2 (if (= (- 2 1) 0) 1 (* (- 2 1) (fact (- (- 2 1) 1)))))
expand:  (* 2 (if (= 1 0) 1 (* (- 2 1) (fact (- (- 2 1) 1)))))
expand:  (* 2 (if #f 1 (* (- 2 1) (fact (- (- 2 1) 1)))))
expand:  (* 2 (* (- 2 1) (if (= (- (- 2 1) 1) 0) 1 (* (- (- 2 1) 1) (fact (- (- (-
      2 1) 1) 1))))))
expand:  (* 2 (* (- 2 1) (if (= (- 1 1) 0) 1 (* (- (- 2 1) 1) (fact (- (- (- 2 1)
     1) 1))))))
expand:  (* 2 (* (- 2 1) (if (= 0 0) 1 (* (- (- 2 1) 1) (fact (- (- (- 2 1) 1) 1))
     ))))
expand:  (* 2 (* (- 2 1) (if #t 1 (* (- (- 2 1) 1) (fact (- (- (- 2 1) 1) 1))))))
expand:  (* 2 (* (- 2 1) 1))
```

```
reduce:  (* 2 (* (- 2 1) 1))
reduce:  (* 2 (* 1 1))
reduce:  (* 2 1)
reduce:  2
done
```

## 5.7 深层递归

```
;;; M-Eval input:
(define (even? n)
   (= (remainder n 2) 0))

;;; M-Eval value:
ok

;;; M-Eval input:
(define (cycle n)
    (cond ((= n 1) 1)
          ((even? n) (cycle (/ n 2)))
          (else (cycle (+ 1 (* n 3))))))

;;; M-Eval value:
ok

;;; M-Eval input:
(cycle 2)

;;; M-Eval value:

expand:  (if (= 2 1) 1 (if (even? 2) (cycle (/ 2 2)) (cycle (+ 1 (* 2 3)))))
expand:  (if #f 1 (if (even? 2) (cycle (/ 2 2)) (cycle (+ 1 (* 2 3)))))
expand:  (if (= (remainder 2 2) 0) (cycle (/ 2 2)) (cycle (+ 1 (* 2 3))))
expand:  (if (= 0 0) (cycle (/ 2 2)) (cycle (+ 1 (* 2 3))))
expand:  (if #t (cycle (/ 2 2)) (cycle (+ 1 (* 2 3))))
expand:  (if (= (/ 2 2) 1) 1 (if (even? (/ 2 2)) (cycle (/ (/ 2 2) 2)) (cycle (+ 1
      (* (/ 2 2) 3)))))
expand:  (if (= 1 1) 1 (if (even? (/ 2 2)) (cycle (/ (/ 2 2) 2)) (cycle (+ 1 (* (/
     2 2) 3)))))
expand:  (if #t 1 (if (even? (/ 2 2)) (cycle (/ (/ 2 2) 2)) (cycle (+ 1 (* (/ 2 2)
     3)))))
expand:  1
reduce:  1
done
```

你可以尝试一下**(cycle 3)**的执行结果。