函数式程序设计

郭　炜

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

第十五讲

用寄存器机器实现scheme解释器

存储分配和废料收集

● 元循环求值器在处理函数调用的时候，会创建新环境，但是在函数调用结束时却没有回收新环境的存储空间（可以改进）

●作为寄存器机器存在的解释器，能够支持表操作，需要完成存储分配和废料收集功能

将存储器看作向量

- scheme的向量操作：

```
(vector-ref <vector> <n>)  ;返回向量中下标为n的元素

(vector-set! <vector> <n> <value>)  ;设置向量中下标为n的元素
```

将存储器看作向量

- Lisp数据的表示

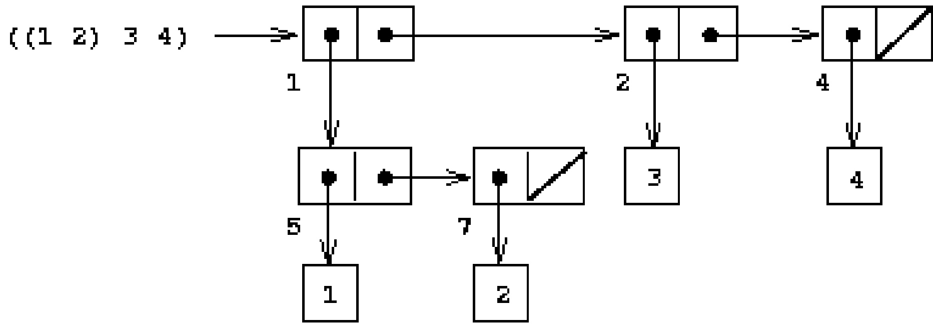
  ➢计算机的全部存储器就是两个向量：
  the-cars和the-cdrs

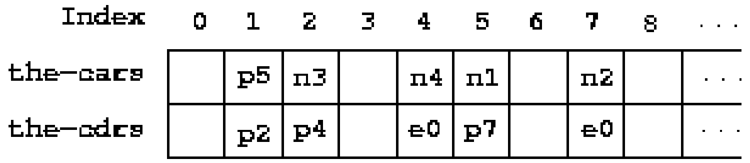
  ➢数据分成两大类，序对和非序对

  ➢每个序对都有一个独特的编号n，序对n的car存在 the-cars[n]，cdr存在 the-cdrs[n]。car和cdr都是指针。

  ➢指针为pi则表示其指向编号为i的序对。指针为ni则表示其指向常数i。空指针是e0。

  ➢box左下角的数字是box的编号





Index	0	1	2	3	4	5	6	7	8	...
the-cars		p5	n3		n4	n1		n2		...
the-cdrs		p2	p4		e0	p7		e0		...

将存储器看作向量

- Lisp数据的表示

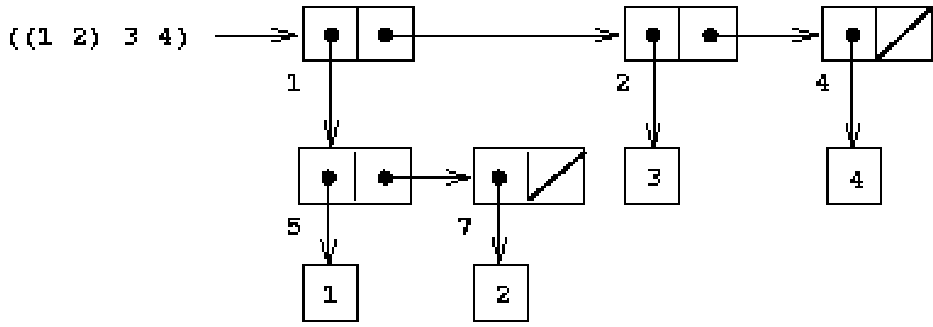
➢指针内部可以包含类型信息。

➢对于小的数据（如一个字符），可以直接放在指针内部。

➢对于大的数据，如字符串，符号，高精度数，都可以表示为一个列表，其指针就是一个序对指针。列表内部可以包含数据的类型信息，也可以在指针内部包含。

➢eq? 检测指针是否相等

➢一个符号只能有一个地方来存放



Index	0	1	2	3	4	5	6	7	8	...
the-cars		p5	n3		n4	n1		n2		...
the-cdrs		p2	p4		e0	p7		e0		...

基本表操作car,cdr,set-car!,set-cdr!的实现

- 两个寄存器the-cars和the-cdrs分别代表两个内存向量
- vector-ref 和 vector-set!是可以使用的基本向量操作
- car, cdr, set-car!, set-cdr! 的实现(它们非基本操作):

```
(assign <reg1> (op car) (reg <reg2>)) ;取编号为reg2的序对的car到reg1 =>
   (assign <reg1> (op vector-ref) (reg the-cars) (reg <reg2>))

(assign <reg1> (op cdr) (reg <reg2>)) ;取编号为reg2的序对的cdr到reg1 =>
   (assign <reg1> (op vector-ref) (reg the-cdrs) (reg <reg2>))

(perform (op set-car!) (reg <reg1>) (reg <reg2>))
   ;将编号为reg1的序对的car设置为reg2 =>
   (perform (op vector-set!) (reg the-cars) (reg <reg1>) (reg <reg2>))
(perform (op set-cdr!) (reg <reg1>) (reg <reg2>)) =>
   ;将编号为reg1的序对的cdr设置为reg2 =>
   (perform (op vector-set!) (reg the-cdrs) (reg <reg1>) (reg <reg2>))
```

基本表操作cons和eq?的实现

- cons的实现

```
assign <reg1> (op cons) (reg <reg2>) (reg <reg3>))
```

;将reg1和reg2的内容组成一个新的序对，其编号放入 reg1 =>

```
(perform (op vector-set!) (reg the-cars) (reg free) (reg <reg2>))
(perform (op vector-set!) (reg the-cdrs) (reg free) (reg <reg3>))
(assign <reg1> (reg free))
(assign free (op +) (reg free) (const 1))
```

free是一个特殊寄存器，其值为n则表示the-cars[n]和the-cdrs[n]都空闲可用

- eq?的实现

```
(op eq?) (reg <reg1>) (reg <reg2>)
```

检测reg1和reg2内容是否完全相同。**symbol? , number?, null?, pair?**等只检测指针的类型域

废料收集系统

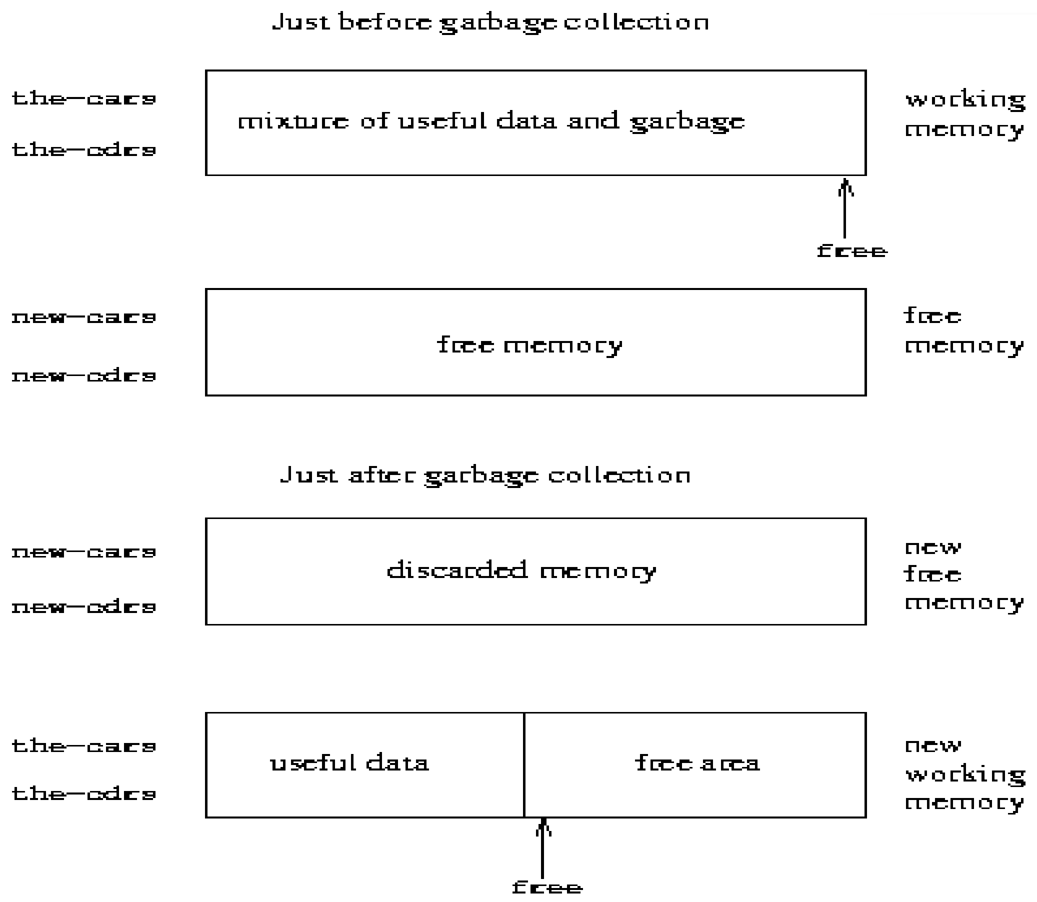
●随着程序的运行，有些存储单元的内容再也没有用处，应当回收，分配给后续的数据使用。

●从当前机器寄存器里的指针出发，经过一系列car和cdr操作能够到达的对象，都是可能有用的对象。除此之外的对象，都不再有用，可以回收。

●将存储器分成两半：工作区和自由区。自由区全空闲。cons需要分配序对，就从工作区分配。工作区满时，执行废料收集。

●废料收集的过程，是将工作区里今后还可能用到的数据，全部复制到自由区里的连续区域。复制完成后，原来自由区变为工作区，原工作区变为自由区。

废料收集系统的寄存器机器实现

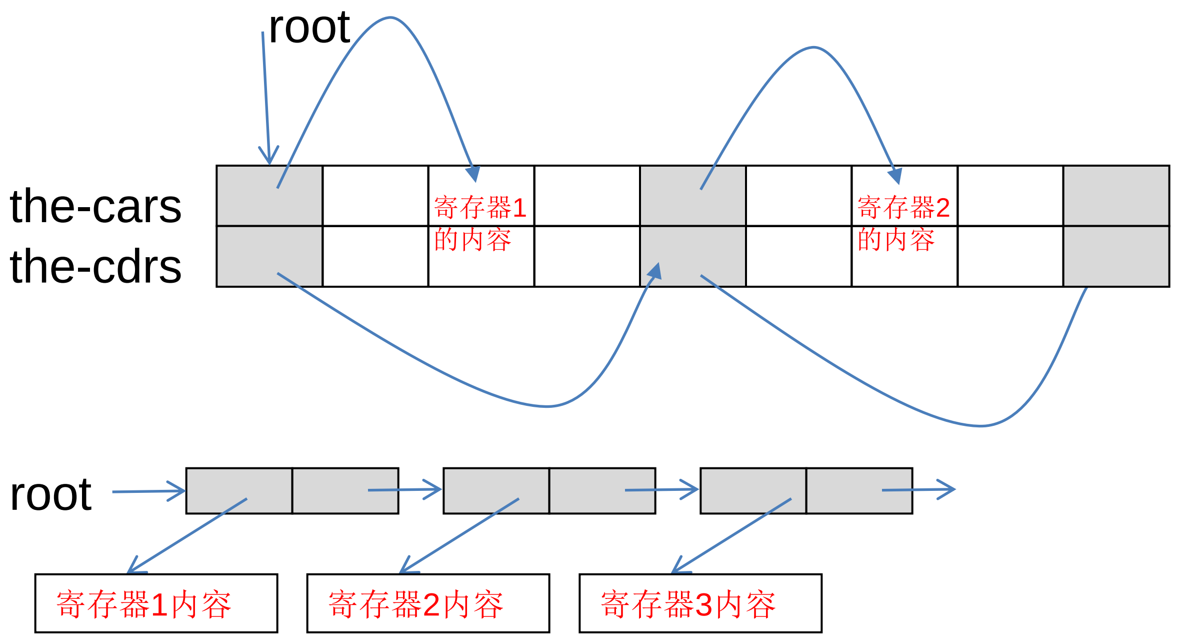


废料收集前

废料收集后

废料收集系统的寄存器机器实现

- 设置root寄存器指向一个列表，该列表包含除内存分配寄存器以外的所有寄存器的值（不包含 root, the-cars, the-cdrs, new-cars, new-cdrs, new, old, scan, free...）。该列表存在工作区中。列表中每个box的car来自向量the-cars, cdr来自向量the-cdrs



废料收集系统的寄存器机器实现

●初始工作

1） 设置指针scan和free, 开始都指向空闲区的地址0

2） 将root指向的第一个序对复制到空闲区地址0处，并修改root让其指向空闲区的地址 0

3） ++ free

● 复制循环

```
while(scan!= free) {
        scan此时指向空闲区里的已经被复制好的某个序对。
        接下来将要先后复制 *(scan->car)和*(scan->cdr),两者都是一个序对
        if ( scan->car 不是指向序对的指针（指向常量） ) {
                不进行复制 ;
        }
        else {
                if( *(scan->car)还没有被复制) {
                        将 *(scan->car)复制到free指向的地方;
                        设置 (*(scan->car)).car 为标记"broken heart",
                            表示*(scan->car)已经被复制;
                        设置 (*(scan->car)).cdr 为free，让它指向刚刚复制好的对象;
                        设置 scan->car 为free, 让它指向刚刚复制好的对象;
                        ++ free;
                }else {  scan->car = (*(scan->car)).cdr ; }
        }
        复制 *(scan->cdr), 过程与复制 *(scan->car)相同 ;
        ++scan;
}   scan==free说明复制工作已经完成。然后将空闲区和工作区互换, 整个废料收集工作完成。
```

废料收集系统的寄存器机器实现

```
begin-garbage-collection
   (assign free (const 0)) ;指向空闲区开头
   (assign scan (const 0)) ;指向空闲区开头
   (assign old (reg root)) ;准备复制root指向的对象到空闲区
   (assign relocate-continue (label reassign-root))
   (goto (label relocate-old-result-in-new))
reassign-root
   (assign root (reg new)) ;root指向空闲区中被复制的第一个对象
   (goto (label gc-loop))
```

废料收集系统的寄存器机器实现

```
;scan指向的序对是已经被复制到自由区中的一个序对
gc-loop
  (test (op =) (reg scan) (reg free))
  (branch (label gc-flip)) ;所有该复制的都已经复制就goto gc-flip
  (assign old (op vector-ref) (reg new-cars) (reg scan))
;此时打算复制scan指向的序对的car所指向的对象, 即 * (scan->car)
;因此将scan->car放入old
  (assign relocate-continue (label update-car))
;复制完后, 被复制对象的地址在new, 要转到update-car去做
;      scan->car=new
  (goto (label relocate-old-result-in-new))
```

```
update-car
   (perform
    (op vector-set!) (reg new-cars) (reg scan) (reg new))
;scan->car = new
   (assign old (op vector-ref) (reg new-cdrs) (reg scan))
;打算复制 *(scan->cdr)
   (assign relocate-continue (label update-cdr))
;复制结束后要转到update-cdr去做 scan->cdr = new
   (goto (label relocate-old-result-in-new))

update-cdr
   (perform
    (op vector-set!) (reg new-cdrs) (reg scan) (reg new))
;scan->cdr = new
   (assign scan (op +) (reg scan) (const 1)) ;++scan,scan指向下
一个已经复制好的序对，然后转gc-loop去复制*(scan->car)和*(scan->cdr)
   (goto (label gc-loop))
```

relocate-old-result-in-new
复制old指向的序对，即 *old。复制结束后new指向被复制序对

```
if (old 指向序对 ) {
        if( 该序对已经被复制过 )
                new = old->cdr; //此时old->cdr指向该序对的复制品
        else {
                new = free;
                复制该序对到空闲区free指向的地方;
                ++ free;
                old->car = broken heart;
                old->cdr = new;
        }
}
else //old指向常量。old可能等于以前某个序对的cdr或car，因此可能指向常量
        new = old;
(goto (reg relocate-continue))
```

```
relocate-old-result-in-new
   (test (op pointer-to-pair?) (reg old))
   (branch (label pair))
   (assign new (reg old))
   (goto (reg relocate-continue))
pair
   (assign oldcr (op vector-ref) (reg the-cars) (reg old))
   (test (op broken-heart?) (reg oldcr))
   (branch (label already-moved))
   (assign new (reg free)) ; new location for pair
   ;; Update free pointer.
   (assign free (op +) (reg free) (const 1))
   ;; Copy the car and cdr to new memory.
   (perform (op vector-set!)
            (reg new-cars) (reg new) (reg oldcr))
   (assign oldcr (op vector-ref) (reg the-cdrs) (reg old))
   (perform (op vector-set!)
            (reg new-cdrs) (reg new) (reg oldcr))
```

```
;; Construct the broken heart.
  (perform (op vector-set!)
                (reg the-cars) (reg old) (const broken-heart))
  (perform
   (op vector-set!) (reg the-cdrs) (reg old) (reg new))
  (goto (reg relocate-continue))

already-moved

  (assign new (op vector-ref) (reg the-cdrs) (reg old))
  (goto (reg relocate-continue))
```

```
gc-flip
   (assign temp (reg the-cdrs))
   (assign the-cdrs (reg new-cdrs))
   (assign new-cdrs (reg temp))
   (assign temp (reg the-cars))
   (assign the-cars (reg new-cars))
   (assign new-cars (reg temp))
```

显示控制的求值器(汇编语言实现的scheme解释器)

- 这是一个scheme程序，包含前述整个寄存器机器的解释器
- 这个程序包含元循环求值器(sehcme编写的scheme解释器)中几乎全部的函数(eval函数不需要)
- 设计一段汇编程序，能够读入scheme程序并运行之
- 将这段汇编程序安装到一个寄存器机器上
- 该寄存器机器有以下7个寄存器

exp	存放待求值的表达式
env	存放表达式求值所需的环境
val	存放求出来的表达式的值
continue	存放返回地址
proc	存放函数对象
argl	存放函数调用时的参数表，表中的参数的值都已经求出
unev	辅助寄存器，存放各种临时值

显示控制的求值器(汇编语言实现的scheme解释器)

- 该寄存器机器有大量操作，是通过元循环求值器中的函数实现的，例如：

```
self-evaluating?
variable?
quoted?
lambda-parameters
make-procedure
......
```

- 该寄存器机器有一些操作，是通过新增的一些函数实现的，等于是对前述寄存器机器解释器做了一些扩充，例如：

```
adjoin-arg
```

- 该寄存器机器也有一些操作，是通过**scheme**基本过程实现的，例如：

```
read
```

运行整个程序

```
(define scheme-machine-controller
        '( .....) )   ;整个汇编程序

(define scheme-machine
   (make-machine
    '(exp env val proc argl continue unev c d)
    eceval-operations ;操作列表
    scheme-machine-controller ;汇编程序
   ))

(define glb-env (setup-environment))
(start scheme-machine)
;启动寄存器机器。该机器读入scheme程序并运行之，输出该scheme程序的结果
```

本章的讲述如何编写能解释scheme程序的汇编程序。重点就是如何使用栈。

寄存器机器的操作列表

```
(define eceval-operations
           (list
               (list 'rem remainder)
               (list 'self-evaluating? self-evaluating?)
               (list 'variable? variable?)
               ......
               (list 'extend-environment extend-environment)
               ......
               (list 'read read)
               (list 'get-global-environment get-global-environment)
               .....
               (list 'eof? eof?)
          )
  )
```

```
(define (get-global-environment)
  glb-env)
```

汇编程序的实现 -- 驱动循环

```
;程序从此开始运行:
read-eval-print-loop
   (perform (op initialize-stack))
   (assign exp (op read)) ;读入需要求值的scheme表达式
   (assign env (op get-global-environment))
   (assign continue (label print-result))  ;before doing
somthing that may change the return address, always assign
continue with right label
   (goto (label eval-dispatch)) ;eval-dispatch相当于元循环求值器的 eval
print-result
   (perform (op user-print) (reg val)) ;the value of exp is
stored in val
   (goto (label read-eval-print-loop))
```

汇编程序的实现 -- eval-dispatch，相当于eval

```
eval-dispatch
;after this is completed, the value of exp is stored in reg val,and
;program goto the address stored in reg continue;
; eval value of exp in env
  (test (op eof?) (reg exp))
  (branch (label program-end))
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
  (test (op quoted?) (reg exp))
  (branch (label ev-quoted))
  (test (op assignment?) (reg exp))
  (branch (label ev-assignment))
  (test (op definition?) (reg exp))
  (branch (label ev-definition))
```

eval-dispatch的各路分支

```
ev-self-eval
   (assign val (reg exp))
   (goto (reg continue))
ev-variable
   (assign val (op lookup-variable-value) (reg exp) (reg env))
   (goto (reg continue))
ev-quoted
   (assign val (op text-of-quotation) (reg exp))
   (goto (reg continue))
ev-lambda
   (assign unev (op lambda-parameters) (reg exp))
   (assign exp (op lambda-body) (reg exp))
   (assign val (op make-procedure)
               (reg unev) (reg exp) (reg env))
   (goto (reg continue))
;just store the value of exp in reg val,and goto continue
;the value of a lambda is a function object, and it is stored also in
val
```

eval-dispatch的各路分支 -- 处理赋值语句

```
ev-assignment
   (assign unev (op assignment-variable) (reg exp))
   (save unev)   ; save variable name for later
   (assign exp (op assignment-value) (reg exp))
   (save env)
   (save continue)
   (assign continue (label ev-assignment-1))
   (goto (label eval-dispatch)) ;evaluate the assignment value stored in exp
ev-assignment-1
   (restore continue)
   (restore env)
   (restore unev)
   (perform
    (op set-variable-value!) (reg unev) (reg val) (reg env))
        ;variable name is stored in uenv
   (assign val (const ok))
   (goto (reg continue))
```

eval-dispatch的各路分支 -- 处理define语句

```
ev-definition
   (assign unev (op definition-variable) (reg exp))
   (save unev)                          ; save variable for later
   (assign exp (op definition-value) (reg exp))
   (save env)
   (save continue)
   (assign continue (label ev-definition-1))
   (goto (label eval-dispatch))   ; evaluate the definition value
ev-definition-1
   (restore continue)
   (restore env)
   (restore unev)
   (perform
    (op define-variable!) (reg unev) (reg val) (reg env))
   (assign val (const ok))
   (goto (reg continue))
```

eval-dispatch的各路分支 -- 处理if表达式

```
ev-if
   (save exp)       ; save the whole if expression for later
   (save env)
   (save continue) ;after ev-if,should goto continue
   (assign continue (label ev-if-decide))
   (assign exp (op if-predicate) (reg exp))
   (goto (label eval-dispatch))  ; evaluate the predicate
ev-if-decide
   (restore continue) ;after ev-if,should goto continue
   (restore env)  ;the env for the whole if exp
   (restore exp)  ;restore the whole if exp
   (test (op true?) (reg val))
   (branch (label ev-if-consequent))
ev-if-alternative
   (assign exp (op if-alternative) (reg exp))
   (goto (label eval-dispatch))
ev-if-consequent
   (assign exp (op if-consequent) (reg exp))
   (goto (label eval-dispatch))
```

eval-dispatch的各路分支 -- 处理begin表达式

```
ev-begin
   (assign unev (op begin-actions) (reg exp))
   (save continue)
   (goto (label ev-sequence))
```

eval-dispatch的各路分支 -- 处理表达式序列

```
ev-sequence   ;compound-apply的时候也会被调用
   (assign exp (op first-exp) (reg unev))
   (test (op last-exp?) (reg unev)) ;最后一个表达式要单独处理
   (branch (label ev-sequence-last-exp))
   (save unev)
   (save env)
   (assign continue (label ev-sequence-continue))
   (goto (label eval-dispatch))
ev-sequence-continue
   (restore env)
   (restore unev)
   (assign unev (op rest-exps) (reg unev))
   (goto (label ev-sequence))
ev-sequence-last-exp
   (restore continue)
   (goto (label eval-dispatch)) ;去求值最后一个表达式
```

上述处理语句序列的写法，保证了尾递归用常数栈空间就可以实现

下面程序不会导致爆栈

```
(define (count n)
   (newline)
   (display n)
   (count (+ n 1)))
```

处理表达式序列的不支持尾递归的写法

```
ev-sequence
   (test (op no-more-exps?) (reg unev)) ;序列已经放在 unev中
   (branch (label ev-sequence-end))
   (assign exp (op first-exp) (reg unev)) ;最后一个表达式和其他表达式一样处理
   (save unev)
   (save env) ;最后一个表达式处理完之前，unev和env都不会出栈
   (assign continue (label ev-sequence-continue))
   (goto (label eval-dispatch))
ev-sequence-continue
   (restore env)
   (restore unev)
   (assign unev (op rest-exps) (reg unev))
   (goto (label ev-sequence))
ev-sequence-end
   (restore continue)
   (goto (reg continue))
```

下面程序导致爆栈

```
(define (count n)
   (newline)
   (display n)
   (count (+ n 1)))
```

eval-dispatch的各路分支 -- 处理函数调用

```
ev-application
   (save continue)
   (save env)
   (assign unev (op operands) (reg exp))
   (save unev);保存全部参数
   (assign exp (op operator) (reg exp))
   (assign continue (label ev-appl-did-operator))
   (goto (label eval-dispatch))
```

```
(define (empty-arglist) '())
```

```
ev-appl-did-operator
   (restore unev)                         ; the operands
   (restore env)
   (assign argl (op empty-arglist)) ;argl存放值已经求出的参数表
   (assign proc (reg val))   ; the operator,is a function object in val
   (test (op no-operands?) (reg unev))
   (branch (label apply-dispatch))
   (save proc)
```

eval-dispatch的各路分支 -- 处理函数调用

```
ev-appl-operand-loop
   (save argl)
   (assign exp (op first-operand) (reg unev))
   (test (op last-operand?) (reg unev))
   (branch (label ev-appl-last-arg))
   (save env)
   (save unev) ;参数表，其中第一个马上就要被求值后丢弃
   (assign continue (label ev-appl-accumulate-arg))
   (goto (label eval-dispatch))

ev-appl-accumulate-arg
   (restore unev)  ;参数表，其中第一个已经求出值了
   (restore env)
   (restore argl)
   (assign argl (op adjoin-arg) (reg val) (reg argl))
   (assign unev (op rest-operands) (reg unev))
   (goto (label ev-appl-operand-loop))
```

```
(define (adjoin-arg arg arglist)
  (append arglist (list arg)))
```

eval-dispatch的各路分支 -- 处理函数调用

```
ev-appl-last-arg
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))
ev-appl-accum-last-arg
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore proc)
  (goto (label apply-dispatch))
```

eval-dispatch的各路分支 -- 处理函数调用

```
apply-dispatch
   (test (op primitive-procedure?) (reg proc))
   (branch (label primitive-apply))
   (test (op compound-procedure?) (reg proc))
   (branch (label compound-apply))
   (goto (label unknown-procedure-type))
primitive-apply
   (assign val (op apply-primitive-procedure)
               (reg proc)
               (reg argl))
   (restore continue)
   (goto (reg continue))
compound-apply
   (assign unev (op procedure-parameters) (reg proc));proc是函数对象
   (assign env (op procedure-environment) (reg proc))
   (assign env (op extend-environment)
               (reg unev) (reg argl) (reg env))
   (assign unev (op procedure-body) (reg proc))
   (goto (label ev-sequence))
```

```
(define (apply-primitive-procedure op args)
  (apply (primitive-implementation op) args))
```

eval-dispatch的各路分支 -- 错误处理

```
unknown-expression-type
  (assign val (const unknown-expression-type-error))
  (goto (label signal-error))
unknown-procedure-type
  (restore continue)     ; clean up stack (from apply-dispatch)
  (assign val (const unknown-procedure-type-error))
  (goto (label signal-error))
signal-error
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))
program-end
```