程序设计技术与方法

郭　炜

http://weibo.com/guoweiofpku

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

第十四讲

寄存器机器的计算

元循环求值器依然不够本质

- 元循环求值器使用了Racket的apply完成最基础的操作,因而不能解释以下过程:

➢ 求出子表达式的值之后如何把它送给使用值的表达式?

➢ 为什么有些递归过程会产生迭代型计算过程（只需要常量空间），而另一些却产生递归型计算过程（需要线性以上的空间）？

模拟实现一个寄存器机器就能回答以上问题

用语言描述的求GCD的寄存器机器

```
;以下 a,b,t都是寄存器
(controller
 test-b   ;语句标号
    (test (op =) (reg b) (const 0)) ;reg表示寄存器,op表示操作
    ;test+branch 表示测试和跳转
    (branch (label gcd-done));在上面条件满足时跳到标号 gcd-done
    (assign t (op rem) (reg a) (reg b));rem表示求余数 t= a rem b
    (assign a (reg b)) ;assign表示赋值 a = b
    (assign b (reg t))
    (goto (label test-b))
 gcd-done) ;语句标号
;最终寄存器a里的值就是答案
```

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

用语言描述的求GCD的寄存器机器

➢ 替换rem这种高级实现：

```
(controller
 test-b ;语句标号
   (test (op =) (reg b) (const 0))
   (branch (label gcd-done))
   (assign t (reg a))
 rem-loop ;语句标号。下面用减法求 t rem b
   (test (op <) (reg t) (reg b)) ;一直做到 t < b，即求出了余数t
   (branch (label rem-done))
   (assign t (op -) (reg t) (reg b))
   (goto (label rem-loop))
 rem-done ;语句标号
   (assign a (reg b))
   (assign b (reg t))
   (goto (label test-b))
 gcd-done)
```

```
(define (remainder n d)
  (if (< n d)
      n
      (remainder (- n d) d)))
```

求a和b的GDC以及c和d的GCD的机器

```
gcd-1 ;求a,b的gdc
  (test (op =) (reg b) (const 0))
  (branch (label after-gcd-1))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd-1))
after-gcd-1 ;最终结果在a
    ....
gcd-2 ;求c,d的gdc
  (test (op =) (reg d) (const 0))
  (branch (label after-gcd-2))
  (assign s (op rem) (reg c) (reg d))
  (assign c (reg d))
  (assign d (reg s))
  (goto (label gcd-2))
after-gcd-2  ;最终结果在c
```

方法非常笨拙！
应该重复利用gcd的部件！

求a和b的GDC以及c和d的GCD的机器 --- 改进一

```
gcd-1   ;此处的改进是不需要使用寄存器 c和d了，但还是重复
  (test (op =) (reg b) (const 0))
  (branch (label after-gcd-1))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd-1))
after-gcd-1
   ....  ;把原本应放在c和d的值放入a和b
gcd-2
  (test (op =) (reg b) (const 0))
  (branch (label after-gcd-2))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd-2))
after-gcd-2
```

求a和b的GDC以及c和d的GCD的机器 --- 改进二

➢用一个continue寄存器记录过程的返回地址

```
gcd
 (test (op =) (reg b) (const 0))
  (branch (label gcd-done))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd))
gcd-done
  (goto (reg continue)) ;可以将标号存入寄存器并且 goto 一个寄存器里的标号
..... ;将需要的值放入 a ,b  机器从这里开始工作
  (assign continue (label after-gcd-1))
  (goto (label gcd))
..... ;将需要的值放入 a ,b
after-gcd-1
  (assign continue (label after-gcd-2))
  (goto (label gcd))
after-gcd-2
```

采用堆栈实现递归

- 在过程嵌套调用的情况下，仅用一个continue寄存器来记录过程的返回地址就不够用了

- 使用栈来解决这个问题

  - 增加 save n 指令将值n 存入栈顶

  - 增加 restore n 指令用于从栈顶取出数据放到n

```
struct Node {
        int n;  int retAdr;
        Node(int n_,int adr):n(n_),retAdr(adr) { }
} ;
int factorial(int n) {
        int retVal;    stack<Node> stk;  stk.push(Node(n,0));
        while(!stk.empty()) {
                Node nd = stk.top();
                if( nd.n == 1) {  retVal = 1;  stk.pop(); }
                else {
                        if( nd.retAdr == 0) {
                                stk.top().retAdr = 1;
                                stk.push(Node(nd.n-1,0));
                        }else {
                                retVal *= nd.n; stk.pop();
                        }
                }
        }
        return retVal;
}
```

用栈实现阶乘机器

```
(controller
    (assign continue (label fact-done));设置最终返回地址
 fact-loop
    (test (op =) (reg n) (const 1))
    (branch (label base-case))
    (save continue) ;保存返回地址到栈里
    (save n) ;保存n到栈里
    (assign n (op -) (reg n) (const 1))
    (assign continue (label after-fact));算出f(n-1)后要返回到和n乘的地方
    (goto (label fact-loop))
 after-fact
    (restore n)
    (restore continue)
    (assign val (op *) (reg n) (reg val)) ; val now contains n(n - 1)!
    (goto (reg continue)); return to caller
 base-case
    (assign val (const 1))             ; base case: 1! = 1
    (goto (reg continue))              ; return to caller
 fact-done) ;最终寄存器 val中存放着结果
```

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

求斐波那契数列第n项的双重递归机器

```
(controller
    (assign continue (label fib-done))
 fib-loop
    (test (op <) (reg n) (const 2))
    (branch (label immediate-answer))
    ;; set up to compute Fib(n - 1)
    (save continue) ;返回地址入栈
    (assign continue (label afterfib-n-1))
    (save n)                              ; save old value of n
    (assign n (op -) (reg n) (const 1)); clobber n to n - 1
    (goto (label fib-loop))          ; perform recursive call
 afterfib-n-1                   ; upon return, val contains Fib(n - 1)
    (restore n)
    (restore continue)
    ;; set up to compute Fib(n - 2)
    (assign n (op -) (reg n) (const 2))
    (save continue)
    (assign continue (label afterfib-n-2))
```

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

求斐波那契数列的双重递归机器

```
   (save val)                              ; save Fib(n - 1)
   (goto (label fib-loop))
afterfib-n-2                         ; upon return, val contains Fib(n - 2)
   (assign n (reg val))                    ; n now contains Fib(n - 2)
   (restore val)                           ; val now contains Fib(n - 1)
   (restore continue)
   (assign val                             ;  Fib(n - 1) +  Fib(n - 2)
           (op +) (reg val) (reg n))
   (goto (reg continue))               ; return to caller, answer is in val
immediate-answer
   (assign val (reg n))                    ; base case:  Fib(n) = n
   (goto (reg continue))
fib-done)
```

机器指令汇总

```
(assign <register-name> (reg <register-name>))
(assign <register-name> (const <constant-value>))
(assign <register-name> (op <operation-name>)
                             <input1> ... <inputn>)
(perform (op <operation-name>) <input1> ... <inputn>)
(test (op <operation-name>) <input1> ... <inputn>)
(branch (label <label-name>))
;上面<inputi>是  (reg <register-name>) 或 (const <constant-value>).
(goto (label <label-name>))
(assign <register-name> (label <label-name>))
(goto (reg <register-name>))
(save <register-name>)
(restore <register-name>)
;constant-value以后还可以是 (const "abc")或(const abc)(符号abc)或
(const (a b c)) 或 (const ()))
```

寄存器机器的模拟

●用下面四个函数从外部操作一台机器：

➢构造一台包含寄存器，操作和控制器的机器：

```
(make-machine <register-names> <operations> <controller>)
```

controller实际上就是指令的序列

➢在给定的机器的寄存器中存放值

```
(set-register-contents! <machine-model> <register-name> <value>)
```

machine-model是个机器，实际上是个闭包

➢取给定的机器的寄存器中的值

```
(get-register-contents <machine-model> <register-name>)
```

➢启动一台机器

```
(start <machine-model>)
```

构造一台机器，给一些寄存器赋值后，就可以启动机器，然后在某个寄存器中得到机器的运行结果。

定义gcd机器

```
(define gcd-machine
  (make-machine
   '(a b t) ;寄存器列表
   (list (list 'rem remainder) (list '= =)) ;操作列表
   '(test-b ;控制器(指令序列)
       (test (op =) (reg b) (const 0))
       (branch (label gcd-done))
       (assign t (op rem) (reg a) (reg b))
       (assign a (reg b))
       (assign b (reg t))
       (goto (label test-b))
     gcd-done)))
```

每台机器由寄存器列表、操作列表和指令序列（控制器）组成。这三项内容在构造机器时即确定，不可更改。指令序列决定机器的功能。一台机器的功能是固定的。

运行gcd机器

```
(set-register-contents! gcd-machine 'a 206)
done
(set-register-contents! gcd-machine 'b 40)
done
(start gcd-machine)
done
(get-register-contents gcd-machine 'a)
2
(set-register-contents! gcd-machine 'a 24)   ;再运行一次
done
(set-register-contents! gcd-machine 'b 18)
done
(start gcd-machine)
done
(get-register-contents gcd-machine 'a)
6
```

创建新机器

●一台空机器实际上是make-new-machine创建的一个闭包，可以接受各种消息，可以扩充

```
(define (make-machine register-names ops controller-text)
  ;参数分别为寄存器列表、操作列表和指令序列
  (let ((machine (make-new-machine)));machine开始是空机器
    (for-each (lambda (register-name)
                ((machine 'allocate-register) register-name))
              register-names)
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
     (assemble controller-text machine)) ;安装指令序列
    machine))
```

make-new-machine创建一台空机器，然后make-machine添加进寄存器列表，操作列表和指令序列。

定义gcd机器

```
(define gcd-machine
  (make-machine
   '(a b t) ;寄存器列表
   (list (list 'rem remainder) (list '= =)) ;操作列表
   '(test-b ;控制器(指令序列)
       (test (op =) (reg b) (const 0))
       (branch (label gcd-done))
       (assign t (op rem) (reg a) (reg b))
       (assign a (reg b))
       (assign b (reg t))
       (goto (label test-b))
     gcd-done)))
```

每台机器由寄存器列表、操作列表和指令序列（控制器）组成。这三项内容在构造机器时即确定，不可更改。指令序列决定机器的功能。

机器的结构

机器是个闭包，有以下内部状态：

1) 栈。空机器的栈是个空表。

2) 寄存器列表。空机器里只有程序计数器**pc**和标志寄存器**flag**。

- **pc**指向指令序列中下一条要执行的指令。**flag**用于**test**和**branch**操作。**test**设置**flag**，**branch**根据**flag**决定是否跳转

- 机器的内部过程**allocate-register**往寄存器表中添加寄存器，**lookup-register**在寄存器表中查找寄存器

3) 操作列表。空机器里只包含初始化栈这一个操作。

4) 指令序列。空机器里指令序列是个空表。

机器上程序的执行

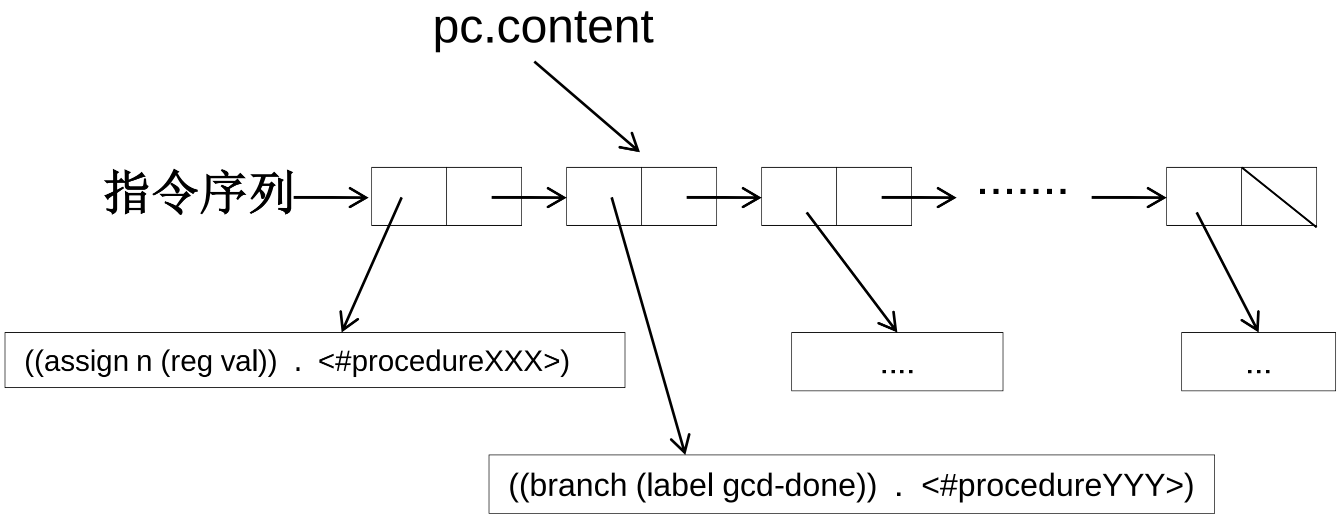
● 指令序列中每条指令是一个数据结构，内部包含指令文本和一个无参数的“指令执行过程”，调用这个过程就模拟了该指令的执行。指令是个序对(不是列表）形如：

```
((assign n (reg val)) .  <#procedureXXX>)
```

●机器内部过程 (assemble controller-text machine) 对原始的纯文字形式的指令序列进行分析，将其变为如上形式的机器指令的序列，然后存入机器内部的 the-instruction-sequence 变量。

机器上程序的执行

- 机器内部过程excute逐条执行程序中的指令。寄存器pc的内部变量content指向下一条要执行的指令。excute根据pc取出下一条指令进行执行，直到没有指令可以执行。



pc.content是机器里指令序列的一个后缀（从指令序列的开头或中间开始，直到结尾的子序列）

pc.content = (cdr pc.content) 即让pc指向下一条指令。

创建新机器

```
(define (make-new-machine)
  (let ((pc (make-register 'pc)) ;名字 'pc 'flag没用
; (make-register的结果是个闭包，内部有状态contents，放着寄存器的值。
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()));机器的指令序列
    (let ((the-ops  ;操作列表
           (list (list 'initialize-stack
                       (lambda () (stack 'initialize)))))
          (register-table
           (list (list 'pc pc) (list 'flag flag))))
      (define (allocate-register name) ;添加一个寄存器表项。表项是(名字 寄存器)
        (if (assoc name register-table)
            (error "Multiply defined register: " name)
            (set! register-table
                  (cons (list name (make-register name))
                        register-table)))
        'register-allocated)
```

创建新机器

```
(define (lookup-register name);根据名字查找寄存器
  (let ((val (assoc name register-table)))
    (if val
        (cadr val) ;返回结果是个闭包，代表寄存器。val形如 ('pc pc)
        (error "Unknown register:" name))))
(define (execute)
  (let ((insts (get-contents pc)));insts是指令序列后缀
    (if (null? insts)
        'done
        (begin
          ((instruction-execution-proc (car insts)))
                ;(car  insts)就是指令序列后缀中的第一条指令
          (execute)))))
```

insts是pc所指的指令序列后缀。insts中的每个元素形如：

((assign n (reg val)) . <#procedureXXX>)

```
(define (instruction-execution-proc inst)
  (cdr inst))
```

创建新机器

```
(define (dispatch message)
  (cond ((eq? message 'start)
          (set-contents! pc the-instruction-sequence)
          (execute)) ;pc指向执行指令序列的开头,从pc开始执行
        ((eq? message 'install-instruction-sequence)
          (lambda (seq) (set! the-instruction-sequence seq)))
          ;安装分析完成后的指令序列seq, 在make-machine中用到
        ((eq? message 'allocate-register) allocate-register)
        ((eq? message 'get-register) lookup-register)
        ((eq? message 'install-operations) ;make-machine中用
          (lambda (ops) (set! the-ops (append the-ops ops))))
        ((eq? message 'stack) stack)
        ((eq? message 'operations) the-ops)
        (else (error "Unknown request -- MACHINE" message))))
dispatch))
```

从外部操作机器

```
(define (start machine)
  (machine 'start)) ;让机器开始运行

(define (get-register-contents machine register-name)
  (get-contents (get-register machine register-name)))

(define (set-register-contents! machine register-name value)
  (set-contents! (get-register machine register-name) value)
  'done)

(define (get-register machine reg-name)
;返回值是个register，就是个闭包，里面只有一个 content状态变量
  ((machine 'get-register) reg-name))
```

寄存器的实现

- 一个寄存器是一个闭包，内部有contents这个状态变量

```
(define (make-register name)  ;name没用
  (let ((contents '*unassigned*))
    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value) (set! contents value)))
            (else
             (error "Unknown request -- REGISTER" message))))
    dispatch))

(define (get-contents register)
  (register 'get))

(define (set-contents! register value)
  ((register 'set) value))
```

栈的实现

```
(define (make-stack)
  (let ((s '())) ;s就是栈
    (define (push x) (set! s (cons x s)))
    (define (pop)
      (if (null? s)
          (error "Empty stack -- POP")
          (let ((top (car s)))
            (set! s (cdr s))
            top)))
    (define (initialize)
      (set! s '())   'done)
    (define (dispatch message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) (pop))
            ((eq? message 'initialize) (initialize))
            (else (error "Unknown request -- STACK"  message))))
   dispatch))
```

访问栈：

```
(define (pop stack)
  (stack 'pop))

(define (push stack value)
  ((stack 'push) value))
```

分析原始指令序列并安装到机器

```
;分析原始的指令序列，并将分析后的指令序列添加到机器
;controller-text是原始形式的指令序列

(define (assemble controller-text machine)
  (extract-labels controller-text
    (lambda (insts labels)
      (update-insts! insts labels machine)
      insts)))
```

insts是尚未分析好的，但去除了标号的不带可执行过程指令序列 **labels**是分析好的标号序列

update-insts!往**machine**中安装分析好的指令序列

分析原始指令序列并安装到机器

```
(define (extract-labels text receive) ;只在assemble中被调用
  (if (null? text)
      (receive '() '());此处这个receive实际上就是assemble中的LBD
      (extract-labels (cdr text)
       (lambda (insts labels)
         (let ((next-inst (car text)))
           (if (symbol? next-inst) ;为true说明是label,如 'gcd-done
               (receive insts
                        (cons (make-label-entry next-inst
                                                insts)
                              labels))
               (receive (cons (make-instruction next-inst)
                              insts)
                        labels)))))))
```

每次递归，extract-labels都在receive外加包一层，最里层就是LBD。LBD 实际上只被调用一次！

```
(define (assemble controller-text machine)
  (extract-labels controller-text
    (lambda (insts labels) ;LBD
      (update-insts! insts labels machine)
      insts)))
```

帮助理解extract-labels

●考查下面的函数：

```
(define (f text receive)
;参数receive一直没被调用，而是越变越复杂，直到 text为null,此时的receive才被调用
  (if (null? text)
      (receive '())
      (f (cdr text)
           (lambda (lst)
             (receive (cons (* 2 (car text)) lst))))))

(f (list 1 2 3 4 5 6)
   (lambda (lst) (display lst))) ; 记此lambda 为 RC
```

=> (2 4 6 8 10 12)

帮助理解extract-labels

执行`(f (list 1 2 3.... n)`

```
    (lambda (lst) (display lst))) ; 记此lambda 为 RC
```

考察 receive参数的变化过程:

```
RC
(lbd (lst)
     (RC (cons (* 2 1) lst)))   ;lbd-1
(lbd (lst)
     (lbd-1 (cons (* 2 2) lst))) ;lbd-2
(lbd (lst)
     (lbd-2 (cons (* 2 3) lst))) ;lbd-3
.....
(lbd (lst)
     (lbd-[n-1] (cons (* 2 n) lst)) ;lbd-n
```

```
(define (f text receive)
  (if (null? text)
      (receive '())
      (f (cdr text)
         (lambda (lst)
           (receive
          (cons (* 2 (car text))
            lst))))))
```

text为空时，执行:

```
  (lbd-n '())
```

每个 lbd-i都是个闭包，里面有个状态变量，就是当时的 (car text)

最终结果是调用 (RC (2 4 6 8 10 12))

分析指令序列

```
(define (update-insts! insts labels machine)
;为insts中的每条指令添加可执行过程，insts本来是文本形式的指令序列
  (let ((pc (get-register machine 'pc))
        (flag (get-register machine 'flag))
        (stack (machine 'stack))
        (ops (machine 'operations)));拿到上面这些东西的指针
   (for-each
      (lambda (inst)
       (set-instruction-execution-proc!
        inst  ;为每条指令加上分析后得到的可执行过程
        (make-execution-procedure ;分析指令的结果是产生一个可执行过程
         (instruction-text inst) labels machine
         pc flag stack ops)))
     insts))) ;返回分析后的指令序列
```

指令相关函数

```
(define (make-instruction text)
  ;text是指令的文字形式。本函数创建一个只有文字形式的指令
  (cons text '()))
(define (instruction-text inst)
  (car inst))
(define (instruction-execution-proc inst)
  (cdr inst))

(define (set-instruction-execution-proc! inst proc)
  (set-cdr! inst proc))
;参数inst形如：  ((test (op =) (reg b) (const 0)))
;执行之后指令变成了一个序对（非列表），形如：
   ((test (op =) (reg b) (const 0)) . <#procedureXXX>)
```

标号相关函数

```
(define (make-label-entry label-name insts)
   (cons label-name insts))
 ;insts是机器里指令序列的一个后缀（从指令序列的开头或中间开始，直到结尾的子序列）
 ;返回值是一个label-entry,形如(L1 (assign...) (test ....) (branch ...)...)
;label-entry的cdr部分就是指令序列的一个后缀



  (define (lookup-label labels label-name)
   ;labels里每个元素都是 label-entry
   (let ((val (assoc label-name labels)))
     (if val
         (cdr val)
         (error "Undefined label -- ASSEMBLE" label-name))))
;返回值就是一个指令序列后缀。表示从label-name开始往后的所有指令
```

创建指令的可执行过程

```
;创建指令所对应的可执行过程。inst是一条指令，形如 (assign n (reg b))
(define (make-execution-procedure inst labels machine pc flag stack ops)
  (cond ((eq? (car inst) 'assign)
         (make-assign inst machine labels ops pc))
        ((eq? (car inst) 'test)
         (make-test inst machine labels ops flag pc))
        ((eq? (car inst) 'branch)
         (make-branch inst machine labels flag pc))
        ((eq? (car inst) 'goto)
         (make-goto inst machine labels pc))
        ((eq? (car inst) 'save)
         (make-save inst machine stack pc))
        ((eq? (car inst) 'restore)
         (make-restore inst machine stack pc))
        ((eq? (car inst) 'perform)
         (make-perform inst machine labels ops pc))
        (else (error "Unknown instruction type -- ASSEMBLE"
                     inst))))
```

创建assign指令的可执行过程

```
(define (make-assign inst machine labels operations pc)
  ;inst 形如:  (assign n (reg b)) n是寄存器名
 (let ((target  ;target是寄存器, 相当于 n
          (get-register machine (assign-reg-name inst)))  ;取得n
         (value-exp (assign-value-exp inst))) ;value-exp 形如 ((reg b))
    (let ((value-proc  ; (value-proc)是寄存器应被赋予的值
             (if (operation-exp? value-exp)
                ;value-exp形如((op rem))则是 operation-exp
                (make-operation-exp
                 value-exp machine labels operations)
                (make-primitive-exp ;此时(car value-exp)形如 (reg b)
                 (car value-exp) machine labels))))
      (lambda ()                          ;assign指令对应的可执行过程
        (set-contents! target (value-proc))
             ;set-contents!设置寄存器target的值为 (value-proc)
        (advance-pc pc)))))  ;程序计数器向前推进即pc.content=(cdr pc.content)
```

assign指令的相关函数

```
 ; assign-instruction 形如:  (assign n (reg b)) n是寄存器名

(define (assign-reg-name assign-instruction)
  (cadr assign-instruction))

(define (assign-value-exp assign-instruction)
  (cddr assign-instruction))


(define (advance-pc pc)
  (set-contents! pc (cdr (get-contents pc))))

;pc里面有状态变量 content，指向指令序列里面某处  (cdr content)就指向再下一条指令。
```

primitive表达式求值相关函数

```
(define (make-primitive-exp exp machine labels) ;返回对基本表达式求值的函数
  ;primitive exp形如: (reg b) 或 (const 3) 或 (label thing-done)
  ;返回值一定是个过程，执行该过程，得到exp的值。如果exp是个标号，则返回该标号代表的指令序列后缀
  (cond ((constant-exp? exp)
         (let ((c (constant-exp-value exp)))
           (lambda () c)))
        ((label-exp? exp)
         (let ((insts
                (lookup-label labels ;lookup-label的返回值是一个指令序列后缀
                              (label-exp-label exp))))
           (lambda () insts))) ;
        ((register-exp? exp)
         (let ((r (get-register machine
                                (register-exp-reg exp))))
           (lambda () (get-contents r)))) ;该lbd返回寄存器的值
        (else (error "Unknown expression type -- ASSEMBLE" exp))))
```

```
(define (get-contents register)
  (register 'get))
```

primitive表达式求值相关函数

```
;primitive exp形如: (reg b) 或 (const 3) 或 (label thing-done)

(define (register-exp? exp) (tagged-list? exp 'reg))
(define (register-exp-reg exp) (cadr exp)) ;取寄存器名
(define (constant-exp? exp) (tagged-list? exp 'const))
(define (constant-exp-value exp) (cadr exp)) ;取常数
(define (label-exp? exp) (tagged-list? exp 'label))
(define (label-exp-label exp) (cadr exp)) ;取标号名
```

operation表达式求值相关函数

```
;operation exp形如: ((op rem) (reg a) (reg b))
(define (make-operation-exp exp machine labels operations)
  (let ((op (lookup-prim (operation-exp-op exp) operations))
         ;op是操作所对应的可执行过程
        (aprocs
         (map (lambda (e)   ;e形如: (reg a)、 (const 3)、 (lable done)
                (make-primitive-exp e machine labels))
              (operation-exp-operands exp))))
    (lambda ()
      (apply op (map (lambda (p) (p)) aprocs)))))

;operations形如:
;(('+ +) ('< <) (initialize-stack  (lambda () (stack 'initialize)))
```

operation表达式求值相关函数

```
;operation exp形如: ((op rem) (reg a) (reg b))

(define (operation-exp? exp)
  (and (pair? exp) (tagged-list? (car exp) 'op)))
(define (operation-exp-op operation-exp)
  (cadr (car operation-exp)))
(define (operation-exp-operands operation-exp)
  (cdr operation-exp))

(define (lookup-prim symbol operations);查找操作对应的可执行过程
;operations形如:
;(('+ +) ('< <) (initialize-stack  (lambda () (stack 'initialize)))
  (let ((val (assoc symbol operations)))
    (if val
        (cadr val) ;返回操作所对应的可执行过程
        (error "Unknown operation -- ASSEMBLE" symbol))))
```

创建test指令的可执行过程

```scheme
(define (make-test inst machine labels operations flag pc)
  (let ((condition (test-condition inst)))
  ;inst 形如 (test (op =) (reg n) (const 1))
    (if (operation-exp? condition)
  ;condition 形如 ((op =) (reg n) (const 1))
        (let ((condition-proc
                (make-operation-exp
                  condition machine labels operations)))
          (lambda ()
            (set-contents! flag (condition-proc))
            (advance-pc pc)))
        (error "Bad TEST instruction -- ASSEMBLE" inst))))

(define (test-condition test-instruction)
  (cdr test-instruction))
```

创建branch指令的可执行过程

```
(define (make-branch inst machine labels flag pc)
  ;branch 形如 (branch (label base-case))
  (let ((dest (branch-dest inst)))
    (if (label-exp? dest)
        (let ((insts ;insts是指令序列后缀
                (lookup-label labels (label-exp-label dest))))
          (lambda ()
            (if (get-contents flag)
                (set-contents! pc insts)
                (advance-pc pc))))
        (error "Bad BRANCH instruction -- ASSEMBLE" inst))))

(define (branch-dest branch-instruction)
  (cadr branch-instruction))
```

创建goto指令的可执行过程

```
(define (make-goto inst machine labels pc)
  ;inst形如(goto (reg continue))或 (goto (label fact-loop))
  (let ((dest (goto-dest inst)))
    (cond ((label-exp? dest)
           (let ((insts
                  (lookup-label labels
                                (label-exp-label dest))))
             (lambda () (set-contents! pc insts))))
          ((register-exp? dest)
           (let ((reg
                  (get-register machine
                                (register-exp-reg dest))))
             (lambda ()
               (set-contents! pc (get-contents reg)))))
          (else (error "Bad GOTO instruction -- ASSEMBLE"
                       inst)))))
(define (goto-dest goto-instruction)
  (cadr goto-instruction))
```

创建save和restore指令的可执行过程

```
(define (make-save inst machine stack pc) ;inst形如 (save n)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (push stack (get-contents reg))
      (advance-pc pc))))

(define (make-restore inst machine stack pc) ;inst形如 (restore n)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (set-contents! reg (pop stack))
      (advance-pc pc))))

(define (stack-inst-reg-name stack-instruction)
  (cadr stack-instruction)) ; stack-instruction形如 (restore n)
```

```
(define (pop stack)
  (stack 'pop))
(define (push stack value)
  ((stack 'push) value))
```

创建perform指令的可执行过程

```
;  inst形如:(perform (op <operation-name>) <input1> ... <inputn>)
(define (make-perform inst machine labels operations pc)
  (let ((action (perform-action inst)))
    (if (operation-exp? action)
        (let ((action-proc
               (make-operation-exp
                action machine labels operations)))
          (lambda ()
            (action-proc)
            (advance-pc pc)))
        (error "Bad PERFORM instruction -- ASSEMBLE" inst))))

(define (perform-action inst) (cdr inst))
```

创建perform指令的可执行过程

```
(define (make-operation-exp exp machine labels operations)
  (let ((op (lookup-prim (operation-exp-op exp) operations))
          ;op是操作所对应的可执行过程
        (aprocs
          (map (lambda (e)   ;e形如: (reg a)、 (const 3)、 (lable
done)
                 (make-primitive-exp e machine labels))
               (operation-exp-operands exp))))
    (lambda ()
      (apply op (map (lambda (p) (p)) aprocs)))))
```

创建并运行斐波那契数列机器

```
(define fib-machine
  (make-machine
   '(a b t n continue val)
   (list (list 'rem remainder) (list '= =)
         (list '< <) (list '+ +) (list '- -))
   '(fib-start
   (assign continue (label fib-done))
 fib-loop
   (test (op <) (reg n) (const 2))
   (branch (label immediate-answer))
   ;; set up to compute Fib(n - 1)
   (save continue)
   (assign continue (label afterfib-n-1))
   (save n)                             ; save old value of n
   (assign n (op -) (reg n) (const 1)); clobber n to n - 1
   (goto (label fib-loop))              ; perform recursive call
```

创建并运行斐波那契数列机器

```
afterfib-n-1                     ; upon return, val contains Fib(n - 1)
   (restore n)
   (restore continue)
   ;; set up to compute Fib(n - 2)
   (assign n (op -) (reg n) (const 2))
   (save continue)
   (assign continue (label afterfib-n-2))
   (save val)                          ; save Fib(n - 1)
   (goto (label fib-loop))
 afterfib-n-2                         ; upon return, val contains Fib(n - 2)
   (assign n (reg val))                ; n now contains Fib(n - 2)
   (restore val)                       ; val now contains Fib(n - 1)
   (restore continue)
   (assign val                         ;  Fib(n - 1) +  Fib(n - 2)
           (op +) (reg val) (reg n))
   (goto (reg continue))              ; return to caller, answer is in val
 immediate-answer
   (assign val (reg n))                ; base case:  Fib(n) = n
   (goto (reg continue))
 fib-done)))
```

创建并运行斐波那契数列机器

```
(set-register-contents! fib-machine 'n 7)
done
(start fib-machine)
done
(get-register-contents fib-machine 'val)
13

(set-register-contents! fib-machine 'n 8)
done
(start fib-machine)
done
(get-register-contents fib-machine 'val)
21
```

监视机器运行的性能

●在栈中可以记录栈操作的次数，以及栈的最大深度
  为此需要增加栈操作，在**make-new-machine**的初始操作列表里增加栈统计操作：

```
(let ((the-ops   ;操作列表
           (list (list 'initialize-stack
                       (lambda () (stack 'initialize)))
                 (list 'print-stack-statistics ;栈统计
                       (lambda () (stack 'print-statistics)))))
```

监视机器运行的性能

- 修改栈的写法以支持栈统计操作

```
(define (make-stack)
  (let ((s '())
        (number-pushes 0) ;总压入次数
        (max-depth 0) ;最大深度
        (current-depth 0)) ;当前深度
    (define (push x)
      (set! s (cons x s))
      (set! number-pushes (+ 1 number-pushes))
      (set! current-depth (+ 1 current-depth))
      (set! max-depth (max current-depth max-depth)))
    (define (pop)
      (if (null? s)
          (error "Empty stack -- POP")
          (let ((top (car s)))
            (set! s (cdr s))
            (set! current-depth (- current-depth 1))
            top)))
```

监视机器运行的性能

```
(define (initialize)
  (set! s '())
  (set! number-pushes 0)
  (set! max-depth 0)
  (set! current-depth 0)
  'done)
(define (print-statistics)
  (newline)
  (display (list 'total-pushes  '= number-pushes
                 'maximum-depth '= max-depth)))
(define (dispatch message)
  (cond ((eq? message 'push) push)
        ((eq? message 'pop) (pop))
        ((eq? message 'initialize) (initialize))
        ((eq? message 'print-statistics)
         (print-statistics))
        (else   (error "Unknown request -- STACK" message))))
dispatch))
```