函数式程序设计

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

第十六讲

用scheme实现的scheme编译器

scheme编译器的结构（与课本同）

- 这是一个scheme程序，包含前述整个寄存器机器的解释器
- 这个程序包含前述整个能够解释scheme语言的寄存器机器，简称scheme机
- 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)

- 编译器读入一段scheme程序A，将其编译生成一段汇编程序，将该程序装入scheme机，启动scheme机的驱动循环。scheme机能读入scheme程序，被读入的scheme程序可以调用A中定义的函数
- scheme机有以下5个寄存器及其他寄存器

    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

scheme编译器的结构（与课本不同）

- 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)

- 编译器读入一段scheme程序，将其编译生成一台寄存器机器scheme机，并启动该scheme机，即能得到读入的scheme程序的运行结果

- scheme机有以下5个寄存器

  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

scheme编译器的结构

- scheme机有一些操作，是通过元循环求值器中的函数实现的，共有7个：

```
lookup-variable-value
set-variable-value!
define-variable!
extend-environment
false?
primitive-procedure?
apply-primitive-procedure
```

- scheme机有一些操作，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，共有3个：

```
make-compiled-procedure
compiled-procedure-env
compiled-procedure-entry
```

- scheme机也有一些操作，是通过scheme基本过程实现的，共有2个：

```
list  cons
```

编译出来的汇编程序的优势

- 作为scheme解释器存在的寄存器机器处理 (f 84 96):

  1) 对运算符f求值前，将运算对象和环境都入栈，求值后再恢复
  2) 对运算符f求值，会将求出的结果先放入val，再移入 proc

- 对 (f 84 96) 进行编译，得到的汇编指令:

```
(assign proc (op lookup-variable-value) (cons f) (reg env))
```

可以避免不必要的栈操作，以及寄存器内容拷贝

- 编译器还可以优化对环境的访问，有时不通过lookup-variable-value就能确定变量的值

编译器核心函数compile

```
(define (compile exp target linkage)
;exp是要编译的表达式 , 编译出来的指令应将表达式的值求，并放入target寄存器，
  (cond ((self-evaluating? exp)
         (compile-self-evaluating exp target linkage))
        ((quoted? exp) (compile-quoted exp target linkage))
        ((variable? exp)
         (compile-variable exp target linkage))
        ((assignment? exp)
         (compile-assignment exp target linkage))
        ((definition? exp)
         (compile-definition exp target linkage))
        ((if? exp) (compile-if exp target linkage))
        ((lambda? exp) (compile-lambda exp target linkage))
        ((begin? exp)
         (compile-sequence (begin-actions exp)
                           target
                           linkage))
;红色函数称为“代码生成器”
```

编译器核心函数compile

```
((cond? exp) (compile (cond->if exp) target linkage))
((application? exp)
 (compile-application exp target linkage))
(else
 (error "Unknown expression type -- COMPILE" exp))))
```

linkage(连接描述符)，描述编译出来的代码执行完后，该如何继续。

1) linkage = 'next : 继续下一条指令
2) linkage = 'return : (goto (reg continue))
3) linkage = 其他，如 'some-label，则linkage被当作标号的名字看待：
   (goto (label some-label))

编译器核心函数compile

```
(compile 5 'val 'next) =>

  (assign val (const 5))

(compile 5 'val 'return) =>

  (assign val (const 5))
  (goto (reg continue))


(compile 5 'val 'somewhere) =>

  (assign val (const 5))
  (goto (label somewhere))
```

指令序列

- 每个“代码生成器”的返回值都是一个“指令序列”
- “指令序列”中包含表达式经过编译后得到的汇编代码
- 复合表达式也对应一个“指令序列”，是通过组合起各成分表达式对应的“指令序列”而成

指令序列

- 一个“指令序列” 包含三部分信息

1） 序列中的指令执行之前，必须被初始化的寄存器集合(简称“需要”的寄存器集合)

2） 指令序列执行中，值有可能会被修改的寄存器集合

3） 序列中的汇编指令(语句)

指令序列相关函数

```
(define (make-instruction-sequence needs modifies statements)
  (list needs modifies statements))
;needs: 要初始化的寄存器集合
;modifies:值可能被修改的寄存器集合
;statements : 汇编指令列表
```

例：

```
(make-instruction-sequence '(env continue) '(val)
 '((assign val
           (op lookup-variable-value) (const x) (reg env))
   (goto (reg continue))))
```

指令序列相关函数

```
;下面的s是指令序列，也可以是标号如 'somewhere
(define (registers-needed s) (if (symbol? s) '() (car s)))
(define (registers-modified s)
  (if (symbol? s) '() (cadr s)))
(define (statements s)
  (if (symbol? s) (list s) (caddr s)))


(define (needs-register? seq reg) ;seq是指令序列
  (memq reg (registers-needed seq)))
(define (modifies-register? seq reg)
  (memq reg (registers-modified seq)))

(define (empty-instruction-sequence) ;空的指令序列
  (make-instruction-sequence '() '() '()))
```

指令序列的组合(一)

- 顺序组合任意多个指令序列，返回新指令序列。不考虑寄存器入栈出栈

```
(define (append-instruction-sequences . seqs)
   (define (append-2-sequences seq1 seq2)
     (make-instruction-sequence
      (list-union (registers-needed seq1)
                  (list-difference (registers-needed seq2)
                                   (registers-modified seq1)))
      (list-union (registers-modified seq1)
                  (registers-modified seq2))
      (append (statements seq1) (statements seq2))))
   (define (append-seq-list seqs)
     (if (null? seqs)
         (empty-instruction-sequence)
         (append-2-sequences (car seqs)
                             (append-seq-list (cdr seqs)))))
   (append-seq-list seqs)) ;seqs中的元素也可以是标号，如'somewhere
```

指令序列的组合(一)

```scheme
(define (list-union s1 s2)
  ;merge s1 & s2 and remove the repeated elements
  (cond ((null? s1) s2)
        ((memq (car s1) s2) (list-union (cdr s1) s2))
        (else (cons (car s1) (list-union (cdr s1) s2)))))


(define (list-difference s1 s2)
 ;remove elements which are in s2 from s1
  (cond ((null? s1) '())
        ((memq (car s1) s2) (list-difference (cdr s1) s2))
        (else (cons (car s1)
                    (list-difference (cdr s1) s2)))))
```

指令序列的组合(二)

- 顺序组合两个指令序列，返回新指令序列。要考虑寄存器入栈出栈

假设指令序列**seq1**和**seq2**中都访问了 寄存器**reg1**和**reg2**,则根据**reg1**和**reg2**在两个指令序列中的使用情况，用**preserving**组合两个指令序列：

$(\texttt{preserving}\ (\texttt{list}\ <reg_1>\ <reg_2>)\ <seq_1>\ <seq_2>)$

结果可能是一下四种情况之一：

⟨seq1⟩	(save ⟨reg1⟩)	(save ⟨reg2⟩)	(save ⟨reg2⟩)
⟨seq2⟩	⟨seq1⟩	⟨seq1⟩	(save ⟨reg1⟩)
	(restore ⟨reg1⟩)	(restore ⟨reg2⟩)	⟨seq1⟩
	⟨seq2⟩	⟨seq2⟩	(restore ⟨reg1⟩)
			(restore ⟨reg2⟩)
			⟨seq2⟩

指令序列的组合(二)

```
(preserving (list <reg1> <reg2>) <seq1> <seq2>)
```

<seq$_1$>	(save <reg$_1$>)	(save <reg$_2$>)	(save <reg$_2$>)
<seq$_2$>	<seq$_1$>	<seq1>	(save <reg$_1$>)
	(restore <reg$_1$>)	(restore <reg$_2$>)	<seq$_1$>
	<seq$_2$>	<seq2>	(restore <reg$_1$>)
			(restore <reg$_2$>)
			<seq$_2$>

1) reg1和reg2都不在(registers-needed seq2)中，或都不在(registered-modified seq1)中
2) reg1同时在(registers-needed seq2)和(registered-modified seq1)中。reg2不是如此
3) reg2同时在(registers-needed seq2)和(registered-modified seq1)中。reg1不是如此
4) reg1和reg2都同时在(registers-needed seq2)和(registered-modified seq1)中

指令序列的组合(二)

```
(define (preserving regs seq1 seq2)  ;regs是可能需要保存到栈里的寄存器集合
  (if (null? regs)
      (append-instruction-sequences seq1 seq2)
      (let ((first-reg (car regs)))
        (if (and (needs-register? seq2 first-reg)
                 (modifies-register? seq1 first-reg))
            (preserving (cdr regs)
             (make-instruction-sequence
              (list-union (list first-reg)
                          (registers-needed seq1))
              (list-difference (registers-modified seq1)
                               (list first-reg))
              (append '((save ,first-reg))
; ,first-reg表示对表达式first-reg求值的结果,Racket不支持
                      (statements seq1)
                      '((restore ,first-reg))))
             seq2)
            (preserving (cdr regs) seq1 seq2)))))
```

编译简单表达式

```
(define (compile-self-evaluating exp target linkage)
  (end-with-linkage linkage ; end-with-linkage 处理后续如何执行(连接代码)
   (make-instruction-sequence '() (list target)
    '((assign ,target (const ,exp))))))

(define (compile-quoted exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '() (list target)
    '((assign ,target (const ,(text-of-quotation exp)))))))

(define (compile-variable exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '(env) (list target)
    '((assign ,target
              (op lookup-variable-value)
              (const ,exp)
              (reg env))))))
```

编译连接代码

```
(define (end-with-linkage linkage instruction-sequence)
;返回一个指令序列
  (preserving '(continue)
;需要关注coutinue是否可能被instruction-sequence修改且在第二个指令序列中用到
   instruction-sequence
   (compile-linkage linkage)))


(define (compile-linkage linkage) ;生成链接后续执行代码的指令列表
  (cond ((eq? linkage 'return)
         (make-instruction-sequence '(continue) '()
          '((goto (reg continue)))))
        ((eq? linkage 'next)
         (empty-instruction-sequence))
        (else
         (make-instruction-sequence '() '()
          '((goto (label ,linkage)))))))
```

编译简单表达式

```
(compile 5 'val 'next) =>
'(() (val) ((assign val (const 5))))

(compile ''(tom jack) 'val 'next) =>
'(() (val) ((assign val (const (tom jack)))))

(compile '(define x 13) 'val 'next) =>
       ......(略)

(compile 'x 'val 'next) =>
'((env) (val)
  ((assign val (op lookup-variable-value) (const x) (reg env))))

(compile ''x 'val 'next) =>
'(() (val) ((assign val (const x))))
```

编译define表达式

```
(define (compile-definition exp target linkage)
  (let ((var (definition-variable exp))
        (get-value-code
         (compile (definition-value exp) 'val 'next)))
; 变量值求出后放入val，接下来用val值对变量名赋值，因此linkage是 'next
    (end-with-linkage linkage
     (preserving '(env) ;env可能在get-value-code中被修改。为何没 val?
      get-value-code
      (make-instruction-sequence '(env val) (list target)
       '((perform (op define-variable!)
                  (const ,var) ;变量名
                  (reg val) ;变量值
                  (reg env)) ;变量所在环境
         (assign ,target (const ok))))))))
```

编译赋值表达式

```
(define (compile-assignment exp target linkage)
  (let ((var (assignment-variable exp))
        (get-value-code
         (compile (assignment-value exp) 'val 'next)))
    (end-with-linkage linkage
     (preserving '(env)
      get-value-code
      (make-instruction-sequence '(env val) (list target)
       '((perform (op set-variable-value!)
                  (const ,var) ;变量名
                  (reg val) ;变量值
                  (reg env)) ;变量所在环境
         (assign ,target (const ok))))))))
```

编译赋值表达式和define表达式

```
(compile '(define x 13) 'val 'next) =>

'((env) (val)
  ((assign val (const 13))
   (perform (op define-variable!) (const x) (reg val) (reg env))
   (assign val (const ok))))

(compile '(set! x 243) 'val 'next)  =>
'((env) (val)
  ((assign val (const 243))
   (perform (op set-variable-value!) (const x) (reg val)
                                     (reg env))
   (assign val (const ok))))
```

编译if表达式

- 对if表达式的编译结果形式如下：

```
<compilation of predicate, target val, linkage next>
 (test (op false?) (reg val))
 (branch (label false-branch))
true-branch
 <compilation of consequent with given target and given
linkage or after-if> ;如果given linkage是'next,则改为after-if
false-branch
 <compilation of alternative with given target and linkage>
after-if
```

given target and linkage 是compile-if 的参数

```scheme
(define (compile-if exp target linkage)
  (let ((t-branch (make-label 'true-branch)) ;make-label生成无重复标号
        (f-branch (make-label 'false-branch))
        (after-if (make-label 'after-if)))
   (let ((consequent-linkage
           (if (eq? linkage 'next) after-if linkage)))
     (let ((p-code (compile (if-predicate exp) 'val 'next))
            (c-code
             (compile
              (if-consequent exp) target consequent-linkage))
            (a-code
             (compile (if-alternative exp) target linkage)))
       (preserving '(env continue)
        p-code
        (append-instruction-sequences
         (make-instruction-sequence '(val) '()
          '((test (op false?) (reg val))
            (branch (label ,f-branch))))
         (parallel-instruction-sequences
          (append-instruction-sequences t-branch c-code)
          (append-instruction-sequences f-branch a-code))
        after-if))))))
```

此处是整个程序中，三处使得linkage可能是标号的地方之一。另二处在compile-procedure-call和compile-lambda

指令序列的组合(三)

- 合并两个指令序列，只有其中一个会被执行：

```
(define (parallel-instruction-sequences seq1 seq2)
  (make-instruction-sequence
   (list-union (registers-needed seq1)
               (registers-needed seq2))
   (list-union (registers-modified seq1)
               (registers-modified seq2))
   (append (statements seq1) (statements seq2))))
```

seq1中的代码决定了seq1和seq2只有一个会被执行(seq1代码的末尾是跳转到别处)

编译if表达式

```
(compile '(if 5 45 50) 'val 'next) =>

'(()
  (val)
  ((assign val (const 5))
   (test (op false?) (reg val))
   (branch (label false-branch2))
   true-branch1
   (assign val (const 45))
   (goto (label after-if3))
   false-branch2
   (assign val (const 50))
   after-if3))
```

标号相关函数

```
(define label-counter 0)

(define (new-label-number)
  (set! label-counter (+ 1 label-counter))
  label-counter)

(define (make-label name)
  (string->symbol
    (string-append (symbol->string name)
                   (number->string (new-label-number)))))
```

编译表达式序列

```
(define (compile-sequence seq target linkage)
  (if (last-exp? seq)
      (compile (first-exp seq) target linkage)
      (preserving '(env continue) ;为何不要val?
       (compile (first-exp seq) target 'next)
       (compile-sequence (rest-exps seq) target linkage))))
```

编译lambda表达式

● lambda表达式编译出来的代码中，应该生成一个“译后过程”(compiled-procedure)，并将该“译后过程” 放入目标寄存器。

● lambda编译出来的代码中，还应该包括lambda表达式函数体部分的编译结果。但是函数体对应的代码，在整个lambda表达式对应的代码序列中，只是出现，不应该被执行到。

● 编译结果如下：

<construct compiled procedure object and assign it to target register>

<code for given linkage> or (goto (label after-lambda))

<compilation of procedure body> ;此部分不应在本段代码中被执行

after-lambda

● 若compile-lambda被调用时，linkage是'return，则

<code for given linkage> 是 (goto (reg continue))

● 若compile-lambda被调用时，linkage是标号'somewhere，则

<code for given linkage> 是 (goto (label somewhere))

● 若compile-lambda被调用时，linkage是'next，则

<code for given linkage> 是(goto (label after-lambda))

编译lambda表达式

```
(compile '(lambda (x) x) 'val 'next)  =>

'((env)
  (val)
  ((assign val (op make-compiled-procedure) (label entry1) (reg env))
   (goto (label after-lambda2))
  entry1
   (assign env (op compiled-procedure-env) (reg proc))
   (assign env (op extend-environment) (const (x)) (reg argl) (reg env))
   (assign val (op lookup-variable-value) (const x) (reg env))
   (goto (reg continue))
  after-lambda2))
```

编译函数定义表达式:

(compile '(define (f x) x) 'val 'next) 生成的代码会往环境中加入f到一个“译后过程”的约束

编译lambda表达式

```
(define (compile-lambda exp target linkage);exp是lambda表达式
   (let ((proc-entry (make-label 'entry))
         (after-lambda (make-label 'after-lambda)))
     (let ((lambda-linkage
              (if (eq? linkage 'next) after-lambda linkage)))
       (append-instruction-sequences
        (tack-on-instruction-sequence
         (end-with-linkage lambda-linkage
          (make-instruction-sequence '(env) (list target)
           '((assign ,target
                     (op make-compiled-procedure)
                     (label ,proc-entry)
                     (reg env)))))
         (compile-lambda-body exp proc-entry))
        after-lambda))))
```

指令序列的组合(三)

- 合并两个指令序列，第一个序列一定被执行，第二个序列一定不被执行

```
(define (tack-on-instruction-sequence seq body-seq)
  (make-instruction-sequence
   (registers-needed seq)
   (registers-modified seq)
   (append (statements seq) (statements body-seq))))
```

;由于第二个序列一定不被执行（第一个指令序列的末尾一定是跳转到别处），因此整个序列的寄存器使用情况就取决于第一个序列

生成“译后过程”

- “译后过程”内包含一个函数的体的起始位置（标号），以及该函数体被调用时所对应的环境

```
(define (make-compiled-procedure entry env)
  (list 'compiled-procedure entry env))

(define (compiled-procedure? proc)
  (tagged-list? proc 'compiled-procedure))

(define (compiled-procedure-entry c-proc) (cadr c-proc))

(define (compiled-procedure-env c-proc) (caddr c-proc))
```

编译函数体

```
(define (compile-lambda-body exp proc-entry)
;proc-entry是个标号
  (let ((formals (lambda-parameters exp)))
    (append-instruction-sequences
     (make-instruction-sequence '(env proc argl) '(env)
;函数体的代码要求执行前env放着环境，proc放着译后过程，argl放着实参值表
       '(,proc-entry
         (assign env (op compiled-procedure-env) (reg proc))
         (assign env
                 (op extend-environment)
                 (const ,formals)
                 (reg argl)
                 (reg env))))
     (compile-sequence (lambda-body exp) 'val 'return))))
;整个程序中唯一使用 'return 作为linkage参数的地方
;函数体执行结束一定是 (goto (reg continue))
```

组合式的编译

- 对于函数调用表达式，编译结果如下：

<compilation of operator, target proc, linkage next>
<evaluate operands and construct argument list in argl>
<compilation of procedure call with given target and linkage>

- 需要保留与恢复**`env`**,**`proc`**和**`argl`**

- 整个编译器中唯一出现的以非`'val`作为 `target`调用代码生成器的地方

组合式的编译

```
(define (compile-application exp target linkage)
  (let ((proc-code (compile (operator exp) 'proc 'next))
;整个程序中，唯一 target不是 val 的地方。proc用来放“译后过程”
        (operand-codes
         (map (lambda (operand) (compile operand 'val 'next))
              (operands exp))))
    (preserving '(env continue);求值运算符会改变env和continue
     proc-code
     (preserving '(proc continue)
      (construct-arglist operand-codes)
      (compile-procedure-call target linkage)))))
```

construct-arglist 生成构造实参表的代码

compile-procedure-call 生成调用过程的代码

组合式的编译

- **`construct-arglist`** 生成构造实参表的代码：

  <compilation of last operand, targeted to val>
  (assign argl (op list) (reg val))
  <compilation of next operand, targeted to val>
  (assign argl (op cons) (reg val) (reg argl))
  ...
  <compilation of first operand, targeted to val>
  (assign argl (op cons) (reg val) (reg argl))

- 除了**`last operand`**以外，在每个**`operand`**求值前后都可能要保留和恢复**`argl`**

- 除了**`first operand`**以外，在每个**`operand`**求值的前后都可能要保留和恢复 **`env`**

组合式的编译

```
(define (construct-arglist operand-codes)
;operand-codes是个列表，每个元素都是对一个参数求值的指令序列
  (let ((operand-codes (reverse operand-codes))) ;倒过来
    (if (null? operand-codes)
        (make-instruction-sequence '() '(argl)
         '((assign argl (const ()))))
        (let ((code-to-get-last-arg
               (append-instruction-sequences
                (car operand-codes)
                (make-instruction-sequence '(val) '(argl)
                 '((assign argl (op list) (reg val)))))))
          (if (null? (cdr operand-codes))
              code-to-get-last-arg
              (preserving '(env)
               code-to-get-last-arg
               (code-to-get-rest-args
                (cdr operand-codes))))))))
```

组合式的编译

```
(define (code-to-get-rest-args operand-codes)
  (let ((code-for-next-arg
         (preserving '(argl)
          (car operand-codes)
          (make-instruction-sequence '(val argl) '(argl)
           '((assign argl
              (op cons) (reg val) (reg argl)))))))
    (if (null? (cdr operand-codes))
        code-for-next-arg
        (preserving '(env)
         code-for-next-arg
         (code-to-get-rest-args (cdr operand-codes))))))
```

```scheme
(compile '(* 4 5 6) 'val 'next) =>
'((env)
  (env proc argl continue val)
  ((assign proc (op lookup-variable-value) (const *) (reg env))
   (assign val (const 6))
   (assign argl (op list) (reg val))
   (assign val (const 5))
   (assign argl (op cons) (reg val) (reg argl))
   (assign val (const 4))
   (assign argl (op cons) (reg val) (reg argl))
   (test (op primitive-procedure?) (reg proc))
   (branch (label primitive-branch1))
  compiled-branch2
   (assign continue (label after-call3))
   (assign val (op compiled-procedure-entry) (reg proc))
   (goto (reg val))
  primitive-branch1
   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  after-call3))
```

过程应用

在运算符和参数求值结束后，调用函数的代码应有如下形式：

```
 (test (op primitive-procedure?) (reg proc)) ;此处proc是函数对象，或译后过程
 (branch (label primitive-branch))
compiled-branch
 <code to apply compiled procedure with given target and appropriate
 linkage> ;此时proc是译后过程，此处的代码要跳过 primitive-branch
primitive-branch
 (assign <target>
         (op apply-primitive-procedure)
         (reg proc)   ;此处proc是函数对象
         (reg argl))
 <linkage>
after-call
```

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

过程应用

- 编译在运算符和参数求值结束后，调用函数的代码：

```
(define (compile-procedure-call target linkage)
  ;仅被compile-application调用
  (let ((primitive-branch (make-label 'primitive-branch))
        (compiled-branch (make-label 'compiled-branch))
        (after-call (make-label 'after-call)))
    (let ((compiled-linkage ;使得调用compile-proc-appl时，linkage不可能为'next
           (if (eq? linkage 'next) after-call linkage)))
      (append-instruction-sequences ;不用考虑寄存器出入栈的指令序列拼接
       (make-instruction-sequence '(proc) '()
        '((test (op primitive-procedure?) (reg proc))
          (branch (label ,primitive-branch))))
       (parallel-instruction-sequences ;两个指令序列拼接，只会执行一个
        (append-instruction-sequences
         compiled-branch
         (compile-proc-appl target compiled-linkage))
```

;compile-proc-appl生成调用复合函数的指令。用compiled-linkage而非linkage是因为如果linkage为'next，则调用复合函数的代码执行完后，应该跳过调用基本函数的代码，转到after-call.

过程应用

```
(append-instruction-sequences
 primitive-branch
 (end-with-linkage linkage
  (make-instruction-sequence '(proc argl)
                             (list target)
    '((assign ,target
              (op apply-primitive-procedure)
              (reg proc)
              (reg argl)))))))
after-call))))
```

compile-proc-appl生成调用复合函数的指令

- 一个编译好的函数（由**compile-lambda**构造）,有一个入口点，即标号，表明了这个函数开始执行的位置。位于这个入口点的代码能够计算出一个结果，将其放入**val**，然后通过**(goto (reg continue))**返回。

- 如果**compile-proc-appl**的**linkage**参数是个标号，则其生成的结果似乎应该是：

```
  (assign continue (label proc-return))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
proc-return
  (assign <target> (reg val)); included if target is not val
  (goto (label <linkage>))    ; linkage code
```

compile-proc-appl生成调用复合函数的指令

- 如果compile-proc-appl的linkage参数是'return ，则生成的结果似乎应该是：

```
    (save continue)
    (assign continue (label proc-return))
    (assign val (op compiled-procedure-entry) (reg proc))
    (goto (reg val))
  proc-return
    (assign <target> (reg val))   ; included if target is not val
    (restore continue)
    (goto (reg continue))     ; linkage code
```

- compile-proc-appl的linkage参数不可能是'next

compile-proc-appl生成调用复合函数的指令

- 若target不是val的情况下，则生成的指令如上。target不是val的情况只有一个，就是：

```
(define (compile-application exp target linkage)
   (let ((proc-code (compile (operator exp) 'proc 'next))
;整个程序中，唯一 target不是 val 的地方
```

- 在target是 val的情况下，生成的指令可以简化,下面的指令是不需要的：

```
(assign <target> (reg val))  ; included if target is not val
```

因此指令应为：

```
<set up continue for linkage> ;(goto (reg val))后据此返回
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

compile-proc-appl生成调用复合函数的指令

- target = 'val && linkage = 标号 ，指令为:

```
(assign continue <label <linkage>)
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

- target = 'val && linkage = 'return ，指令为:

```
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

不需要在栈中保存 continue，直接跳转，实现了尾递归

- target != 'val && linkage = 'return

报错。target != 'val 则target = 'proc,此时必有 linkage = 'next

compile-proc-appl生成调用复合函数的指令

```
(define (compile-proc-appl target linkage); linkage不可能是'next
 (cond ((and (eq? target 'val) (not (eq? linkage 'return)))
          (make-instruction-sequence '(proc) all-regs ;所有寄存器都可能修改
            '((assign continue (label ,linkage))
              (assign val (op compiled-procedure-entry)
                          (reg proc))
              (goto (reg val)))))
         ((and (not (eq? target 'val)) ;此时target是 'proc
              (not (eq? linkage 'return)))
          (let ((proc-return (make-label 'proc-return)))
            (make-instruction-sequence '(proc) all-regs
             '((assign continue (label ,proc-return))
               (assign val (op compiled-procedure-entry)
                           (reg proc))
               (goto (reg val))
               ,proc-return
               (assign ,target (reg val))
               (goto (label ,linkage))))))
```

compile-proc-appl生成调用复合函数的指令

```
((and (eq? target 'val) (eq? linkage 'return))
 (make-instruction-sequence '(proc continue) all-regs
  '((assign val (op compiled-procedure-entry)
                (reg proc))
    (goto (reg val)))))
((and (not (eq? target 'val)) (eq? linkage 'return))
 (error "return linkage, target not val -- COMPILE"
        target))))
```

(compile '(some-func 5 6) 'val 'next) =>

```
'((env)
  (env proc argl continue val)
  ((assign proc (op lookup-variable-value) (const some-func) (reg env))
   (assign val (const 6))
   (assign argl (op list) (reg val))
   (assign val (const 5))
   (assign argl (op cons) (reg val) (reg argl))
   (test (op primitive-procedure?) (reg proc))
   (branch (label primitive-branch1))
   compiled-branch2
   (assign continue (label after-call3))
   (assign val (op compiled-procedure-entry) (reg proc))
   (goto (reg val))
   primitive-branch1
   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
   after-call3))
```

编译代码实例

```
(compile
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n)))
 'val
 'next)
```

编译出来的代码框架：

```
  <save env if modified by code to compute value>
  <compilation of definition value, target val, linkage next>
  <restore env if saved above>
  (perform (op define-variable!)
           (const factorial)
           (reg val)
           (reg env))
  (assign val (const ok))
```

编译define表达式

```
(define (compile-definition exp target linkage)
  (let ((var (definition-variable exp))
        (get-value-code
         (compile (definition-value exp) 'val 'next)))
    (end-with-linkage linkage
     (preserving '(env)
      get-value-code
      (make-instruction-sequence '(env val) (list target)
       '((perform (op define-variable!)
                  (const ,var)
                  (reg val)
                  (reg env))
         (assign ,target (const ok))))))))
```

编译lambda表达式

```
(define (compile-lambda exp target linkage) ;exp是lambda表达式
  (let ((proc-entry (make-label 'entry))
        (after-lambda (make-label 'after-lambda)))
    (let ((lambda-linkage
           (if (eq? linkage 'next) after-lambda linkage)))
      (append-instruction-sequences
       (tack-on-instruction-sequence
        (end-with-linkage lambda-linkage
         (make-instruction-sequence '(env) (list target)
          '((assign ,target
                    (op make-compiled-procedure)
                    (label ,proc-entry)
                    (reg env)))))
        (compile-lambda-body exp proc-entry))
      after-lambda))))
```

框架细化

```
 <compilation of definition value, target val, linkage next>  =>
 (assign val (op make-compiled-procedure)
             (label entry2)
             (reg env))
  (goto (label after-lambda1))
entry2
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env (op extend-environment)
              (const (n))
              (reg argl)
              (reg env))
 <compilation of procedure body>
  ;由compile-lambda-body编译, target= 'val, linkage = 'return
after-lambda1
  (perform (op define-variable!)
           (const factorial)
           (reg val)
           (reg env))
  (assign val (const ok))
```

编译过程体

编译 (if (= n 1) 1 (* (factorial (- n 1)) n))
target = 'val, linkage = 'return

```
   <save continue, env if modified by predicate and needed by branches>
   <compilation of predicate, target val, linkage next>
   <restore continue, env if saved above>
   (test (op false?) (reg val))
   (branch (label false-branch4))
true-branch5
   <compilation of true branch, target val, linkage return>
false-branch4
   <compilation of false branch, target val, linkage return>
after-if3
```

<compilation of predicate, target val, linkage next> 的结果

谓词:　 (= n 1)

```
(assign proc
            (op lookup-variable-value) (const =) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch17))
compiled-branch16
  (assign continue (label after-call15))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch17
  (assign val (op apply-primitive-procedure)
              (reg proc)
              (reg argl))
after-call15
```

*<compilation of true branch, target val, linkage return>*的结果

真分支： 1

```
(assign val (const 1))
(goto (reg continue))
```

实例的编译结果

```
;; construct the procedure and skip over code for the procedure body
  (assign val
          (op make-compiled-procedure) (label entry2) (reg env))
  (goto (label after-lambda1)) ;after-lambda1会在环境中加入变量factorial的定义

entry2      ; calls to factorial will enter here
  (assign env (op compiled-procedure-env) (reg proc)) ;proc是译后过程
  (assign env
          (op extend-environment) (const (n)) (reg argl) (reg env))
;; begin actual procedure body
  (save continue)
  (save env)
;; compute (= n 1)
  (assign proc (op lookup-variable-value) (const =) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
```

实例的编译结果

```
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch17))
compiled-branch16
  (assign continue (label after-call15))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch17
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call15   ; val now contains result of (= n 1)
  (restore env)
  (restore continue)
  (test (op false?) (reg val))
  (branch (label false-branch4))
true-branch5  ; return 1
  (assign val (const 1))
  (goto (reg continue))
```

实例的编译结果

```
false-branch4
;; compute and return (* (factorial (- n 1)) n)
  (assign proc (op lookup-variable-value) (const *) (reg env))
  (save continue)
  (save proc)    ; save * procedure
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op list) (reg val))
  (save argl)    ; save partial argument list for *

;; compute (factorial (- n 1)), which is the other argument for *
  (assign proc
          (op lookup-variable-value) (const factorial) (reg env))
  (save proc)   ; save factorial procedure
```

实例的编译结果

```
;; compute (- n 1), which is the argument for factorial
   (assign proc (op lookup-variable-value) (const -) (reg env))
   (assign val (const 1))
   (assign argl (op list) (reg val))
   (assign val (op lookup-variable-value) (const n) (reg env))
   (assign argl (op cons) (reg val) (reg argl))
   (test (op primitive-procedure?) (reg proc))
   (branch (label primitive-branch8))
compiled-branch7
   (assign continue (label after-call6))
   (assign val (op compiled-procedure-entry) (reg proc))
   (goto (reg val))
primitive-branch8
   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call6   ; val now contains result of (- n 1)
   (assign argl (op list) (reg val))
   (restore proc) ; restore factorial
```

实例的编译结果

```
;; apply factorial
   (test (op primitive-procedure?) (reg proc))
   (branch (label primitive-branch11))
compiled-branch10
   (assign continue (label after-call9))
   (assign val (op compiled-procedure-entry) (reg proc))
   (goto (reg val))
primitive-branch11
   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))

after-call9       ; val now contains result of (factorial (- n 1))
   (restore argl) ; restore partial argument list for *
   (assign argl (op cons) (reg val) (reg argl))
   (restore proc) ; restore *
   (restore continue)
```

实例的编译结果

```
;; apply * and return its value
   (test (op primitive-procedure?) (reg proc))
   (branch (label primitive-branch14))
compiled-branch13
;; note that a compound procedure here is called tail-recursively
   (assign val (op compiled-procedure-entry) (reg proc))
   (goto (reg val))
primitive-branch14
   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
   (goto (reg continue))
after-call12
after-if3
after-lambda1
;; assign the procedure to the variable factorial
   (perform
    (op define-variable!) (const factorial) (reg val) (reg env))
   (assign val (const ok)) ;factorial被约束到一个译后过程
```

编译代码与scheme机器的互联

运行方式:

```
(compile-and-go
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n))))
 ;;; EC-Eval value:
ok
 ;;; EC-Eval input:
(factorial 5)
;;; EC-Eval value:
120
```

compile-and-go 编译出一段代码，将其加入scheme机器，然后启动scheme机器。scheme机器读入scheme表达式，输出结果。 读入的scheme表达式可以调用factorial

修改scheme机器的apply-dispatch以能调用编译的代码

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (test (op compiled-procedure?) (reg proc))
  (branch (label compiled-apply))
  (goto (label unknown-procedure-type))
compiled-apply
  (restore continue) ;编译代码执行前要求返回地址在continue
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
```

修改scheme机器的开始代码

```
   (branch (label external-entry))
; 新scheme机器起点 , branches if flag is set
read-eval-print-loop ;原scheme机器起始点
   (perform (op initialize-stack))
 ...

external-entry
   (perform (op initialize-stack))
   (assign env (op get-global-environment))
   (assign continue (label print-result))
   (goto (reg val))
```

compile-and-go

```
(define (compile-and-go expression)
  (let ((instructions
          (assemble (statements  ;assemble由文本形式的指令列表生成scheme机器的指令列表（带可执行过程的指令的列表，该可执行过程是scheme可执行过程）
                      (compile expression 'val 'return))
                     eceval))) ;eceval是个scheme机器
    (set! the-global-environment (setup-environment))
    (set-register-contents! eceval 'val instructions)
    (set-register-contents! eceval 'flag true)
    (start eceval)))
```

;val放着编译出来的代码的起始地址