5. 寄存器机器里的计算(3)

本节讨论编译，实现从 Scheme 到寄存器机器语言的映射

- 编译和解释的概念和问题
- 编译器结构
- 基本表达式和组合式的编译
- 代码序列的组合
- 编译代码与求值器的互连

编译和解释

- 显式控制求值器是一部寄存器机器，其控制器能解释 Scheme 程序
- 下面要研究的是如何在控制器不是 Scheme 解释器的寄存器机器上运行 Scheme程序
- 显式控制求值器是解释Scheme程序的通用机器
  - 其控制器与数据通路配合，可执行任何计算过程
  - 其数据通路也是通用的
- 商品计算机都是通用寄存器机器，基于一组寄存器和一组操作（高效方便的数据通路），其控制器是一个寄存器机器语言（与寄存器机器语言类似，本机语言）解释器。程序是使用机器的数据通路的指令序列
- 完全可以把显式控制求值器的指令序列看作一台通用计算机的机器语言程序，而不看作一部特定解释器机器的控制器
- 要在寄存器机器上运行一种高级语言程序，需要在高级语言和寄存器机器语言间架一座桥梁。两种策略：一种称为解释，一种称为编译

编译和解释

- 显式控制求值器采用的是解释策略：
  - 用本机语言写一个解释器，它配置该机器使之能执行源语言（与本机语言完全不同）的程序
  - 源语言的基本过程采用本机语言写出的子程序库实现。被解释程序（源程序）用数据结构表示
  - 解释器遍历表示程序的数据结构，分析源程序的情况，模拟其行为（需要利用库里的子程序）
- 另一种策略是编译：
  - 针对给定源语言和给定机器的编译器，将源程序翻译为本机语言的等价程序（目标程序）
  - 然后就可以让目标程序自己运行了
- 下面实现的编译器把 Scheme 写的程序（源程序）翻译为显式控制求值器的数据通路能执行的指令序列（目标程序）

编译和解释

- 与解释相比，编译能大大提高程序执行效率，下面会说明一些情况
- 另一方面，解释器能为程序的开发和排错提供强大帮助
  - 被执行源代码在运行期间可用，可以方便地检查和修改
  - 由于所有基本操作的库都在，因此可以支持在修改程序和排错过程中构造新程序，并将新程序随时加入系统
- 编译和解释优势互补，现代开发环境推崇混合策略
  - Lisp解释器一般都允许解释性程序和编译性程序相互调用，这样，已完成的部分可以用编译的方式，取得效率优势；正在开发而不断变化的部分通过解释执行
  - 下面做完编译器后，还要说明如何将它与解释器连接，形成一个集成的编译器-解释器开发环境
- 本书后面部分讨论如何做这个编译器，给出了大量代码和简单解释

  我们适当地看一些部分，首先是整体想法（目标程序的结构）

编译器的结构

- 编译器分析表达式的机制与解释器类似
- 为使编译代码能与解释代码互连，生成代码遵循同样寄存器规则：
  - 执行环境在env
  - 实参表在argl里积累
  - 被应用过程在proc
  - 过程值通过val返回
  - 过程将要用的返回地址放在 continue
- 从一个源程序生成的目标程序在执行中所做的寄存器操作，本质上就是解释器求值同一个源程序时执行的操作
- 编译器在技术上与前面分析型求值器类似，像解释器一样遍历表达式
  - 遇到解释器求值表达式时执行一条寄存器指令的情况，它将这条指令放到一个序列里。最终得到的指令序列就是所需目标代码

编译器的结构

- 表达式分类。以求值（f 84 96）为例：
  - 解释器每次求值表达式时都要做表达式分类（发现是过程应用），并检查表示表达式的表是否结束（弄清有两个运算对象）
  - 编译器只在编译源程序并生成指令序列时做一次表达式分析

    产生的目标代码里只有对运算符和运算对象求值的指令，以及将过程（保存在 proc）应用于实参（保存在 argl）的指令
- 一般和特殊处理
  - 解释器必须考虑处理所有表达式
  - 编译结果代码完成特定表达式
  - 以保存寄存器为例：

    解释器要准备处理所有可能，求值子表达式前必须把后来可能用的所有寄存器进栈（不知道子表达式会做什么）

    编译器可以根据具体表达式的情况，只生成必要的栈操作

编译和解释

- 考虑对 (f 84 96) 的处理：
  - 解释器求值 f 前后保存/恢复运算对象和环境寄存器 (都可能有用)，最后把 val 的值移到 proc （先放到 val 再移到proc）
  - 由于运算符就是 f，求值由机器操作 lookup-variable-value 完成，不修改寄存器。编译代码只需要一条指令做运算符求值：

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

    不需要保存和恢复，值直接赋给 proc
- 优化环境访问
  - 通过分析代码，许多情况时候可以确定特定值所在框架，然后直接访问该框架，不需要用 lookup-variable-value 搜索
  - 直接处理某些基本操作，而不通过通用的apply（练习5.38）
- 下面不准备特别强调各种优化，主要目标是在一个简化（但仍然很有意思）的上下文中展示编译过程的各种情况

编译器结构

- 工作方式与分析求值器类似，但不产生执行过程，生成能在寄存器机器上运行的指令序列
- 操作表达式的语法过程就是元循环求值器用的Scheme过程。显式控制求值器假定语法过程是寄存器机器操作
- compile做最高层分派（对应eval或analyze），按表达式语法类型指派特定代码生成器

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

编译器结构

- compile 及其调用的代码生成器都另有两个参数：target 所生成代码段将表达式的值存入的寄存器；连接描述符 linkage 描述表达式的编译结果代码完成后如何继续，有几种可能动作：
  - 继续序列里的下一条指令（连接描述符为 next）
  - 从被编译的过程返回（连接描述符为 return）
  - 跳到一个命名入口点（以指定标号作为连接描述符）
- 例：以 val 寄存器为目标以 next 为连接描述符编译表达式5产生：

  (assign val (const 5))

  接着执行下一指令。以 return 作为连接描述符，则生成：

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

  要求从一个过程返回

编译器结构

- 代码生成器返回由被编译表达式生成的目标代码指令序列。复合表达式的代码是通过组合成分表达式的代码建立的
- 组合指令序列的最简单方式是调用 append-instruction-sequences 过程。它直接顺序拼接任意数目的参数指令序列，返回组合指令序列。如果 $<seq_1>$ 和 $<seq_2>$ 都是指令序列

  (append-instruction-sequences $<seq_1>$ $<seq_2>$)

  产生的序列是

  $<seq_1>$
  $<seq_2>$
- 如果执行中有可能需要保存寄存器，就用preserving实现精细组合。过程的参数是寄存器集合 R 和两个顺序执行的指令序列$<seq_1>$,$<seq_2>$。preserving 保证如果 R 中任一寄存器 s 的值在$<seq_2>$里用，其值就不会受到$<seq_1>$的影响。如$<seq_1>$修改 s 而 $<seq_2>$ 需要 s 的原值，preserving 会在$<seq_1>$外面包上对 s 的 save 和restore。没这种情况时就简单连接

编译器结构

- (preserving (list $<reg_1>$ $<reg_2>$) $<seq_1>$ $<seq_2>$) 可能产生四种结果：

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

- 用preserving组合指令序列可避免不必要的堆栈操作，也把生成save和restore的细节封装在preserving内部，与代码生成的具体情况分离。所有代码生成器都不显式生成save和restore
- 如果用表表示指令序列，append-instruction-sequences就是append。但preserving会很复杂（要分析指令序列确定寄存器使用情况），也很低效（需要分析每个指令序列，即使原就是由preserving构造的，已经分析过）。为避免重复分析，给每个指令序列关联寄存器使用信息。简单指令序列可直接确定，组合指令时推导出组合的使用信息

指令序列的构造

- 这样做，指令序列要包含三部分信息
  - 序列中指令执行前必须初始化的寄存器集合（使用的寄存器）
  - 这一序列执行时修改的寄存器的集合
  - 序列里的实际指令（也称语句）
- 指令序列表示为包含这三个部分的表，构造函数：

```
(define (make-instruction-sequence needs modifies statements)
  (list needs modifies statements))
```

- 下面是一个包含两条指令的序列，它找出变量x的值赋给val后返回，要求执行前初始化寄存器env和continue，并修改寄存器val：

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

指令序列的构造

- 有时需要构造不含语句的指令序列：

```
(define (empty-instruction-sequence)
  (make-instruction-sequence '() '() '()))
```

- 组合指令序列的各种过程下面讨论
- 下面考虑各种表达式的编译，实现由compile分派的各种代码生成器
- 下面用到反引号表达式 `(...)。与引号表达式类似，被引表达式不求值。但表达式里的 ,exp 情况特殊，需要把 exp 求出的值放在该位置
- 例如：设 place 的值是 (China Beijing)，howlong 的值是 (5 years)

```
`(I live in ,place for ,howlong)
```

的值是

```
(I live in (China Beijing) for (5 years))
```

这种表达式提供了一个框架，其中 ,e 的内容用 e 求出的值填充

连接代码的编译

- 生成代码最后总是compile-linkage生成的连接指令。连接是return时生成(goto (reg continue))，不修改寄存器；连接是next时不生成指令。否则把连接看作标号生成goto指令，不需要也不修改寄存器

```
(define (compile-linkage linkage)
  (cond ((eq? linkage 'return)
         (make-instruction-sequence '(continue) '()
          '((goto (reg continue)))))
        ((eq? linkage 'next)
         (empty-instruction-sequence))
        (else
         (make-instruction-sequence '() '()
          `((goto (label ,linkage)))))))
```

- 把连接代码附在指令序列后需要维持continue，因为return连接要用。如果指令序列修改continue而连接代码需要它，就应保存和恢复

```
(define (end-with-linkage linkage instruction-sequence)
  (preserving
    '(continue) instruction-sequence (compile-linkage linkage)))
```


简单表达式的编译

- 对自求值表达式、引号表达式和变量，代码生成器构造的指令序列将所需值赋给指定目标寄存器，而后根据连接描述符继续

```
(define (compile-self-evaluating exp target linkage)
  (end-with-linkage linkage
   (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))))))
```


赋值和定义表达式的编译

- 赋值和定义的处理与解释器类似。递归生成计算值（准备赋给变量）的代码，后附一个两条指令的序列完成赋值并把值 ok 赋给目标寄存器。递归编译要用目标val和连接next，所以生成的代码把值放入val后执行随后的代码。所用拼接方式要求维持env，因为设置或定义变量都需要当时环境，而产生变量值的代码可能是复杂表达式的编译结果，其中完全可能修改env寄存器（可能需要save和restore）

```
(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)  (reg val)  (reg env) )
         (assign ,target (const ok))))))))
```

赋值和定义表达式的编译

- 定义表达式与赋值类似

```
(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))))))))
```

- 拼接两指令序列时需要env和val，修改目标寄存器。这个序列只保留env但却不保留val，因为get-value-code将把返回值放入val供序列里的指令用。（维护val是不对的，因为这将导致get-value-code运行后又恢复val的原来内容）

条件表达式的编译

- 给定目标和连接，编译 if 表达式生成的指令序列形式如下

```
<编译 predicate 部分的结果, 目标在 val, 连接在 next>
(test (op false?) (reg val))
(branch (label false-branch))
true-branch
<用给定 target, linkage 和 after-if 编译 consequence 部分的结果>
false-branch
<用给定 target 和 linkage 编译 alternative 的结果>
after-if
```

- 生成前需要编译 if 的三个子部分。将得到的代码与检查谓词结果的代码组合时需生成标识真假分支和条件表达式结束的新标号。谓词为假时跳过真分支。如果if的连接是return或标号，真/假分支都使用该连接。如果连接是next，真分支最后应加入跳过假分支的指令
- 不能直接用标号true-branch, false-branch和after-if，因程序里可能有多个 if。设make-label生成新标号，它以一个符号为参数返回一个新符号作为标号，用与查询语言中生成唯一变量名类似的方式实现

条件表达式的编译

生成三个新标号

分别编译出三段代码

```
(define (compile-if exp target linkage)
  (let ((t-branch (make-label 'true-branch))
        (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))))))
```

根据连接确定then最后的动作

求值条件前后保留回复两个寄存器

后面定义

拼接两段不会同时执行的代码

表达式序列的编译

- 对表达式序列（过程体或begin表达式），先分别编译子表达式，最后一个子表达式用整个序列的连接，其他表达式用next连接（执行序列剩下部分）。结果序列由拼接子表达式的指令序列得到。需要保留env（序列其余部分可能用它）和continue（最后的连接可能用它）

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

Lambda 表达式的编译

- lambda表达式构造过程对象，目标代码具有下面形式

  <构造过程对象并将其赋给 target 寄存器>
  <linkage>

- 编译lambda表达式时要生成过程体的代码。虽然构造过程时不执行体，但需要找地方安置其目标代码。lambda的代码后是合适的位置：如果lambda表达式的连接是标号或return，这样正合适。如果连接是next就用转跳连接跳过过程体代码，相应标号放在过程体后面

  <构造过程对象并将其赋给 target 寄存器>
  <给定linkage 的代码 >or (goto (label after-lambda))
  <构成体的编译结果>
  after-lambda

- compile-lambda生成构造过程对象的代码，随后是过程体代码。实际过程对象是运行时构造的，用到当时环境和编译后的过程体入口点

Lambda表达式的编译

- compile-lambda生成构造过程对象的代码，随后是过程体代码。过程对象将在运行时构造，其中组合当时环境和编译后过程体的入口点

```
(define (compile-lambda exp target linkage)
  (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))))
```

组合操作，直接把过程体代码放在lambda表达式代码之后。它们相互无关，只是放在这里合适

Lambda表达式的编译

- compile-lambda-body构造过程体代码，入口点标号后的指令把运行时环境转到求值过程体的正确环境（过程的定义环境）并做环境扩充。此后是过程体表达式序列的编译代码。序列用连接return和目标val编译，结束连接是从过程返回，过程的执行结果放在val

```
(define (compile-lambda-body exp proc-entry)
  (let ((formals (lambda-parameters exp)))
    (append-instruction-sequences
     (make-instruction-sequence '(env proc argl) '(env)
      `(,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) ) ) )
```


组合式的编译

- 过程应用的编译最关键。组合式的编译结果代码的形式：

  <运算符的编译结果, 目标为 proc, 连接为 next>
  <求值运算对象并在 argl 里构造实参表的代码 >
  <用给定目标和连接编译过程调用的结果 >

  运算符和运算对象求值期间可能保留与恢复寄存器env, proc和argl。注意，整个编译器里只有这一处的目标描述不是val 而是 proc

- compile-application递归编译运算符，生成的代码把要应用的过程放入proc；编译各运算对象，生成求值各运算对象的代码
- 将运算对象指令序列与在argl里构造实参表的代码组合（construct-arglist），组合的结果再与过程代码和过程调用代码（compile-procedure-call生成）组合

组合式的编译

- 求值运算符前后需要保留和恢复env（求值运算符时可能修改它们，求值运算对象时需要它们），构造实际参数表前后需要保留proc（运算对象求值可能修改它，实际过程应用需要它）。整个段前后需要保留和恢复continue，过程调用的连接需要它
- ```
  (define (compile-application exp target linkage)
    (let ((proc-code (compile (operator exp) 'proc 'next))
          (operand-codes
            (map (lambda (operand) (compile operand 'val 'next))
              (operands exp))))
      (preserving '(env continue)
       proc-code
       (preserving '(proc continue)
        (construct-arglist operand-codes)
        (compile-procedure-call target linkage)))))
  ```


组合式的编译

- 构造实参表的代码求值运算对象，结果放在val，将该值积累到argl里的实参表中。由于是顺序处理，从最后参数开始反向做才能得到正确顺序。这里让第一个代码序列构造初始的空argl表。代码形式为

  ```
  <最后一个运算对象的编译结果, 目标为 val>
  (assign argl (op list) (reg val))
  <前一个运算对象的编译结果, 目标为 val >
  (assign argl (op cons) (reg val) (reg argl))
  ...<第一个运算对象的编译结果, 目标为 val >
  (assign argl (op cons) (reg val) (reg argl))
  ```

- 除第一个运算对象外，其余运算对象求值前后都保存恢复argl（保证已积累的实际参数不丢失）；除最后一个参数外，每个运算对象求值前后都必须保留和恢复env（以便后续运算对象的求值中使用）
- 第一个参数需要特殊处理，保存argl和env的地方与处理其他参数时不同，编译这段实参代码中有些小麻烦。construct-arglist以求值各运算对象的代码段为参数。如果没有运算对象就直接送出

  ```
  (assign argl (const ()))
  ```


组合式的编译

- 存在参数时，construct-arglist处理最后一个实参时创建初始化argl的代码，而后将求值其他参数的代码顺序结合到argl里。为了反向处理实参，先反转compile-application送来的运算对象代码序列表

```
(define (construct-arglist operand-codes)
 (let ((operand-codes (reverse operand-codes)))
  (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))))))
  ```

过程应用

- 组合式各元素的求值后，编译结果代码要把proc里的过程应用于argl里的实参。这也是分派，类似元循环求值器的apply或显式控制求值器里apply-dispatch：基本过程用apply-primitive-procedure，编译得到的过程另外处理。代码形式

```
(test (op primitive-procedure?) (reg proc))
 (branch (label primitive-branch))
compiled-branch
 <以给定目标和连接应用编译过程的代码>
primitive-branch
 (assign <目标>
         (op apply-primitive-procedure) (reg proc) (reg argl))
 <连接>
after-call
```

- 编译后过程的分支必须跳过处理基本过程的分支。如果过程调用的原连接是next，复合分支就要用跳到插入在基本分支最后的标号的连接（类似于compile-if里真分支所用的连接）

过程应用

```
(define (compile-procedure-call target linkage)
  (let ((primitive-branch (make-label 'primitive-branch))
        (compiled-branch (make-label 'compiled-branch))
        (after-call (make-label 'after-call)))
    (let ((compiled-linkage
           (if (eq? linkage 'next) after-call linkage)))
      (append-instruction-sequences
       (make-instruction-sequence '(proc) '()
        `((test (op primitive-procedure?) (reg proc))
          (branch (label ,primitive-branch))))
       (parallel-instruction-sequences
        (append-instruction-sequences
         compiled-branch
         (compile-proc-appl target compiled-linkage))
        (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))))
```

后面定义

拼接两段不会同时执行的代码（编译 if 表达式时也用了这个过程）

应用编译得到的过程

- 处理过程调用的代码是本编译器最复杂的部分，虽然生成的指令序列很短
- 编译过程（由compile-lambda构造）的入口点标号标明开始位置，代码将算出的结果放入val后执行(goto (reg continue))返回。很容易觉得，如果连接是标号，针对给定目标和连接应用编译过程的代码应是下面形式

```
(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
```

如果连接是 return，代码应该是

```
(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
```


应用编译得到的过程

- 如果目标不是val，编译器就应该生成上面代码。但实际目标通常是val（仅有一处以proc寄存器作为求值的目标），过程结果可以直接放入目标寄存器。代码还可以简化，先设置好continue使得过程直接“返回”到调用者的连接指定的位置

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

- 连接是标号时设置continue使过程直接返回该标号。过程结束的(goto (reg continue))变为等价于proc-return处的(goto (label <linkage>))

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

- 连接是return时不需要再设置continue（它已经存着所需地址），作为过程结束的(goto (reg continue)) 能直接跳到应该的地方

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

应用编译得到的过程

- 这样的return连接生成的是尾递归代码：过程调用中过程体最后一步直接转移，不在栈里保存信息。如果对具有return连接和val目标的过程调用也像前面非val目标的代码一样处理，就会破坏尾递归。得到的代码语义相同，但每次调用过程时都保存continue，调用后撤销这一无用保存的效果，造成线性存储需求
- （书上有一段注释）“看来生成尾递归代码不难。常见语言（C等）的编译器没这样做，因此不能用过程描述迭代。常规语言的栈里不仅保存着返回地址，还保存实参和局部变量。本书Scheme实现里的实参和变量都放在能做废料收集的存储区。用栈保存实参和局部变量可以不依靠废料收集，一般认为效率高。实际上Lisp编译器也可以用栈保存实参又保证尾递归。至于用栈分配的效率是否比废料收集高，有许多争论，有些细节依赖于计算机体系结构”
- 根据调用目标是否val，连接是否return，compile-proc-appl生成调用代码时分四种情况。所有代码序列都说明为要修改所有寄存器（被执行过程体可能修改任何寄存器）。目标val和连接return情况的代码序列说明了需要continue：即使continue并没有用在两个指令序列里，也必须保证进入编译得到的过程时continue的值正确

应用编译得到的过程

```
(define (compile-proc-appl target linkage)
  (cond ((and (eq? target 'val) (not (eq? linkage 'return)))
         (make-instruction-sequence '(proc) all-regs
           `((assign continue (label ,linkage))
             (assign val (op compiled-procedure-entry) (reg proc))
             (goto (reg val)))))
        ((and (not (eq? target 'val)) (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))))))
        ((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))))
```


指令序列的组合

- 考虑指令序列表示和组合的一些细节
- 前面说指令序列用表表示，其中包含所需寄存器集合，所修改寄存器集合，及一串实际指令。标号（符号）看作退化指令，它不需要也不修改寄存器。确定一个指令序列需要哪些寄存器，修改哪些寄存器时用下面选择函数

```
(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)))
```

- 确定某指令序列是否需要或者修某特定寄存器时使用下面谓词

```
(define (needs-register? seq reg)
  (memq reg (registers-needed seq)))
(define (modifies-register? seq reg)
  (memq reg (registers-modified seq)))
```

- 基于这些谓词和选择函数就可以实现编译器的各种指令序列组合过程

指令序列的组合

- 最基本的组合过程append-instruction-sequences以任意个要求顺序执行的指令序列为实参，返回一个指令序列，其中
  - 语句是所有参数序列的语句的顺序拼接
  - 修改的寄存器是被任一实参序列修改的寄存器
  - 需要的寄存器是第一个序列运行前应初始化的寄存器，加上其他序列需要而又没被前面序列初始化（或修改）的寄存器
- 序列用append-2-sequences逐次拼接。此过程以两个指令序列seq1和seq2为参数，返回的指令序列里的语句是两个序列的语句的顺序拼接，所修改的寄存器是所有被seq1或seq2修改的寄存器，需要的寄存器是seq1所需寄存器加上seq2需要而又没有被seq1修改的寄存器（新序列需要的寄存器是seq1的需要寄存器集，与seq2的需要寄存器集与seq1的修改寄存器集的差集之并集）
- append-instruction-sequences实现如下

指令序列的组合

- append-instruction-sequences的实现

```
(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))
```


指令序列的组合

- 过程里用了一些简单操作完成对表形式表示的集合的运算

```
(define (list-union s1 s2)
  (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)
  (cond ((null? s1) '())
        ((memq (car s1) s2) (list-difference (cdr s1) s2))
        (else (cons (car s1) (list-difference (cdr s1) s2)))))
```

- 最重要的是preserving，其参数是寄存器表regs和两个要求顺序执行的指令序列seq1和seq2。返回的指令序列中包括它们的语句，加上围在seq1的语句前后适当save和restore，以保护regs里seq2需要而会被seq1修改的寄存器。preserving先创建一个序列：所需save，seq1的语句，所需restore。其中所需寄存器包括seq1需要的和这里保留恢复的寄存器，修改的寄存器是seq1修改的寄存器除去这里保留和恢复的寄存器。最后按常规方式将这一扩充序列与seq2拼接

指令序列的组合

- 下面以递归方式实现上述策略，逐一处理要保留的寄存器表里的寄存器

```
(define (preserving regs seq1 seq2)
  (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)) (statements seq1)
                        `((restore ,first-reg))))
              seq2)
            (preserving (cdr regs) seq1 seq2)))))
```


指令序列的组合

- tack-on-instruction-sequence用在compile-lambda里，将过程与另一序列拼接。由于过程体不作为组合序列的一部分而“在线”执行，它用的寄存器对它嵌入其中的序列的寄存器使用没有影响。在将过程体纳入其他序列时，应忽略它所需要和修改的寄存器集合

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

- compile-if和compile-procedure-call用了特殊组合过程完成两个分支的拼接，parallel-instruction-sequences。两个分支不顺序执行

```
(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))))
```

编译代码的实例

- 现在考察一个编译代码实例，看前面定义的东西如何相互配合
- 按下面形式调用compile，编译递归定义的factorial过程

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

前面说过，define表达式的值应放入val，这里不关心执行define后的编译代码是什么。因此随意选择next作为连接描述符

编译代码的实例

- compile看到表达式是定义，调用compile-definition去编译计算被赋的值的代码（以val为目标），随后是安装这一定义的代码，随后是将define的值（符号ok）放入目标寄存器的代码，最后是连接代码。env保留绕过值计算部分，因为后来还要用它安装定义。由于连接是next，不需要连接代码。编译结果代码的框架如下

```
<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))
```


编译代码的实例

- 产生factorial值的是lambda表达式，compile调用compile-lambda去编译过程体，用新标号标记其入口点，生成一些指令将位于这个入口的过程体组合到运行时环境中，最后将结果赋给val。编译好的过程代码放在这里，整个序列要跳过这些代码。过程代码开始扩充过程定义环境，增加一个框架；随后是过程体。由于求变量值的代码不修改env，不做save和restore（位于entry2的过程代码在这点还没有执行，它对env的使用与此无关）。代码框架变成

```
  (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>
after-lambda1
  (perform (op define-variable!)
           (const factorial)  (reg val) (reg env))
  (assign val (const ok))
```


编译代码的实例

- 过程体（用compile-lambda-body）编译为一个序列，以val为目标，用连接return。目前这个序列来自if表达式

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

- compile-if生成的代码先算谓词（目标为val），谓词假时跳过真分支。谓词代码前后保留恢复env、continue，因为其他部分可能用。这个if是过程体的序列的最后表达式，目标是val且连接是return，所以真假分支都用目标val和连接return编译（即，条件表达式的值就是其分支算出的值，是整个过程的值）

```
  <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
```

编译代码的实例

- 谓词(= n 1)是过程调用，要找运算符（符号 =）并把相应值放入proc。而后把实参1和n的值装进argl。代码检查proc里是基本过程还是复合过程，并根据情况分派，两个分支都结束在after-call标号。目前情况不需要寄存器保留动作，因为这里的求值都不修改要考虑的寄存器

```
(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
```


编译代码的实例

- 真分支是常数1，编译为（用目标val和连接return）

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

- 假分支的代码是另一过程调用，其中过程是符号 * 的值，参数是n和另一过程调用的结果（对factorial的递归调用）。每个调用都要设置proc和argl及其基本分支和复合分支
- 下面是factorial过程定义的完整编译结果。注意，围绕着谓词的还有实际生成的对continue和env的save和restore，因为谓词里的过程调用要修改它们，两个分支里的过程调用和return连接都需要它们

编译结果（100行，包括空行）

```
;; 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))

entry2    ; calls to factorial will enter here
  (assign env (op compiled-procedure-env) (reg proc))
  (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))
  (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))

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

;; 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
;; 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)
;; 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))
```


词法地址

- 现在考虑一种可能的代码优化，优化变量查找
- 至今生成的代码用求值器机器的lookup-variable-value查找变量，一个个框架顺序查找。如果框架嵌套很深或变量很多，代价非常高
- 考虑某过程应用里求值(* x y z) 时的情况，过程由下面表达式返回

```
(let ((x 3) (y 4))
  (lambda (a b c d e)
    (let ((y (* a b x)) (z (+ c d x)))
      (* x y z)))
```

let是lambda的语法包装，等价于

```
((lambda (x y)
   (lambda (a b c d e)
     ((lambda (y z) (* x y z))
      (* a b x)
      (+ c d x))))
 3
 4)
```

lookup-variable-value每次查找x时都要确定x不是y或z（第一个框架里）也不是a, b, c, d或e（第二个框架）

假定没有define而只有lambda建立变量约束。采用词法作用域规则，表达式运行时环境的结构与其出现所在的过程的词法结构一样。编译器分析这个表达式时就知道过程应用时 (* x y z)里的x总会在当前框架外面第二个框架找到，而且是框架里第二项

词法地址

- 利用这一事实，可实现一个新变量查找过程 lexical-address-lookup，其参数是一个环境和一个词法地址（词法地址由两个数组成：框架号描述要跳过几个框架；移位数描述在框架里应跳过几个变量）。过程在当前环境里找出存在给定词法地址处的变量值
- 如果把lexical-address-lookup操作加进前面的机器，就可以在编译生成的代码里用它引用变量。还可以用一个新操作lexical-address-set!
- 为生成这种代码，编译一个变量引用时就必须确定变量的词法地址。变量在程序里的词法地址依赖于具体变量在代码里出现的位置

例如对右边程序

表达式<*e1*>里变量x的地址是(2,0)，向回两个框架里的最前面一个变量。在这一点y的地址是(0,0)，c的地址是(1,2)

表达式<*e2*>里，变量x的地址是(1,0)，y的地址是(1,1)，c的地址是(0,2)

```
((lambda (x y)
   (lambda (a b c d e)
     ((lambda (y z) <e1>)
      <e2>
      (+ c d x))))
 3
 4)
```


词法地址

- 编译器要生成使用词法地址的代码，一种方法是维持一个称为编译时环境的数据结构，其中保存各种变化的轨迹，说明在程序执行到特定变量访问操作时，各变量出现在运行环境的哪个框架的哪个位置
- 编译时环境也用框架的表，框架是变量表（无值，编译时不可能算值）。把这种环境作为compile的另一参数，与原有参数一起传给代码生成器
- 对compile的最高层调用给一个空编译时环境。编译lambda体时，compile-lambda-body用一个包含该过程的所有变量的框架扩充当时的编译时环境，构成lambda体的表达式序列在扩充后的环境里编译。编译中每一点，compile-variable和compile-assignment都用这个环境生成出正确的词法地址
- 练习5.39到5.43说明如何完成这一策略，将词法查找结合到编译器里。练习5.44描述编译时环境的另一用途

编译代码与求值器的互连

- 还应解释如何将编译得到的代码装入求值器，以及怎样运行
- 假设已定义好显式控制求值器，包括必要的操作。下面实现一个过程compile-and-go，它编译一个Scheme表达式，将目标代码装入求值器机器并启动该机器，使之在求值器的全局环境里运行这一代码，打印结果后再次进入求值器的驱动循环。还要修改求值器，使解释性的表达式除能调用其他编译代码外，也能调用编译后的过程。此后就可以将编译后的过程放进机器，并用求值器调用它们了

```
(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
```


编译代码与求值器的互连

- 为使求值器能处理编译后的过程，需要修改位于apply-dispatch的代码，使它能识别编译后的过程（与基本过程和复合过程具有同等地位），并将控制直接传到编译后代码的入口点

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

注意，compiled-apply处需要恢复continue。回忆求值器各方面的安排情况，在apply-dispatch处继续点位于堆栈顶。而编译代码的入口点却期望继续点在continue里。因此，执行编译代码前必须恢复continue

编译代码与求值器的互连

- 为在启动求值器机器时能运行一些编译代码，在求值器机器开始加一条branch指令，如果寄存器flag被设置，就要求机器转向一个新入口点

```
  (branch (label external-entry))     ; branches if flag is set
read-eval-print-loop
  (perform (op initialize-stack))
  ...
```

- external-entry入口假定启动时val包含一个指令序列的位置，该指令序列将结果放在val里并以 (goto (reg continue))结束。从这一入口点启动，执行就会跳到val指定的位置，但先设置continue使执行还能转回print-result，打印val的值后转到求值器的读入-求值-打印循环开始

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


编译代码与求值器的互连

- 现在已经可以按下面方式编译过程的定义，执行编译后的代码，然后运行读入-求值-打印循环，能试验这个过程了
- 因为希望编译后的代码能将结果放入val并返回continue里的地址，应该用目标val和连接return去编译表达式。为将编译器生成的目标代码转换到求值器寄存器机器的可执行指令，下面用来自寄存器机器模拟器的过程assemble。最后设置val寄存器使之指向指令的表，设置flag使求值器转向入口点external-entry，并启动求值器

```
(define (compile-and-go expression)
    (let ((instructions
           (assemble (statements
                      (compile expression 'val 'return))
                     eceval)))
      (set! the-global-environment (setup-environment))
      (set-register-contents! eceval 'val instructions)
      (set-register-contents! eceval 'flag true)
      (start eceval)))
```


编译代码与求值器的互连

- 如果设了堆栈监视器（第5.4.4节）就可以检测编译代码的堆栈使用情况

```
(compile-and-go
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n))))

(total-pushes = 0 maximum-depth = 0)
 ;;; EC-Eval value:
ok
 ;;; EC-Eval input:
(factorial 5)
(total-pushes = 31 maximum-depth = 14)
;;; EC-Eval value:
120
```

与同一过程的解释性版本的求值(factorial 5)比较（第5.4.4节最后），解释性版本需要144次压栈，最大栈深28。可看出编译的优化

解释和编译

- 有了上面程序，现在可以对解释和编译的不同策略做各种试验
- 解释器将所用机器提升到源程序层面；而编译器将源程序降低到机器语言层面。可以认为Scheme语言（或任何高级语言）是矗立在机器语言之上的一族有效抽象。解释器能很好支持交互式的程序开发和排错，因为程序执行的细节步骤以这套抽象方式组织起来，程序员更容易理解。编译后的代码更快，因为程序执行步骤在机器语言层面上，编译器也可以自由地做各种跨越高层抽象的优化
- 解释和编译之间的相互替代关系也带来将一种语言移植到新计算机的不同策略。假定希望在一种新机器上实现Lisp。一种策略是从第5.4节的显式控制求值器出发，将其中的指令一条一条翻译到新机器。另一种策略是从编译器出发，修改其代码生成器，使之能为这种新计算机生成代码。第二种策略使人可以在新机器上运行任何Lisp程序，方式是先用原有的Lisp机器上的编译器去编译它，并将它与有关运行库的编译后的版本连接。事情还可以更好，可以编译这个编译器本身，并在新机器上运行编译结果，去编译其他Lisp程序。或者可以编译第4.1节的一个解释器，生成一个可以在这一新机器上运行的解释器