```scheme
;;; file: compiler.scm (from section 5.5 of Structure and
;;; Interpretation of Computer Programs)

;;; Comments and small modifications made 2002-2004 by Carl Offner:

;;; To compile a file, load compiler-shell.scm.  That in turn loads
;;; this file, and prompts for a source file.  It creates an
;;; executable file that you can then run by loading
;;; machine-shell.scm.

;;; This file is identical to the code in the textbook with a few
;;; exceptions:
;;;
;;; 1.  compile-assignment and compile-definition end by generating
;;; code to return the name of the variable being defined or assigned
;;; to, rather than the symbol 'ok.
;;;
;;; 2.  The instruction sequence attributes (needs, modifies) are
;;; expanded to (possibly_needs, possibly_modifies,
;;; definitely_modifies) to reflect the way they are actually used and
;;; to allow for the possibility of adding more registers that have
;;; different patterns of use.  The need for doing something like this
;;; was pointed out in Spring 2002 by Nick Anzalone.
;;;
;;; 3.  I renamed parallel-instruction-sequences to
;;; generate-alternative-sequences because it seems much clearer that
;;; way.  There is no parallelism in this code or in our model of
;;; execution.  The purpose of generate-alternative-sequences is to
;;; generate two sequences of code only one of which will be executed,
;;; the decision to be made at run-time.  (Thus, the code must come
;;; after a test and branch.)

(load "syntax.scm")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      The main dispatch procedure
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(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))))
```

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      Instruction sequences
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; An instruction sequence is now of the form
;;; (<possibly needs> <possibly modifies> <definitely modifies> <statements>)

(define (make-instruction-sequence needs pmodifies dmodifies statements)
  (list needs pmodifies dmodifies statements))

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


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      Compiling linkage code
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Every generated code fragment has to end by "going somewhere".
;;; The linkage parameter specifies exactly what happens.  There are
;;; three possible values for this parameter:
;;;
;;; 'return
;;;
;;;    This compiles into '(goto (reg continue)).  It originates in two places:
;;;
;;;    a) at the top level call (in compiler-shell.scm).  This has the
;;;    effect of returning to the outermost print routine.
;;;
;;;    b) at the end of a compiled procedure call (see the end of
;;;    compile-lambda-body).
;;;
;;; 'next
;;;
;;;    This means just fall through to the next instruction, so
;;;    it compiles to an empty instruction sequence.
;;;
;;; anything else
;;;
;;;    This must be a label.  It compiles to `(goto (label ,linkage)).

(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)))))))

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

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      self-evaluating expressions
;;;      quoted expressions
;;;      variable names
;;;      assignments
;;;      definitions
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (compile-self-evaluating exp target linkage)
  (end-with-linkage linkage
                    (make-instruction-sequence '() (list target) (list target)
                                               `((assign ,target (const ,exp)))))
)

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

(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) (list target)
                                   `((perform (op set-variable-value!)
                                              (const ,var)
                                              (reg val)
                                              (reg env))
                                     (assign ,target (const ,var))))))))

(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) (list target)
                                   `((perform (op define-variable!)
                                              (const ,var)
                                              (reg val)
                                              (reg env))
                                     (assign ,target (const ,var))))))))
```

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      sequences
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; This is the usual kind of recursive construction.  The only
;;; special thing to notice is that the linkage for each expression is
;;; changed to 'next except for the last one, which is the linkage
;;; that was passed in.  That is, the code for each expression simply
;;; falls through to evaluate the next, and the code for the last
;;; expression exits by going where the original linkage specified.

(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))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      The label generator
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(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)))))
```

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      conditional expressions
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; The generated code looks like this:
;;;
;;;       <code to evaluate the if test into the val register>
;;;       (test (op false?) (reg val))
;;;       (branch (label false-branch25))
;;;     true-branch24
;;;       <code for true "consequent">
;;;       <linkage code for "consequent">
;;;     false-branch25
;;;       <code for "alternative">
;;;       <linkage code for "alternative">
;;;     after-if26
;;;
;;; Now there are three possibilities for the passed in linkage:
;;;
;;;     1.  'return
;;;
;;;         In this case, both the consequent-linkage and the
;;;         alternative-linkage are
;;;
;;;         (goto (reg continue))
;;;
;;;     2.  Some label
;;;
;;;         In this case, both the consequent-linkage and the
;;;         alternative-linkage are
;;;
;;;         (goto (label <linkage>))
;;;
;;;     3.  'next
;;;
;;;         In this case the alternative linkage is empty, because the
;;;         code can just continue executing whatever comes afterward.
;;;
;;;         But the consequent linkage can't be empty, because then the
;;;         machine would continue executing the alternative code, which
;;;         we don't want.  So in this case, we want the consequent
;;;         linkage to be
;;;
;;;         (goto (label after-if26))
;;;
;;;         That way we jump over the alternative code and continue
;;;         execution after it.
```

```scheme
(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))))
                       (generate-alternative-sequences
                        (append-instruction-sequences t-branch c-code)
                        (append-instruction-sequences f-branch a-code))
                       after-if))))))
```

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      lambda expressions
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; The code generated for a lambda expression looks like this:
;;;
;;;   (assign <target> (op make-compiled-procedure) (label ENTRY) (reg env))
;;;   <linkage code>
;;;   ;; The linkage code jumps to AFTER-LAMBDA if the passed in linkage is
;;;   ;; 'next.  Otherwise it generates a (goto continue) or (goto <label>)
;;;
;;; ;; Now comes the code for the "body" of the lambda expression.  At this
;;; ;; point, we know that the proc register holds the procedure (made by
;;; ;; make-compiled-procedure -- so it is made from the entry label and the
;;; ;; procedure environment).
;;;
;;; ENTRY
;;;   (assign env (op compiled-procedure-env) (reg proc))
;;;   ;; extract <formals> from exp
;;;   (assign env (op extend-environment) <formals> (reg argl) (reg env))
;;;   ;; generate code for the body of the lambda expression, using
;;;   ;; compile-sequence.
;;; AFTER-LAMBDA

;; Note that the body of the compiled lambda expression is always set up to
;; return to what is in the continue register (via a linkage of 'return).  This
;; is key to understanding the code in compile-proc-appl below.

(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) (list target)
                           `((assign ,target
                                     (op make-compiled-procedure)
                                     (label ,proc-entry)
                                     (reg env)))))
        (compile-lambda-body exp proc-entry))
       after-lambda))))

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

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      procedure calls; setting up for the call
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; compile-application generates code to evaluate the procedure and
;; its arguments, and then invokes compile-procedure-call to generate
;; code to evaluate the procedure body.

;; The procedure code is stored in the variable
;;
;;    proc-code
;;
;; When that code is executed, it produces a compiled-procedure object
;; in the proc register.

;; The argument evaluation code is produced by the call
;;
;;    (construct-arglist operand-codes)
;;
;; When that code is executed, it evaluates the operands into a list
;; in the argl register.

;; Note: The call to compile in the second line below (producing
;; proc-code) is the only place in the compiler where a target
;; different from 'val is specified.

(define (compile-application exp target linkage)
    (let ((proc-code (compile (operator exp) 'proc 'next)) ;; See note above.
          (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)))))

(define (construct-arglist operand-codes)
  (let ((operand-codes (reverse operand-codes)))
    (if (null? operand-codes)
        (make-instruction-sequence '() '(argl) '(argl)
                                   '((assign argl (const ()))))
        (let ((code-to-get-last-arg
               (append-instruction-sequences
                (car operand-codes)
                (make-instruction-sequence
                 '(val) '(argl) '(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))))))))

(define (code-to-get-rest-args operand-codes)
  (let ((code-for-next-arg
         (preserving '(argl)
                     (car operand-codes)
                     (make-instruction-sequence
                      '(val argl) '(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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      procedure calls: making the actual call
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; compile-procedure-call generates code to see (at run-time) if the
;; procedure is primitive or user-defined.  It generates code for both
;; possibilities: for a primitive procedure, the code calls the
;; underlying Scheme (via apply-primitive-procedure, which is just the
;; Scheme apply) to evaluate the procedure call.  For a user-defined
;; procedure, the code calls compile-proc-appl below.

(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))))
       (generate-alternative-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) (list target)
                            `((assign ,target
                                      (op apply-primitive-procedure)
                                      (reg proc)
                                      (reg argl)))))))
       after-call))))

;;; compile-proc-appl (below) generates code to save registers as
;;; appropriate, to set up a return address, and to jump to the code
;;; for the procedure body.
;;;
;;; The tricky part is setting up a return address for the whole procedure call.
;;; This of course depends on the linkage.  To do this, we make use of the
;;; following contract, which we noted above:
;;;
;;; We know that the code generated for the body of a lambda expression will
;;; always returns to what is in the continue register.  So we can set the
;;; continue register to whatever we need.
;;;
;;; Now the compile-proc-appl is called from only 1 place:
;;; compile-procedure-call.  And compile-procedure-call has already replaced any
;;; 'next linkage by a label.  So there are only two possibilities for the
;;; linkage:
;;;
;;;    'return
;;;
;;;    some label
;;;
;;; How they are handled depends on the target, and there are two possibilities
;;; for that as well:
;;;
;;; target is not 'val
;;;
;;;   This happens only when compiling an operator into the proc register.  The
;;;   linkage was originally (in compile-application) 'next, which (as explained
;;;   above) is turned into a label in compile-procedure-call.  So the
;;;   possibilities are:
;;;
;;;   linkage is 'return (case 4 below)
;;;
;;;     As we've just pointed out, this must be an error.
;;;
;;;   linkage is not 'return (case 2 below); it must be a label.
;;;
;;;     Make a new label ('proc-return).  Have the generated code return to that
;;;     label, (by assigning that label to the continue register) assign what is
;;;     in (reg val) to the target, and then goto the passed in linkage.
;;;
;;; target is 'val -- the usual case
;;;
;;;   In this case there is no need to return to the call site because
;;;   we can make sure the result winds up in the val register
;;;   automatically.  That is, we can make the call "tail-recursive":
;;;
;;;   linkage is not 'return (case 1 below)
;;;
;;;     The linkage must be a label.  (If it was 'next, it was turned into a
;;;     label in compile-procedure-call.)  Put the label in the continue
;;;     register (so the procedure call will return there), put the procedure
;;;     entry in the val register, and go there.
;;;
;;;   linkage is 'return (case 3 below)
;;;
;;;     It's the same except that the continue register already holds
;;;     the place to return to.  So just put the procedure entry in
;;;     the val register and go there.
```

```scheme
(define (compile-proc-appl target linkage)
    (cond ((and (eq? target 'val) (not (eq? linkage 'return)))    ;;; CASE 1
           (make-instruction-sequence
            '(proc) all-regs '(val continue) ;; !!!
            `((assign continue (label ,linkage))
              (assign val (op compiled-procedure-entry)
                      (reg proc))
              (goto (reg val)))))
          ;; ---------------------------------------------------------------
          ((and (not (eq? target 'val))                             ;;; CASE 2
                (not (eq? linkage 'return)))
           (let ((proc-return (make-label 'proc-return)))
             (make-instruction-sequence
              '(proc) all-regs
              `(val continue ,target) ;; !!!
              `((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))          ;;; CASE 3
           (make-instruction-sequence
            '(proc continue) all-regs '(val) ;; !!!
            '((assign val (op compiled-procedure-entry)
                      (reg proc))
              (goto (reg val)))))
          ;; ---------------------------------------------------------------
          ((and (not (eq? target 'val)) (eq? linkage 'return))    ;;; CASE 4
           (error "return linkage, target not val -- COMPILE:  "
                  target))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;      Generating linkage code
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Selectors and predicates for instruction-sequences:

(define all-regs '(env proc val argl unev continue temp1 temp2))

(define (registers-needed s)
  (if (symbol? s) '() (car s)))

(define (registers-pmodified s)
  (if (symbol? s) '() (cadr s)))

(define (registers-dmodified s)
  (if (symbol? s) '() (caddr s)))

(define (statements s)
  (if (symbol? s) (list s) (cadddr s)))

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

(define (pmodifies-register? seq reg)
  (memq reg (registers-pmodified seq)))

(define (dmodifies-register? seq reg)
  (memq reg (registers-dmodified seq)))

;;; Set operations on sets of registers:

(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)))))

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

```scheme
;;; There are four ways of putting together instruction sequences:

;;; 1: append-instruction-sequences.

;;; This simply appends the statement lists and propagates the
;;; register attributes.

;;; append-instruction-sequences is the one place where the set of
;;; definitely-modified registers is used.

(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-dmodified seq1)))
     (list-union (registers-pmodified seq1)
                 (registers-pmodified seq2))
     (list-union (registers-dmodified seq1)
                 (registers-dmodified 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))

;;; 2: preserving.

;;; The first argument is a list of registers.  Each of those
;;; registers will be saved and restored over the first
;;; instruction-sequence if both
;;;
;;;     a) the second instruction sequence needs it, and
;;;     b) the first instruction sequence possibly modifies it.

;;; preserving is the one place where the set of possibly-modified
;;; registers is used.

(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)
                 (pmodifies-register? seq1 first-reg))
            (preserving (cdr regs)
                        (make-instruction-sequence
                         (list-union (list first-reg)
                                     (registers-needed seq1))
                         (list-difference (registers-pmodified seq1)
                                          (list first-reg))
                         (list-difference (registers-dmodified seq1)
                                          (list first-reg))
                         (append `((save ,first-reg))
                                 (statements seq1)
                                 `((restore ,first-reg))))
                        seq2)
            (preserving (cdr regs) seq1 seq2)))))

;;; 3: tack-on-instruction-sequence.
;;;
;;; This just propagates the register attributes from the first
;;; instruction-sequence, ignoring those of the second
;;; instruction-sequence.  It is only used in compile-lambda, where
;;; the second instruction-sequence is the body of the lambda
;;; expression and is code that will always be jumped into.  So that
;;; code does not care what the pattern of register usage for the
;;; previous code is.

(define (tack-on-instruction-sequence seq body-seq)
  (make-instruction-sequence
   (registers-needed seq)
   (registers-pmodified seq)
   (registers-dmodified seq)
   (append (statements seq) (statements body-seq))))

;;; 4: generate-alternative-sequences.

;;; This procedure is used to generate two sequences of code, only one
;;; of which will be executed.  The decision of which one to execute
;;; is made at run-time.  Thus, this must follow the generation of a
;;; test and branch.  This is used in two places:
;;;
;;;   a) in generating code for the consequent and alternative of an
;;;   if special form.
;;;
;;;   b) in generating code to implement a procedure call, since we do
;;;   not in general know at compile-time (i.e., before evaluating the
;;;   procedure name) whether the procedure is a primitive or a
;;;   user-defined procedure.  We have to generate code for both, and
;;;   decide at run-time which to execute.

;;; generate-alternative-sequences is the one place where the
;;; possibly-modified registers and the definitely-modified registers
;;; are propagated differently.


(define (generate-alternative-sequences seq1 seq2)
  (make-instruction-sequence
   (list-union (registers-needed seq1)
               (registers-needed seq2))
   (list-union (registers-pmodified seq1)
               (registers-pmodified seq2))
   (list-intersection (registers-dmodified seq1)
                      (registers-dmodified seq2))
   (append (statements seq1) (statements seq2))))
```