```
;;; File:  compiler−shell.scm.
;;;
;;; This file implements the user interface to the compiler.  So to
;;; invoke the compiler, type (at the Unix prompt)
;;;
;;;          scheme compiler−shell.scm
;;;
;;; This interface replaces that provided by A&S.  The compiler
;;; provided by the text compiles just one Scheme expression.  In this
;;; implementation, we read in an entire file, wrap it in a "begin",
;;; and compile that expression.  In this way, we are able to compile
;;; a whole sequence of expressions, including definitions and
;;; mutually recursive procedures.

;;; Note that "write" is used below in one place instead of "display".
;;; This is to preserve quoted strings in the expression.  "display"
;;; removes the quotes.

(load "compiler.scm")

(display "File:  ")
(let* ((file (read))
       (filename (symbol->string file))
       (length (string−length filename))
       (filename−base "")
       (source '())
       (outfile '()))

  ;; Note:  read−file produces a *list* whose elements are the statements
  ;; in the source file.  Below we will cons 'begin onto the beginning
  ;; of this list.

  (define (read−file)
    (let ((expr (read)))
      (if (not (eof−object? expr))
          (begin
            (set! source (append source (list expr)))
            (read−file)
            )))
    source)

  (define (emit)
    (define (write−with−newlines statements)
      (define (add−newline stmt)
        (if (list? stmt)    ;; If it's a real statement, not a label,
            (display "  ")  ;;   then prepend 2 spaces for indentation;
            (display " "))  ;;   otherwise 1 space.
        (write stmt)        ;; preserve strings by using "write"
        (newline))
      (for−each add−newline statements))

    ;; Here is where the compilation happens and the output file is
    ;; written:

    (let ((output (compile source 'val 'return)))
      (display "(")
      (display (car output))   ;; list of needed registers;
      (display* #\newline " ")
      (display (cadr output))  ;; list of possibly modified registers;
      (display* #\newline " ")
      (display (caddr output)) ;; list of definitely modified registers;
      (display* #\newline " (" #\newline)
      (write−with−newlines (cadddr output))
      (display* " )" #\newline)
      (display ")"))) ;;; end of (define (emit) ...

  ;; If the filename ends in ".scm", the output file name will be the same
  ;; as the filename with ".cmp" substituted for ".scm".  If the filename
  ;; does not end in ".scm", the output file name is just the filename
  ;; with ".cmp" appended.

  (cond ((and (> length 4)
              (string=? (substring filename (− length 4) length) ".scm"))
         (set! filename−base (substring filename 0 (− length 4))))
        (else
         (set! filename−base filename)
         (set! filename (string−append filename ".scm"))))
  (set! outfile (string−append filename−base ".cmp"))
  (set! source (with−input−from−file filename read−file))
  (set! source (cons 'begin source))
  (with−output−to−file outfile emit)
  ) ;;; end of (let* ...

;;;(exit) ; This works if we add (exit) as a primitive to the underlying Scheme.
```