```
;;; File:  machine−shell.scm
;;;
;;; This file is used to execute a .cmp file.  To do this, type (at
;;; the Unix prompt)
;;;
;;;          scheme machine−shell.scm
;;;

(load "regsim.scm")
(load "eceval−support.scm")

(define statements cadddr)

;; any old value to create the variable so it can be set! later:
(define the−global−environment '())

;; Modification of section 4.1.4 procedure
;; **replaces version in syntax file
(define (user−print object)
  (cond ((compound−procedure? object)
         (display (list 'compound−procedure
                        (procedure−parameters object)
                        (procedure−body object)
                        '<procedure−env>)))
        ((compiled−procedure? object)
         (display '<compiled−procedure>))
        (else (display object))))


;; These are the machine−primitive operations used in generated code.
;; That is, they are machine operations and occur in (op xxx) and
;; (perform xxx) constructs.

(define eceval−operations
  (list
   ;; operations defined by Scheme primitives
   (list 'list list)
   (list 'cons cons)
   (list 'car car)
   (list 'cdr cdr)
   (list 'null? null?)
   (list 'reverse reverse)
   (list 'newline newline)
   ;; Uncomment the following line if needed for debugging.
   ;;   (list 'display display)

   ;; operations in eceval−support.scm

   (list 'announce−output announce−output)
   (list 'user−print user−print)
   (list 'get−global−environment get−global−environment)
   (list 'define−variable! define−variable!)
   (list 'extend−environment extend−environment)
   (list 'lookup−variable−value lookup−variable−value)
   (list 'primitive−procedure? primitive−procedure?)
   (list 'apply−primitive−procedure apply−primitive−procedure)
   (list 'false? false?)
   (list 'set−variable−value! set−variable−value!)
   (list 'make−compiled−procedure make−compiled−procedure)
   (list 'compiled−procedure? compiled−procedure?)
   (list 'compiled−procedure−entry compiled−procedure−entry)
   (list 'compiled−procedure−env compiled−procedure−env)
   ))

;;**NB. To not monitor stack operations, comment out the line after
;; the print−result label in the machine controller below, and choose
;; the non−monitored version of make−stack in regsim.scm.

(define eceval
  (make−machine
   '(env val proc argl continue
         unev temp1 temp2)            ;; registers; note "temp[12]" added.
    eceval−operations                  ;; machine−primitives
    ;; Now comes the machine code.  This is really just a stub; the
    ;; compiled code is pointed to by the val register.
    '(
      ;;
      ;; We start here on entry to the machine.
      ;;
      (perform (op initialize−stack))
      (assign env (op get−global−environment))
      (assign continue (label print−result))
      (goto (reg val))

    print−result
      ;;**following instruction optional −− if use it, need monitored stack
      (perform (op print−stack−statistics))
      (perform (op user−print) (reg val))
      (perform (op newline))
      )))


;;; machine−shell.scm starts executing here with a prompt to the user:

(display "File:  ")
(let* ((file (read))
       (filename (symbol−>string file))
       (length (string−length filename))
       (source '())
       (instructions '()))
  (if (not (and (> length 4)
                (string=? (substring filename (− length 4) length) ".cmp")))
      (set! filename (string−append filename ".cmp")))
  (set! source (with−input−from−file filename read))
  (set! instructions
        (assemble (statements source) eceval))
  (set! the−global−environment (setup−environment))
  (set−register−contents! eceval 'val instructions)
  (set−register−contents! eceval 'flag #t)
  ;; Comment this next line out if you want to set breakpoints
  ;; before starting the register machine.  Of course then you
  ;; have to (start eceval) yourself from the Scheme prompt.
  (start eceval)
  )

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