This file contains procedures that are taken from the Chapter 4 interpreter. They are used in two contexts:

It is loaded by eceval-support.scm to provide implementations of additional machine-primitive operators in the register machines of Chapter 5.

compiler.scm to support syntax analysis in the compiler itself.

Numbers, strings, and booleans are all represented as themselves. (Not characters though; they don’t seem to work out as well because of an interaction with read and display.)

Variables — represented as symbols

(define (variable? exp) (symbol? exp))

Quote — represented as (quote <text-of-quotation>)

(define (quoted? exp) (tagged-list? exp 'quote))

(define (text-of-quotation exp) (cadr exp))

(define (tagged-list? exp tag) (if (pair? exp) (eq? (car exp) tag) #f))

Assignment — represented as (set! <var> <value>)

(define (assignment? exp) (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))

Definitions — represented as

(define <var> <value>
  ...<parameter_n>) <body>)

The second form is immediately turned into the equivalent lambda expression.

(define (definition? exp) (tagged-list? exp 'define))

(lambda expressions -- represented as (lambda ...)

That is, any list starting with lambda. The list must have at least one other element, or an error will be generated.

(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (caddr exp))

(define (make-lambda parameters body) (cons 'lambda (cons parameters body)))

Conditionals — (if <predicate> <consequent> <alternative>?)

(define (if? exp) (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp) (if (not (null? (cdddr exp))) (cadddr exp) 'false))

**following needed only to implement COND as derived expression, not needed by eceval machine in text. But used by compiler

(define (make-if predicate consequent alternative) (list 'if predicate consequent alternative))

-Sequences — (begin <list of expressions>)

(define (begin? exp) (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))

(define (sequence->exp seq) (cond ((null? seq) seq) ((last-exp? seq) (first-exp seq)) (else (make-begin seq)))

(define (make-begin seq) (cons 'begin seq))

Procedure applications -- any compound expression that is not one of the above expression types.

(define (application? exp) (pair? exp))

(define (definition-variable exp)
  (if (symbol? (cadr exp)) (cadr exp) (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadadr exp)) (make-lambda (cdadr exp) (cdcdr exp))
  )

;;; lambda expressions -- represented as (lambda ...)

;;; That is, any list starting with lambda. The list must have at least one other element, or an error will be generated.

(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (caddr exp))

(define (make-lambda parameters body) (cons 'lambda (cons parameters body)))

;;; Conditionals -- (if <predicate> <consequent> <alternative>?)

(define (if? exp) (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp) (if (not (null? (cdddr exp))) (cadddr exp) 'false))

;;;**following needed only to implement COND as derived expression, not needed by eceval machine in text. But used by compiler

(define (make-if predicate consequent alternative) (list 'if predicate consequent alternative))

;;; Sequences -- (begin <list of expressions>)

(define (begin? exp) (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))

(define (sequence->exp seq) (cond ((null? seq) seq) ((last-exp? seq) (first-exp seq)) (else (make-begin seq)))

(define (make-begin seq) (cons 'begin seq))

;;; Procedure applications -- any compound expression that is not one of the above expression types.

(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))

;;; Derived expressions -- the only one we include initially is cond, 
;;; which is a special form that is syntactically transformed into a 
;;; nest of if expressions.

(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))
(define (cond-else-clause? clause) (eq? (cond-predicate clause) 'else))
(define (cond-actions clause) (car clause))
(define (cond-actions clause) (cdr clause))

(define (cond->if exp) (expand-clauses (cond-clauses exp)))

(define (expand-clauses clauses)
  (if (null? clauses) 'false 
      ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF" clauses))
            (make-if (cond-predicate first)
                        (sequence->exp (cond-actions first))
                        (expand-clauses rest))))))

;;; end of Cond support