(defun startEval (e)

(xeval e nil nil)

)

;

; PURPOSE : Evaluate an expression

;

; NOTES : Expressions can have the following forms

;

; x - a variable

; (xquote s) - a constant

; (+ e1 e2)

; (- e1 e2)

; (\* e1 e2)

; (/ e1 e2)

; (rem e1 e2)

; (atom e1)

; (car e1)

; (cdr e1)

; (cons e1 e2)

; (eq e1 e2)

; (leq e1 e2)

; (if e1 e2 e3)

; (e e1 e2 ... en)

; (let (x1 x2 ... xk) (e1 e2 ... ek) e)

; (letrec (x1 x2 ... xk) (e1 e2 ... ek) e)

; (lambda (x1 x2 ... xn) e )

;

; NOTES :

; This implementation does not allow t or nil to be rebound

; (it does not look in the name and value list for their values)

; Similarily, integers are bound to themselves, and cannot be rebound.

; Unlike lisp, no error is generated by this code

(defun xeval (e n v)

(if (eq e t)

; t is bound to itself

t

(if (null e)

; nil is bound to itself

nil

(if (numberp e)

; numbers are bound to themselves

e

(if (atom e)

; a variable - return result of searching context

; for variable represented by 'e'.

(xassoc e n v)

(let ( (func-name (car e))

(e1 (car (cdr e)))

(e2 (car (cdr (cdr e))))

(e3 (car (cdr (cdr (cdr e)))))

)

;

; NOTE : e1, e2 and e3 will not be semantically

; meaningful for all functions. For example,

; the (lambda (x1 x2 ... xn) e ) expression

; does not have an expression as its first argument.

; For lambda, let, letrec and function calls, don't

; use these variables, since they are not meaningful.

;

(if (eq func-name 'xquote)

; return the first argument unevaluated

e1

(if (member func-name '(+ - \* / rem atom car cdr cons eq leq))

;

; Functions with 1 or 2 expressions as arguments,

; where each argument needs to be evaluated.

;

(let ( (ev-e1 (xeval e1 n v))

(ev-e2 (xeval e2 n v))

)

(if (eq func-name '+)

; return the sum of the two evaluated arguments.

; obviously, the evaluated arguments should be integers.

(+ ev-e1 ev-e2)

(if (eq func-name '-)

; return the difference of the two evaluated arguments.

; obviously, the evaluated arguments should be integers.

(- ev-e1 ev-e2)

(if (eq func-name '\*)

; return the product of the two evaluated arguments.

; The evaluated arguments should be integers.

(\* ev-e1 ev-e2)

(if (eq func-name '/)

; return the quotient of the two evaluated arguments.

; The evaluated arguments should be integers.

(/ ev-e1 ev-e2)

(if (eq func-name 'atom)

; if the evaluated argument is atomic, return constant T,

; otherwise return constant nil.

(if (atom ev-e1)

t

nil

)

(if (eq func-name 'car)

; return the first element in the evaluated

; arguments list. If ev-e1 is an atom,

; return nil (not the constant nil, just nil -

; this will allow callers to test for execution

; errors).

(if (atom ev-e1)

nil

(car ev-e1)

)

(if (eq func-name 'cdr)

; return the list represented by ev-e1 with its first

; element removed. If ev-e1 is an atom,

; return nil (not the constant nil, just nil -

; this will allow callers to test for execution

; errors)

(if (atom ev-e1)

nil

(cdr ev-e1)

)

(if (eq func-name 'cons)

; add the evaluated first argument as the first

; element of the evaluated second argument.

; If the second element is an atom, how do

; we represent this? Leave it up to list.

(cons ev-e1 ev-e2)

(if (eq func-name 'eq)

; return true if two evaluate arguments

; are pointer equal, false otherwise

(eq ev-e1 ev-e2)

(if (eq func-name 'leq)

; return true if first evaluated argument

; is less than second evaluated argument,

; false otherse. Obviously, arguments should

; be integers when evaluated.

(leq ev-e1 ev-e2)

))))))))))

)

;

; Functions with other argument formats

;

(let ()

(if (eq func-name 'if)

; If first argument evaluate to true,

; return evaluated second argument,

; otherwise return evaluted third argument.

(if (xeval e1 n v)

(xeval e2 n v)

(xeval e3 n v)

)

(if (eq func-name 'let)

; Simply add the names and values defined in the let

; to the current name and value lists, then evaluate

; the body of the let.

;

(let ((new-names (cadr e))

(new-values (caddr e))

(body (cadddr e)))

(let ( (all-names (cons new-names n))

(ev-parms (evlis new-values n v)) )

(let ((all-values (cons ev-parms v)))

(xeval body all-names all-values)

)

)

)

(if (eq func-name 'letrec)

; Do the Henderson rplaca trick so recursive name bindings

; are possible.

;

(let ((all-values (cons '(PENDING) v)))

(let ((new-names (cadr e))

(parm-exprs (caddr e))

(body (cadddr e)))

(let ((all-names (cons new-names n)))

(let ((ev-pending-parms

(evlis parm-exprs

all-names all-values)))

(rplaca all-values ev-pending-parms)

(xeval body all-names all-values)

)

)

)

)

(if (eq func-name 'lambda)

(let ( (arg-list (car (cdr e)))

(body (car (cdr (cdr e))))

)

; Return the closure - abstracted to allow

; different implementations.

(closure arg-list body n v)

)

; else ASSUME a function call.

(let ((ev-args (evlis (cdr e) n v))

(closure (xeval func-name n v)))

(let ((new-names

(cons (closure-parameters closure) (closure-names closure)))

(new-values

(cons ev-args (closure-values closure)))

(body (closure-body closure))

)

(xeval body new-names new-values)

)

)

))))

)

))

)

))))

)

;

; n is a namelist in the form :

; ( (var1.1 var1.2 ... var1.n ) (var2.1 ... var2.k ) ... (varm.1 ... varm.j))

; where each var is a symbol.

;

; l is a valuelist parallel to n

; representing the value of the

; corresponding variable.

;

(defun xassoc (var n v)

(if (null n)

nil

; else

(if (member var (car n))

(locate var (car n) (car v))

; else

(xassoc var (cdr n) (cdr v))

)

)

)

;

; Note that 'locate' is NEVER called unless

; 'var' is guaranteed to be in the

; name-sublist.

(defun locate (var name-sublist value-sublist)

(if (eq var (car name-sublist))

(car value-sublist)

; else

(locate var (cdr name-sublist) (cdr value-sublist))

)

)

(defun evlis ( arg-list n v)

(if (null arg-list)

nil

(cons (xeval (car arg-list) n v) (evlis (cdr arg-list) n v))

)

)

(defun closure (arg-list body n v)

(cons (cons arg-list body) (cons n v))

)

(defun closure-parameters (c)

(car (car c))

)

(defun closure-body (c)

(cdr (car c))

)

(defun closure-values (c)

(cdr (cdr c))

)

(defun closure-names (c)

(car (cdr c))

)

(defun leq (x y)

(or (< x y) (equal x y))

)