;A17

(load "chez-init.ss")

**; DATATYPES**

; parsed expression

(define (implist-of pred?)

(lambda (implst)

(let helper ([ls implst])

(or (null? ls) (pred? ls)

(and (pred? (car ls)) (helper (cdr ls)))))))

(define-datatype expression expression?

[var-exp

(id symbol?)]

[lambda-exp

(id (list-of symbol?))

(body (list-of expression?))]

[lambda-single-exp

(id symbol?)

(body (list-of expression?))]

[lambda-pair-exp

(id1 (list-of symbol?))

(id2 symbol?)

(body (list-of expression?))]

[prim-exp

(id symbol?)

(body (list-of expression?))]

[lit-exp

(datum

(lambda (x)

(ormap

(lambda (pred) (pred x))

(list number? vector? boolean? symbol? string? pair? null?))))]

[if-exp

(condition expression?)

(tcase expression?)

(other expression?)]

[if-one-exp

(condition expression?)

(tcase expression?)]

[let-exp

(syms (list-of symbol?))

(exps (list-of expression?))

(bodies (list-of expression?))]

[let-name-exp

(name symbol?)

(syms (list-of symbol?))

(exps (list-of expression?))

(bodies (list-of expression?))]

[let\*-exp

(syms (list-of symbol?))

(exps (list-of expression?))

(bodies (list-of expression?))]

[letrec-exp

(proc-names (list-of symbol?))

(idss (list-of (implist-of symbol?)))

(bodiess (list-of (list-of expression?)))

(letrec-bodies (list-of expression?))]

[cond-exp

(condition (list-of expression?))

(elsecase expression?)]

[begin-exp

(exps (list-of expression?))]

[or-exp

(exps (list-of expression?))]

[and-exp

(exps (list-of expression?))]

[case-exp

(calc expression?)

(condition (list-of expression?))

(result (list-of expression?))

(elsecase expression?)]

[while-exp

(condition expression?)

(exp (list-of expression?))]

[set!-exp

(id symbol?)

(exp expression?)]

[define-exp

(id symbol?)

(exp expression?)]

[app-exp

(rator expression?)

(rand (list-of expression?))])

;; environment type definitions

(define scheme-value?

(lambda (x) #t))

(define-datatype environment environment?

[empty-env-record]

[extended-env-record

(syms (implist-of symbol?))

(vals (list-of scheme-value?))

(env environment?)]

[recursively-extended-env-record

(proc-names (list-of symbol?))

(idss (list-of (implist-of symbol?)))

(bodiess (list-of (list-of expression?)))

(env environment?)])

; datatype for procedures. At first there is only one

; kind of procedure, but more kinds will be added later.

(define-datatype proc-val proc-val?

[prim-proc

(name symbol?)]

[closure

(syms (lambda (n) (or (or (symbol? n) ((list-of symbol?) n)) (pair? n))))

(bodies (list-of expression?))

(env environment?)]

[closure-single

(sym symbol?)

(bodies (list-of expression?))

(env environment?)]

[closure-pair

(syms (lambda (n) (or (or (symbol? n) ((list-of symbol?) n)) (pair? n))))

(bodies (list-of expression?))

(env environment?)])

**; PARSER**

; This is a parser for simple Scheme expressions, such as those in EOPL, 3.1 thru 3.3.

; You will want to replace this with your parser that includes more expression types, more options for these types, and error-checking.

; Procedures to make the parser a little bit saner.

(define 1st car)

(define 2nd cadr)

(define 3rd caddr)

(define parse-exp

(lambda (datum)

(cond

[(null? datum) datum]

[(number? datum) (lit-exp datum)]

[(boolean? datum) (lit-exp datum)]

[(string? datum) (lit-exp datum)]

[(vector? datum) (lit-exp datum)]

[(symbol? datum) (var-exp datum)]

[(pair? datum)

(cond

[(equal? (1st datum) 'quote)

(lit-exp `,(2nd datum))];get help from Jizhou Huang on `

[(equal? (1st datum) 'lambda)

(cond [(list? (2nd datum))

(lambda-exp (2nd datum)

(map parse-exp (cddr datum)))]

[(symbol? (2nd datum))

(lambda-single-exp (2nd datum)

(map parse-exp (cddr datum)))]

[else (lambda-pair-exp (get-before-dot (2nd datum))

(after-dot (2nd datum))

(map parse-exp (cddr datum)))])]

[(equal? (1st datum) 'set!)

(set!-exp (2nd datum)

(parse-exp (3rd datum)))]

[(equal? (1st datum) 'if)

(if (equal? (length datum) 4)

(if-exp (parse-exp (2nd datum))

(parse-exp (3rd datum))

(parse-exp (cadddr datum)))

(if-one-exp (parse-exp (2nd datum))

(parse-exp (3rd datum))))]

[(equal? (1st datum) 'let)

(if (symbol? (2nd datum))

(let-name-exp (2nd datum) (map car (3rd datum)) (map parse-exp (map cadr (3rd datum))) (map parse-exp (cdddr datum)))

(let-exp (map car (2nd datum)) (map parse-exp (map cadr (2nd datum))) (map parse-exp (cddr datum))))]

[(equal? (1st datum) 'let\*)

(let\*-exp (map car (2nd datum)) (map parse-exp (map cadr (2nd datum))) (map parse-exp (cddr datum)))]

[(equal? (1st datum) 'letrec)

(letrec-exp (map 1st (2nd datum))

(map (lambda (n) (choose-id-lambda (parse-exp n))) (map 2nd (2nd datum)))

(map (lambda (n) (choose-body-lambda (parse-exp n))) (map 2nd (2nd datum)))

(map parse-exp (cddr datum)))]

[(equal? (1st datum) 'cond)

(let ([else-part (else-cond (cdr datum))])

(if (equal? (car else-part) 'else)

(cond-exp (map parse-exp (first-cond (cdr datum))) (parse-exp (2nd else-part)))

(cond-exp (map parse-exp (cdr datum)) (parse-exp 'void))))] ;Get the void from Yuqi Zhou

[(equal? (1st datum) 'begin)

(begin-exp (map parse-exp (cdr datum)))]

[(equal? (1st datum) 'or)

(or-exp (map parse-exp (cdr datum)))]

[(equal? (1st datum) 'and)

(and-exp (map parse-exp (cdr datum)))]

[(equal? (1st datum) 'case)

(case-exp (parse-exp (2nd datum))

(map (lambda (x) (parse-exp (1st x))) (first-cond (cddr datum)))

(map (lambda (x) (parse-exp (2nd x))) (first-cond (cddr datum)))

(parse-exp (cadr (else-cond (cddr datum)))))]

[(equal? (1st datum) 'while)

(while-exp (parse-exp (2nd datum)) (map parse-exp (cddr datum)))]

[(equal? (1st datum) 'define)

(define-exp (2nd datum) (syntax-expand (parse-exp (3rd datum))))]

[(number? (1st datum))

(lit-exp datum)]

[else (app-exp (parse-exp (1st datum))

(map parse-exp (cdr datum)))])]

[else (eopl:error 'parse-exp "bad expression: ~s" datum)])))

(define choose-body-lambda

(lambda (exp)

(cases expression exp

[lambda-exp (ids body)

body]

[lambda-single-exp (id body)

body]

[lambda-pair-exp (id1 id2 body)

body]

[else exp])))

(define choose-id-lambda

(lambda (exp)

(cases expression exp

[lambda-exp (ids body)

ids]

[lambda-single-exp (id body)

id]

[lambda-pair-exp (id1 id2 body)

(append id1 id2)]

[else exp])))

(define get-before-dot

(lambda (exp)

(if (symbol? (cdr exp))

(list (car exp))

(cons (car exp) (get-before-dot (cdr exp))))))

(define after-dot

(lambda (exp)

(if (symbol? (cdr exp))

(cdr exp)

(after-dot (cdr exp)))))

(define first-cond

(lambda (exp)

(if (null? (cdr exp))

'()

(cons (car exp) (first-cond (cdr exp))))))

(define else-cond

(lambda (exp)

(if (null? (cdr exp))

(car exp)

(else-cond (cdr exp)))))

**; ENVIRONMENTS |**

; Environment definitions for CSSE 304 Scheme interpreter.

; Based on EoPL sections 2.2 and 2.3

(define cell

(lambda (x)

(cons x 'this-is-a-cell)))

(define cell-ref car)

(define cell-set! set-car!)

(define cell?

(lambda (obj)

(and (pair? obj) (eq? (cdr obj) 'this-is-a-cell))))

(define deref cell-ref)

(define set-ref! cell-set!)

(define empty-env

(lambda ()

(empty-env-record)))

(define extend-env

(lambda (syms vals env)

(extended-env-record syms (map cell vals) env)))

(define list-find-position

(lambda (sym los)

(list-index (lambda (xsym) (eqv? sym xsym)) los)))

(define list-index

(lambda (pred ls)

(cond

((null? ls) #f)

((pred (car ls)) 0)

(else (let ((list-index-r (list-index pred (cdr ls))))

(if (number? list-index-r)

(+ 1 list-index-r)

#f))))))

(define apply-env-ref

(lambda (env sym succeed fail) ; succeed and fail are "callback procedures,

(cases environment env ; succeed is appluied if sym is found, otherwise

[empty-env-record () ; fail is applied.

(fail)]

[extended-env-record (syms vals env)

(let ((pos (list-find-position sym syms)))

(if (number? pos)

(succeed (list-ref vals pos))

(apply-env-ref env sym succeed fail)))]

[recursively-extended-env-record (procnames idss bodiess old-env)

(let ([pos (list-find-position sym procnames)])

(if (number? pos)

(let ([ids (list-ref idss pos)])

(if (list? ids)

(closure ids (list-ref bodiess pos) env)

(closure-pair (append (list (car ids)) (list (cdr ids))) (list-ref bodiess pos) env)))

(apply-env-ref old-env sym succeed fail)))])))

(define apply-env

(lambda (env var succeed fail)

(deref (apply-env-ref env var succeed fail))))

(define extend-env-recursively

(lambda (proc-names idss bodiess old-env)

(recursively-extended-env-record proc-names idss bodiess old-env)))

(define reset-global-env

(lambda ()

(set! init-env

(extend-env \*prim-proc-names\*

(map prim-proc \*prim-proc-names\*)

(empty-env)))))

**; SYNTAX EXPANSION**

(define syntax-expand

(lambda (exp)

(cases expression exp

[lambda-exp (ids body)

(lambda-exp ids (map syntax-expand body))]

[lambda-single-exp (id body)

(lambda-single-exp id (map syntax-expand body))]

[lambda-pair-exp (id1 id2 body)

(lambda-pair-exp id1 id2 (map syntax-expand body))]

[if-exp (condition tcase other)

(if-exp (syntax-expand condition) (syntax-expand tcase) (syntax-expand other))]

[if-one-exp (condition tcase)

(if-one-exp (syntax-expand condition) (syntax-expand tcase))]

[let-exp (syms exps bodies)

(app-exp (lambda-exp syms (map syntax-expand bodies)) (map syntax-expand exps))]

[let-name-exp (name syms exps bodies)

(syntax-expand (letrec-exp (list name) (list syms) (list bodies) (list (app-exp (var-exp name) exps))))]

[let\*-exp (syms exps bodies)

(syntax-expand (if (null? (cdr syms))

(let-exp (list (car syms)) (list (car exps)) bodies)

(let-exp (list (car syms)) (list (car exps)) (list (let\*-exp (cdr syms) (cdr exps) bodies)))))]

[begin-exp (exps)

(app-exp (lambda-exp (list) (map syntax-expand exps)) (list))];get help from Yiyu Ma

[or-exp (exps)

(if (null? exps)

(lit-exp #f)

(syntax-expand (let-exp '(jyhtbgvfc) (list (1st exps)) (list (if-exp (var-exp 'jyhtbgvfc) (var-exp 'jyhtbgvfc) (or-exp (cdr exps)))))))]

[and-exp (exps)

(if (null? exps)

(lit-exp #t)

(syntax-expand (let-exp '(kujyhtgrfed) (list (1st exps)) (list (if-exp (var-exp 'kujyhtgrfed) (and-exp (cdr exps)) (lit-exp #f))))))]

[cond-exp (condition elsecase)

(if (null? condition)

(syntax-expand elsecase)

(if (and (equal? elsecase (void)) (null? (cdr condition)))

(syntax-expand (if-one-exp (cadar condition) (car (caddar condition))))

(syntax-expand (if-exp (cadar condition) (car (caddar condition)) (cond-exp (cdr condition) elsecase)))))]

[case-exp (calc condition result elsecase)

(if (null? condition)

elsecase

(syntax-expand (if-exp (app-exp (var-exp 'member) (list calc (car condition)))

(car result)

(case-exp calc (cdr condition) (cdr result) elsecase))))]

[app-exp (rator rands)

(app-exp (syntax-expand rator) (map syntax-expand rands))]

[define-exp (id exp)

(define-exp id (syntax-expand exp))]

[var-exp (id)

exp]

[prim-exp (id body)

(prim-exp id body)]

[lit-exp (id)

exp]

[letrec-exp (proc-names idss bodiess letrec-bodies)

(letrec-exp proc-names idss (map (lambda (n) (map syntax-expand n)) bodiess) (map syntax-expand letrec-bodies))]

[while-exp (cond exp)

(while-exp cond exp)]

[set!-exp (id exp)

(set!-exp id (syntax-expand exp))])))

**; INTERPRETER**

; top-level-eval evaluates a form in the global environment

(define top-level-eval

(lambda (input)

; later we may add things that are not expressions.

(eval-exp input (empty-env))))

; eval-exp is the main component of the interpreter

(define eval-exp

(lambda (exp env)

(cases expression exp

[lit-exp (datum) datum]

[var-exp (id)

(apply-env-ref env

id; look up its value.

(lambda (x) (deref x)) ; procedure to call if id is in the environment

(lambda ()

(apply-env-ref init-env

id

(lambda (x) (deref x))

(lambda () (eopl:error 'apply-env ; procedure to call if id not in env

"variable not found in environment: ~s" id)))))]

[app-exp (rator rands)

(let ([proc-value (eval-exp rator env)]

[args (eval-rands rands env)])

(apply-proc proc-value args))]

[if-exp (condition tcase other)

(if (eval-exp condition env)

(eval-exp tcase env)

(eval-exp other env))]

[if-one-exp (condition tcase)

(if (eval-exp condition env)

(eval-exp tcase env))]

[lambda-exp (id body)

(closure id body env)]

[lambda-single-exp (id body)

(closure-single id body env)]

[lambda-pair-exp (id1 id2 body)

(closure-pair (append id1 (list id2)) body env)]

[prim-exp (id body)

(apply-prim-proc id (map (lambda (x) (eval-exp x env)) body))]

[while-exp (condition exp)

(if (eval-exp condition env) (begin (eval-bodies exp env) (eval-exp (while-exp condition exp) env)))]

[let-exp (syms exps bodies)

(let ([extended-env

(extend-env syms

(map (lambda (x) (eval-exp x env)) exps) env)])

(eval-bodies bodies extended-env))]

[letrec-exp (proc-names idss bodiess letrec-bodies)

(eval-bodies letrec-bodies

(extend-env-recursively proc-names

idss

bodiess

env))]

[cond-exp (condition elsecase)

(eval-exp (syntax-expand (cond-exp condition elsecase)) env)]

[let-name-exp (name syms exps bodies)

(eval-exp (syntax-expand (let-name-exp name syms exps bodies)) env)]

[set!-exp (id exp)

(let ([result (apply-env-ref env

id

(lambda (x) x)

(lambda ()

(apply-env-ref init-env

id

(lambda (x) x)

(lambda () #f))))])

(if (null? result)

(set! init-env (extend-env (list id) (list (eval-exp exp env)) init-env))

(set-ref! result (eval-exp exp env))))]

;define from lexical, but does not always works

; [define-exp (id exp)

; (set! init-env (list (1st init-env)

; (cons (3rd id) (2nd init-env))

; (cons (cell (eval-exp exp init-env)) (3rd init-env))

; (cadddr init-env)))]

[define-exp (id exp)

(set! init-env (extend-env (list id) (list (eval-exp exp init-env)) init-env))

;(display init-env)

]

[else (eopl:error 'eval-exp "Bad abstract syntax: ~a" exp)])))

; evaluate the list of operands, putting results into a list

(define eval-rands

(lambda (rands env)

(map (lambda (e)

(eval-exp e env)) rands)))

; Apply a procedure to its arguments.

; At this point, we only have primitive procedures.

; User-defined procedures will be added later.

(define apply-proc

(lambda (proc-value args)

(cases proc-val proc-value

[prim-proc (op) (apply-prim-proc op args)]

; You will add other cases

[closure (ids bodies env)

(eval-bodies bodies

(extend-env ids

args

env))]

[closure-single (id bodies env)

(eval-bodies bodies

(extend-env (list id)

(list args)

env))]

[closure-pair (ids bodies env)

(eval-bodies bodies

(extend-env ids

(make-imporper-lambda-args ids args)

env))]

[else (eopl:error 'apply-proc

"Attempt to apply bad procedure: ~s"

proc-value)])))

(define make-imporper-lambda-args

(lambda (id args)

(if (null? (cdr id))

(list args)

(cons (car args) (make-imporper-lambda-args (cdr id) (cdr args))))))

(define eval-bodies

(lambda (bodies env)

(if (null? (cdr bodies))

(eval-exp (car bodies) env)

(begin (eval-exp (car bodies) env)

(eval-bodies (cdr bodies) env)))))

(define \*prim-proc-names\* '(+ - \* / add1 sub1 zero? not = < > <= >= cons car cdr list null?

assq eq? equal? atom? length list->vector list? pair? procedure?

vector->list vector make-vector vector-ref vector? number? symbol?

set-car! set-cdr! vector-set! display newline cadr caar cdar cadar

apply map member quotient list-tail eqv? append))

(define init-env ; for now, our initial global environment only contains

(extend-env ; procedure names. Recall that an environment associates

\*prim-proc-names\* ; a value (not an expression) with an identifier.

(map prim-proc

\*prim-proc-names\*)

(empty-env)))

; Usually an interpreter must define each

; built-in procedure individually. We are "cheating" a little bit.

(define apply-prim-proc

(lambda (prim-proc args)

(case prim-proc

[(+) (apply + args)]

[(-) (apply - args)]

[(\*) (apply \* args)]

[(/) (apply / args)]

[(add1) (+ (1st args) 1)]

[(sub1) (- (1st args) 1)]

[(zero?) (zero? (1st args))]

[(not) (not (1st args))]

[(=) (= (1st args) (2nd args))]

[(<) (< (1st args) (2nd args))]

[(>) (> (1st args) (2nd args))]

[(<=) (<= (1st args) (2nd args))]

[(>=) (>= (1st args) (2nd args))]

[(cons) (cons (1st args) (2nd args))]

[(car) (car (1st args))]

[(cdr) (cdr (1st args))]

[(list) args]

[(null?) (null? (1st args))]

[(assq) (assq (1st args) (2nd args))]

[(eq?) (eq? (1st args) (2nd args))]

[(eqv?) (eqv? (1st args) (2nd args))]

[(equal?) (equal? (1st args) (2nd args))]

[(atom?) (atom? (1st args))]

[(length) (length (1st args))]

[(list->vector) (list->vector (1st args))]

[(list?) (list? (1st args))]

[(pair?) (pair? (1st args))]

[(procedure?) (proc-val? (1st args))]

[(vector->list) (vector->list (1st args))]

[(vector) (list->vector args)]

[(make-vector) (make-vector (1st args))]

[(vector-ref) (vector-ref (1st args) (2nd args))]

[(vector?) (vector? (1st args))]

[(number?) (number? (1st args))]

[(symbol?) (symbol? (1st args))]

[(set-car!) (set-car! (1st args) (2nd args))]

[(set-cdr!) (set-cdr! (1st args) (2nd args))]

[(vector-set!) (vector-set! (1st args) (2nd args) (3rd args))]

[(display) (display (1st args))]

[(newline) (newline)]

[(cadr) (cadr (1st args))]

[(caar) (caar (1st args))]

[(cdar) (cdar (1st args))]

[(cadar) (cadar (1st args))]

[(apply) (apply-proc (1st args) (2nd args))]

[(map) (map (lambda (n) (apply-proc (1st args) (list n))) (2nd args))]

[(member) (member (1st args) (2nd args))]

[(quotient) (quotient (1st args) (2nd args))]

[(list-tail) (list-tail (1st args) (2nd args))]

[(append) (append (1st args) (2nd args))]

[else (error 'apply-prim-proc

"Bad primitive procedure name: ~s"

prim-proc)])))

(define rep ; "read-eval-print" loop.

(lambda ()

(display "--> ")

;; notice that we don't save changes to the environment...

(let ([answer (top-level-eval (parse-exp (read)))])

;; TODO: are there answers that should display differently?

(eopl:pretty-print answer) (newline)

(rep)))) ; tail-recursive, so stack doesn't grow.

(define eval-one-exp

(lambda (x) (top-level-eval (syntax-expand (parse-exp x)))))