; CSE341, Programming Languages
; Lecture 17: Implementing Languages Including Closures

#lang racket

(provide (all-defined-out))

; a larger language with two kinds of values, booleans and numbers
; an expression is any of these:
(struct const (int) #transparent) ; int should hold a number
(struct negate (el) #transparent) ; el should hold an expression
(struct add (el e2) #transparent) ; el, e2 should hold expressions
(struct multiply (el e2) #transparent) ; el, e2 should hold expressions
(struct boolean (b) #transparent) ; b should hold #t or #f
(struct equal? (el e2) #transparent) ; el, e2 should hold expressions
; a value in this language is a legal const or bool

; And example of our MUPL expression
(define exp (eq-num (const 12) (add (const 16) (negate (const 4)))))

; Here are some Racket functions that given language-being-implemented syntax,
; produce language-being-implemented syntax
(define (andalso el e2)
  (if (then-else el e2 (bool #f))))

(define (double e)
  (multiply e (const 2)))

(define (negative-x-plus-one e)
  (negate (add (const 1) e)))

; this one takes a Racket list of language-being-implemented syntax
; and makes language-being-implemented syntax
(define (list-product es)
  (if (null? es)
      (const 1)
      (multiply (car es) (list-product (cdr es)))))

(define test (andalso (eq-num (double (const 4))
                           (list-product (list (const 2) (const 2) (const 1) (const 2)))
                           (bool #t)))

; we will talk about this eval function tomorrow
(define (eval-exp e)
  (cond
    [(const? e) e]
    [(negate? e) (let ([v (eval-exp (negate-el e))])
                   (if (const? v)
                       (const (- (const-int v)))
                       (error "negate applied to non-number")))]
    [(add? e) (let ([v1 (eval-exp (add-el e))]
                    [v2 (eval-exp (add-e2 e))]
                    (if (and (const? v1) (const? v2))
                        (const (+ (const-int v1) (const-int v2)))
                        (error "add applied to non-number")))]
    [(multiply? e) (let ([v1 (eval-exp (multiply-el e))]
                          [v2 (eval-exp (multiply-e2 e))]
                          (if (and (const? v1) (const? v2))
                              (const (* (const-int v1) (const-int v2)))
                              (error "multiply applied to non-number")))]
    [(boolean? e) e]
    [(eq-num? e) (let ([v1 (eval-exp (eq-num-el e))])
                   (if (and (const? v1) (const? v2))
                       (bool (= (const-int v1) (const-int v2)))
                       (error "eq-num applied to non-number")))]
    [(if-then-else? e)
     (let ([v-test (eval-exp (if-then-else-e1 e))]
           [v (if (bool? v-test)
                  (eval-exp (if-then-else-e2 e))
                  (eval-exp (if-then-else-e3 e))]
           (error "if-then-else applied to non-boolean")))]
    [(else? (error "eval-exp expected an exp")) ; not strictly necessary but helpful for debugging]
  )

; notice we have not have to change our interpreter at all
(define result (eval-exp test))

; quoting

(define 10 (list 1 2 (list 3 4 5))
(define 11 (quote (1 2 (3 4 5)))
(define 12 '(1 2 (3 4 5))
(define s0 (list 1 2 (+ 1 2))
(define s1 '(1 2 (+ 1 2))
(define s2 (quasiquote (1 2 (unquote (+ 1 2))))
(define s3 '(1 2 ,(+ 1 2))

; A more useful thing to do: partial evaluation
;
; We might want to specialize the pow function
;
; normal version below, calculates x ^ y
;
; never call with y < 1
(define (pow-normal x y)
  (if (eq? y 1)
      x
      (* x (pow-normal (- 1) x))))

; Now suppose we want a version specialized to a particular y
; Here we can specialize for a given y, and get back a function
; which doesn't use recursion
;
; pow x y is x ^ y
; never call with y < 1
(define (pow y)
  (if (eq? y 1)
      (quote x)
      (quasiquote (* x (unquote (powh (- 1) x))))))

(define (pow y)
  (let ([mul-exp (powh y)]
         [eval (quasiquote (lambda (x) (unquote mul-exp)))]
         (for-each mul-exp y)
         (cons y))
  )

; Racket is cool, and lets us do something even simpler
(define (pow2 y)
  (let ([xs (build-list y (lambda (z) 'x))]
         [cons '*']
         (eq-apply xs))

; define (pow2 y)
  (let ([mul-exp (powh2 y)])

Winter 2018 CSE 341: Programming Languages
(eval (quasiquote (lambda (x) (unquote mul-exp)))))

;; Another example below, probably no time in section
;
; eval
; ======
;
(setup
(current-namespace (make-base-namespace))

(define (join-symbols s0 s1)
  (string->symbol
    (string-append (symbol->string s0)
                  (symbol->string s1))))

; quoting, simple example
(define f (lambda (x) (+ x x)))

(define f-data '(lambda (x) (+ x x)))

(define f-from-data (eval f-data))

; (quasi)quoting, realistic example

(struct employee (name age salary manager address jobtitle hiredate vacation sickdays overtime))

(define-name-space-anchor a)
(current-namespace (namespace-anchored->namespace a))

(define me (employee "Tam" 20 1000000 'no "Somewhere" "TA" 2017 10 2 20))

(define print-employee (lambda (e)
  (printf "-a: ~a\n" "name" (employee-name e))
  (printf "-a: ~a\n" "age" (employee-age e))
  (printf "-a: ~a\n" "salary" (employee-salary e))
  (printf "-a: ~a\n" "manager" (employee-manager e))
  (printf "-a: ~a\n" "address" (employee-address e))
  (printf "-a: ~a\n" "jobtitle" (employee-jobtitle e))
  (printf "-a: ~a\n" "hiredate" (employee-hiredate e))
  (printf "-a: ~a\n" "vacation" (employee-vacation e))
  (printf "-a: ~a\n" "sickdays" (employee-sickdays e))
  (printf "-a: ~a\n" "overtime" (employee-overtime e))))

(define print-employee-data
  (append '(lambda (e))
    (map (lambda (field)
           (printf "-a: ~a\n", (symbol->string field) ,((join-symbols 'employee
             field) e))
           '((name age salary manager address jobtitle hiredate vacation sickdays
              overtime))))))

(define print-employee-from-data (eval print-employee-data))