; 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 el2) #transparent) ; el, el2 should hold expressions
(struct multiply (el el2) #transparent) ; el, el2 should hold expressions
(struct bool (b) #transparent) ; b should hold #t or #f
(struct eqqnum (el el2) #transparent) ; el, el2 should hold expressions
(struct if-then-else (el el2 el3) #transparent) ; el, el2, el3 should hold expressions

; a value in this language is a legal const or bool

define test1 (multiply (negate (add (const 2)))
  (const 3)))

(define test2 (multiply (negate (add (const 2)))
  (if-then-else (bool #f)
    (const 7))
  (bool #t)))

(define non-test (multiply (negate (add (const #t)))
  (const 2)))

(define (eval-exp-wrong e)
  (cond [[const? e] e]
    [(negate? e) (const (- (const-int (eval-exp-wrong (negate-el e)))))]
    [(add? e) (let ([i1 (const-int (eval-exp-wrong (add-el e)))]
                  [i2 (const-int (eval-exp-wrong (add-el e)))]
                  (const (+ i1 i2)))]
    [(multiply? e) (let ([i1 (const-int (eval-exp-wrong (multiply-el e)))]
                           [i2 (const-int (eval-exp-wrong (multiply-el e)))]
                           (const (* i1 i2)))
      (bool? e)
      [(eqqnum? e) (let ([i1 (const-int (eval-exp-wrong (eqqnum-el e)))]
                           [i2 (const-int (eval-exp-wrong (eqqnum-el e)))]
                           (bool (= i1 i2)))]
        (creates (bool #t) or (bool #f)]
      [(if-then-else? e) (if (bool-b (eval-exp-wrong (if-then-else-el e)))
                               (eval-exp-wrong (if-then-else-el e))
                               (eval-exp-wrong (if-then-else-e3 el)))]
      [else (error "eval-exp-wrong")]]
  ; not strictly necessary but helps debugging)

; Here are two Racket functions that given language-being-implemented syntax,
; produce language-being-implemented syntax
(define (andalso el el2)
  (if (and EL el2) (bool #f))
  (define (double e)
    (multiply e (const 2)))

; 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 (eqqnum (double (const 4))
                           (list-product (list (const 2) (const 2) (const 1)
                                          (const 2)))))

(define result (eval-exp test))