; CSE341, Programming Languages
; Lecture 16: Datatype-Style Programming With Lists or Structs
#lang racket

(provide (all-defined-out))

; we do not need a datatype to create a list holding a mix of numbers and string
; Note: arguably bad style to not have else clause but makes example more like ML code
(define (funny-sum xs)
  (cond [(null? xs) 0]
        [(number? (car xs)) (+ (car xs) (funny-sum (cdr xs)))]
        [(string? (car xs)) (+ (string-length (car xs)) (funny-sum (cdr xs)))]))

; suppose we wanted a "one-of type" using nothing but lists and symbols and such
; like ML's
; datatype exp = Const of int | Negate of exp | Add of exp * exp | Multiply of e
; exp * exp

; just helper functions that make lists where first element is a symbol
; Note: More robust could check at run-time the type of thing being put in
(define (Const i) (list 'Const i))
(define (Negate e) (list 'Negate e))
(define (Add e1 e2) (list 'Add e1 e2))
(define (Multiply e1 e2) (list 'Multiply e1 e2))

; just helper functions that test what "kind of exp"
; Note: More robust could raise better errors for non-exp values
(define (Constr? x) (eq? (car x) 'Const))
(define (Neg? x) (eq? (car x) 'Negate))
(define (Add? x) (eq? (car x) 'Add))
(define (Multiply? x) (eq? (car x) 'Multiply))

; just helper functions that get the pieces for "one kind of exp"
; Note: More robust could check "what kind of exp"
(define (Constr-int e) (car (cdr e)))
(define (Negate-e e) (car (cdr e)))
(define (Add-e1 e) (car (cdr e)))
(define (Add-e2 e) (car (cdr (cdr e))))
(define (Multiply-e1 e) (car (cdr e)))
(define (Multiply-e2 e) (car (cdr (cdr e))))

; fyi: there are built-in functions for getting 2nd, 3rd list elements that would
; have made the above simpler:
;(define Constr-int cdr)
;(define Negate-e cdr)
;(define Add-e1 cdr)
;(define Add-e2 caddr)
;(define Multiply-e1 cadr)
;(define Multiply-e2 caddr)

; same recursive structure as we have in ML, just without pattern-matching
; And one change from what we did before: returning an exp, in particular
; a Constant, rather than an int
(define (eval-expl e)
  (cond [(Constr? e) (e) ; note returning an exp, not a number
            (Negate-e e) (Constr-int (eval-expl (Negate-e e)))]
        [(Add? e) (let ([v1 (Constr-int (eval-expl (Add-e1 e)))]
                       [v2 (Constr-int (eval-expl (Add-e2 e)))]
                       [Const (+ v1 v2)])]
        [(Multiply? e) (let ([v1 (Constr-int (eval-expl (Multiply-e1 e)))]
                            [v2 (Constr-int (eval-expl (Multiply-e2 e)))]
                            [Const (* v1 v2)])]
        [
        ]))

(define a-test1 (eval-expl (Multiply (Negate (Add (Const 2) (Const 2))) (Const 7)))))

; same idea using Racket's structs, which are more convenient /and/ less