(provide (all-defined-out))

(define (f x) (+ x (* x y))) ; forward reference okay here
(define y 3)
(define z (+ y 4)); backward reference okay
(define w (+ v 4)); not okay (get an error)
(define v 5)
(define f 17); not okay: f already defined in this module

; what is c, d, and e?
(define b 3)
(define g (lambda (x) (* 1 (+ x b))))
(define c (+ b 4))
(set! b 5)
(define d (g 4))
(define e c)

; the truth about cons: it just makes a pair
(define pr (cons 1 (cons #:hi)))
(define lst (cons #:t (cons #:hi #:null)))
(define hi (cdr (cdr pr)))
(define hi-again (car (cdr (cdr lst))))
(define hi-again-shorter (cddr lst))
(define no (list? pr))
(define yes (pair? pr))
(define of-course (and (list? lst) (pair? lst)))
(define do-not-do-this (length pr))

; cons cells are immutable -- this does not change a cell’s contents
(define lst1 (cons 14 null))
(define allased_lst1 lst1)
(set! lst1 (cons 42 null))
(define fourteen (car allased_lst1))

; but since mutable pairs are useful, Racket has them too:
(define mcons mcar mcdr set-mcar set-mcdr)
(define mpr (mcons 1 (mcons #:hi #:null)))
(define circular-mcons (mcons #:hi #:null))
(define mpr "bye")
(define bye (mcdr (mcdr mpr)))

; Note: run-time error to use mcar on a cons or car on an mcons
; (mcar (cons 1 2)); nope!

; set! vs. set-mcar! / set-mcdr!
(define mp (mcons 1 2))
(define mp2 mp)
(set-mcdr! mp #:z); change first value in mutable pair mp to #:z
(set-mcdr! mp #:x); change second value in mutable pair mp to #:x
(set (mcdr mp #:z))

; Note that (set! mp ...),
; unlike (set-mcar! mp ...),
; did *not* modify the data structure!
; It just made mp point to a new object!

;;;;; zero-argument functions (thunks) delay evaluation
(define (factorial-normal x)
  (if (= x 0)
      1
      (* x (factorial-normal (- x 1)))))

(define (factorial-okay x)
  (my-if-strange-but-works (= x 0)
    1
    (* x
       (factorial-okay (- x 1)))))

; how do these procedures differ?
(define accumulate-1
  (let ((acc 0))
    (lambda (x)
      (begin
        (set! acc (+ acc x))
        acc))))

(define accumulate-2
  (lambda (x)
    (let ((acc 0))
      (begin
        (set! acc (+ acc x))
        acc))))

; this distinction really matters even without mutation...
; this is a silly addition function that purposely runs slowly for
; demonstration purposes
(define (slow-add x y)
  (letrec ((slow-id (lambda (y z)
                            (if (= z 0 z)
                                y
                                (slow-id y (- z 1)))))
            (+ (slow-id x 500000000)y))
    (slow-id y x))

; multiplies x and result of y-thunk, calling y-thunk x times
(define (my-mult x y-thunk) ;; assumes x is >= 0
  (cond [(= x 0) 0]
        [ (= x 1) (y-thunk)]
        [ (and (> x 1) (my-mult (- x 1) (y-thunk)))
          #t (+ (y-thunk) (my-mult (- x 1) y-thunk)))])

; these calls: great for 0, okay for 1, bad for > 1
;(my-mult 0 (lambda () (slow-add 3 4)))
;(my-mult 1 (lambda () (slow-add 3 4)))
;(my-mult 2 (lambda () (slow-add 3 4)))

; these calls: okay for all
;(my-mult 0 (let ((x (slow-add 3 4))) (lambda () x)))
;(my-mult 1 (let ((x (slow-add 3 4))) (lambda () x)))
;(my-mult 2 (let ((x (slow-add 3 4))) (lambda () x)))