### Racket Introduction

always make this the first (non-comment, non-blank) line of your file

```
#lang racket
```

not needed here, but a workaround so we could write tests in a second file

```
(define three 3)
```

see getting-started-with-Racket instructions for more explanation

```
(provide (all-defined-out))
```

basic definitions

```
(define five (+ three 2)) ; function call is (el e2 ... en): parens matter!
```

```
; basic function
(define cube1
  (lambda (x)
    (* x (* x x))))

; many functions, such as *, take a variable number of arguments
(define cube2
  (lambda (x)
    (* x x x)))

; syntactic sugar for function definitions
(define (cube3 x) (* x x x))
```

```
; conditional
(define (powl x y)
  (if (= y 0)
    1
    (* (powl x (- y 1)))))
```

```
; currying
(define pow2
  (lambda (x)
    (lambda (y)
      (powl x y))))
```

```
; sugar for currying (fairly new to Racket)
(define (pow2b x y) (powl x y))
```

```
; define three-to-the (pow2 3))
(define eightyone (three-to-the 4)) ; need exactly these parens
```

```
; list processing: null, cons, null?, car, cdr
; we won't use pattern-matching in Racket
(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs) (sum (cdr xs)))))
```

```
; (define (my-append xs ys) ; same as append already provided
  (if (null? xs)
      ys
      (cons (car xs) (my-append (cdr xs) ys))))
```

```
; (define (my-map f xs) ; same as map already provided
  (if (null? xs)
      null
      (cons (f (car xs)) (my-map f (cdr xs)))))
```

```
; better style: use cond instead of nested if
```

```
; sum3 is equivalent to suml above but better style
(define (sum3 xs)
  (cond [(null? xs) 0]
        [(number? (car xs)) (+ (car xs) (sum3 (cdr xs)))]
        [else (+ (sum3 (car xs)) (sum3 (cdr xs)))]))
```

### Dynamic Typing!!

```
; this function counts how many #f are in a (non-nested) list
; it uses the "controversial" idiom of anything not #f is true
```

```
; so why do this work (hint: it’s not recursive
; and there is no type system):
(define (fact-wrong1 n) (if (= n 0) 1 (* n (fact-wrong1 (- n 1)))))
```

```
; passing 5 arguments to if: =, n, 0, 1, ( ... )
; this is bad syntax
```

```
; (define (fact-wrong2 n) (if = n 0 1 (* n (fact-wrong2 (- n 1)))))
```

```
; calling n with zero arguments and also having an if
; this is not a legal definition: bad syntax
```

```
; (define fact-wrong3 (n) (if (= n 0) 1 (* n (fact-wrong3 (- n 1)))))
```

```
; calling multiply with three arguments, which would be fine
; except the second one is fact-wrong4
```

```
; (define fact-wrong4 n) (if (= n 0) 1 (* n (fact-wrong4 (- n 1)))))
```

```
; calling fact-wrong5 with zero arguments, calling result of that
; with n-1
```

```
; (define fact-wrong5 n) (if (= n 0) 1 (* n (fact-wrong5 (- n 1))))
```

```
; treating n as a function of two arguments, passing it *
```

```
; [second big difference from ML (and Java)] Dynamic Typing!!
```

```
; dynamic typing: can use values of any type anywhere
; e.g., a list that holds numbers or other lists
```

```
; this function sums lists of (numbers or lists of (numbers or ...)),
; but it does assume it only encounters lists or numbers (else run-time error)
```

```
; (define (sum1 xs)
    (if (null? xs)
        0
        (if (number? (car xs))
            (+ (car xs) (sum1 (cdr xs)))
            (+ (sum1 (car xs)) (sum1 (cdr xs))))))
```

```
; this version does not fail on non-lists -- it treats them as 0
```

```
; (define (sum2 xs)
    (if (null? xs)
        0
        (if (number? (car xs))
            (+ (car xs) (sum2 (cdr xs)))
            (+ (sum2 (car xs)) (sum2 (cdr xs))))))
```

```
; ; [first big difference from ML (and Java)] PARENS MATTER!!
```

```
; (define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
```

```
; ; so why do this work (hint: it’s not recursive
```

```
; ; and there is no type system:
```

```
; ; passing 5 arguments to if: =, n, 0, 1, ( ... )
```

```
; ; this is bad syntax
```

```
; ; (define (fact-wrong2 n) (if = n 0 1 (* n (fact-wrong2 (- n 1)))))
```

```
; ; calling n with zero arguments and also having an if
```

```
; ; this is not a legal definition: bad syntax
```

```
; ; (define fact-wrong3 (n) (if (= n 0) 1 (* n (fact-wrong3 (- n 1)))))
```

```
; ; calling multiply with three arguments, which would be fine
```

```
; ; except the second one is fact-wrong4
```

```
; ; (define fact-wrong4 n) (if (= n 0) 1 (* n (fact-wrong4 (- n 1)))))
```

```
; ; calling fact-wrong5 with zero arguments, calling result of that
```

```
; ; with n-1
```

```
; ; (define fact-wrong5 n) (if (= n 0) 1 (* n (fact-wrong5 (- n 1))))
```

```
; ; treating n as a function of two arguments, passing it *
```

```
; ; [second big difference from ML (and Java)] Dynamic Typing!!
```

```
; ; dynamic typing: can use values of any type anywhere
```

```
; ; e.g., a list that holds numbers or other lists
```

```
; ; this function sums lists of (numbers or lists of (numbers or ...)),
```

```
; ; but it does assume it only encounters lists or numbers (else run-time error)
```

```
; ; (define (sum1 xs)
```

```
; ; (define (sum2 xs)
```

```
; ; ; [first big difference from ML (and Java)] PARENS MATTER!!
```

```
; ; (define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
```
(car xs) (count-falses (cdr xs))); (car xs) can have any type
 #t (+ 1 (count-falses (cdr xs))))

; different kinds of local bindings
(define (max-of-list xs)
  (cond [(null? xs) (error "max-of-list given empty list")]
        [(null? (cdr xs)) (car xs)]
        [($t (let ([trns (max-of-list (cdr xs))])
                      (if (> trns (car xs))
                          trns
                          (car xs)))]))

; let evaluates all expressions using outer environment,
; *not* earlier bindings
(define (double1 x)
  (let ([x (+ x 1])
         [y (+ x 2)])
    (+ x y -5)))

; let* is like ML's let: environment includes previous bindings
(define (double2 x)
  (let* ([x (+ x 3])
         [y (+ x 2)])
    (+ x y -8)))

; letrec uses an environment where all bindings in scope
; * like ML's use of and for mutual recursion
; * you get #<undefined> if you use a variable before it's defined
; * where as always function bodies not used until called
; (bindings still evaluated in order)
(define (triple x)
  (letrec ([y (+ x 2)]
            [f (lambda (z) (+ z y x))]
            [w (+ x 7)])
    (f -9)))

(define (mod2 x)
  (letrec ([even? (lambda (x) (if (zero? x) #t (odd? (- x 1))))]
            [odd? (lambda (x) (if (zero? x) #f (even? (- x 1))))]
            [if (even? x) 0 1])
    (if (even? x) 0 1)))

(define (bad-letrec-example x)
  (letrec [[y z]]
    (if y undefined
      [
        [z 13]]
      (if x y z))))

; and you can use define locally (in some positions)
; the same as letrec when binding local variables
(define (mod2-b x)
  (define even? (lambda (x) (if (zero? x) #t (odd? (- x 1)))))
  (define odd? (lambda (x) (if (zero? x) #f (even? (- x 1)))))
  (if (even? x) 0 1))

; at the top-level (*)
; same letrec-like rules: can have forward references, but
; definitions still evaluate in order and cannot be repeated
; (*) we are not actually at top-level -- we are in a module called lec3.rkt

(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 instead of #<undefined>)
(define v 5)
(define f 17); not okay: f already defined in this module

(define b 3)
(define g (lambda (x) (* 1 (+ x b))))
(define c (+ b 4))
(set! b 5)