

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Local Bindings

Local bindings

- Racket has 4 ways to define local variables
 - **let**
 - **let***
 - **letrec**
 - **define**
- Variety is good: They have different semantics
 - Use the one most convenient for your needs, which helps communicate your intent to people reading your code
 - If any will work, use **let**
 - Will help us better learn scope and environments
- Like in ML, the 3 kinds of let-expressions can appear anywhere

Let

A let expression can bind any number of local variables

- Notice where all the parentheses are

The expressions are all evaluated in the environment from **before the let-expression**

- Except the body can use all the local variables of course
- This is **not** how ML let-expressions work
- Convenient for things like `(let ([x y] [y x]) ...)`

```
(define (silly-double x)
  (let ([x (+ x 3)]
        [y (+ x 2)])
    (+ x y -5)))
```

Let*

Syntactically, a let* expression is a let-expression with 1 more character

The expressions are evaluated in the environment produced from the **previous bindings**

- Can repeat bindings (later ones shadow)
- This **is** how ML let-expressions work

```
(define (silly-double x)
  (let* ([x (+ x 3)]
         [y (+ x 2)])
    (+ x y -8)))
```

Letrec

Syntactically, a letrec expression is also the same

The expressions are evaluated in the environment that includes **all the bindings**

```
(define (silly-triple x)
  (letrec ([y (+ x 2)]
           [f (lambda (z) (+ z y w x))]
           [w (+ x 7)])
    (f -9)))
```

- Needed for mutual recursion
- But expressions are still *evaluated in order*: accessing an uninitialized binding produces an error
 - Remember function bodies not evaluated until called

More letrec

- Letrec is ideal for recursion (including mutual recursion)

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

- Do not use later bindings except inside functions
 - This example will produce an error if **x** is not **#f**

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

Local defines

- In certain positions, like the beginning of function bodies, you can put defines
 - For defining local variables, same semantics as **letrec**

```
(define (silly-mod2 x)
  (define (even? x) (if (zero? x) #t (odd? (- x 1))))
  (define (odd? x) (if (zero? x) #f (even? (- x 1))))
  (if (even? x) 0 1))
```

- Local defines is preferred Racket style, but course materials will avoid them to emphasize let, let*, letrec distinction
 - You can choose to use them on homework or not