**Bindings, environments, and scope**

For style, convenience, and efficiency

---

**Let expressions**

2 questions:

- **Syntax:** \( (\text{let } ([x_1 e_1] \ldots [x_n e_n]) \ e) \)
  - Each \( x_i \) is any variable, and \( e \) and each \( e_i \) are any expressions

- **Evaluation:**
  - Evaluate each \( e_i \) to \( v_i \) in the current dynamic environment.
  - Evaluate \( e \) in the current dynamic environment extended with each \( x_i \) bound to the corresponding \( v_i \).
  - Result of whole let-expression is result of evaluating \( e \).

---

**It is an expression**

A let-expression is *just an expression*, so we can use it anywhere an expression can go.

Silly example:

\[
(+ (\text{let } ([x 1]) \ x) (\text{let } ([y 2] [z 4]) (~ z y)))
\]
Shadowing and Scope

; Environment *after* this line
; env: .
(let ([x 2])
  (+ x
    (let ([x (* x x)])
      ; env: x --> 4, x --> 2, .
      (+ x 3)))
  ; env: .
)

What's new is scope: where a binding is in the environment

Only in body of the let-expression

Error: last use of x outside scope of binding:

(+ (let ([x 4]) x) x)

Better:

(define (count-up-from-1-better x)
  (letrec ([count (lambda (from)
                     (if (= from x)
                        (cons x null)
                        (cons from (count (+ from 1)))))]))
    (count 1)))

- Functions can use bindings in the environment where they are defined:
  - Bindings from "outer" environments
    - Such as parameters to the outer function
    - Earlier bindings in the let-expression
  - Unnecessary parameters are usually bad style
    - Like to in previous example

Even function bindings...

- Silly example:
  (define (quad x)
    (let ([square (lambda (x) (* x x))])
      (square (square x))))

- Private helper functions bound locally = good style.
- But no define-style recursion... for that we need letrec

(define (count-up-from-1 x)
  (letrec ([count (lambda (from to)
                     (if (= from to)
                        (cons to null)
                        (cons from (count (+ from 1) to))))])
    (count 1 x)))

Nested functions: style

- Good style to define helper functions inside the functions they help if they are:
  - unlikely to be useful elsewhere
  - likely to be misused if available elsewhere
  - likely to be changed or removed later

- A fundamental trade-off in code design: reusing code saves effort and avoids bugs, but makes the reused code harder to change later
Avoid repeated recursion

Consider this code and the recursive calls it makes

- Don’t worry about calls to car, cdr, and null? because they do a small constant amount of work

\[
\text{(define (bad-max xs)} \\
\text{  (if (null? xs)} \\
\text{    null ; max is not defined on empty list} \\
\text{  (if (null? (cdr xs)} \\
\text{    (car xs)} \\
\text{    (if (> (car xs) (bad-max (cdr xs)))} \\
\text{      (car xs)} \\
\text{      (bad-max (cdr xs)))))})
\]

Some calculations

Suppose one bad-max call’s if logic and calls to car, null?, cdr take 10^7 seconds

- Then (bad-max (list 50 49 ... 1)) takes 50 x 10^7 sec
- And (bad-max (list 1 2 ... 50)) takes 1.12 x 10^9 sec
  - (over 3.5 years)
  - (bad-max (list 1 2 ... 55)) takes over 1 century
  - Buying a faster computer won’t help much 😞

The key is not to do repeated work that might do repeated work that might do...

- Saving recursive results in local bindings is essential...

Fast vs. unusable

\[
\text{(if (> (car xs) (bad-max (cdr xs)))} \\
\text{  (car xs))} \\
\text{  (bad-max (cdr xs)))}
\]

Efficient max

\[
\text{(define (good-max xs)} \\
\text{  (if (null? xs)} \\
\text{    null)} \\
\text{  (if (null? (cdr xs))} \\
\text{    (car xs)} \\
\text{    (let ([rest-max (good-max (cdr xs))]} \\
\text{      (if (> (car xs) rest-max)} \\
\text{        (car xs)} \\
\text{        rest-max))])})
\]

Let as sugar, jump to list-append