Bindings, environments, and scope

For style, convenience, and efficiency

Let expressions

- The big thing we need: local bindings
  - For style and convenience
  - A big but natural idea: nested function bindings
  - For efficiency (*not* "just a little faster")

Let expressions

2 questions:

- Syntax: `(let ([x1 e1] ... [xn en]) e)`
  - Each `xi` is any *variable,* and `e` and each `ei` are any *expressions*
- Evaluation:
  - Evaluate each `ei` to `vi` in the current dynamic environment.
  - Evaluate `e` in the current dynamic environment extended with each `xi` bound to the corresponding `vi.`

  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:

```
(+ (let ([x 1]) x) (let ([y 2]
                         [z 4])
                     (- z y)))
```

Shadowing and Scope

```
                          ; Environment *after* this line
                          ; env: .
(let ([x 2])              ; env: 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)
```


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)))
```


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

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

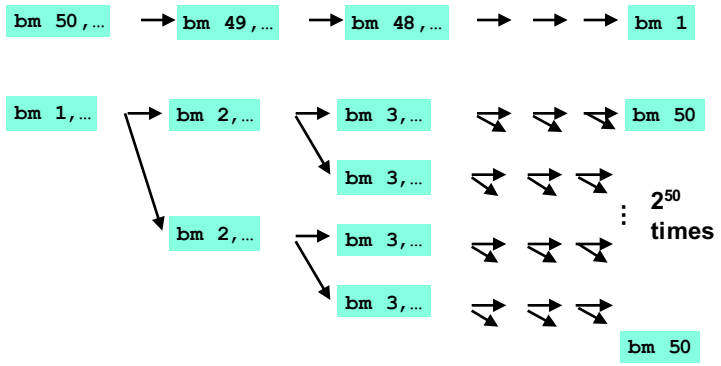
- Don't worry about calls to **car**, **cdr**, and **null?** because they do a small constant amount of work

```
(define (bad-max xs)
    (if (null? xs)
        null ; max is not defined on empty list
        (if (null? (cdr xs))
            (car xs)
            (if (> (car xs) (bad-max (cdr xs)))
                (car xs)
                (bad-max (cdr xs))))))
```

9

Fast vs. unusable

```
(if (> (car xs) (bad-max (cdr xs)))
    (car xs)
    (bad-max (cdr xs)))
```



10

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^{8} 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…

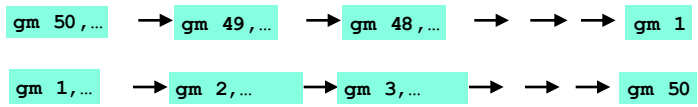
- Saving recursive results in local bindings is essential…

11

Efficient max

Should we rewrite any earlier functions?

```
(define (good-max xs)
    (if (null? xs)
        null
        (if (null? (cdr xs))
            (car xs)
            (let ([rest-max (good-max (cdr xs))])
                (if (> (car xs) rest-max)
                    (car xs)
                    rest-max)))))
```



Let as sugar, jump to list-append

12