Local Bindings and Scope

Topics
- Control scope with local bindings
- Shadowing
- Scope sugar
- Nested function bindings
- Avoid duplicate computations
  - style and convenience
  - efficiency \( \text{(big-O)} \)

let expressions

Syntax: \( \text{(let ([} \{x1\} \text{el}] \ldots [\{xn\} \text{en}]) \text{e})} \)

Each \( {x1} \) is any variable. \( e \) and each \( {el} \) are any expressions.

Evaluation:
1. Under the current dynamic environment, \( E \), evaluate \( {el} \) through \( {en} \) to values \( v1, ..., vn \).
2. The result is the result of evaluating \( e \) under the environment, \( E \), extended with bindings \( {x1} \rightarrow v1, ..., {xn} \rightarrow vn \).

\[
\begin{align*}
E \triangleright {el} &\downarrow v1 \\
& \ldots \\
E \triangleright {en} &\downarrow vn \\
{x1} \rightarrow v1, ..., {xn} \rightarrow vn, E \triangleright e &\downarrow v \\
E \triangleright (\text{let ([} \{x1\} \text{el}] \ldots [\{xn\} \text{en}]) \text{e}) &\downarrow \nu \quad \text{[let]}
\end{align*}
\]

(\( + (\text{let ([}x\ 1]\ )\ x) \)
(\( \text{(let ([}y\ \text{(let ([}a\ 2]\ )\ a]\]) \)
(\( [z\ 4]\) 
(\( (- z\ y)\)\)) \)
let expressions control scope.

Scope of a binding = area of program that is evaluated while that binding is in environment. Visualize scope via lexical contours.

```
(define add-n (lambda (x) (+ n x)))
(define add-2n (lambda (y) (add-n (add-n y))))
(define n 17)
(define (f z)
  (let ([c (add-2n z)]
        [d (- z 3)])
    (+ z (* c d))))
```

Errors: cannot use x or y outside scope of bindings.

```plaintext
; E = .
(+ (let ([x 4]
          [y (* 2 x)])
    ; E = x→4, y→8, .
    (+ x y))
  (* x y))
```

Shadowing

```
; E = .
(let ([x 2])
  ; E = x→2, .
  (+ x
    (let ([x (* x x)])
      ; E = x→4, x→2, .
      (+ x 3))))
; E = .
```

and and or are sugar!

```
(and e1 e2)
```
desugars to
```
(if e1 e2 #f)
```
```
(or e1 e2)
```
desugars to
```
(let ([x1 e1])
  (if x1 x1 e2))
```
where x1 is not used (without first being bound) in e2
(easiest: "fresh" identifier used nowhere in entire program)
let is sugar!

Syntax:  
\[
\text{(let ([x1 e1] ... [xn en]) e)}
\]
Each \(x_i\) is any variable. \(e\) and each \(e_i\) are any expressions.

Evaluation:
1. Under the current dynamic environment, \(E\), evaluate \(e_1\) through \(e_n\) to values \(v_1, \ldots, v_n\).
2. The result is the result of evaluating \(e\) under the environment, \(E\), extended with bindings \(x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\).

\[
\begin{align*}
E & \vdash e_1 \downarrow v_1 \\
& \quad \ldots \\
E & \vdash e_n \downarrow v_n \\
x_1 & \mapsto v_1, \ldots, x_n \mapsto v_n, \\
E & \vdash e \downarrow v
\end{align*}
\]

Local Bindings and Scope

- Local function bindings

```
(define (quad x)
  (let ([square (lambda (x) (* x x))])
    (square (square x))))
```

Private helper functions bound locally can be good style.
Need \text{letrec} to allow recursion*.

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

- Better style:

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

- Functions can use bindings in the environment where they are defined: \text{count-to-x} can use \(x\).
- Unnecessary parameters are usually bad style:
  - to in previous example

*Not just lambda sugar. We will wait to define it precisely later.*
**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

Trade-off in code design:
- reusing code saves effort and avoids bugs
- makes the reused code harder to change later

**Avoid repeated recursion**

Consider this code and the recursive calls it makes
- Ignore calls to first, rest, and null?
  (small constant amounts of work)

```
(define (bad-max xs)
  (if (null? xs)
      null ; not defined on empty list
      (if (> (first xs) 
          (bad-max (rest xs)))
        (first xs)
        (bad-max (rest xs)))))
```

**Fast vs. unusable**

Assume $10^{-7}$ seconds each
Then: $50 \times 10^{-7}$ sec vs $1.12 \times 10^{8}$ sec = 3.5 years
(bad-max (list 1 2 ... 100)) takes $> 4 \times 10^{15}$ years.

**Efficient max**

```
(define (good-max xs)
  (if (null? xs)
      null ; not defined on empty list
      (if (> (first xs) 
          (good-max (rest xs)))
        (first xs)
        (good-max (rest xs))))
```

Our sun is predicted to die in about $5 \times 10^9$ years.
Efficient and concise max

(define (maxlist xs)
  (if (null? xs)
      null ; not defined on empty list
      (max (first xs) (maxlist (rest xs))))

; even better implementations to come later