Local Bindings and Scope
Topics

• Control scope with local bindings
• Shadowing
• Scope sugar
• Nested function bindings
• Avoid duplicate computations
  – style and convenience
  – efficiency (big-O)
let expressions

Interchangeable: ( ), [ ], or { }

Syntax:

\[
\text{(let } ([x_1 e_1] \ldots [x_n e_n]) \ e)\]

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 \(x_1 \leftrightarrow v_1, \ldots, x_n \leftrightarrow v_n\).

\[
\begin{align*}
E \vdash e_1 & \downarrow v_1 \\
& \quad \ldots \\
E \vdash e_n & \downarrow v_n \\
\end{align*}
\]

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

\[
E \vdash (\text{let } ([x_1 e_1] \ldots [x_n e_n]) \ e) \downarrow v
\]

Local Bindings and Scope
let **expressions**

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

Let expression bindings are in the environment only during evaluation of the body.

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

; E = .
(+ (let ([x 4] [y (* 2 x)])
  ; E = x\rightarrow4, y\rightarrow8, .
  (+ x y))
; E = .
(* x y))
Shadowing

; E = .

(let ([x 2])
 ; E = x↦2,
 (+ x
 (let ([x (* x x x)])
 ; E = x↦4, x↦2,
 (+ x 3)
 ) ) )
 ; E = .
and and or are sugar!

(and \( e_1 \ e_2 \)) desugars to

(if \( e_1 \ e_2 \ \#f \))

(or \( e_1 \ e_2 \)) desugars to

(let (\([x_1 \ e_1 ]\))

(if \( x_1 \ x_1 \ e_2 \))

where \( x_1 \) is not used (without first being bound) in \( e_2 \)

(easiest: "fresh" identifier used nowhere in entire program)
let is sugar!

Syntax: 

\[
(\text{let} ([x_1 e_1] \ldots [x_n e_n]) e)
\]

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 \(x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\).

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

\[
E \vdash (\text{let} ([x_1 e_1] \ldots [x_n e_n]) e) \downarrow v
\]
let is sugar!

(let ([x1 e1] ... [xn en]) e)

desugars to

(((lambda (x1 ... xn) e) e1 ... en)

Example:

(let ([x (* 3 5)]) (+ x x))

desugars to

(((lambda (x) (+ x x)) (* 3 5))

Local Bindings and Scope 10
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 letrec to allow recursion*.

(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)))

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

(define (count-up-from-1 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: `count-to-x` can use `x`.
- Unnecessary parameters are usually bad style: – `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

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 (null? (rest xs))
          (first xs)
          (if (> (first xs)
                  (bad-max (rest xs)))
              (first xs)
              (bad-max (rest xs)))))
```
**Fast vs. unusable**

```
(if (> (first xs)
      (bad-max (rest xs)))
  (first xs)
  (bad-max (rest xs)))
```

Assume $10^{-7}$ seconds each

Then: $50 \times 10^{-7}$ sec vs $1.12 \times 10^8$ sec = 3.5 years

```
(bad-max (range 50 0 -1))
```

- `bm 50,...` ➔ `bm 49,...` ➔ `bm 48,...` ➔ `bm 1`

```
(bad-max (range 1 51))
```

- `bm 1,...` ➔ `bm 2,...` ➔ `bm 3,...` ➔ `bm 50`

$O(2^n)$

$2^{50}$ times

$O(2^n)$

```
(bad-max (list 1 2 ... 100)) takes > 4 x 10^{15} years.
```

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

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

gm 50,… → gm 49,… → gm 48,… → gm 1

gm 1,… → gm 2,… → gm 3,… → gm 50
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