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

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

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

let expressions

\[
(+ (\text{let } ([x 1]) x) \\
(\text{let } ([y (\text{let } ([a 2]) a)]) \\
[z 4])) \\
(- z y))
\]
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))))
```

Local Bindings and Scope

**Shadowing**

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

Local Bindings and Scope

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))

Local Bindings and Scope
let is sugar!

Syntax: 
(\[x1 e1\] ... [xn en]) e

Each xi is any variable. e and each ei are any expressions.

Evaluation:
1. Under the current dynamic environment, E, evaluate e1 through en to values v1, ..., vn.
2. The result is the result of evaluating e under the environment, E, extended with bindings x1 ⟷ v1, ..., xn ⟷ vn.

\[ E \vdash e1 \downarrow v1 \]

\[ \vdots \]

\[ E \vdash en \downarrow vn \]

\[ x1 \rightarrow v1, ..., xn \rightarrow vn, E \vdash e \downarrow v \]

\[ E \vdash (let ([x1 e1] ... [xn en]) e) \downarrow v \]

Local Bindings and Scope

10

Better style:

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

*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 x 10⁻⁷ sec vs 1.12 x 10⁻⁸ sec = 3.5 years
(bad-max (list 1 2 ... 100)) takes > 4 x 10¹⁸ years.

Our sun is predicted to die in about 5 x 10⁹ 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 (> (rest xs) rest-max)
            (rest xs)
            (rest-max))))))

O(2^n)

\[ \text{gm 50,} \quad \text{gm 49,} \quad \text{gm 48,} \quad \text{gm 1} \]
\[ \text{gm 1,} \quad \text{gm 2,} \quad \text{gm 3,} \quad \text{gm 50} \]
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