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

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

\[
\begin{array}{c}
E \vdash \text{let } (((x_1 \ e_1) \ldots) (x_n \ e_n)) \ e \downarrow v \\
\end{array}
\]
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)])
    (+ x y))
; E = x→4, y→8, .
(* 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))
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 \(x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\).

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

Local Bindings and Scope

Better style:

\[
\begin{align*}
& \quad \text{(define (count-up-from-1 x)} \\
& \quad \quad \text{(letrec} \\
& \quad \quad \quad ([\text{count-to-x} \ (\lambda (\text{from}) \\
& \quad \quad \quad \quad (\text{if} \ (= \text{from} \ x) \\
& \quad \quad \quad \quad \ (\text{cons} \ x \ \text{null}) \\
& \quad \quad \quad \quad \ (\text{cons} \ \text{from} \\
& \quad \quad \quad \quad \quad \ (\text{count-to-x} \ (+ \ \text{from} \ 1) \ x))))]) \\
& \quad \quad \quad (\text{count-to-x} \ 1)))
\end{align*}
\]

- 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)))))
```

Local Bindings and Scope

13

Efficient max

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

```
(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
   bm 2, ...
   bm 3, ...
   bm 49, ...
```

Assume 10⁻⁷ seconds each
Then: 50 x 10⁻⁷ sec vs 1.12 x 10⁸ sec = 3.5 years
(bad-max (list 1 2 ... 100)) takes > 4 x 10¹⁵ years.
Our sun is predicted to die in about 5 x 10⁹ years.

Fast vs. unusable

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

```
(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))))))
```

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

```
(bad-max (range 1 51))
   gm 1, ...
   gm 2, ...
   gm 3, ...
   gm 50
   gm 2, ...
   gm 3, ...
   gm 49, ...
```

Assume 10⁻⁷ seconds each
Then: 50 x 10⁻⁷ sec vs 1.12 x 10⁸ sec = 3.5 years
(bad-max (list 1 2 ... 100)) takes > 4 x 10¹⁵ years.
Our sun is predicted to die in about 5 x 10⁹ 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