Local Naming and Scope

These slides borrow heavily from Ben Wood’s Fall ‘15 slides, some of which are in turn based on Dan Grossman’s material from the University of Washington.

CS251 Programming Languages
Spring 2016, Lyn Turbak
Department of Computer Science
Wellesley College

Motivation for local bindings

We want local bindings = a way to name things locally in functions and other expressions.

Why?
– For style and convenience
– Avoiding duplicate computations
– A big but natural idea: nested function bindings
– Improving algorithmic efficiency (not “just a little faster”)

let expressions

2 questions:

• Syntax: \( \text{let } \{ \{id1 e1\} \ldots \{idn en\} \text{ e_body} \} \)
  – Each \( xi \) is any variable, and e_body and each ei are any expressions

• Evaluation:
  – Evaluate each ei to \( vi \) in the current dynamic environment.
  – Evaluate e_body[v1,…vn/id1,…,idn] in the current dynamic environment.
  Result of whole let expression is result of evaluating e_body.

Example

Pretty printed form

\[
\begin{align*}
\text{let } & \{ \{a (+ 1 2)\} \{b (* 3 4)\} \} \quad (\text{list a b}) \\
\rightarrow & (3 12)
\end{align*}
\]
**Parens vs. Braces vs. Brackets**

As matched pairs, they are interchangeable. Differences can be used to enhance readability.

```racket
> (let {[[a (+ 1 2)] [b (* 3 4)]]} (list a b))
'(3 12)
```

```racket
> (let {((a (+ 1 2)) (b (* 3 4)))} (list a b))
'(3 12)
```

```racket
> (let {[a (+ 1 2)] [b (* 3 4)]} (list a b))
'(3 12)
```

```racket
> (let [{a (+ 1 2)] [b (* 3 4)]} (list a b))
'(3 12)
```

**let is an expression**

A let-expression is *just an expression*, so we can use it *anywhere* an expression can go. Silly example:

```racket
(+ (let {[[x 1]} x)
    (let {[[y 2]
            [z 4]]
         (+ z y))}
   (+ (let {[[x 1]} x)
       (let {[[y 2]
              [z 4]]
          (+ z y))})
   (+ (let {[[x 1]} x)
       (let {[[y 2]
              [z 4]]
          (+ z y))})
```

**let is just syntactic sugar!**

`(let {[[id1 e1] ... [idn en]]} e_body)`

desugars to

```racket
((lambda (id1 ... idn) e_body) e1 ... en)
```

Example:

`(let {[[a (+ 1 2)] [b (* 3 4)]]} (list a b))`

desugars to

```racket
((lambda (a b) (list a b)) (+ 1 2) (* 3 4))
```

**Scope and Lexical Contours**

`scope` = area of program where declared name can be used. Show scope in Racket via *lexical contours* in *scope diagrams*.

```racket
(define add-n (λ (x) (+ n x)))
(define add-2n (λ (y) (add-n (add-n y))))
(define n 17)
(define f (λ (z)
          (let {[(c (add-2n z))]
                [(d (- z 3))]
                (+ z (* c d))})
          ))
```
**Declarations vs. References**

A **declaration** introduces an identifier (variable) into a scope.

A **reference** is a use of an identifier (variable) within a scope.

We can box declarations, circle references, and draw a line from each reference to its declaration. Dr. Racket does this for us (except it puts ovals around both declarations and references).

An identifier (variable) reference is **unbound** if there is no declaration to which it refers.

---

**Scope and Define Sugar**

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

---

**Shadowing**

An inner declaration of a name **shadows** uses of outer declarations of the same name.

```
(let {
  [x 2]
  (- (let {
        [x (* x x)]
      }
        (+ x 3)))
  x ))
```

---

**Alpha-renaming**

Can consistently rename identifiers as long as it doesn’t change the connections between uses and declarations.

```
(define (f w z)
  (* w
    (let {
      [c (add-2n z)]
      [d (- z 3)]
    }
      (+ z (* c d)))))
```

---

```
(define (f w z)
  (* w
    (let {
      [c (add-2n z)]
      [d (- z 3)]
    }
      (+ z (* c d)))))
```

---

```
(define (f x y)
  (* x
    (let {
      [x (add-2n y)]
      [y (- d x)]
    }
      (+ y (* x y)))))
```

---
Scope, Free Variables, and Higher-order Functions

In a lexical contour, an identifier is a **free variable** if it is not defined by a declaration within that contour. Scope diagrams are especially helpful for understanding the meaning of free variables in higher order functions.

```
(define (make-sub n )
  (λ ( x ) (- x n)))

(define (map-scale factor ns )
  (map (λ ( num ) (* factor num )) ns) )
```

Your Turn: Compare the Following

```
(let { [a 3] [b 12]}
  (list a b
    (let { [a (- b a)]
        [b (* a a)]}
      (list a b))))
```

```
(let { [a 3] [b 12]}
  (list a b
    (let { [a (- b a)]}
      (let { [b (* a a)]}
        (list a b)))))
```

New sugar: **let***

```
(let* {} e_body) desugars to e_body
```

```
(let* { [id1 e1] ...} e_body)
  desugars to (let { [id1 e1]}
    (let* { ...} e_body))
```

Example:

```
(let { [a 3] [b 12]}
  (list a b
    (let* { [a (- b a)]
        [b (* a a)]}
      (list a b)))))
```

and  and or sugar

```
(and) desugars to #t
(and el) desugars to el
(and el ...) desugars to (if el (and ...) #f)
```

```
(or) desugars to #f
(or el) desugars to el
(or el ...) desugars to
  (let ((id1 el))
    (if el el (or ...)))
```

where **id1** must be **fresh** – i.e., not used elsewhere in the program.

- Why is let needed in or desugaring but not and?
- Why must id1 be fresh?
Avoid repeated recursion

Consider this code and the recursive calls it makes
- Don’t worry about calls to first, rest, and null? because they do a small constant amount of work

```
(define (bad-maxlist xs)
  (if (null? xs)
      -inf.0
      (if (> (first xs) (bad-maxlist (rest xs)))
          (first xs)
          (bad-maxlist (rest xs)))))
```

Fast vs. unusable

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

bm 50, ...  bm 49, ...  bm 48, ...
```

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

bm 1, ...  bm 2, ...  bm 3, ...
```

Efficient maxlist

```
(define (good-maxlist xs)
  (if (null? xs)
      -inf.0
      (let { [rest-max (good-maxlist (rest xs))]
          (if (> (first xs) rest-max)
              (first xs)
              rest-max))})
```

Some calculations

Suppose one bad-maxlist call’s if logic and calls to null?, first?, rest take 10⁻⁷ seconds total
- Then (bad-maxlist (list 50 49 ... 1)) takes 50 x 10⁻⁷ sec
- And (bad-maxlist (list 1 2 ... 50)) takes
  \(1 + 2 + 2^2 + 2^3 + \ldots + 2^{49}\) x 10⁻⁷
  \(= (2^{49} - 1) \times 10^{-7} = 1.12 \times 10^8 \text{ sec}\)
  - over 3.5 years
  - (bad-maxlist (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...
Transforming good-maxlist

```scheme
(define (good-maxlist xs)
  (if (null? xs)
      -inf.0
      (let {{[rest-max (good-maxlist (rest xs))])
        (if (> (first xs) rest-max)
          (first xs)
          rest-max))}))
```

```scheme
(define (good-maxlist xs)
  (if (null? xs)
      -inf.0
      (let {{[rest-max (good-maxlist (rest xs))])
        (if (> (first xs) rest-max)
          (first xs)
          rest-max))}))
```

```scheme
(define (good-maxlist xs)
  (if (null? xs)
      -inf.0
      (max (first xs) (good-maxlist (rest xs))))))
```

Local function bindings with \textit{let}

- Silly example:
  ```scheme
  (define (quad x)
    (let ([square (lambda (x) (* x x))])
      (square (square x)))
  )
  ```
- Private helper functions bound locally = good style.
- But can’t use \textit{let} for local recursion. Why not?
  ```scheme
  (define (up-to-broken x)
    (let ([between (lambda (from to)
                    (if (> from to)
                      null
                      (cons from
                        (between (+ from 1) to))))]
          (between 1 x)))
  )
  ```

\textit{letrec} to the rescue!

```scheme
(define (up-to x)
  (letrec {[between (lambda (from to)
                      (if (> from to)
                        null
                        (cons from
                          (between (+ from 1) to))))]
            (between 1 x)))
)
```

In \texttt{(let \{[id1 e1] ... [idn en]} e\_body)},
e1 ... en are in the scope of id1 ... idn.

Better

```scheme
(define (up-to-better x)
  (letrec {[up-to-x (lambda (from)
                        (if (> from x)
                          null
                          (cons from
                            (up-to-x (+ from 1))))]
            (between 1 x))}
            (up-to-x 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 \texttt{to} in previous example
Mutual Recursion with letrec

(define (test-even-odd num)
  (letrec {[
    even? (λ (x)
      (if (= x 0) #t (not (odd? (- x 1))))))
    odd? (λ (y)
      (if (= y 0) #f (not (even? (- y 1))))))
  (list (even? num) (odd? num))))

> (test-even-odd 17)
'(#t #f)

Local definitions are sugar for letrec

(define (up-to-alt2 x)
  (define (up-to-x from)
    (if (> from x) null
      (cons from
        (up-to-x (+ from 1)))))
  (up-to-x 1))

(define (test-even-odd-alt num)
  (define (even? x)
    (if (= x 0) #t (not (odd? (- x 1))))
  (define (odd? y)
    (if (= y 0) #f (not (even? (- y 1))))
  (list (even? num) (odd? num)))

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

Local Scope in other languages

What support is there for local scope in Python?
JavaScript?
Java?

You will explore this in PS5!
Pragmatics: Programming Language Layers

Where We Stand

Kernel  Sugar  Built-in library functions  User-defined library functions