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.

**Local Bindings and Scope**

**SOLUTIONS**

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: Example

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

Pretty printed form

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

**let** in the quadratic formula

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

```lisp
(define (quadratic-roots a b c)
  (let { [-b (- b)]
         [sqrt-discriminant
          (sqrt (- (* b b) (* 4 a c)))]
         [2a (* 2 a)]
         (list (/ (+ -b sqrt-discriminant) 2a)
               (/ (- -b sqrt-discriminant) 2a)))})

> (quadratic-roots 1 -5 6)
'(3 2)
> (quadratic-roots 2 7 -15)
'(1/2 -5)```
Formalizing let expressions

2 questions: a new keyword!

- Syntax: \( \text{let} \{ [\text{Id}_1 \ E_1] \ldots [\text{Id}_n \ E_n] \} \ E_{\text{body}} \)
  - Each \( \text{Id}_i \) is any identifier, and \( E_{\text{body}} \) and each \( E_i \) are any expressions

- Evaluation:
  - Evaluate each expression \( E_i \) to value \( V_i \) in the current dynamic environment.
  - Evaluate \( E_{\text{body}}[V_1,\ldots,V_n]/[\text{Id}_1,\ldots,\text{Id}_n] \) in the current dynamic environment.

Result of whole let expression is result of evaluating \( E_{\text{body}} \).

let is an expression

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

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

let is just syntactic sugar!

\( \text{let} \ \{ [\text{Id}_1 \ E_1] \ldots [\text{Id}_n \ E_n] \} \ E_{\text{body}} \)

desugars to

\( ((\lambda (\text{Id}_1 \ldots \text{Id}_n) \ E_{\text{body}}) \ E_1 \ldots E_n) \)

Example:

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

desugars to

\( ((\lambda (a \ b) \ (\text{list} \ a \ b)) \ (+ \ 1 \ 2) \ (* \ 3 \ 4)) \)
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

\[
\text{(define (bad-maxlist xs)}
  \text{(if (null? xs)}
    \text{-inf.0)}
  \text{(if (> (first xs) (bad-maxlist (rest xs)))}
    \text{(first xs)}
    \text{(bad-maxlist (rest xs)))))}
\]

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 \((1 + 2 + 2^2 + 2^3 + \ldots + 2^{49}) \times 10^{-7}
    = (2^{50} - 1) \times 10^{-7} = 1.12 \times 10^8\) sec
    = over 3.5 years
  − And (bad-maxlist (list 1 2 ... 55))
    takes over 114 years
  − And (bad-maxlist (list 1 2 ... 100))
    takes over 4 \times 10^{15} years.
    (Our sun is predicted to die in about 5 \times 10^9 years)
  − Buying a faster computer won’t help much 😞

The key is not to do repeated work!
  − Saving recursive results in local bindings is essential…

Efficient maxlist

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

Your turn: sumProdList  Solution

Given a list of numbers, sumProdList returns a pair of
(1) the sum of the numbers in the list and
(2) The product of the numbers in the list

(sumProdList '(5 2 4 3)) -> (14 . 120)
(sumProdList '()) -> (0 . 1)

Define sumProdList. Why is it a good idea to use let in your definition?

```
(define (sumProdList ns)
  (if (null? ns)
      '(0 . 1) ; (cons 0 1)
      (let {[[sumProdRest (sumProdList (rest ns))]] (cons (+ (first ns) (car sumProdRest))
          (* (first ns) (cdr sumProdRest)))))))
```

and and or sugar

(and) desugars to #t
(and E1) desugars to E1
(and E1 ...) desugars to (if E1 (and ...) #f)

(or) desugars to #f
(or E1) desugars to E1
(or E1 ...) desugars to
(let ((Id1 E1))
  (if Id1 Id1 (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?

Scope and Lexical Contours

scope = area of program where declared name can be used.

Show scope in Racket via lexical contours in scope diagrams.

```
(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

```
(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))))
```

Local Bindings & Scope

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))
```

Can’t refer to outer `x` here.

Alpha-renaming

Can consistently rename identifiers as long as it doesn’t change the “wiring diagram” between uses and declarations.

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

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

OK

```
(define (f x y)
  (* x
    (let {
      [x (add-2n y)]
      [y (- y 3)]
    }
      (+ y (* x y))))))
```

Not OK (because `y` in `(y ...)` refers to let-bound `y`, not function parameter `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))
```

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))
```

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))
```

More sugar: `let*`

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

```
(let* {[Id1 E1] ...} Ebody)
```

desugars to
```
(let* {[Id1 E1]}
  (let* {...} Ebody))
```

Example (same as 2nd example on previous slide)
```
(let* {[a (+ 2 3)] [b (* 3 4)]}
  (list a
    (let* {[a (- b a)]
      [b (* a a)]
      (list a b))
    b))
```

Local function bindings with `let`

- Silly example:
```
(define (quad x)
  (let ([square (λ (x) (* x x))]
        (square (square x)))))
```

- Private helper functions bound locally = good style.
- But can’t use let for local recursion. Why not?

```
(define (up-to-broken x)
  (let* {[between (λ (from to)
      (if (> from to)
        null
        (cons from
          (between (+ from 1) to)))]}
    (between 1 x)))
```
**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 **(letrec {[Id1 E1] ... [Idn E1]} Ebody)**, 
Id1 ... Idn are in the scope of E1 ... En.

---

**Even Better**

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

---

**Mutual Recursion with letrec**

```scheme
(define (test-even-odd num)
  (letrec {[[even? (lambda (x)
                  (if (= x 0)
                    #t
                    (odd? (- x 1)))]
          [odd? (lambda (y)
                    (if (= y 0)
                      #f
                      (even? (- y 1)))]}
    (list (even? num) (odd? num)))
```

> (test-even-odd 42) 
'(#t #f)

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

---

**Exercise: let vs. let* vs. letrec Solutions**

```scheme
(let {[[f (lambda (x) (/ x 2))]
        [g (lambda (y) (+ y 1))]
        [h (lambda (a b) (+ a b))]}
  (let {[[f (lambda (y) (- y 1))]
        [g (lambda (n)
          (if (<= n 0)
            1
            (h n (g (f n)))))]
        [h (lambda (a b) (* a b))]} (list (f 10) (g 4) (h 2 3))))
```

- What is the value of the above expression? '(9 7 6)
- What is its value if the inner let is replaced by let*? '(9 8 6)
- What is its value if the inner let is replaced by letrec? '(9 24 6)
  (in this case, g is the factorial function!)

---

**Local Bindings & Scope**

25

26

27

28
Local definitions are sugar for `letrec`

The following internal definitions desugar to the `letrec` studied in previous slides:

```racket
(define (up-to-alt 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 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)))
```

Local Scope in other languages

<table>
<thead>
<tr>
<th>Java</th>
<th>JavaScript</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static int w = 2;</td>
<td>var w = 2;</td>
<td>def f(y):</td>
</tr>
<tr>
<td>public static int x = 3;</td>
<td>var x = 3;</td>
<td>global w:</td>
</tr>
<tr>
<td>public static int f (int y) {</td>
<td></td>
<td>if y &gt; x:</td>
</tr>
<tr>
<td></td>
<td>int z;</td>
<td>z = y - x;</td>
</tr>
<tr>
<td></td>
<td>if (y &gt; x) {</td>
<td>} else {</td>
</tr>
<tr>
<td></td>
<td>z = y * w;</td>
<td>w = y + z;</td>
</tr>
<tr>
<td></td>
<td>} else {</td>
<td>return y * z;</td>
</tr>
<tr>
<td></td>
<td>z = y * w;</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>w = y + z;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return y * z;</td>
<td>}</td>
</tr>
</tbody>
</table>

In all 3 languages, (f(8)) returns 28 and a following (f(10)) returns 70.

- Java requires z to be declared outside if it's used in both branches, because each {...} defines a new scope. But in JavaScript and Python, any declaration has scope of entire function body regardless of where declaration is.
- Python uses = to both declare and re-assign, so needs `global` declaration when assigning to global variable.
- JavaScript and Python allow local function decls; Java has local class (not method) decls.
- No let-like expression in Python/JavaScript, but can be simulated by calling local or anonymous function.

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.

Racket Language Summary So Far

**Racket kernel declarations:**
- definitions: `(define Id E)`

**Racket kernel expressions**
- literal values (numbers, boolean, strings): e.g. 251, 3.141, #t, "Lyn"
- variable references: e.g., x, fact, positive?, fib_n-1
- conditionals: `(if Etest Ethen Eelse)`
- function values: `(lambda (Id1 ... Idn) Ebody)`
- function calls: `(Erator Erand1 ... Erandn)`

*Note: arithmetic and relational operations are really just function calls!*

- (new) local recursion: `(letrec {[[Id1 E1] ... [Idn En]] Ebody})`

**Racket Syntactic Sugar**
- `(define (Idfun Id1 ... Idn) Ebody)`
- `(and E1 ... E2)`
- `(or E1 ... E2)`
- `(let {[[Id1 E1] ... [Id1 E1]] Ebody})`
- `(let* {[[Id1 E1] ... [Id1 E1]] Ebody})`

**Racket Built-in Functions**
- `+`, `-%`, `*/`, `min`, `max`, ...
- `<`, `<=`, `>=`, `>`,
- `cons`, `cdr`,
- `list`, `first`, `second`, ..., `rest`