Local Bindings 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
Fall 2018, 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: Example

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

Pretty printed form

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

let in the quadratic formula

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

(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 1/2 -5)
Formalizing \texttt{let} expressions

2 questions:

- Syntax: \(\texttt{(let \{[I1 \; E1] \; \ldots \; [In \; En]\} \; \textit{Ebody})}\)
  - Each \texttt{Idi} is any identifier, and \textit{Ebody} and each \texttt{Ei} are any expressions

- Evaluation:
  - Evaluate each expression \texttt{Ei} to value \texttt{Vi} in the current dynamic environment.
  - Evaluate \textit{Ebody}[\texttt{V1, \ldots \; Vn}/\texttt{Id1, \ldots , In}] in the current dynamic environment.
  - Result of whole \texttt{let} expression is result of evaluating \textit{Ebody}.

---

\texttt{let} is an expression

A \texttt{let}-expression is \textit{just an expression}, so we can use it \textit{anywhere} an expression can go.

Silly example:

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

---

\texttt{let} is just syntactic sugar!

\(\texttt{(let} \; \{[I1 \; E1] \; \ldots \; [In \; En]\} \; \textit{Ebody})\)

desugars to

\[
((\lambda (I1 \; \ldots \; In) \; \textit{Ebody}) \; E1 \; \ldots \; En)
\]

Example:

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

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 \texttt{first}, \texttt{rest}, and \texttt{null?} because they do a small constant amount of work

\begin{verbatim}
(define (bad-maxlist xs)
  (if (null? xs)
      -inf.0
      (if (> (first xs) (bad-maxlist (rest xs)))
        (first xs)
        (bad-maxlist (rest xs))))
\end{verbatim}

Some calculations

Suppose one \texttt{bad-maxlist} call’s \texttt{if} logic and calls to \texttt{null?}, \texttt{first?}, \texttt{rest} take $10^{-7}$ seconds total
– Then (\texttt{bad-maxlist (list 50 49 \ldots 1)}) takes $50 \times 10^{-7}$ sec
– And (\texttt{bad-maxlist (list 1 2 \ldots 50)}) takes \((1 + 2 + 2^2 + 2^3 + \ldots + 2^{49}) \times 10^{-7} = (2^{50} - 1) \times 10^{-7} = 1.12 \times 10^8 \text{ sec} = \text{over 3.5 years}\)
– And (\texttt{bad-maxlist (list 1 2 \ldots 100)}) takes over 114 years
– And (\texttt{bad-maxlist (list 1 2 \ldots 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

\begin{verbatim}
(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))))
\end{verbatim}
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

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?

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))))
```

---

**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 “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)))))
```

---

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

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 El] …} Ebody)
  desugars to (let {[Id1 El]}
                 (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 (lambda (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 (lambda (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**

```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?
- What is its value if the inner `let` is replaced by `let*`?
- What is its value if the inner `let` is replace by `letrec`?
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)))
```

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

**Java**
```
public static int w = 2;
public static int x = 3;
public static int f (int y) {
  int z;
  if (y > x) {
    z = y - x;
  } else {
    z = y * w;
  }
  w = y + z;
  return y * z;
}
```

**JavaScript**
```
var w = 2;
var x = 3;
function f(y) {
  var z = y;
  if (y > x) {
    z = y - x;
  } else {
    z = y * w;
  }
  w = y + z;
  return y * z;
}
```

**Python**
```
def f(y):
    global w
    if y > x:
        z = y - x
    else:
        z = y * w
    w = y + z
    return y * z
```

- Java requires `z` to be declared outside `if` if it’s used in both branches, because each `if` 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.

Racket Language Summary So Far

**Racket kernel declarations:**
- definitions: `(define Id E)`
- parameters:` (letrec {{Id1 E1} … {Idn En} Ebody})`

**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: `(Eoperator 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 En} Ebody})`
- `(let* {{Id1 E1} … {Id1 En} Ebody})`
- `(+, -, *, /, min, max, …)`
- `<, <=, >=, >, =, cons, car, cdr, list, first, second, …, rest`