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.

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

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

Pretty printed form

```scheme
> (let ([a (+ 1 2)]

[b (* 3 4)])

(list a b))

'(3 12)
```

let in the quadratic formula

```scheme
(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: \textcolor{red}{a new keyword!}

- Syntax: \texttt{(let \{[id1 e1] ... [idn en]\) e\_body})
  - Each \(x_i\) is any variable, and \texttt{e\_body} and each \(e_i\) are any expressions

- Evaluation:
  - Evaluate each \(e_i\) to \(v_i\) in the current dynamic environment.
  - Evaluate \texttt{e\_body[v1,...vn/id1,...,idn]} in the current dynamic environment.
  - Result of whole \texttt{let} expression is result of evaluating \texttt{e\_body}.

\texttt{let is an expression}

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

Silly example:

\begin{verbatim}
(+ (let \{[x 1]\) x)
  (let \{[y 2]
    [z 4]}
    (- z y)))
\end{verbatim}

\texttt{let is just syntactic sugar!}

\texttt{(let \{[id1 e1] ... [idn en]\) e\_body})}

desugars to

\texttt{((lambda (id1 ... idn) e\_body) e1 ... en)}

Example:

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

desugars to

\begin{verbatim}
((lambda (a b) (list a b)) (+ 1 2) (* 3 4))
\end{verbatim}

Parens vs. Braces vs. Brackets

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

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

> (let \((a (+ 1 2)) (b (* 3 4))\) (list a b))
'(3 12)

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

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

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

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 \times 10^{-7} sec
- And (bad-maxlist (list 1 2 ... 50))
takes \(1 + 2^2 + 2^3 + ... + 2^{49}\) \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^{16} 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

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

```racket
(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))}}))
```

```racket
(define (good-maxlist xs)
  (if (null? xs)
      -inf.0
      (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*.

```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.

```scheme
(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.

```scheme
(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 (- d y)]
              (+ y (* x y))})))
```

Local Bindings & Scope

17

18

19

20
**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.

\[
\text{define (make-sub n )} \\
(\lambda (x) (- x n)) \\
\]

\[
\text{define (map-scale factor ns )} \\
(\text{map} (\lambda (num)(* factor num) ) ns) \\
\]

**More sugar: let**

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

Example:

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

**Local function bindings with let**

- Silly example:
  
  \[
  (\text{define (quad x)} \\
  (\text{let (square (lambda (x) (* x x)))} \\
  (square (square x)))\]) \\
  \]

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

\[
(\text{define (up-to-broken x)} \\
(\text{let (between (lambda (from to)) (if (> from to) null (cons from (between (+ from 1) to))))}) \\
\]

\[
(\text{let ([between (lambda (x) (* x x))])} \\
(square (square x)))\]) \\
\]
letrec to the rescue!

(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 en]} e_body),

id1 ... idn are in the scope of e1 ... en.

Mutual Recursion with letrec

(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 17)
'(#t #f)

Even Better

(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

Exercise: let vs. let* vs. letrec

(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)))))]
      (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 Bindings & Scope 25

Local Bindings & Scope 26

Local Bindings & Scope 27

Local Bindings & Scope 28
Local definitions are sugar for \texttt{letrec}

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

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

Pragmatics: Programming Language Layers

- \texttt{kernel}
- \texttt{primitive values/datatypes}
- \texttt{syntactic sugar}
- \texttt{system libraries}
- \texttt{user libraries}

Local Scope in other languages

What support is there for local scope in Python? JavaScript? Java?
| Kernel | Sugar | Built-in library functions | User-defined library functions |

*Local Bindings & Scope* 33