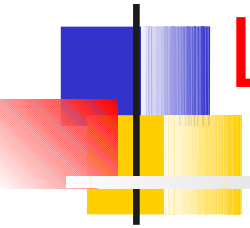


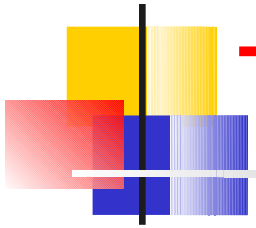
Local definitions and lexical scope

Corky Cartwright

Vivek Sarkar

Department of Computer Science

Rice University



Top-Level Definitions

- We have learned three kinds of definitions thus far:
  1. Function definitions e.g.,

     ```
     (define (f x) (+ x 1))
     ```

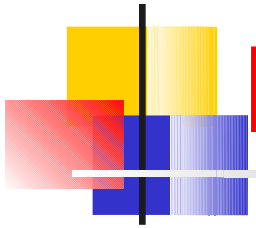
  1. Variable (constant) definitions e.g.,

     ```
     (define two (f 1))
     ```

  1. Structure definitions e.g.,

     ```
     (define-struct cmplx (real imag))
     ```

- They appear in Dr. Scheme's `Definitions` window and are called *top-level definitions*

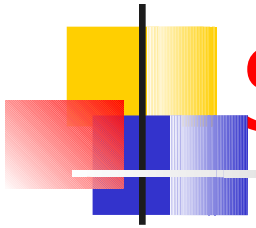


Local Expression

- A *local expression* groups together a set of *disjoint* definitions for use in a subcomputation:

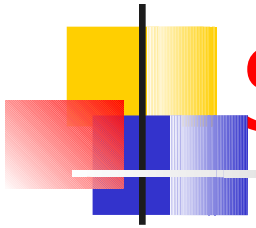
  (**local** (*def*$_1$ *def*$_2$ … *def*$_n$) *exp)*

  - *exp* is an arbitrary expression
  - *def*$_i$ is a definition in the set
  - *def*$_i$ is only available for use within the local expression i.e., within *def*$_1$ *def*$_2$ … *def*$_n$ and *exp*



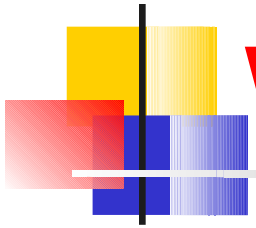
Simple Examples

```
(define x 3) ;; top-level definition

(local ((define x 3)) (+ x 1))   ;; local expression

(define (f x) (+ x 1))  ;; top-level definition

(local ((define x 3)    ;; local definition
        (define (f x) (+ x 1)))    ;; local definition
        (f x))   ;; body

(+ (local ((define x 3) (define (f x) (+ x 1))) (f x)) 1)
;; local-expression as part of another expression
```



Some Incorrect Examples

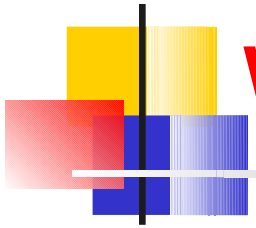
- What's wrong with following expressions?
  - `(local ((define x 1)))`
  - `(local ((define x 1)`
    `(define x 2))`
    `x)`
  - `(local ((define x 1)`
    `(define f (+ x 1)))`
    `(f x))`



Why local?

Reason 1: Avoid namespace pollution

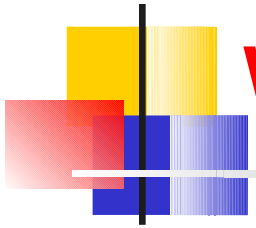
```
;; sort: list-of-numbers -> list-of-number
(define (sort alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon) (insert (first alon)(sort (rest alon)))]))

;; insert: number list-of-numbers (sorted) -> list-of number
;; auxiliary function for sort
(define (insert an alon)
  (cond
    [(empty? alon) (list an)]
    [else (cond   [(> an (first alon)) (cons an alon)]
                  [else (cons   (first alon)
                                (insert an (rest alon)))])]))
```



Why local?

- Reason 1: Avoid namespace pollution (contd)

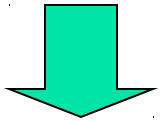
```
;; sort: list-of-numbers -> list-of-numbers
(define (sort alon)

   (local
      (;; insert: number list-of-numbers (sorted) -> list-of numbers
       (define (insert an alon)
               (cond
                  [(empty? alon) (list an)]
                  [else (cond
                          [(> an (first alon)) (cons an alon)]
                  [else  (cons   (first alon)
   (insert an (rest alon)))])]))))


(cond
     [(empty? alon) empty]
     [(cons? alon) (insert (first alon) (sort (rest alon)))]))
```



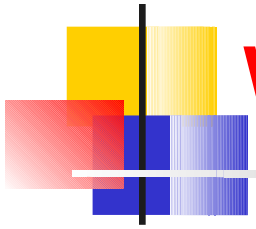
Why local?

Reason 1: Avoid namespace pollution

```
(define (mainFun x) exp)
(define (auxFun1 ...) exp1)
(define (auxFun2 ...) exp2)
```



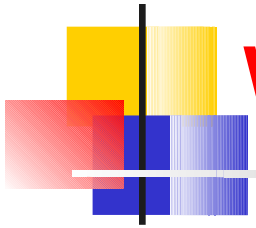
```
(define (mainFun x)
(local ((define (mainFun x) exp)
                (define (auxFun1 …) exp1)
        (define (auxFun2 …) exp2))
 (mainFun x)))
```



Why local?

- Reason 2: Avoid repeated computation

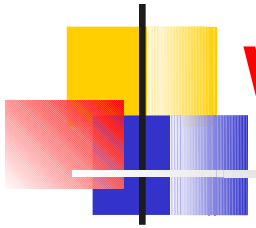
```
;; last-occurrence: number list-of-posn -> number or false
;; (last-occurrence x lop) returns y such that (make-posn x y)
;;   is the last posn p in lop with (posn-x p) = x;
;;   returns "false" if no such posn is found.
(define (last-occurrence x lop)
      (cond
          [(empty? lop) ...]
          [else ... (first lop)
           ... (last-occurrence x (rest lop)) ...]))
```



Why local?

- Reason 2: Avoid repeated computation

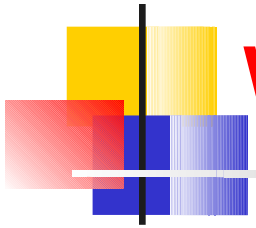
```
(define (last-occurrence x lop)
    (cond
     [(empty? lop) false]
     [else
        (cond
           [(number? (last-occurrence x (rest lop)))
            (last-occurrence x (rest lop))]
                 [(equal? (posn-x (first lop)) x) (posn-y (first lop))]
                 [else false])]))
```



Why local?

- Reason 2: Avoid repeated computation

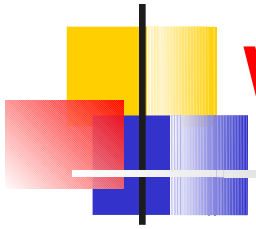
```
(define (last-occurrence x lop)
    (cond
       [(empty? lop) false]
       [else
          (cond
             [(number? (last-occurrence x (rest lop)))
              (last-occurrence x (rest lop))]
                   [(equal? (posn-x (first lop)) x) (posn-y (first lop))]
                   [else false])]))
```



Why local?

- Reason 2: Avoid repeated computation

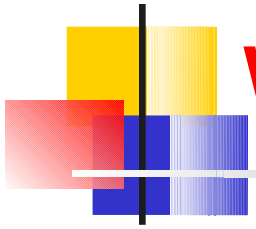
```
(define (last-occurrence x lop)
    (cond
      [(empty? lop) false]
      [else (local ((define y (last-occurrence x (rest lop))))
                   (cond
                      [(number? y)  y]
                            [(equal? (posn-x (first lop)) x) (posn-y (first lop))]
                            [else false]))]))
```



Why local?

- Reason 3: Naming complicated expressions

```
;; mult10 : list-of-digits  ->  list-of-numbers
;; creates a list of numbers by multiplying each digit in alod
;; by (expt 10 p) where p is the number of digits that follow
;; This is bad code used only as an example.  Good code
;; requires refactoring techniques we haven't learned yet.
(define (mult10 alod)
  (cond
    [(empty? alod) empty]
    [else (cons (* (expt 10 (length (rest alod))) (first alod))
                (mult10 (rest alod)))]))
```



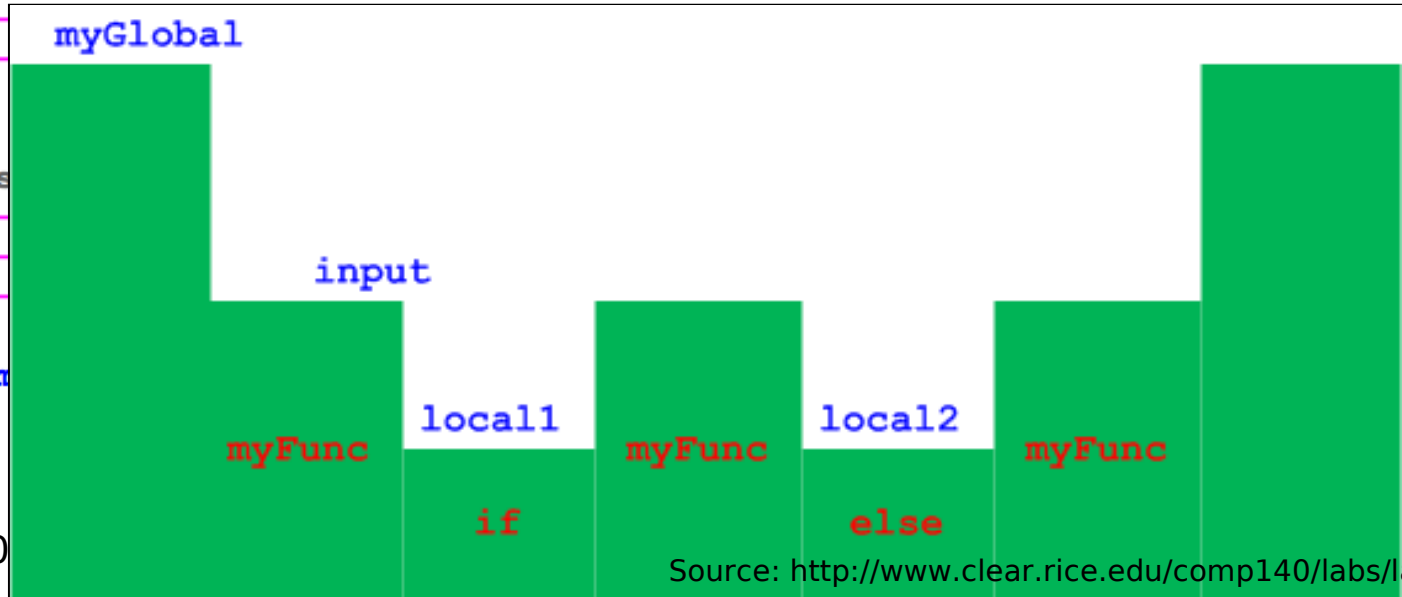
Why local?

- Reason 3: Naming complicated expressions

```
;; mult10 : list-of-digits  ->  list-of-numbers
;; creates a list of numbers by multiplying each digit on alod
;; by (expt 10 p) where p is the number of digits that follow
(define (mult10 alod)
  (cond
    [(empty? alod) 0]
    [else (local ((define a-digit (first alod))
                  (define the-rest (rest alon))
                  (define p (length the-rest)))
            (cons (* (expt 10 p) a-digit) (mult10 the-rest)))]))
```

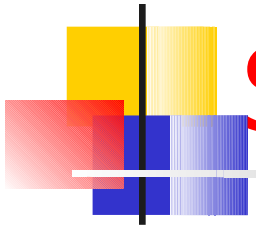
Recap of Variable Scopes from COMP 140

```
myGlobal = 42

def myFunc(input):
    print "myFunc: input = ", input
    print "myFunc: myGlobal = ", myGlobal # global variable visible here
    # neither local1 nor local2 are accessible here.
    if input > 0:
        local1 = 100
        # cannot access local2 from here.
        print "myFunc-if: local1 = ", local1
        print "myFunc-
        print "myFunc-
    else:
        local2 = -100
        # cannot acces
        print "myFunc-
        print "myFunc-
        print "myFunc-

print "myGlobal = ", m
myFunc(5)
myFunc(-5)
```

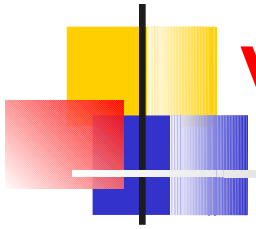


Source: http://www.clear.rice.edu/comp140/labs/l

Variables and Scope in Scheme



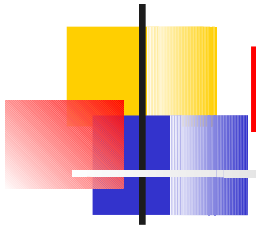
- Example:
  - (local ((define answer$_1$ 42)
    (define (f_2 x_3) (+ 1 x_4)))
    (f_5 answer$_6$))
- Variable occurrences: 1-6
  - Binding (or defining) occurrences: 1,2,3
  - Use occurrences: 4,5,6
  - Scope = code region where a definition may be used
- Scopes of definitions
  - 1:?
  - 2:?
  - 3:?



Variables and Scope

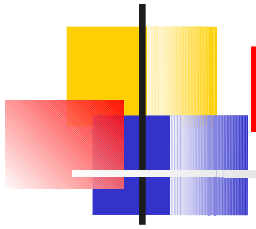
- What will g evaluate to?

```
(define x 0)
(define f x)
(define g (local ((define x 1)) f))
```



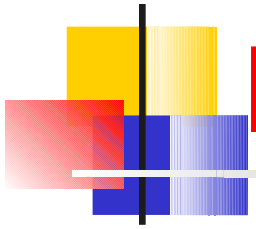
Renaming

- Recall:
  - (local ((define answer$_1$ 42)
    (define (f$_2$ x$_3$) (+ 1 x$_4$)))
    (f$_5$ answer$_6$))
- Which variables can be renamed within the local expression?
- Use the same name for “binding occurrence” and all its “use occurrences”
  - (local ((define answer 42)
    (define (f x) (+ 1 x)))
    (f answer))
- What name choices can be used? Any name that does not clash with variable names already visible in same scope. A “fresh” variable name.



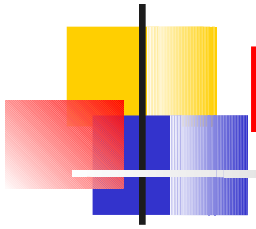
Renaming

- Recall:
  - (local ((define $answer_1$ 42)
    (define (f_2 x_3) (+ 1 x_4)))
    (f_5 $answer_6$))
- Which variables can be renamed?
- Use the same new name for “binding occurrence” and “use occurrences”
  - (local ((define answer' 42)
    (define (f x) (+ 1 x)))
    (f answer'))



Renaming

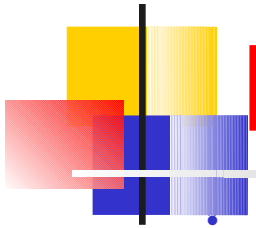
- Recall:
  - (local ((define answer$_1$ 42)
    (define (f_2 x_3) (+ 1 x_4)))
    (f_5 answer$_6$))
- Which variables can be renamed?
- Use the same name for “binding occurrence” and “use occurrence”
  - (local ((define answer 42)
    (define (f' x) (+ 1 x)))
    (f' answer))



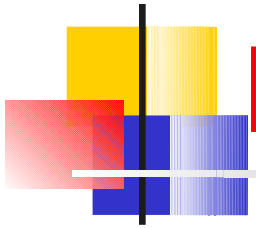
Renaming

- Recall:
  - (local ((define answer$_1$ 42)
    (define (f_2 x_3) (+ 1 x_4)))
    (f_5 answer$_6$))
- Which variables can be renamed?
- Use the same name for “binding occurrence” and “use occurrence”
  - (local ((define answer 42)
    (define (f x') (+ 1 x')))
    (f answer))

Hand Evaluation of Local Expressions



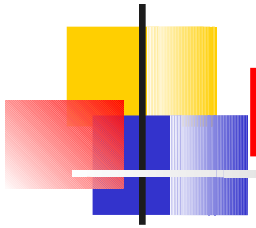
- How do we (hand) evaluate Scheme programs with local?
- By lifting local definitions to the top level and renaming all of the variables that they introduce with *fresh* names to avoid any collisions with variables already defined at the top level.
- To express these laws we need a new format for expressing rules. Why? Because promoting local constructs revises the set of definitions that constitute the *environment* in which evaluation takes place.



Hand Evaluation Example

```
(define x 2)                    ;; top-level definition

;; local-expression as part of another expression
(+ (local ((define x 3) (define (f x) (+ x 1))) (f x)) 1)
=> ???
```

When naming can cause problems



Romeo, Romeo! wherefore art thou Romeo?

. . .

What's in a name? That which we call a roseBy any other name would smell as sweet.

Romeo and Juliet (II, ii)