



# Local definitions and lexical scope

---

Corky Cartwright  
Department of Computer Science  
Rice University



# Issues in HW1

---

- Follow the instructions in HW Guide, particularly the Requirements for programs
  - Hand-evaluation:
    - Missing/combined steps: *exactly one* reduction per step
    - repeated step (only happens in trivial infinite loop)
    - some `cond` rules eliminate a question-answer pair, which *is* a reduction step



# Issues in HW1 cont.

---

- Follow the instructions in HW Guide ...
  - Programming
    - Include every inductive data definition (including whatever forms of lists you may use) as a comment.
    - Data definition (other than a trivial struct) includes the template for processing that form of data and examples of the data.
    - All function definitions (*including* auxiliary/help functions) must be preceded by the (type) contract, purpose statement, examples (which will be used for testing), and the template instantiation (which customizes the function name and the form of recursive calls).
    - In some cases the template instantiation is degenerate (e.g., the definitions of `area-of-ring` or `average-price`):
      - `(define (average-price lon) ... )`



# Issues in HW1 cont.

---

- Follow the instructions in HW Guide ...
  - Programming ...
    - Follow the examples in the HW Guide and use the same formatting.
    - Devise a representative set of tests. Focus on different possible cases in small examples including obvious boundary cases. Typically, at least 5 examples are necessary for reasonable "coverage".
    - DO NOT test for errors that are inconsistent with the input type!
      - Extraneous error testing makes code ugly, hard to understand
      - Computationally wasteful
      - Inhibits code factoring (polymorphism), *e.g.* `how-many-symbols`, `how-many-numbers`



# Definition

- BNF Syntax (cryptic inductive definition) for **local**
  - $exp ::= \dots \mid (\mathbf{local} (def_1 def_2 \dots def_n) exp)$
  - $def ::= (\mathbf{define} var exp) \mid (\mathbf{define} (var_1 var_2 \dots var_n) exp)$

In many contexts, the names of syntactic categories are enclosed in pointy brackets rather than italicized, *e.g.* `<var>` instead of *var*

- Simple examples
  - `(define x 3)` ;; Top-level variable definition
  - `(define (f x) (+ x 1))` ;; Top-level function definition
  - `(define-struct entry (name zip phone))` ;; Structure definition



# Definition

---

- Simple examples
  - `(define x 3)`
  - `(local ((define x 3)) (+ x 1))`
  - `(define (f x) (+ x 1))`
  - `(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



# Definition

---

- 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-numbers
(define (sort alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon) (insert (first alon)
                          (sort (rest alon)))]))

;; 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)))]))]))
```





# Why local?

- Reason 1: Avoid namespace pollution

```
;; 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]
```

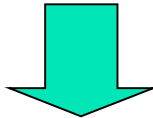
```
    [(cons? alon) (insert (first alon) (sort (rest alon)))]))
```



# Why local?

- Reason 1: Avoid namespace pollution

```
(define (main_fun x) exp)
(define (aux_fun1 ...) exp1)
(define (aux_fun2 ...) exp2)
```



```
(define (main_fun x)
  (local ((define (main_fun x) exp)
    (define (aux_fun1 ...) exp1)
    (define (aux_fun2 ...) exp2))
  (main_fun 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

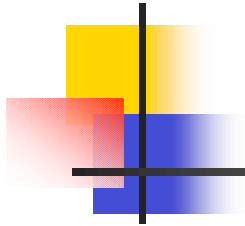
;; returns "false" if no such posn is found.

```
(define (last-occurrence x lop)
```

```
  (cond
```

```
    [(empty? lop) ...]
```

```
    [else ... (first lop) ... (last-occurrence x (rest lop))...]))
```



- (last-occurrence x lop)
  - lop = empty
    - (last-occurrence x lop) = ?
  - lop = cons((make-posn x' y') lop-rest)
    - (last-occurrence x lop-rest) = y ;; y is a number
      - (last-occurrence x lop) = ?
    - (last-occurrence x lop-rest) = false and x' = x
      - (last-occurrence x lop) = ?
    - (last-occurrence x lop-rest) = false and x' != x
      - (last-occurrence x 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]]))
```



repeated work



## 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 refacotring 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)))]))
```



# Variables and Scope

---

- Example:
  - `(local ((define answer1 42)`  
    `(define (f2 x3) (+ 1 x4)))`  
    `(f5 answer6))`
- Variable occurrences: 1-6
  - Binding (or defining) occurrences: 1,2,3
  - Use occurrences: 4,5,6
- Scopes:
  - 1:?
  - 2:?
  - 3:?



# Variables and Scope

---

- Recall:
  - `(local ((define answer1 42)  
          (define (f2 x3) (+ 1 x4)))  
          (f5 answer6))`
- Variable occurrences: 1-6
  - Binding (or defining) occurrences: 1,2,3
  - Use occurrences: 4,5,6
- Scopes:
  - 1: (all of local expression)
  - 2: (all of local expression)
  - 3: (+1 x)



# Variables and Scope

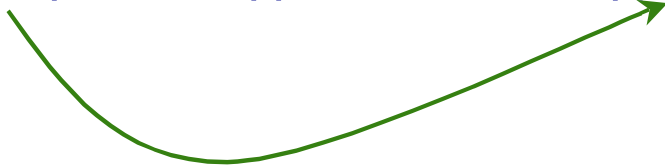
---

- What will g evaluate to?
  - (define x 0)
  - (define f x)
  - (define g (local ((define x 1)) f))



# Variables and Scope

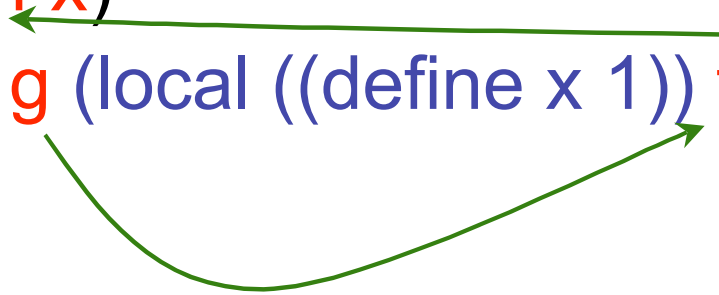
---

- What will g evaluate to?
    - (define x 0)
    - (define f x)
    - (define g (local ((define x 1)) f))
- 



# Variables and Scope


---

- What will g evaluate to?
    - (define x 0)
    - (define f x)
    - (define g (local ((define x 1)) f))
- 



# 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 answer1 42)`  
`(define (f2 x3) (+ 1 x4)))`  
`(f5 answer6))`
- 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))`
- 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 answer1 42)  
          (define (f2 x3) (+ 1 x4)))  
          (f5 answer6))`
- 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 answer1 42)`  
`(define (f2 x3) (+ 1 x4)))`  
`(f5 answer6))`
- 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 answer1 42)`  
`(define (f2 x3) (+ 1 x4)))`  
`(f5 answer6))`
- 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))`



## Evaluation Laws (bonus material)

- 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 (for which they create binding occurrences) 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.
- New format: we evaluate a sequence of **define** forms followed by an expression (which we formerly called the program application) which yields the answer for the computation.

## Evaluation Laws (bonus)

- To simplify formalizing semantics of **local**, we need to introduce new notation for functions which is explained in more depth in Lectures 9-10. A *function value* is **(lambda**  $(x_1 \dots x_n)$   $M$ ) where  $x_1 \dots x_n$  are variables and  $M$  is an expression. Note that we are defining a new form of value.
- The form **(define**  $(y \ x_1 \dots x_n)$   $M$ ) where  $y \ x_1 \dots x_n$  are variables and  $M$  is an expression abbreviates **(define**  $y$  **(lambda**  $(x_1 \dots x_n)$   $M$ )).
- A define form **(define**  $y$   $M$ ) is *reduced* iff  $M$  is a value.
- A *program suite* is:  
**(define**  $y_1$   $M_1$ ) ... **(define**  $y_n$   $M_n$ )  $M$   
 where  $y_1 \dots y_n$  are variables and  $M_1 \dots M_n \ M$  are expressions.



## Revised Evaluation Laws cont.

---

- Our revised rewriting semantics explains how to evaluate program suites. All of our former laws, specifying reductions (the rewriting of program text) still apply with two revisions/exceptions.
  - The law for reducing the application of a program-defined function `f` to values does not use a “given” program; it looks up the value of `f` in the list of `define` forms preceding the application in the program suite. If no such value exists, the application is a run-time error.
  - The law for reducing `error` applications will be given along with the law for evaluating `local`.
- We need to introduce another definition first.

## Evaluation Laws (bonus)

- An *evaluation context*  $E$  is either:
  - $[ \ ]$  (called "hole")
  - $(p \ V_1 \ \dots \ V_m \ E \ M_1 \ \dots \ M_n)$
  - $((\text{lambda } (x_1 \ \dots \ x_k) \ M) \ V_1 \ \dots \ V_m \ E \ M_1 \ \dots \ M_n)$
  - $(\text{cond } (E \ M) \ \dots \ )$
  - $(\text{local } (R_1 \ \dots \ R_m \ (\text{define } y \ E) \ D_1 \ \dots \ D_n) \ M)$
- where  $p$  is a primitive function symbol;  $V_1 \ \dots \ V_m$  are values;  $M_1 \ \dots \ M_n$  are expressions;  $x_1 \ \dots \ x_k \ y$  are variables;  $R_1 \ \dots \ R_m$  are reduced define forms; and  $D_1 \ \dots \ D_n$  are define forms.
-



## Evaluation Laws cont.

---

- If  $E$  is an evaluation context and  $M$  is an expression,  $E[M]$  denotes  $E$  with the hole replaced by  $M$ .
- The revised law for **error** is
- $R_1 \dots R_m E[(\text{error } v_1 v_2)] D_1 \dots D_n \Rightarrow$   
`error: sym string`



## Evaluation Laws cont.

- $$\begin{array}{l}
 R_1 \dots R_m \\
 E[(\text{local } ((\text{define } v_1 M_1 \dots (\text{define } v_n M_n)) M)] \\
 D_1 \dots D_n M
 \end{array}$$

$\Rightarrow$

$$\begin{array}{l}
 D_1 \dots D_m \\
 (\text{define } v_1' M_1') \\
 \dots \\
 (\text{define } v_n' M_n') \\
 E[M'] \\
 D_1 \dots D_n M
 \end{array}$$



## For Next Class

---

- Homework due on Friday
  - Potential additions/updates
- Reading:
  - Ch 18: Local Definitions and Lexical Scope
  - Ch 19: Similarities in definition (and “refactoring”)