

Local definitions and lexical scope



Instructor: Prof. Walid Taha
Department of Computer Science
Rice University



Questions from quiz

- Please explain (in some fashion that makes more sense than the book) how using local saves computation.
- Can we spend some time going over local in class?



Definition

- Syntax of local

- `<exp> ::= ... | (local (<def-1><def-2>...<def-n>) body) exp`
- `<def> ::= (define <var> exp)`
`| (define (<var><var>...<var>) exp)` } right hand side
`| (define-struct <var> (<var>...<var>))`

- Simple examples

- `(define x 3)` ;; Top-level variable definition
- `(define (f x) (+ x 2))` ;; 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)
```

```

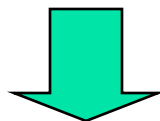
(local ((define (sort alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon) (insert (first alon)
                          (sort (rest alon)))]))
  ;; insert: number list-of-numbers (sorted) -> list-of numbers
  (define (insert an alon)
    (cond
      [(empty? alon) (list an)]
      [else (cond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

```

Why local?

- Reason 1: Avoid namespace pollution

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

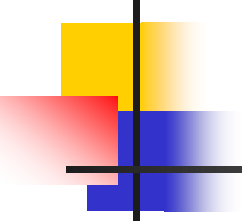


```
(define (main_fun x)
  (local ((define (main_fun x) exp)
          (define (aux_fun1 ...) exp1)
          (define (aux_fun2 ...) exp2)))
  (main_fun x))
```



Why local?

- Reason 2: Avoid repeated computation
 - ;; last-occurrence: x-value, [posn] -> y-value or false
 - ;; give an x value and a list of posn, return y value of a posn s.t.
 - ;; (1) posn-x of that position is x
 - ;; (2) no other position in the list satisfies (1)
 - ;; return “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)**
    - $\text{lop} = \text{empty}$ 
      - $(\text{last-occurrence } x \text{ lop}) = ?$
    - $\text{lop} = (\text{cons } (\text{make-posn } x' \text{ } y') \text{ lop\_rest})$ 
      - $(\text{last-occurrence } x \text{ lop\_rest}) = y \quad ;; y \text{ is a number}$ 
        - $(\text{last-occurrence } x \text{ lop}) = ?$
      - $(\text{last-occurrence } x \text{ lop\_rest}) = \text{false}$  and  $x' = x$ 
        - $(\text{last-occurrence } x \text{ lop}) = ?$
      - $(\text{last-occurrence } x \text{ lop\_rest}) = \text{false}$  and  $x' \neq x$ 
        - $(\text{last-occurrence } x \text{ lop}) = ?$



# 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))]
 [(= (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))]
 [(= (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 lo-rest (last-occurrence x (rest lop))))
 (cond
 [(number? Lo-rest)
 lo-rest]
 [(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
    - ;; to create a list of numbers by multiplying each digit on alod
    - ;; by (expt 10 p) where p is the number of digits that follow
    - ;; Example: (mult10 (list 1 2 3)) => (list 100 20 3)
- ```
(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
```

```
;; to create 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 alod))
```

```
                  (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 statement)
- 2:(all of local statement)
- 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

---

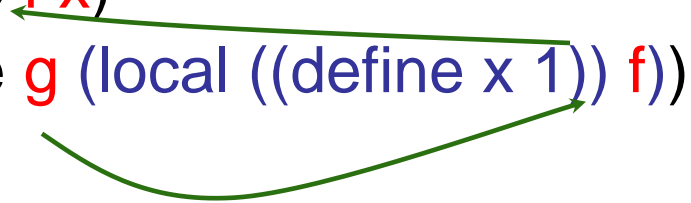
- What will “g” evaluate to?
  - (define x 0)
  - (define f x)
  - (define **g** (local ((define x 1)) **f**))





# 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))`



# 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))`



# 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)`

- Which variable names need to be renamed?

```
(define (sort alon)
```

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

```
 (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)))])])))
```

```
 (sort alon))
```

- Which variable names need to be renamed?

```
(define (sort alon)
```

```
 (local ((define (local_sort alon)
 (cond
 [(empty? alon) empty]
 [(cons? alon) (insert (first alon)
 (local_sort (rest alon)))])))
```

```
 (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)))])])))
```

```
 (local_sort alon)))
```



# For Next Class

---

- Homework due on Monday
  - Note additions/updates
- Reading:
  - Ch 19: Similarities in definition (and “re-factoring”)
- Quiz on reading