



Local definitions and lexical scope

Corky Cartwright
Department of Computer Science
Rice University



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)]  
    [(cons? alon) (if (< an (first alon))  
                      (cons an alon)  
                      (cons (first alon) (insert an (rest alon))))]))
```



Why local?

- Reason 1: Avoid namespace pollution

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

```
    ;; insert: number list-of-numbers (sorted) -> list-of numbers
```

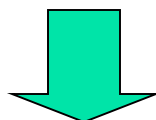
```
    ((define (insert an alon)
      (cond
        [(empty? alon) (list an)]
        [else (if (< an (first alon))
                  (cons an alon)
                  (cons (first alon) (insert an (rest alon))))]))
```

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



repeated work

```
(define (power los)
  (cond [(empty? los) (list empty)]
        [(cons? los)
         (append (cons-all (first los) (power (rest los)))
                  (power (rest los)))]))
```




Why local?

- Reason 2: Avoid repeated computation

```
(define (power los)
  (cond [(empty? los) (list empty)]
        [(cons? los)
         (local ((define pow (power (rest los)))
                 (append (cons-all (first los) pow) pow)))]))
```



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 following digits
;; 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))]))])
```



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

- In the following code segment, what will `g` evaluate to?

```
(define x 0)
```

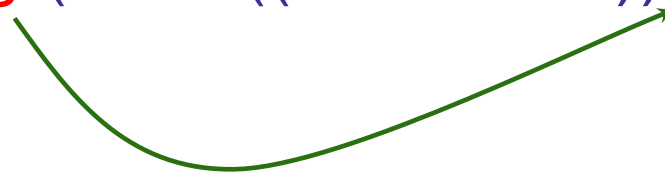
```
(define f x)
```

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



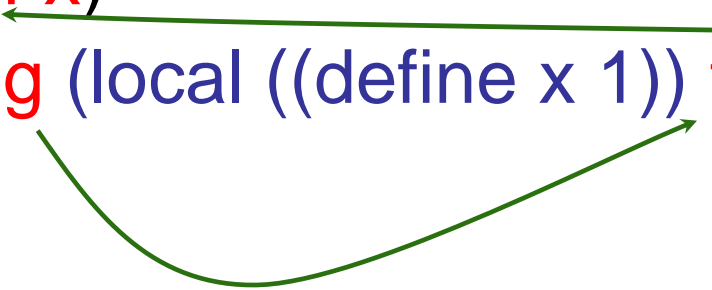
Variables and Scope

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



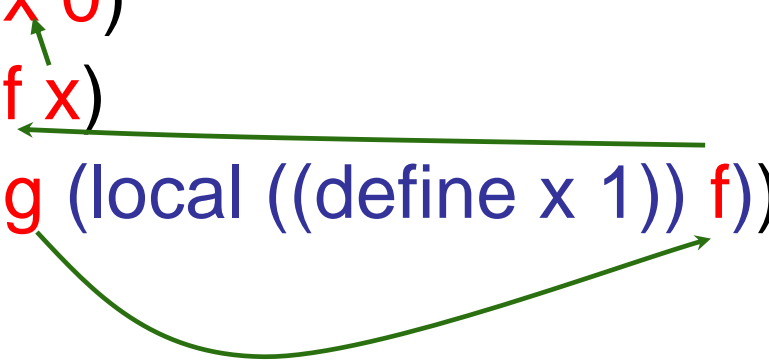


Variables and Scope

- 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

- How do we (hand) evaluate Racket 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

- To be continued ...