Local Definitions, Scope, Functional Abstraction, and Polymorphism
Definitions

- We've seen **global** definitions:
  - `(define answer 42)    ;; "42" is RHS
  - `(define (f x) (+ 1 x))

- Today we'll look at **local** definitions:
  - `(local ((define answer 42)
               (define (f x) (+ 1 x)))
       (f answer))    ;; body
Formalities

- What does the BNF for local definitions look like?
  - `<exp> ::= … | (local (def+) exp)

- Indentation style

- Semantics (informally)
Why "local"?

- **Avoiding namespace pollution**
  - Example: Our insert sort function
  - This is an example of encapsulation
  - How useful is this concept?
    - Reuse name, avoid complex contracts, organize, hiding implementation detail

- **Avoiding repeated work**
  - Consider:
    
    ```
    ;; last-occurrence:
    ;;    x-value, [posn] -> y-value or false
    ```
Variables and Scope

- Recall:
  - `(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**

- Scopes: 1: *(all of local statement)*, 2: *(all of local statement)*, 3: *(+1 x)*
Renaming

- Recall:
  - `(local ((define answer₁ 42)
              (define (f₂ x₃) (+ 1 x₄)))
    (f₅ answer₆))`
  - ? variables can be safely renamed
  - But only ? can be renamed in
    - `(define answer₁ 42)`
    - `(define (f₂ x₃) (+ 1 x₄))`
Abstracting Designs

- “The elimination of repetitions is the most important step in the (program) editing process” – Textbook
- Lots of “cut and paste” is generally a bad thing. Anyone have this experience?
- “Abstractions” avoid this problem
Similarities in Functions

- Simple example:

  ;; contains-doll? : los -> boolean
  ;; to determine whether alos contains
  ;; the symbol 'doll

  (define (contains-doll? alos)
    (cond [(empty? alos) false]
      [else (or (symbol=? (first alos) 'doll)
                  (contains-doll? (rest alos)))]]))
Similarities in Functions

- Simple example:

  ;; contains-car? : los -> boolean
  ;; to determine whether alos contains
  ;; the symbol 'car

  (define (contains-car? alos)
    (cond [(empty? alos) false]
          [else (or (symbol=? (first alos) 'car)
                    (contains-car? (rest alos)))]))
Similarities in Functions

Fix:

;;; contains? : symbol, los -> boolean
;;; to determine whether alos contains
;;; the symbol s

(define (contains? s alos)
    (cond [(empty? alos) false]
          [else (or (symbol=? (first alos) s)
                     (contains? s (rest alos)))]))
Similarities in Functions

- Fix:
- What used to be redundant code is now

\[
\text{(define } \text{ (contains-doll? alos) (contains? 'doll alos)})
\]

\[
\text{(define } \text{ (contains-car? alos) (contains? 'car alos)})
\]

- We’ll take this idea and run with it
Second example:

;; below : lon number  ->  lon
;; to construct a list of those numbers
;; in alon that are below t
(define (below alon t)
  (cond [(empty? alon) empty]
        [else (cond [(< (first alon) t)
                     (cons (first alon) (below (rest alon) t))]
                  [else (below (rest alon) t)]]]))
Second example:

;; above : Ion number  ->  Ion
;; to construct a list of those numbers
;; in alon that are above t
(define (above alon t)
  (cond [(empty? alon) empty]
        [else (cond [(> (first alon) t)
                        (cons (first alon) (above (rest alon) t))]
                   [else (above (rest alon) t)])))))
It will get real cool up here...

- **Fix:**

```lisp
;; filter : comparison, lon number -> lon
;; to construct a list of those numbers n
;; in alon such that (test t n) is true
(define (filter test alon t)
  (cond [(empty? alon) empty]
        [else (cond [(test (first alon) t)
                           (cons (first alon) (filter test (rest alon) t))]
                   [else (filter test (rest alon) t)]))])
```

- How can you even use this thing?
It will get real cool up here…

- Fix:

- The magic moment:

  (define (below alon t) (filter > alon t))

  (define (below alon t) (filter < alon t))

- Both functions will work just as before
Similarity in Types

- Good news: You already know this:
  - [number], [symbol], [bool] have similar definitions
- We’ve already been using [X]
- Defining [X] requires is variables at the level of types...the X in [X]!
- The book uses the intuitive (listOf X)
- Often called polymorphism, or generics.
Final thought

- Function abstraction gives a lot of expressivity
- Polymorphism does the same
- Together, both work *really* well
- Things will continue to get better as we add abstraction mechanisms to our language