Reminders:
1. Next exam is a take-home; handed out 3/17/00, due 3/22/00 (5pm in my office).
2. Exam will cover through last lecture. I will post the lecture notes for last lecture this afternoon, along with a revised explanation of how local actually translates the name space.
3. This week's homework will be a half-homework.

Review
1. We worked more examples with local. We tried to hammer home, by repetition, the ideas behind local.

On to Functional Abstraction
Write a simple function that consumes a list of numbers and produces a list of numbers. The numbers in the returned list should be exactly those numbers in the original list that are less than 5 (in the same order as the original list).

;; keep-lt-5 : list of numbers -> list of numbers
;; Purpose: keeps all input numbers less than 5
(define (keep-lt-5 alon)
  (cond
   [emptyp? alon] empty
   [(cons? alon)
    (cond
     [(< (first alon) 5) (cons (first alon) (keep-lt-5 (rest alon)))]
     [else (keep-lt-5 (rest alon))]]))
)

What about keep-lt-9 ?

;; keep-lt-9 : list of numbers -> list of numbers
;; Purpose: keeps all input numbers less than 9
(define (keep-lt-9 alon)
  (cond
   [emptyp? alon] empty
   [(cons? alon)
    (cond
     [(< (first alon) 9) (cons (first alon) (keep-lt-9 (rest alon)))]
     [else (keep-lt-9 (rest alon))]]))
Notice how these two functions have in common. Can we write one function that captures all this common code (single-point of control) and use it to implement keep-lt-5 and keep-lt-9?

;; keep-lt: number list-of-numbers -> list-of-numbers
;; Purpose: keep all input numbers that are less than the given number
(define (keep-lt num alon)
  (cond
   [(empty? alon) empty]
   [(cons? alon)
     (cond
      [(< (first alon) num)
        (cons (first alon) (keep-lt num (rest alon)))]
      [else (keep-lt num (rest alon))])]
   ))

Notice that num never changes. We could use a local to avoid passing it around in so many places (and save work) [But, efficiency isn't a concern in the 1st part of Comp 210]

;; keep-lt: number list-of-numbers -> list-of-numbers
;; Purpose: keep all input numbers that are less than the given number
(define (keep-lt num alon)
  (local
   [(define (filter-lt alon)
     (cond
      [(empty? alon) empty]
      [(cons? alon)
        (cond
         [(< (first alon) num)
          (cons (first alon) (filter-lt num (rest alon)))]
         [else (filter-lt num (rest alon))])]
      ]
    )]
  (filter-lt alon))
))

Using keep-lt, we can define keep-lt-5 and keep-lt-9

(define (keep-lt-5 alon)
  (keep-lt 5 alon))

(define (keep-lt-9 alon)
  (keep-lt 9 alon))
What if we wanted to write keep-gt-5

;; keep-gt-5 : list of numbers -> list of numbers
;; Purpose: keeps all input numbers greater than 5
(define (keep-gt-5 alon)
  (cond
   [(empty? alon)   empty]
   [(cons? alon)
    (cond
     [(> (first alon) 5)
      (cons (first alon) (keep-gt-5 (rest alon)))]
     [else  (keep-gt-5 (rest alon))]
    )]
  ))

Where do these functions differ? Only in the comparison operator and in the names of the functions. [The last lecture should have convinced you that the names are malleable.] How can we reuse the common code here? Previously, we made the upper limit on the value into a parameter. Now, we need to make the comparison operation itself be a parameter.

Aside
How do we represent > in the contract?  (number number -> number)
We've been writing these contracts for eight weeks now. This should be pretty natural.

Back To Abstracting Out Comparison

;; keep-rel-5 : (num num -> num) list of numbers -> list of numbers
;; Purpose: keeps all input numbers that have relation than 5
(define (keep-rel-5 relation alon)
  (cond
   [(empty? alon)   empty]
   [(cons? alon)
    (cond
     [(relation (first alon) 5)
      (cons (first alon) (keep-rel-5 relation (rest alon)))]
     [else  (keep-rel-5 relation (rest alon))]
    )]
  ))

and

(define (keep-lt-5 alon)
  (keep-rel-5 < alon))

(define (keep-gt-5 alon)
  (keep-rel-5 > alon))
As before, we can use local in the obvious way to avoid passing relation as a parameter.

;; keep-rel-5 : (num num -> num) list of numbers -> list of numbers
;; Purpose: keeps all input numbers that have relation than 5
(define (keep-rel-5 relation alon)
  (local
    [(define (filter-rel alon)
        (cond
          [(empty? alon)   empty]
          [(cons?   alon)
            (cond
              [(relation (first alon) 5)
                (cons (first alon) (filter-rel (rest alon)))
              ]
              [else (filter-rel (rest alon))])
          ])
    ]
    (filter-rel alon)))

(define (keep-lt-5 alon)
  (keep-rel-5 < alon))

Of course, the next thing we want to do is abstract out the number 5. We should be able to write a function that takes both the relation and the limit as parameters and returns a list containing the specified subset of the numbers in the original list.

;; keep-rel  (num num -> num) num  list-of-nums -> list-of-nums
;; Purpose: keep all the numbers in the input list that have the relation given
;; by the function argument to the number argument (whew!)
(define (keep-rel relation num alon)
  (local [(define filter-rel alon)        ;; treat relation & num as invariant
    (cond
      [(empty? alon)   empty]
      [(cons? alon)
        (cond
          [(relation (first alon) num)
            (cons (first alon) (filter-rel (rest alon)))
          ]
          [else (filter-rel (rest alon))])
      ])
    (filter-rel alon))
  )

(define (keep-gt-9 alon)
  (keep-rel > 9 alon))

*Enough for one day.*