## *Administrative Announcements*

- Welcome back
- Homework due Wednesday, as usual
- Exam next week
  - → Will cover material through Friday's lecture
  - → Will cover lab lectures including this week's lab
  - → Either Wednesday in class (1 hour) or in *7-9pm (2 hours)*
  - → Closed notes, closed book



## *Functional Abstraction*

What's abstraction?

- Have built programs that take parameters
  - → Fundamental form of abstraction
- What about higher levels of abstraction?
  - → Abstracting over functionality

## Simple programs

Program to pick out the numbers < 5 in a list

```
;; keep-lt-5 : list-of-numbers -> list-of-numbers
;; Purpose: result contains every number < 5 in the input list
(define (keep-lt-5 alon)
   (cond
      [(empty? alon)          empty]
      [(cons?   alon)
       (cond
          [(< (first alon) 5)  (cons (first alon) (keep-lt-5 (rest alon)))]
          [else                (keep-lt-5 (rest alon))]
       )] ))
```

*Followed the standard list template*

*Might clean it up with a helper function, but …*



## Simple programs

Program to pick out the numbers < 9 in a list

```
;; keep-lt-9 : list-of-numbers -> list-of-numbers
;; Purpose: result contains every number < 9 in the input list
(define (keep-lt-9 alon)
   (cond
      [(empty? alon)          empty]
      [(cons?   alon)
       (cond
          [(< (first alon) 9)  (cons (first alon) (keep-lt-9 (rest alon)))]
          [else                (keep-lt-9 (rest alon))]
       )] ))
```

## Simple programs

Program to pick out the numbers < x in a list

```
;; keep-lt : list-of-numbers number -> list-of-numbers
;; Purpose: result contains every number < x in the input list
(define (keep-lt alon x)
   (cond
      [(empty? alon)          empty]
      [(cons?   alon)
       (cond
          [(< (first alon) x)   (cons (first alon) (keep-lt (rest alon) x))]
          [else                 (keep-lt (rest alon) x)]
        )] ))
```

*Abstracted out the upper bound*



## Simple programs

Using local to ellide the invariant parameter

```
;; keep-lt : list-of-numbers number -> list-of-numbers
;; Purpose: result contains every number < x in the input list
(define (keep-lt alon x)
   (local
      [(define (filter-lt alon)
          (cond
             [(empty? alon)   empty]
             [(cons?   alon)
              (cond
                 [(< (first alon) x) (cons (first alon) (filter-lt (rest alon)))]
                 [else                (filter-lt (rest alon))]
               )] )) ]
       (filter-lt alon)
      ))
```

## Simple programs

### Back to keep-lt-5 and keep-lt-9

```
;; with keep-lt defined as on the last slide …

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

(define (keep-lt-9 alon)
    (keep-lt  alon 9))
```

- *Looks a lot easier than writing separate code for each one*
- *Creates single-point-of-control on keep-lt*



## Simple programs

### Program to pick out the numbers > 5 in a list

```
;; keep-gt-5 : list-of-numbers -> list-of-numbers
;; Purpose: result contains every number > 5 in the input list
(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))]
       )] ))
```

*All we did was change the comparison operator*

*Next, we can abstract the number and ellide the invariant*

*Eventually, we end up with keep-gt*

## Simple programs

```
;; keep-gt : list-of-numbers number -> list-of-numbers
;; Purpose: result contains every number > x in the input list
(define (keep-gt alon x)
   (local
      [(define (filter-gt alon)
          (cond
             [(empty? alon)   empty]
             [(cons?   alon)
              (cond
                 [(> (first alon) x) (cons (first alon) (filter-gt (rest alon)))]
                 [else                (filter-gt (rest alon))]
              )] )) ]
       (filter-gt alon)
     ))
```

*Can we abstract out < and > ?*



## Critical Aside

How would we represent < and > ?

- We need a contract
  - → <: (number number -> boolean)
  - → >: (number number -> boolean)
- We need a name
  - → What do we call < and > ?
  - → How about < and > ?
  - → (fee 1) invokes a program named fee
  - → Does (< 2 3) invoke a program named < ?
- Programs have names and can be passed around like values
  - → Programs are values in some more complicated algebraic space

## Back to abstraction

Abstract keep-lt-5 and keep-gt-5

```
;; keep-rel-5 : list-of-numbers (number number -> bool)  -> list-of-numbers
;; Purpose: result contains every number where "relation n 5" is true
(define (keep-rel-5 alon rel)
   (cond
      [(empty? alon)          empty]
      [(cons?   alon)
       (cond
          [(rel  (first alon) 5)(cons (first alon) (keep-rel-5 (rest alon) rel))]
          [else                 (keep-rel-5 (rest alon) rel)]
        )] ))

(define (keep-lt-5 alon)
  (keep-rel-5 alon <))
(define (keep-gt-5 alon)
   (keep-rel-5 alon >))
```



## Following the yellow brick road …

And, we can pass in the number and ellide invariants …

```
;; keep-rel: list-of-nums (num num -> bool) num -> list-of-nums
;; Purpose: keep the numbers specified by relation and x
(define (keep-rel alon rel x)
   (local
     [(define (filter-rel alon)   ;; treat rel and x as invariants
        (cond
          [(empty? alon)     empty]
          [(cons?   alon)
           (cond
              [(rel  (first alon) x)
               (cons (first alon) (filter-rel (rest alon)))]
              [else              (filter-rel (rest alon))]  )] )) ]
      (filter-rel alon) ))

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

} *And so on …*

## What about other comparisons?

Does this work for programs you write?

- < and > are built into Scheme
- What about a program you write?

```
;; between?: number number number -> boolean
;; Purpose: takes lower and upper bound, plus number
;;          returns true if number is between lower & upper bound,
;;          inclusive (lb <= x <= ub)
(define (between? lb ub x)
   (and (<= lb x) (<= x ub)))
```

- Can we pass this to keep-rel and have it work?
  - → No, the contract is wrong
  - → But, we can develop keep-bet with all the gory details



## Abstracting out the differences

So far, we've abstracted out numbers and programs

- All had the same contracts …
- Can we abstract away the contract?
  - → Lets sidestep this for a slide or two
- Look at the common code in all these applications

*Reordered parameters to make the differences come first*

```
(define (keep  … alon)
  (local
    [(define (filter  alon)
       (cond [(empty? alon) empty]
             [(cons?   alon)
              (cond
                [( … (first alon) …)  (cons (first alon) (filter (rest alon)))]
                [else  (filter (rest alon))] )] ))]
      (filter alon) ))
```

## Abstracting out the differences

Let's fill in the gaps

*keep-elt? must be (number -> boolean)*

```
(define (keep  keep-elt?  alon)
  (local
    [(define (filter  alon)
       (cond [(empty? alon) empty]
             [(cons?   alon)
              (cond
                [( keep-elt? (first alon)) (cons (first alon) (filter (rest alon)))]
                [else  (filter (rest alon))] )] ))]
      (filter alon) ))
```



## Using keep

```
(define (keep  keep-elt?  alon)
  (local
    [(define (filter  alon)
       (cond [(empty? alon) empty]
             [(cons?   alon)
              (cond
                [( keep-elt? (first alon)  (cons (first alon) (filter (rest alon)))]
                [else  (filter (rest alon))] )] ))]
      (filter alon) ))

(define (keep-lt-5 alon)
   (local [(define (lt-5? x) (< x 5))]
          (keep  lt-5?  alon)
    ))
(define (keep-bet-5-9 alon)
   (local [(define (bet-5-9? x) (and (<= 5 x) (<= x 9)))]
          (keep  bet-5-9?  alon) ))
```

# Filter

```
(define (keep  keep-elt?  alon)
  (local
    [(define (filter  alon)
       (cond [(empty? alon) empty]
             [(cons?   alon)
              (cond
                [( keep-elt? (first alon)  (cons (first alon) (filter (rest alon)))]
                [else  (filter (rest alon))] )] ))]
      (filter alon) ))
```

*Keep is so useful that Scheme provides a built-in version*

- *Of course, Scheme's version is less restrictive*
- *It isn't limited to numbers*

*filter: (alpha->boolean) list-of-alpha -> list-of-alpha*

*where alpha is a kind of data, i.e, number, symbol, list, structure, …*