## Administrative Announcements

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

COMP 210, Spring 2002

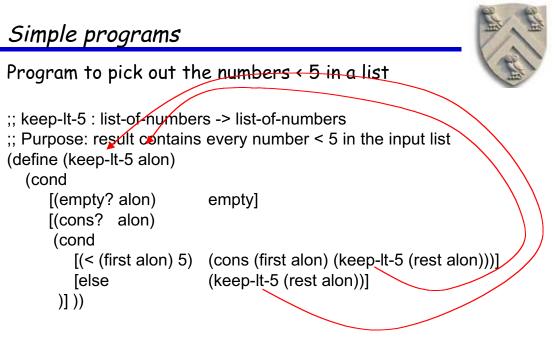
# Functional Abstraction

What's abstraction?

- Have built programs that take parameters
  - $\rightarrow\,$  Fundamental form of abstraction
- What about higher levels of abstraction?
  - $\rightarrow$  Abstracting over functionality







Followed the standard list template Might clean it up with a helper function, but ...

COMP 210, Spring 2002

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

COMP 210, Spring 2002

Simple programs

Using <u>local</u> 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)))]
                                (filter-lt (rest alon))]
              [else
             )]))]
      (filter-lt alon)
    ))
```





## Simple programs

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

;; with keep-It 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

Program to pick out the numbers > 5 in a list

COMP 210, Spring 2002

#### Simple programs

;; 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 > ?

COMP 210, Spring 2002

## Critical Aside

How would we represent < and > ?

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





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))]
                            (keep-rel-5 (rest alon) rel)]
         [else
       )] ))
(define (keep-lt-5 alon)
  (keep-rel-5 alon <))
(define (keep-gt-5 alon)
  (keep-rel-5 alon >))
```

```
COMP 210, Spring 2002
```

Following the yellow brick road ...

```
;; 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
```

And, we can pass in the number and ellide invariants

```
[(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) ))

efine (keep-gt-9 alon)
```

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

```
And so on ...
```





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?
  - $\rightarrow$  No, the contract is wrong
  - $\rightarrow$  But, we can develop keep-bet with all the gory details

COMP 210, Spring 2002

## Abstracting out the differences

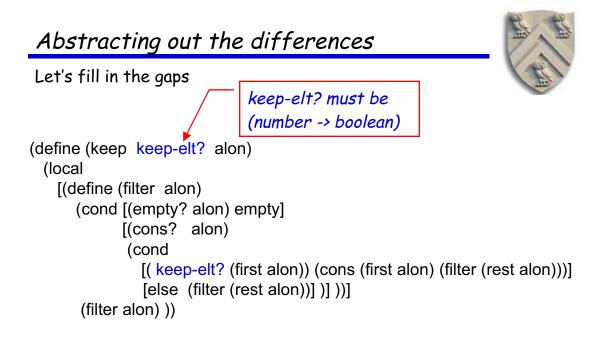
So far, we've abstracted out numbers and programs

- All had the same contracts ...
- Can we abstract away the contract?
  - $\rightarrow$  Lets sidestep this for a slide or two
- Look at the common code in all these applications representations to make

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







COMP 210, Spring 2002

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 It-5? alon)
   ))
(define (keep-bet-5-9 alon)
  (\text{local } [(\text{define (bet-5-9? x) (and (<= 5 x) (<= x 9))})]
         (keep bet-5-9? alon)))
```

16

## 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, ...

COMP 210, Spring 2002