Administrative Announcements

- Who went to the challenge lab?

- Exam
  - → Covers through middle of today's lecture, plus lab lectures
  - → Take home? (hand out Wednesday, due Monday)
  - → Closed notes, closed book
  - → This means we can have Wednesday night lab

Abstract Functions

Capture common functionality

- Scheme provides built-in versions of some important ones
  - → Filter, map, foldl, foldr, ormap, andmap, …
  - → See the lab notes and the book for examples
- Idea is simple
  - → Rather than rewriting code a second time, try to abstract the basic form into a function that you can use for both
- Implementation requires practice
  - → Learning to see patterns, extract them, and use them
  - → Do the homework

Abstract Functions

Consider *map*

- Applies a function to a list, element-by-element
  - → map: (alpha->beta) list-of-alpha -> list-of-beta
  - → Works for any kind of data – alpha & beta
  - → Simple example of phenomenon called parametric polymorphism

Example

```
;; triple: list-of-number -> list-of-number
;; Purpose: compute 3x each number in the list
(define (triple alon)
   (map (lambda (x)(+ x x x)) alon))
```

This example uses lambda

Lambda

Lambda creates anonymous functions

- Quick, compact syntax
- Creates full-fledged functions, albeit without names

- Lambda is the function constructor for Scheme

  (lambda (arg_1 arg_2 … arg_n) expression)
  - → Creates an anonymous function of n arguments

```
(define (is-fee? asym)
   (symbol=? asym 'fee))
```

≡

```
(lambda (asym)
   (symbol=? asym 'fee))
```

Using lambda

What does lambda do?

Dr. Scheme rewrites (lambda (arg_1 arg_2 … arg_n) expression) as

```
(local [(define (a-unique-name arg1 arg2 … argn)
          expression)
        ]
        a-unique-name)
```

Subtle points

- The rewriting process has to concoct the name, not you
- This creates the function & returns it

Lambda

How do lambda & define differ?

```
;; times3: number -> number
(define (times3 x)
   (* 3 x))
```

- *Creates a function that multiplies its input by three*
- *Associates that function with the Scheme object "times3"*

```
;; same function, no name
(lambda (x) (* 3 x))
```

- *Creates an anonymous function that multiplies its input by three*

```
;; times3: number -> number
(define times3
    (lambda (x) (* 3 x)) )
```

- *Binds the anonymous function to the Scheme object "times3"*

Exam Preparation

Major themes since the last test

- Programs that manipulate trees
  - → Child-centric & parent-centric family trees, directories & files
- Programs that have multiple complicated arguments
  - → Merge, flatten, …
  - → Work out the cases, then write the template
- Using local
  - → Replace multiple invocations with single one
  - → Break up complex expressions into simpler, more readable ones
- Abstract functions
  - → Looked at (& used) filter, map, foldl, foldr
  - → Learned to use lambda

Moving on

Structural recursion

- Follows a relationship in the data
  - → Traversing a list, counting down natural numbers
- Derived naturally (almost) from data analysis
- Finite data implies termination

Generative recursion

- Comes from insight into the algorithm
  - → Enumerating possible solutions, applying some rule
- Create new problem instances and manipulate them

This is the final third of COMP 210

Sorting a List of Numbers

You develop mergesort in the homework

→ Let's try a generative approach

The Plan:

1. Pick a representative list element, the pivot
2. Partition the list into two list around the pivot
   - One list has values < pivot, other has values > pivot
3. Sort the smaller lists
   - Use recursion on non-trivial cases
4. Combine the sorted lists
   - Append smaller, pivot, and larger

Such a plan is called an algorithm

Sorting a List of Numbers

Developing the code

- Start with the standard list template
- Fill it in

We know (from the contract) that this is filled with empty

```
;; qsort: list-of-numbers -> list-of-numbers
;; Purpose: return a list containing the input numbers, in ascending order
(define (qsort alon)
   (cond
     [(empty?  alon)          … ]
     [(cons?   alon)
        … (first alon) … (qsort (rest alon)) … ]
   )
)
```

How do we use this stuff to implement our plan

Sorting a List of Numbers

Developing the code

- Filling it in from the English description (*algorithm*)

```
;; qsort: list-of-numbers -> list-of-numbers
;; Purpose: return a list containing the input numbers, in ascending order
(define (qsort alon)
   (cond
     [(empty?  alon)          empty ]
     [(cons?   alon)
      (local [(define pivot (first alon))]
              … (first alon) … (qsort (rest alon)) … ]
   )
)
```

Step 1. Pick a pivot

Step 2. Partition alon around pivot

This task does not fit the template (or the methodology!)

Sorting a List of Numbers

Developing the code

- Implementing Step 2 – Partition alon around pivot

```
;; qsort: list-of-numbers -> list-of-numbers
;; Purpose: return a list containing the input numbers, in ascending order
(define (qsort alon)
   (cond
     [(empty?  alon)          empty ]
     [(cons?   alon)
      (local [(define pivot (first alon))]
              … Start from a clean slate …  ]
   )
)
```

1. *Use helper functions (smaller-items alon) & (larger-items alon)*
2. *Recur on qsort*
3. *Combine results with append*

Sorting a List of Numbers

Developing the code

```
;; qsort: list-of-numbers -> list-of-numbers
;; Purpose: return a list containing the input numbers, in ascending order
(define (qsort alon)
   (cond
     [(empty?  alon)          empty ]
     [(cons?   alon)
      (local [(define pivot (first alon))]
             (append  (qsort (smaller-items alon pivot))
                      (list pivot)
                      (qsort (larger-items alon pivot)))]
   )
)
                ;; smaller-items: list-of-numbers number -> list-of-numbers
                ;; larger-items: list-of-numbers number -> list-of-numbers
```


Sorting a List of Numbers

Developing the code

```
;; qsort: list-of-numbers -> list-of-numbers
;; Purpose: return a list containing the input numbers, in ascending order
(define (qsort alon)
   (cond
     [(empty?  alon)          empty ]    Step 2: Partition alon around pivot
     [(cons?   alon)
      (local [(define pivot (first alon))]
             (append  (qsort (smaller-items alon pivot))
                      (list pivot)
                      (qsort (larger-items alon pivot)))]
   )
)
                ;; smaller-items: list-of-numbers number -> list-of-numbers
                ;; larger-items: list-of-numbers number -> list-of-numbers
```

Sorting a List of Numbers

Developing the code

```
;; qsort: list-of-numbers -> list-of-numbers
;; Purpose: return a list containing the input numbers, in ascending order
(define (qsort alon)
   (cond
     [(empty?  alon)          empty ]
     [(cons?   alon)
      (local [(define pivot (first alon))]
             (append  (qsort (smaller-items alon pivot))
                      (list pivot)
                      (qsort (larger-items alon pivot))))]
   )
)
```

Step 3: Recur on smaller lists

```
                ;; smaller-items: list-of-numbers number -> list-of-numbers
                ;; larger-items: list-of-numbers number -> list-of-numbers
```


Sorting a List of Numbers

Developing the code

```
;; qsort: list-of-numbers -> list-of-numbers
;; Purpose: return a list containing the input numbers, in ascending order
(define (qsort alon)
   (cond
     [(empty?  alon)          empty ]
     [(cons?   alon)
      (local [(define pivot (first alon))]
             (append  (qsort (smaller-items alon pivot))
                      (list pivot)
                      (qsort (larger-items alon pivot))))]
   )
)
```

Step 4: Combine the results

must be a list

```
                ;; smaller-items: list-of-numbers number -> list-of-numbers
                ;; larger-items: list-of-numbers number -> list-of-numbers
```

Sorting a List of Numbers

Developing the code

- What about smaller-items and larger-items?

```
;; smaller-items: list-of-numbers number -> list-of-numbers
(define (smaller-items alon threshold)
   (filter (lambda (n) (< n threshold)) alon))

;; larger-items: list-of-numbers number -> list-of-numbers
(define (larger-items alon threshold)
   (filter (lambda (n) (> n threshold)) alon))
```

- Can hide both of these in the local
  - → Simplify a complex expression

Sorting a List of Numbers

The code

```
;; qsort: list-of-numbers -> list-of-numbers
;; Purpose: return a list containing the input numbers, in ascending order
(define (qsort alon)
   (cond
     [(empty?  alon)          … ]
     [(cons?   alon)
      (local [  (define pivot (first alon))
                (define (smaller-items alon threshold)
                   (filter (lambda (n) (< n threshold)) alon))
                (define (larger-items alon threshold)
                   (filter (lambda (n) (> n threshold)) alon))]
             (append  (qsort (smaller-items alon pivot))
                      (list pivot)
                      (qsort (larger-items alon pivot)) )]
   )
)
```

Sorting a List of Numbers

Quicksort

- Tony Hoare's brilliant insight
- One of fastest sorts known to man

Our version

- Naïve choice of pivot
  - → Always takes first element
  - → Ordered lists generate unbalanced partitions
- Naïve handling of pivot elements
  - → Need to find duplicate elements
  - → Another filter-based helper function

Sorting a List of Numbers

```
;; qsort: list-of-numbers -> list-of-numbers
;; Purpose: return a list containing the input numbers, in ascending order
(define (qsort alon)
   (cond
     [(empty?  alon)          … ]
     [(cons?   alon)
      (local [  (define pivot (first alon))
                (define (smaller-items alon threshold)
                   (filter (lambda (n) (< n threshold)) alon))
                (define (larger-items alon threshold)
                   (filter (lambda (n) (> n threshold)) alon))
                (define (equal-items alon threshold)
                    (filter (lambda (n) (= n threshold)) alon))]
             (append  (qsort (smaller-items alon pivot))
                      (equal-items alon pivot)
                      (qsort (larger-items alon pivot)) )]
   )
)
```