Administrative Announcements

- Homework due today
- Next homework available today, due next Friday
- Challenge lab: tonight at 8:30 in Ryon

- Exam next Wednesday night
  - → Covers lecture through Friday, lab lectures
  - → 7 to 9 pm
  - → Closed notes, closed book
  - → Location TBA
  - → Wednesday night lab folks should attend another lab

Review

Last lecture:

- Did a whole series of examples
  - → keep-lt-x, keep-gt-y, keep-bet-u-and-v
- Used parameterization to share code
- Used local to simplify the code

- Finally, abstracted out the conceptual heart of the code

  filter: (alpha->boolean) list-of-alpha -> list-of-alpha
  - → We call filter an abstract function (*abstracted?*)
  - → We will encounter more abstract functions
  - → We will make heavy use of them (*reuse?*)

Review

Develop keep-fee

```
;; keep-fee:  list-of-symbol -> list-of-symbol
;; Purpose: returns a list containing every occurrence of 'fee' in the list
;; (define (keep-fee alos) …)

(keep-fee (list 'fee 'fie 'foe 'fum 'fee)) -> (list 'fee 'fee)

(keep-fee empty) -> empty
```

```
(define (keep-fee alos)
   (local [(define is-fee? asym) (symbol=? asym 'fee))]
          (filter  is-fee?  alos)
   ))
```


Review

Critical points

- Pass a program as an argument
  - → Description is its contract in parentheses
  - → cons would be (alpha list-of-alpha -> list-of-alpha)

- Scheme functions are just programs*
  - → Can pass cons, <, >, +, symbol=? as arguments
  - → Programs <u>are</u> data

This is not your basic high-school AP programming course

- Concept is called <u>*functional abstraction*</u>

Helper functions

Abstract functions usually require helper functions

- Create many new names
  - → Cognitive overhead of inventing and tracking names
  - → Helper functions are used once, as was is-fee?
- Can hide them inside a local
  - → Works fine
  - → Well-understood rewriting rules
- But, …
  - → A fairly heavy price to pay for creating and using a function
  - → Lots of typing, lots of steps in rewriting rules

Local for helper functions

Using local for this purpose is hard to justify

- Our rules for local

Elliding invariants fits either one

1. Use local to avoid computing some complicated value more than once. This made a huge difference in the cost of max.
2. Use local to make complex expressions more readable by introducing helper functions that break it into tractable parts.

- This case doesn't really fit either criterion
  - → The expression is used once, not twice, or thrice, or …
  - → The expression is not complicated.
  - → is-fee? is about as simple as Scheme gets …
- We used a local just to create a function that we can pass to filter

Helper functions

Need the ability to create <u>anonymous</u> functions

- Want a quick, easy, compact syntax
- Should create full-fledged functions

Enter λ, written <u>lambda</u>

- Lambda is a constructor for anonymous functions

(lambda (arg_1 arg_2 … arg_n) expression)

→ Creates an anonymous function of n arguments

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


Using lambda

We can use an anonymous function in keep-fee

```
;; keep-fee: list-of-symbol -> list-of-symbol
;; Purpose: return a list containing each occurrence of 'fee
(define (keep-fee alos)
   (filter (lambda (asym)(symbol=? asym 'fee)) alos))
```

This is equivalent to our earlier version of keep-fee

```
;; keep-fee: list-of-symbol -> list-of-symbol
;; Purpose: return a list containing each occurrence of 'fee
(define (keep-fee alos)
   (local [(define is-fee? asym) (symbol=? asym 'fee))]
          (filter  is-fee?  alos)
   ))
```

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

Another example

Develop squares

```
;; squares:  list-of-number -> list-of-number
;; Purpose: returns a list containing the squares of the input list
 (define (squares alon)
   (cond
     [(empty? alon)           empty]
     [(cons?   alon) (cons (* (first alon) (first alon))
                           (squares (rest alon)) )]
   ))
```

It would be cleaner to use a helper function, <u>square</u>

Another example

Develop squares

```
;; squares:  list-of-number -> list-of-number
;; Purpose: returns a list containing the squares of the input list
 (define (squares alon)
   (local [(define (square x)(* x x))]
           (cond
              [(empty? alon)  empty]
              [(cons?   alon) (cons (square (first alon)) (squares (rest alon)) )]
            )))
```

We could develop cubes, & quads, & quints, & …

- These need helper functions cube, quad, quint, …
- They fit a pattern: apply function to every element of a list

Another abstract function

Scheme provides the abstract function map

- Takes function & list
- Applies function to list, element-by-element

```
;; squares:  list-of-number -> list-of-number
;; Purpose: returns a list containing the squares of the input list
 (define (squares alon)
   (map (lambda (x)(* x x)) alon))
```