Class Announcements

- Fourth homework: Deadline extension - Now Wednesday, March 29. Parameter passing styles will be discussed in recitation
- First project: Deadline extension - Now Monday, March 27. REMINDER: You have to check for memory leaks and uninitialized variables using valgrind.
- Midterms
  - First midterm. If you want to request a regrade, you must do so by Wednesday, April 29.
  - Second midterm date sometime early April.
  - Final (third midterm) exam on Thursday, May 4, noon - 3:00pm timeslot. Location: most likely this room. Any CONFLICTS with other classes?
- Functional programming resources: Online book on Scheme; we will use racket and drracket which you can install on your laptop or machine. Both interpreters are available on our ilab cluster.
Lists in Scheme

The building blocks for lists are pairs or cons-cells. Lists use the empty list () as an “end-of-list” marker.

Note: (a.b) is not a list!
Special (Primitive) Functions

- **eq?**: identity on names (atoms)
- **null?**: is list empty?
- **car**: selects first element of list *(contents of address part of register)*
- **cdr**: selects rest of list *(contents of decrement part of register)*
- **(cons element list)**: constructs lists by adding **element** to front of **list**
- **quote** or ’**: produces constants
Other Functions

• + − * / numeric operators, e.g.,
  (+ 5 3) = 8, (- 5 3) = 2
  (* 5 3) = 15, (/ 5 3) = 1.6666666

• = < > comparison operators for numbers

• Explicit type determination and test functions:
  ⇒ All return Boolean values: #f and #t
  – (number? 5) evaluates to #t
  – (zero? 0) evaluates to #t
  – (symbol? ’sam) evaluates to #t
  – (list? ’(a b)) evaluates to #t
  – (null? ’()) evaluates to #t

Note: SCHEME is a strongly typed language.
Other Functions

- `(number? 'sam)` evaluates to `#f`
- `(null? '(a))` evaluates to `#f`
- `(zero? (- 3 3))` evaluates to `#t`
- `(zero? '(- 3 3))` ⇒ type error
- `(list? (+ 3 4))` evaluates to `#f`
- `(list? '(+ 3 4))` evaluates to `#t`
READ-EVAL-PRINT Loop

The Scheme interpreters on the ilab machines are called mzscheme, racket, and drracket. “drracket” is an interactive environment, the others are command-line based. For example: Type mzscheme, and you are in the READ-EVAL-PRINT loop. Use Control D to exit the interpreter.

READ: Read input from user:
  a function application

EVAL: Evaluate input:
  (f arg1 arg2 ...argn)
  1. evaluate f to obtain a function
  2. evaluate each argi to obtain a value
  3. apply function to argument values

PRINT: Print resulting value:
  the result of the function application

You can write your Scheme program in file <name>.ss and then read it into the Scheme interpreter by saying at the interpreter prompt: (load "<name>.ss")
READ-EVAL-PRINT Loop Example

> (cons 'a (cons 'b '(c d)))
(a b c d)

1. Read the function application
   (cons 'a (cons 'b '(c d)))

2. Evaluate cons to obtain a function

3. Evaluate 'a to obtain a itself

4. Evaluate (cons 'b '(c d)):
   (a) Evaluate cons to obtain a function
   (b) Evaluate 'b to obtain b itself
   (c) Evaluate '(c d) to obtain (c d) itself
   (d) Apply the cons function to b and (c d) to obtain (b c d)

5. Apply the cons function to a and (b c d) to obtain (a b c d)

6. Print the result of the application:
   (a b c d)
Quotes Inhibit Evaluation

;; Same as before:
> (cons 'a (cons 'b '(c d)))
(a b c d)

;; Now quote the second argument:
> (cons 'a (cons 'b '(c d)))
(a cons (quote b) (quote (c d)))

;; Instead, un-quote the first argument:
> (cons a (cons 'b '(c d)))
ERROR: unbound variable: a
Defining Global Variables

The **define** constructs extends the current interpreter environment by the new defined (name, value) association.

```lisp
> (define foo '(a b c))
#<unspecified>

> (define bar '(d e f))
#<unspecified>

> (append foo bar)
(a b c d e f)

> (cons foo bar)
((a b c) d e f)

> (cons 'foo bar)
(foo d e f)
```
Recursive Scheme Functions: count-elem

(count-elem 'a '(a b a c a) ⇒ 3
(count-elem 'a '(a b (a c) ((a))) ⇒ 1
(count-elem* 'a '(a b (a c) ((a))) ⇒ 3

(define count-elem
  (lambda (x l)

(define count-elem*
  (lambda (x l)

)
Recursive Scheme Functions: Append

(append '(1 2) '(3 4 5) ⇒ (1 2 3 4 5)
(append '(1 2) '(3 (4) 5) ⇒ (1 2 3 (4) 5)
(append () '(1 4 5)) ⇒ (1 4 5)
(append '(1 4 5) '()) ⇒ (1 4 5)
(append '() '()) ⇒ ()

(define append
  (lambda (x y) )
)
Equality Checking

The `eq?` predicate doesn’t work for lists.

Why not?

1. `(cons 'a '())` produces a new list
2. `(cons 'a '())` produces another new list
3. `eq?` checks if its two arguments are *the same*
4. `(eq? (cons 'a '()) (cons 'a '()))` evaluates to `#f`

Lists are stored as pointers to the first element (car) and the rest of the list (cdr). This elementary “data structure”, the building block of lists, is called a **pair**.

![pair](image)

Symbols are stored uniquely, so `eq?` works on them.
Equality Checking for Lists

For lists, need a comparison function to check for the same structure in two lists

```
(define equal?
  (lambda (x y)
    (or (and (atom? x) (atom? y) (eq? x y))
        (and (not (atom? x)) (not (atom? y))
            (equal? (car x) (car y))
            (equal? (cdr x) (cdr y))))))
```

- `(equal? 'a 'a)` evaluates to `#t`
- `(equal? 'a 'b)` evaluates to `#f`
- `(equal? '(a) '(a))` evaluates to `#t`
- `(equal? '((a)) '(a))` evaluates to `#f`
Practice programming in Scheme

Next time:

- Lambda calculus