Class Information

- Homework 6 deadline extended to this coming Wednesday 11/9.
The building blocks for lists are pairs or cons-cells. Lists that use the empty list ( ) as an “end-of-list” marker are called proper lists, otherwise improper lists.

Note: (a.b) is not a proper list!
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:

\[(f \ arg_1 \ arg_2 \ldots \ arg_n)\]

1. evaluate \(f\) to obtain a function
2. evaluate each \(arg_i\) 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")`
Review - Conditional Execution: if

(if <condition> <result1> <result2>)

1. Evaluate <condition>
2. If the result is a “true value” (i.e., anything but #f), then evaluate and return <result1>
3. Otherwise, evaluate and return <result2>

(define abs-val
  (lambda (x)
    (if (>= x 0) x (- x))))

(define rest-if-first
  (lambda (e l)
    (if (eq? e (car l)) (cdr l) '())))
(cond (<condition1> <result1>)
  (<condition2> <result2>)
  ...
  (<conditionN> <resultN>)
  (else <else-result>)); optional else clause

1. Evaluate conditions in order until obtaining one that returns a true value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return <else-result>
(define abs-val
    (lambda (x)
        (cond ((>= x 0) x)
                (else (- x)))))

(define rest-if-first
    (lambda (e l)
        (cond ((null? l) '())
              ((eq? e (car l)) (cdr l))
              (else '()))))
Recursive Scheme Functions: Abs-List

▶ (abs-list '(1 -2 -3 4 0)) ⇒ (1 2 3 4 0)
▶ (abs-list '()) ⇒ ()

(define abs-list
  (lambda (l)
    (if (null? l)
      '()
      (cons (abs-val (car l)) (abs-list (cdr 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)
    (cond ((null? x) y)
           ((null? y) x)
           (else (cons (car x)
                        (append (cdr 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.

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

\[
\text{(define equal?}
  \begin{align*}
    &\quad \text{(lambda} (x \ y) \\
    &\quad \quad \text{(or} (\text{and} (\text{atom?} \ x) (\text{atom?} \ y) (\text{eq?} \ x \ y)) \\
    &\quad \quad \text{(and} (\text{not} (\text{atom?} \ x)) (\text{not} (\text{atom?} \ y)) \\
    &\quad \quad \quad \text{(equal?} (\text{car} \ x) (\text{car} \ y)) \\
    &\quad \quad \quad \text{(equal?} (\text{cdr} \ x) (\text{cdr} \ y))))))
  \end{align*}
\]

- (equal? ’a ’a) evaluates to #t
- (equal? ’a ’b) evaluates to #f
- (equal? ’(a) ’(a)) evaluates to #t
- (equal? ’((a)) ’(a)) evaluates to #f
Functions as arguments:
(define f (lambda (g x) (g x)))

▶ (f number? 0)
⇒ (number? 0) ⇒ #t

▶ (f length '(1 2))
⇒ (length '(1 2)) ⇒ 2

▶ (f (lambda (x) (* 2 x)) 3)
⇒ ((lambda (x) (* 2 x)) 3)
⇒ (* 2 3) ⇒ 6

**REMINDER:** Computation, i.e., function application is performed by reducing the initial S-expression (program) to an S-expression that represents a value. Reduction is performed by substitution, i.e., replacing formal by actual arguments in the function body. Examples for S-expressions that directly represent values, i.e., cannot be further reduced:

▶ function values (e.g.: (lambda(x) e))
▶ constants (e.g.: 3, #t)
Higher-order Functions (Cont.)

Functions as returned values:
(define plusn
  (lambda (n) (lambda (x) (+ n x))))

▶ (plusn 5) evaluates to a function that adds 5 to its argument

Question: How would you write down the value of (plusn 5)?

▶ ((plusn 5) 6) ⇒ 11
Higher-order Functions (Cont.)

In general, any n-ary function

\[(\text{lambda} \ (x_1 \ x_2 \ \ldots \ x_n) \ e)\]

can be rewritten as a nest of \(n\) unary functions:

\[(\text{lambda} \ (x_1) \ \\
(\text{lambda} \ (x_2) \ \\
(\ldots \ (\text{lambda} \ (x_n) \ e) \ \ldots \ )))\]

This translation process is called **currying**. It means that having functions with multiple parameters do not add anything to the expressiveness of the language.

\[((\text{lambda} \ (x_1 \ x_2 \ \ldots \ x_n) \ e) \ v_1 \ v_2 \ \ldots \ v_n)\]

\[((\ldots \ \\
((\text{lambda} \ (x_1) \ \\
(\text{lambda} \ (x_2) \ \\
\ldots \ \\
(\text{lambda} \ (x_n) \ e) \ldots )) \ v_1) \ v_2) \ \ldots \ v_n)\]
Higher-order Functions: map

(define map
  (lambda (f l)
    (if (null? l)
        '()
        (cons (f (car l)) (map f (cdr l))))))

- map takes two arguments: a function and a list
- map builds a new list by applying the function to every element of the (old) list
Higher-order Functions: map

- Example:

  \[
  (\text{map abs } '(-1 2 -3 4)) \Rightarrow \\
  (1 2 3 4)
  \]

  \[
  (\text{map (lambda (x) (+ 1 x)) '(-1 2 -3)) } \Rightarrow \\
  (0 3 -2)
  \]

- Actually, the built-in map can take more than two arguments:

  \[
  (\text{map + } '(1 2 3) ' (4 5 6)) \Rightarrow \\
  (5 7 9)
  \]
More on Higher Order Functions

reduce

Higher order function that takes a binary, associative operation and uses it to “roll-up” a list

\[
\text{(define reduce}
\begin{align*}
\quad & \text{(lambda (op l id)} \\
\quad & \quad \text{(if (null? l)} \\
\quad & \quad \quad \text{id} \\
\quad & \quad \quad \text{(op (car l) (reduce op (cdr l) id))})
\end{align*}
\]

Example:

\[
\begin{align*}
\text{(reduce + '(10 20 30) 0)} & \Rightarrow \\
\text{(reduce + '(20 30) 0)} & \Rightarrow \\
\text{(reduce + '(30) 0)} & \Rightarrow \\
\text{(reduce + '() 0)} & \Rightarrow \\
\text{60}
\end{align*}
\]
More on Higher Order Functions

Now we can compose higher order functions to form compact powerful functions

Examples:

```
(define sum
  (lambda (f l)
    (reduce + (map f l) 0))

(sum (lambda (x) (* 2 x)) '(1 2 3)) ⇒
(reduce (lambda (x y) (+ 1 y)) '(a b c) 0) ⇒
```

Lexical Scoping and let, let*, and letrec

All are variable binding operations:

\[ \text{LET} = \text{let, let*}, \text{letrec} \]

\[
\begin{align*}
\text{(LET ((v1 e1)} \\
\quad (v2 e2)} \\
\quad \ldots \\
\quad (vn en)) \\
\end{align*}
\]
\[ e \]

- **let**: binds variables to values (no specific order), and evaluates body \( e \) using the bindings; new bindings are not effective during evaluation of any \( e_i \).
- **let***: binds variables to values in textual order of write-up (left to right, or here: top down); new binding is effective for next \( e_i \) (nested scopes).
- **letrec**: bindings of variables to values in no specific order; independent **evaluations of all** \( e_i \) **to values** have to be possible; new bindings effective for all \( e_i \); mainly used for recursive function definitions.
Next Lecture

Reading: Scott Chapter 11.1 to 11.3

More Scheme and higher-order functions