Class Information

INFORMATION and REMINDERS

- Homework 6 will be posted by tomorrow.
- Project 2 will be posted next week.
- Midterm has been mostly graded.
  Will be returned in recitation.
  Please look at sample solutions before asking questions about grading. Thanks!
Recursive Scheme Functions: Abs-List

(define abs-val
  (lambda (x)
    (cond ((>= x 0) x)
          (else (- x)))))

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

(define abs-list
  (lambda (l)
    )

Recursive Scheme Functions: Append

(define append
  (lambda (x y) ...) )
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**.

![Diagram of 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

(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
Scheme: Functions as Values (Higher-order)

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:

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

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

\textit{Question}: How to write an application of the original vs. the curried version?

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

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

```
(define reduce
  (lambda (op l id)
    (if (null? l)
        id
        (op (car l) (reduce op (cdr l) id))))
```

Example:

```
(reduce + '(10 20 30) 0) ⇒
(+ 10 (reduce + ’(20 30) 0)) ⇒
(+ 10 (+ 20 (reduce + ’(30) 0))) ⇒
(+ 10 (+ 20 (+ 30 (reduce + ’() 0)))) ⇒
(+ 10 (+ 20 (+ 30 0))) ⇒
60
```
More on Higher Order Functions

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

Examples:

\[
(\text{define } \text{sum} \\
(\lambda (f \ l) \\
(\text{reduce } + (\text{map } f \ l) \ 0) ))
\]

\[
(\text{sum } (\lambda (x) (* 2 x)) \ '(1 2 3)) \Rightarrow
\]

\[
(\text{reduce } (\lambda (x \ y) (+ 1 y)) \ '(a \ b \ c) \ 0) \Rightarrow
\]
Lexical Scoping and let, let*, and letrec

All are variable binding operations:

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

\[
\text{(LET } ((v1 \ e1) \\
(v2 \ e2) \\
\ldots \\
(vn \ en))) \\
e\text{)}
\]

- **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

Things to do:

• Practice programming in Scheme
• Work on Homework 6 programming examples.

Next time:

• Lambda calculus