Class Information

REMINDERS

• Homework 6 due on Tuesday, November 17.

• Wednesday, November 25, is officially a Friday, so there will be lecture and recitation.
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`

Remember: 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

\[(\text{define equal?})\]
\[\text{(lambda } (x \ y)\]
\[\text{(or } (\text{and } (\text{atom? } x) (\text{atom? } y) (\text{eq? } x \ y))\]
\[\text{(and } (\text{not } (\text{atom? } x)) (\text{not } (\text{atom? } y))\]
\[\text{(equal? } (\text{car } x) (\text{car } y))\]
\[\text{(equal? } (\text{cdr } x) (\text{cdr } y))))))\]

• (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 textual substitution (re-write system), 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)
Functions as returned values:

(\(\text{define plusn}
\)  
\((\text{lambda} (n) (\text{lambda} (x) (+ n x))))\)

- \((\text{plusn} 5)\) evaluates to a function that adds 5 to its argument

\text{Question:} How would you write down the value of \((\text{plusn} 5)\)?

- \([(\text{plusn} 5) 6] \Rightarrow 11\)
Higher-order Functions (Cont.)

In general, any n-ary function

$$(\lambda (x_1 \ x_2 \ldots \ x_n) \ e)$$

can be rewritten as a nest of $n$ unary functions:

$$\begin{align*}
(\lambda (x_1) \\
(\lambda (x_2) \\
(\ldots (\lambda (x_n) \ e ) \ldots )))
\end{align*}$$

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

*Question*: How to write an application of the original vs. the curried version?

$$((\lambda (x_1 \ x_2 \ldots \ x_n) \ e) \ v_1 \ v_2 \ldots \ v_n)$$

$$((\ldots ((\lambda (x_1) \\
(\lambda (x_2) \\
(\ldots \\
(\lambda (x_n) \ e )\ldots )))) \ v_1) \ v_2) \ldots v_n)$$
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:
  
  \[
  \text{(map abs '(-1 2 -3 4)) ⇒ (1 2 3 4)} \\
  \text{(map (lambda (x) (+ 1 x)) '(-1 2 -3)) ⇒ (0 3 -2)}
  \]

• Actually, the built-in map can take more than two arguments:
  
  \[
  \text{(map + '}(1 2 3) '}(4 5 6)) ⇒ (5 7 9)
  \]
More on Higher Order Functions

reduce

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

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

Example:

\[
\text{(reduce + ’(10 20 30) 0) ⇒} \\
(+ 10 (reduce + ’(20 30) 0)) ⇒ \\
(+ 10 (+ 20 (reduce + ’(30) 0))) ⇒ \\
(+ 10 (+ 20 (+ 30 (reduce + ’() 0)))) ⇒ \\
(+ 10 (+ 20 (+ 30 0))) ⇒ \\
60
\]

CS 314

lecture 20, fall’15, page 10
More on Higher Order Functions

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

Examples:

\[
\text{(define sum } \ \\
\text{ (lambda (f l} \ \\
\text{ \ \ \ \ (reduce + (map f l) 0)) ) } \\
\text{(sum (lambda (x) (* 2 x)) ’(1 2 3) ) } \Rightarrow \\
\text{(reduce (lambda (x y) (+ 1 y)) ’(a b c) 0) } \Rightarrow
\]
Lexical Scoping and \texttt{let}, \texttt{let*}, and \texttt{letrec}

All are variable binding operations:

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

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

- \texttt{let}: binds variables to values (no specific order), and evaluates body \texttt{e} using the bindings; new bindings are not effective during evaluation of any \(e_i\).

- \texttt{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).

- \texttt{letrec}: bindings of variables to values in no specific order; independent \textbf{evaluations of all} \(e_i\) \textbf{to values} have to be possible; new bindings effective for all \(e_i\); mainly used for recursive function definitions.
Scheme Project

A spell checker generator in Scheme.
Next Lecture

Things to do:

- Homework problem set 6 has been posted.
- Project 2 (Scheme) will be posted soon; start programming in Scheme!

Next time:

- lambda calculus