Class Information

- Midterm exam sample solutions are available.
- Second project will be posted by tomorrow.

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

```
(lambda (x_1 x_2 ... x_n) e)
```

can be rewritten as a nest of n unary functions:

```
(lambda (x_1)
   (lambda (x_2)
     ( ... (lambda (x_n)  e ) ... )))
```

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 ... x_n) e) v_1 v_2 ... v_n)


(( ...
  ((lambda (x_1)
    (lambda (x_2)
      ...
       (lambda (x_n) e )...)) v_1) v_2) ... 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:

  `(map abs '(-1 2 -3 4))` $\Rightarrow$
  `(1 2 3 4)`

  `(map (lambda (x) (+ 1 x)) '(-1 2 -3))` $\Rightarrow$
  `(0 3 -2)`

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

  `(map + '(1 2 3) '(4 5 6))` $\Rightarrow$
  `(5 7 9)`

More on Higher Order Functions

$\boxed{\text{reduce}}$

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) $\Rightarrow$
(+ 10 (reduce + '(20 30) 0)) $\Rightarrow$
(+ 10 (+ 20 (reduce + '(30) 0))) $\Rightarrow$
(+ 10 (+ 20 (+ 30 (reduce + '() 0)))) $\Rightarrow$
(+ 10 (+ 20 (+ 30 0))) $\Rightarrow$
60

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)) $\Rightarrow$

(reduce (lambda (x y) (+ 1 y)) '(a b c) 0) $\Rightarrow$

Lexical Scoping and let, let∗, and letrec

All are variable binding operations:

LET = let, let∗,letrec

```
(LET ((v1 e1)
      (v2 e2)
        ...
      (vn en))
   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.

let and let∗ examples

```
(let ((a 5)
      (b 6))
   (+ a b))     ;; ==> 11

(let ((a 5)
      (b (+ a 6)))
   (+ a b))  ;; ==> ERROR: unbound variable: a


(let* ((a 5)
       (b (+ a 6)))
   (+ a b))  ;; ==> 16
```

Note: **let** and **let*** do not add anything to the expressiveness of the language, i.e., they are only a convenient shorthand. For instance,

`(let ((x v1) (y v2)) e)` can be rewritten as
`((lambda (x y) e) v1 v2)`

letrec examples

Typically used for local definitions of recursive functions

```
(letrec ((a 5)
         (b (+ a 6)))
   (+ a b)) ;; ==> ERROR: unbound variable: a

(letrec ((a 5)
         (b (lambda ()(+ a 6))))
   (+ a (b)))  ;; ==> 16

(letrec ((b (lambda ()(+ a 6)))
         (a 5))
   (+ a (b)))  ;; ==> 16

(letrec ((even? (lambda (x)
                  (or (= x 0)
                      (odd? (- x 1)))))
         (odd?  (lambda (x)
                  (and (not (= x 0))
                      (even? (- x 1))))))
  (list (even? 3) (even? 20) (odd? 21)))
            ;;  ==> (#f #t #t)
```

Second Project (Scheme)

Implement a function that takes as input a sequence of words encoded via an unknown Caesar's Cipher, and returns a function that decodes words in that cipher back into plain text. You then use this function to write a code-breaker that decodes an entire document back into its plain text.

$\boxed{\text{Caesar's Cipher}}$: Each letter "shifted" by a fixed amount.

There are 26 possible ciphers in the English language (lower case letters only)

Two basic approaches to break Caesar's ciphers

- Brute Force
- Letter frequency analysis

Lot's of `map` and some `reduce` applications.

Next Lecture

Things to do:

- Project 2 (Scheme) will be posted this Saturday; start programming in Scheme!

Next time:

- foundations of lambda calculus
- programming in lambda calculus