Class Announcements

- Fourth homework and first project: Deadline extension - Wednesday, March 29.
- First project: More ILOC test cases have been posted (test0.out, ... test14.out)

  REMINDER: Please do not copy or share code. We will use plagiarism tools to detect software similarities.
- Fifth homework has been posted.
- Midterms
  - First midterm. If you want to request a regrade, you must do so by Wednesday, April 29.
  - Second midterm will be on April 11.
  - Final (third midterm) exam on Thursday, May 4, noon - 3:00pm timeslot. Location: most likely this room.

    Any CONFLICTS with other classes?

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. This is a rewrite system.

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)` $\Rightarrow$ `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))` ⇒
  `(1 2 3 4)`

  `(map (lambda (x) (+ 1 x)) '(-1 2 -3))` ⇒
  `(0 3 -2)`

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

  `(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

```
(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:

```
(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:

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.

Next Lecture

Things to do:

- Finish project and homeworks #4 and #5
- Practice Scheme/Racket programming