# SICP Study Group Episode VII

March 17, 2014

# How This Works

- No google, stackoverflow, wikipedia, et cetera. Referencing the book is allowed.
- Pair programming - find a partner!
- We'll have three programming exercises, and you and your partner have 20-30min for each one.
- When time is up, one pair will be volunteered to share their solution.
- We'll change partners after each exercise.

# What we've done thus far:

- Used basic, pure, higher-order functions to implement numerical analysis methods.
- Used linked list (cons, car, cdr) to implement "classic" functional programming interfaces.
- Combined the two to create representations for symbolic differentiation, sets, and encoding trees.

# What you should know this week

- How to “tag” data to create a mock-type system using only functions and linked lists.
- How to generically dispatch functions based on this type system.
- How to implement polar and rectangular representations of complex numbers using these methods.
- Message passing - data types deal with their own operations.

# Background

The last part of 2.4 introduced to smalltalk-style object oriented programming, or **message passing**. We saw that we could implement a complex number in rectangular form by a procedure that returns a procedure that can dispatch messages asking it to convert to a different type:

```
(define (make-from-real-imag x y)
 (define (dispatch op)
  (cond ((eq? op 'real-part) x)
        ((eq? op 'imag-part) y)
        ((eq? op 'magnitude)
         (sqrt (+ (square x) (square y))))
        ((eq? op 'angle) (atan y x))
        (else
         (error "Unknown op -- MAKE-FROM-REAL-IMAG"
    op))))
 dispatch)
```

# Background Cont

The corresponding apply-generic procedure, which applies a generic operation to an argument, now simply feeds the operation's name to the data object and lets the object do the work:

```
(define (apply-generic op arg) (arg op))
```

# Background Sec 2.4.1

Last week, in section 2.3.3, we learned how to use data abstractions to handle multiple representations of data. This week, we're going to learn how to use similar techniques to perform operations and all those data structures.

Similar to last week's *install-rectangular-package* and *install-polar-package*, we have packages for various types of numbers: ordinary integers, rational numbers, and complex numbers, each with their own respective package. The source code for this is in the book and in this week's git.

The *install-complex-package* actually requires last week's two packages to implement its procedures.

# Background Problem 2b

The internal procedures in the *scheme-number* package are essentially nothing more than calls to the primitive procedures +, -, etc. It was not possible to use the primitives of the language directly because our type-tag system requires that each data object have a type attached to it. In fact, however, all Lisp implementations do have a type system, which they use internally. Primitive predicates such as *symbol?* and *number?* determine whether data objects have particular types.