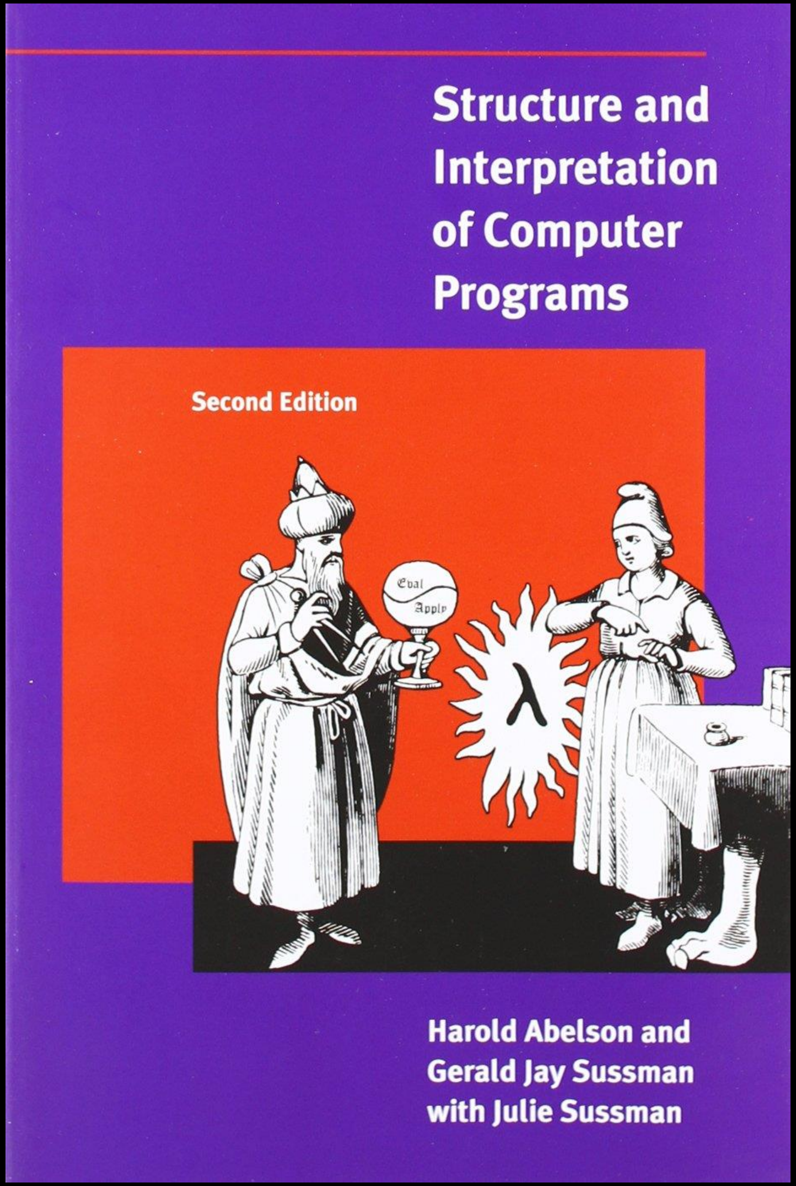
Java

Meetup



Structure and Interpretation of Computer Programs

Chaper 2.5

Before we start …

Friendly Environment Policy

Berlin Code of Conduct

DISCORD

Programming Languages Virtual Meetup

1 Tweet

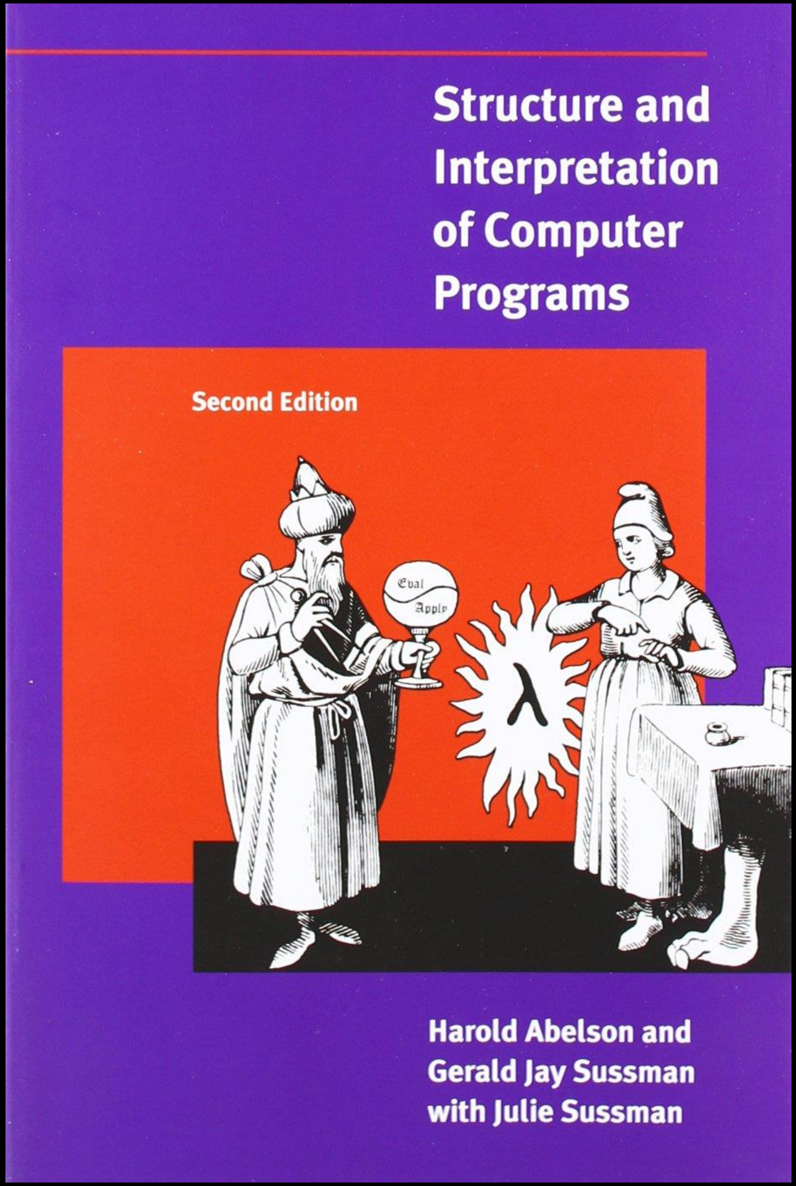
Following

Programming Languages Virtual Meetup

@PLvirtualmeetup

Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

Toronto, CA meetup.com/Programming-La... Joined March 2020



Structure and Interpretation of Computer Programs

Chaper 2.5

In the previous section, we saw how to design systems in which data objects can be represented in more than one way. The key idea is to link the code that specifies the data operations to the several representations by means of generic interface procedures. Now we will see how to use this same idea not only to define operations that are generic over different representations but also to define operations that are generic over different kinds of arguments.

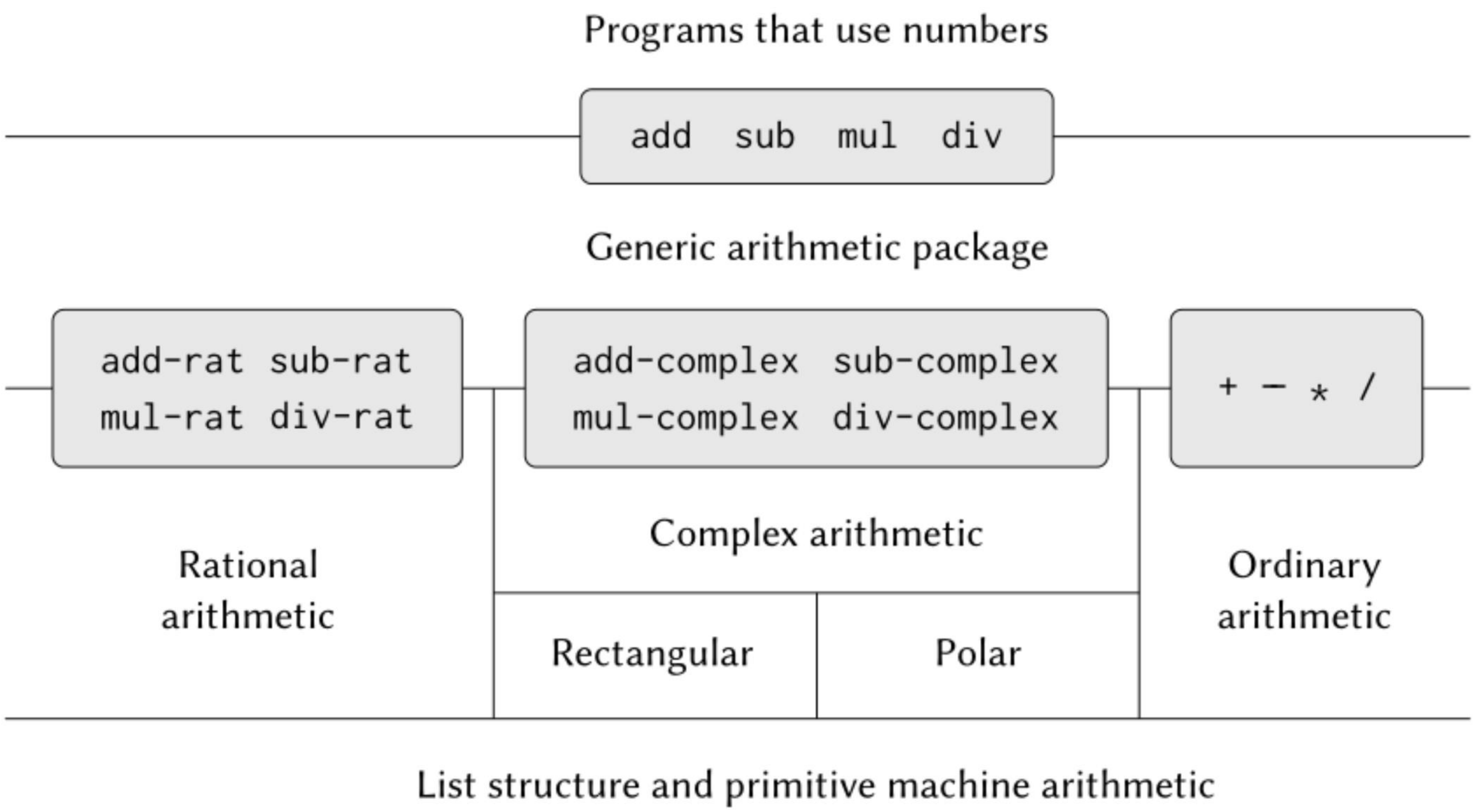
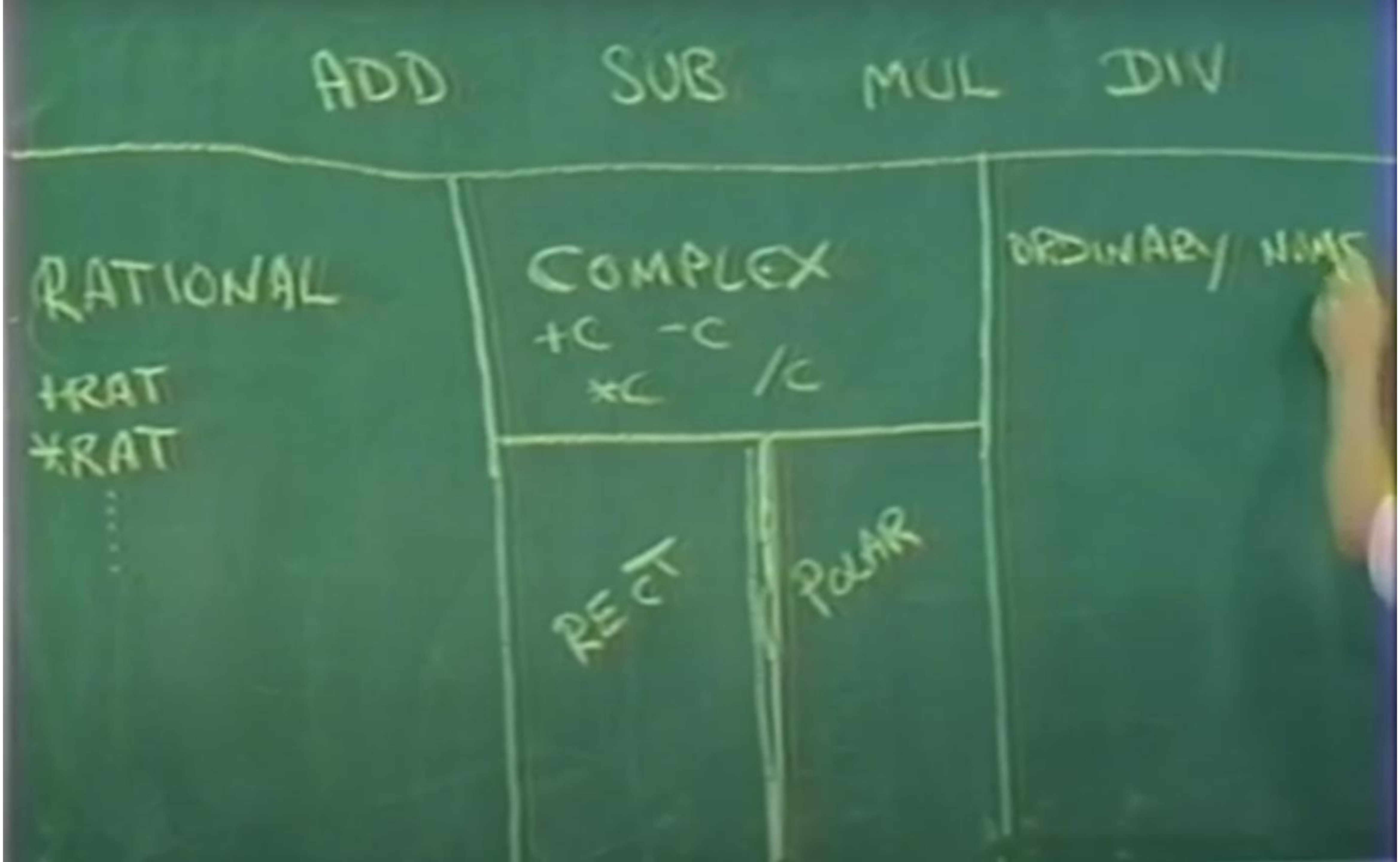


Figure 2.23: Generic arithmetic system.


ADD
SUB
MUL
DIV
RATIONAL
+RAT
*RAT
COMPLEX
+C -C
*C /C
RECT
POLAR
ORDINARY/

```
(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div x y))
```

```
;; from previous chapter

(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error "No method for these types: APPLY-GENERIC"
                 (list op type-tags))))))
```

```racket
(define (install-ordinary-package)
  (define (tag x) (attach-tag 'number x))
  (put 'add '(number number)
       (λ (x y) (tag (+ x y))))
  (put 'sub '(number number)
       (λ (x y) (tag (- x y))))
  (put 'mul '(number number)
       (λ (x y) (tag (* x y))))
  (put 'div '(number number)
       (λ (x y) (tag (/ x y))))
  (put 'make 'number (λ (x) (tag x)))
  'done)
```

```
(define (install-rational-package)
  ;; internal procedures
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d)
    (let ((g (gcd n d)))
      (cons (/ n g) (/ d g))))
  (define (add-rat x y)
```

...

```
(define (install-complex-package)
  ;; imported procedures from rectangular and polar packages
  (define (make-from-real-imag x y)
    ((get 'make-from-real-imag 'rectangular) x y))
  (define (make-from-mag-ang r a)
    ((get 'make-from-mag-ang 'polar) r a))
  ;; internal procedures
  (define (add-complex z1 z2)
```

…

```racket
(put 'add '(rational rational)
     (λ (x y) (tag (add-rat x y))))
(put 'sub '(rational rational)
     (λ (x y) (tag (sub-rat x y))))
(put 'mul '(rational rational)
     (λ (x y) (tag (mul-rat x y))))
(put 'div '(rational rational)
     (λ (x y) (tag (div-rat x y))))
(put 'make 'rational
     (λ (n d) (tag (make-rat n d))))
```

```racket
(put 'add '(complex complex)
     (λ (z1 z2) (tag (add-complex z1 z2))))
(put 'sub '(complex complex)
     (λ (z1 z2) (tag (sub-complex z1 z2))))
(put 'mul '(complex complex)
     (λ (z1 z2) (tag (mul-complex z1 z2))))
(put 'div '(complex complex)
     (λ (z1 z2) (tag (div-complex z1 z2))))
(put 'make-from-real-imag 'complex
     (λ (x y) (tag (make-from-real-imag x y))))
(put 'make-from-mag-ang 'complex
     (λ (r a) (tag (make-from-mag-ang r a))))
```

Exercise 2.78: 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. Modify the definitions of `type-tag`, `contents`, and `attach-tag` from Section 2.4.2 so that our generic system takes advantage of Scheme's internal type system. That is to say, the system should work as before except that ordinary numbers should be represented simply as Scheme numbers rather than as pairs whose `car` is the symbol `scheme-number`.

```scheme
;; Exercise 2.78 (page 261)

(define (attach-tag type-tag contents)
  (if (number? contents)
      contents
      (cons type-tag contents)))

(define (type-tag datum)
  (cond ((number? datum) 'number)
        ((pair? datum) (car datum))
        (else (error "Wrong datum -- TYPE-TAG" datum))))

(define (contents datum)
  (cond ((number? datum) datum)
        ((pair? datum) (cdr datum))
        (else (error "Wrong datum -- CONTENTS" datum))))
```

Exercise 2.79: Define a generic equality predicate `equ?` that tests the equality of two numbers, and install it in the generic arithmetic package. This operation should work for ordinary numbers, rational numbers, and complex numbers.

```racket
;; Exercise 2.79 (261)

(put 'equ? '(number number) =)                             ; put in number package
(put 'equ? '(rational rational) (λ (x y)                   ; put in rational package
                                  (= (* (numer x) (denom y))
                                     (* (numer y) (denom x)))))
(put 'equ? '(complex complex) (λ (x y)                     ; put in complex package
                                (and (= (real-part x) (real-part y))
                                     (= (imag-part x) (imag-part y)))))

(define (equ? x y) (apply-generic 'equ? x y))

(check-equal? (equ? 1 1) #t)
(check-equal? (equ? 1 2) #f)
(check-equal? (equ? (make-rational 1 2) (make-rational 2 4)) #t)
(check-equal? (equ? (make-rational 1 2) (make-rational 1 3)) #f)
(check-equal? (equ? (make-complex-from-real-imag 1 2)
                    (make-complex-from-real-imag 1 2)) #t)
```

2.5.2 Combining Data of Different Types

Coercion

```
(define (scheme-number->complex n)
  (make-complex-from-real-imag (contents n) 0))
```

We install these coercion procedures in a special coercion table, indexed under the names of the two types:

```
(put-coercion 'scheme-number
              'complex
              scheme-number->complex)
```

Hierarchies of types

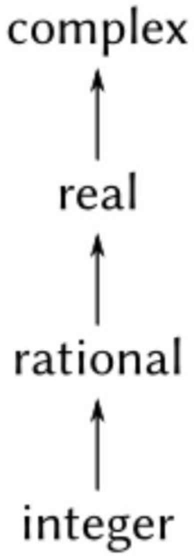


Figure 2.25: A tower of types.

Exercise 2.83: Suppose you are designing a generic arithmetic system for dealing with the tower of types shown in Figure 2.25: integer, rational, real, complex. For each type (except complex), design a procedure that raises objects of that type one level in the tower. Show how to install a generic `raise` operation that will work for each type (except complex).

```
;; Exercise 2.83 (page 272)

(put 'raise 'integer  (λ (x) (make-rational x 1)))
(put 'raise 'rational (λ (x) (make-real (/ (numer x) (denom x)))))
(put 'raise 'real     (λ (x) (make-from-real-imag x 0)))

(define (raise x) (apply-generic 'raise x))
```

Exercise 2.84: Using the `raise` operation of Exercise 2.83, modify the `apply-generic` procedure so that it coerces its arguments to have the same type by the method of successive raising, as discussed in this section. You will need to devise a way to test which of two types is higher in the tower. Do this in a manner that is "compatible" with the rest of the system and will not lead to problems in adding new levels to the tower.

```scheme
;; Exercise 2.84 (page 272)

(define (do-raise a b)
  (let ((a-type (type-tag a))
        (b-type (type-tag b)))
    (cond ((equal? a-type b-type) a)
          ((get 'raise a-type)
           (do-raise ((get 'raise a-type) (contents a)) b))
          (else #f))))
```

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((a (car args))
                    (b (cadr args)))
                (cond ((do-raise a b) (apply-generic op (do-raise a b) b))
                      ((do-raise b a) (apply-generic op a (do-raise b a)))
                      (else (error "Not supported" (list op type-tags)))))
              (error "Not supported" (list op type-tags)))))))

(check-equal? (do-raise 2 3) 2)
(check-equal? (do-raise 2 (make-rational 3 1)) (make-rational 2 1))
(check-equal? (add 2 (make-rational 3 1)) (make-rational 5 1))
(check-equal? (mul 2 (make-rational 3 1)) (make-rational 6 1))
```

2.5.3 Example: Symbolic Algebra

$A: \quad x^5 + 2x^4 + 3x^2 - 2x - 5$ `(1 2 0 3 -2 -5)`

$B: \quad x^{100} + 2x^2 + 1$ `((100 1) (2 2) (0 1))`

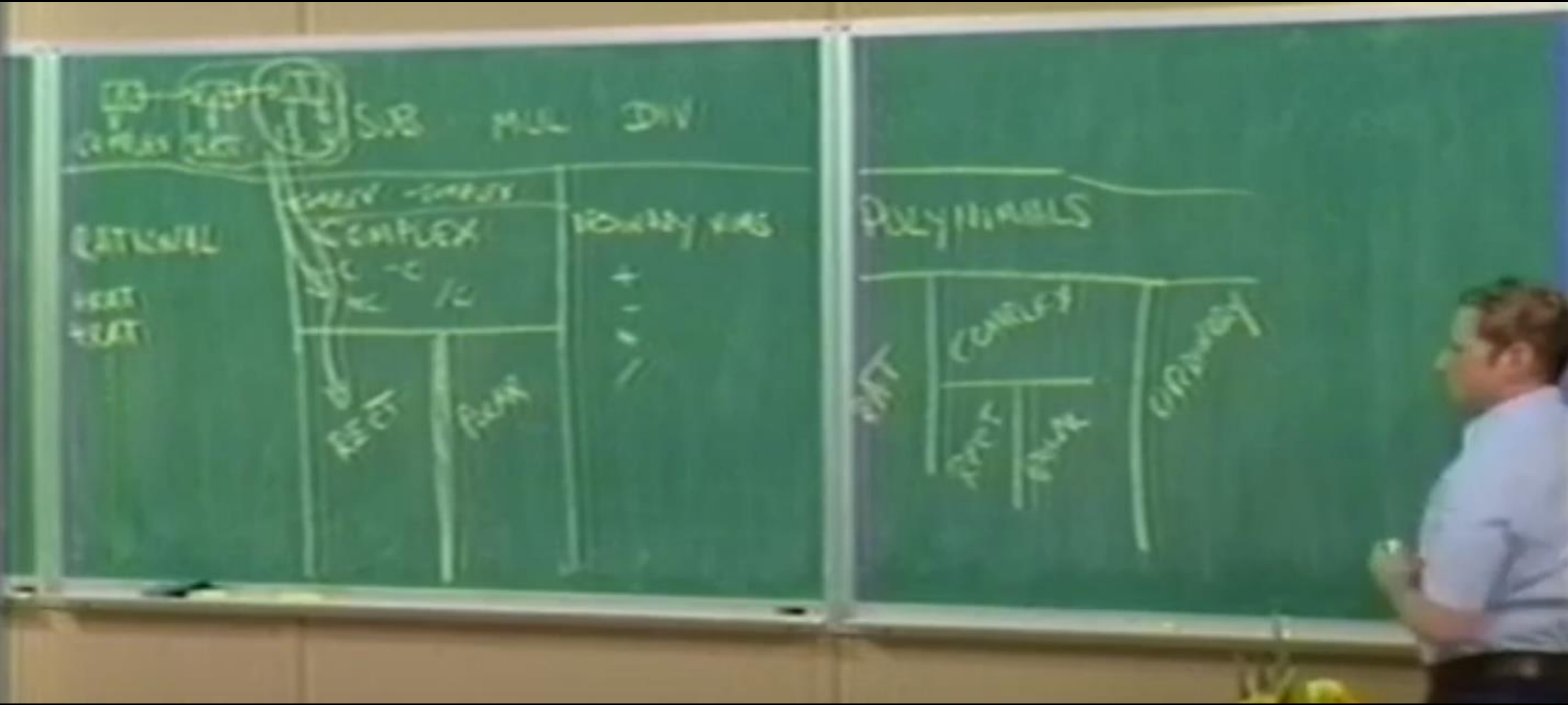
```
(define (add-terms L1 L2)
  (cond ((empty-termlist? L1) L2)
        ((empty-termlist? L2) L1)
        (else
         (let ((t1 (first-term L1))
               (t2 (first-term L2)))
           (cond ((> (order t1) (order t2))
                  (adjoin-term
                   t1 (add-terms (rest-terms L1) L2)))
                 ((< (order t1) (order t2))
```

```
(define (mul-terms L1 L2)
  (if (empty-termlist? L1)
      (the-empty-termlist)
      (add-terms (mul-term-by-all-terms (first-term L1) L2)
                 (mul-terms (rest-terms L1) L2))))
```

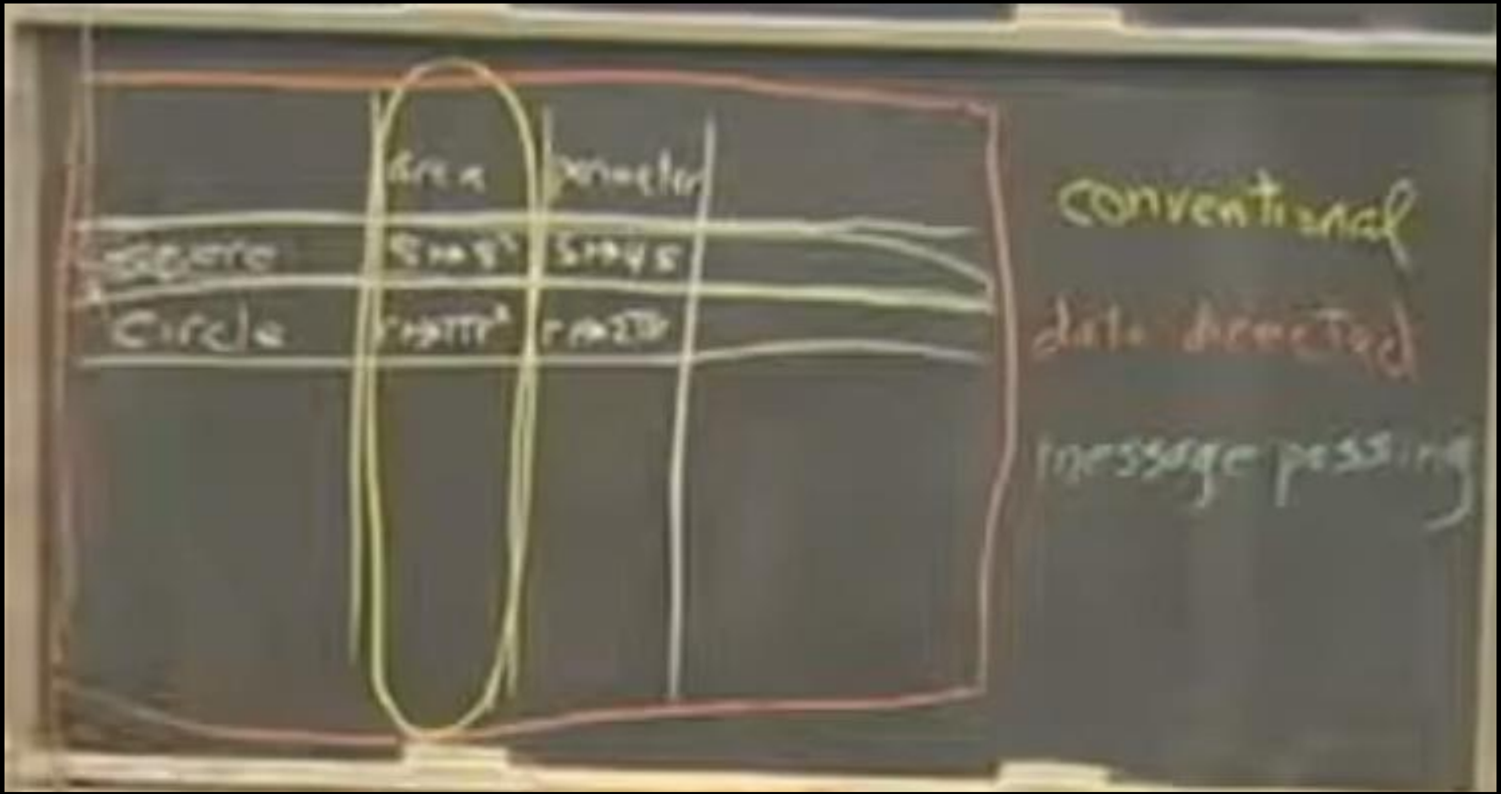
Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman


SUB
MUL
DIV
RATIONAL
COMPLEX
RECT
POLAR
RAT
RECT
POLAR

	perimeter	area
square	$s \mapsto 4s$	$s \mapsto s^2$
circle	$r \mapsto 2\pi r$	$r \mapsto \pi r^2$


area
perimeter
circle
conventional
data directed
message passing

# of operands	regular people	groovy people
1	unary	monadic
2	binary	dyadic

"**monadic** and **dyadic** … these names were proposed by Ken Iverson who invented the important but obscure programming language **APL**"

Brian Harvey

"… they are great names and they don't have any other confusing meanings"

Brian Harvey

L17 Generic Operators | UC Berkeley CS 61A

Java

Meetup