Strategies for Managing Data Complexity

- Separate specification from implementation
- Procedural interface to data (enables alternative representations)
- Manifest typing (enables multiple representations)
- Generic operations
  - Dispatch on type
  - Table-driven generic interface (Data-Directed Programming)

- Packages
- Closure
- Coercion
- Message-Passing

Rectangular Package

(define (install-rectangular-package)
  ;; Internal rep: RepRect = Sch-Num X Sch-Num
  ;; internal procedures on RepRect...
  (define (make-from-real-imag x y) (cons x y))
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (magnitude z) (+ (* (square (real-part z)) (square (imag-part z))))))
  (define (angle z) (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a) (cons (* r (cos a)) (* r (sin a))))

  ;; interface to the rest of the system
  ;; External rep: Rectangular = 'rectangular X RepRect
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle ' (rectangular) angle)
  ; Constructors: Sch-Num, Sch-Num -> Rectangular
  (put 'make-from-real-imag 'rectangular
    (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
    (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)

  ;; External constructor: Sch-Num, Sch-Num -> Rectangular
  (define (make-from-real-imag x y)
    ((get 'make-from-real-imag 'rectangular) x y))
Operation/Type Table

Trace the evaluation of \((\text{define } \textit{my-z} (\text{make-from-real-imag} \ 3 \ 4))\)

Data-Directed Generic Operations

\begin{verbatim}
(define (real-part z) (apply-generic 'real-part z))
(define (magnitude z) (apply-generic 'magnitude z))

(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 types - APPLY GENERIC" (list op type-tags))))))
\end{verbatim}
Trace the evaluation of \((\text{real-part } \text{my-z})\)

**Complex Package - generic arithmetic**

```
(define (install-complex-package)
  ;; Internal Rep: RepComplex = Rectangular U Polar
  ;; import 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 definitions...
  (define (add-complex z1 z2)
    (make-from-real-imag (+ (real-part z1) (real-part z2))
                         (+ (imag-part z1) (imag-part z2))))
  (define (sub-complex z1 z2)
    (make-from-real-imag (- (real-part z1) (real-part z2))
                         (- (imag-part z1) (imag-part z2))))
  (define (mul-complex z1 z2)
    (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                       (+ (angle z1) (angle z2))))
  (define (div-complex z1 z2)
    (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                       (- (angle z1) (angle z2))))

  ;; interface to rest of system -- export to table
  ;; External Rep: Complex = 'complex X RepComplex
  (define (tag z) (attach-tag 'complex z))
  (put 'add 'complex-complex) (lambda (z1 z2) (tag (add-complex z1 z2))))
  (put 'sub 'complex-complex) (lambda (z1 z2) (tag (sub-complex z1 z2))))
  (put 'mul 'complex-complex) (lambda (z1 z2) (tag (mul-complex z1 z2))))
  (put 'div 'complex-complex) (lambda (z1 z2) (tag (div-complex z1 z2))))
  (put 'make-from-real-imag 'complex
        (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'complex
        (lambda (r a) (tag (make-from-mag-ang r a))))
'done)
```
Generic Arithmetic Operators

(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))

Ordinary Number Package

(define (install-number-package)
  ;; Internal rep: RepNum = Sch-Num
  ;; internal procedures -- just use Scheme!

  ;; External rep: Number = 'number X RepNum
  (define (tag x) (attach-tag 'number x))
  (put 'make 'number tag)
  (put 'add '(number number) (lambda (x y) (tag (+ x y))))
  (put 'sub '(number number) (lambda (x y) (tag (- x y))))
  (put 'mul '(number number) (lambda (x y) (tag (* x y))))
  (put 'div '(number number) (lambda (x y) (tag (/ x y))))
  'done)

  ;; External constructor for ordinary numbers: Sch-Num --> Number
  (define (create-number x) ((get 'make 'number) x))

Rational Number Package - Generic Number Arithmetic

(define (install-rational-package)
  ;; Internal rep: RepRat = Generic-Num X Generic-Num
  ;; internal procedures on RepRat
  (define (make-rat n d) (cons n d))
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (add-rat x y) (make-rat (add (mul (numer x) (denom y))
                                      (mul (denom x) (numer y)))
                                  (mul (denom x) (denom y))))
  (define (sub-rat x y) (make-rat (sub (mul (numer x) (denom y))
                                      (mul (denom x) (numer y)))
                                  (mul (denom x) (denom y))))
  (define (mul-rat x y) (make-rat (mul (numer x) (numer y))
                                  (mul (denom x) (denom y))))
  (define (div-rat x y) (make-rat (mul (numer x) (denom y))
                                  (mul (denom x) (numer y))))

  ;; External rep: Rational = 'rational X RepRat
  (define (tag x) (attach-tag 'rational x))
  (put 'make 'rational (lambda (n d) (tag (make-rat n d))))
  (put 'add '(rational rational) (lambda (x y) (tag (add-rat x y))))
  (put 'sub '(rational rational) (lambda (x y) (tag (sub-rat x y))))
  (put 'mul '(rational rational) (lambda (x y) (tag (mul-rat x y))))
  (put 'div '(rational rational) (lambda (x y) (tag (div-rat x y))))
'done)

;; External constructor interface
(define (create-rational n d) ((get 'make 'rational) n d))

**Coercion**

(define (number->rational x)
  (create-rational x (create-number 1)))

;; (put-coercion <from-type> <to-type> <procedure>)
(put-coercion 'number 'rational number->rational)

(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 ((t1 (car type-tags))
                (t2 (cadr type-tags))
                (arg1 (car args))
                (arg2 (cadr args)))
              (let ((t1->t2 (get-coercion t1 t2)))
                (t2->t1 (get-coercion t2 t1)))
              (cond (t1->t2
                      (apply-generic op (t1->t2 a1 a2))
                    (t2->t1
                      (apply-generic op a1 (t2->t1 a2)))
                (else (error "No method"))))))))
      (error "No method"))))

**Polynomials**

;; Term = <order> X <coefficient>
;;      = Pos-Integer X Generic-Num
;;
;; make-term: Pos-Integer, GenericNum -> Term
;; order: Term -> Pos-Integer
;; coeff: Term -> Generic-Num

;; TermList = empty-termlist U (Term X Termlist)
;;
;; the-empty-termlist: () -> empty-termlist
;; empty-termlist?: TermList -> Bool
;; adjoin-term: Term, TermList -> TermList
;; first-term: TermList -> Term
;; rest-terms: TermList -> TermList

(define p1
  (make-poly 'x
(adjoin-term (make-term 5 2)
  (adjoin-term (make-term 7 1)
    (adjoin-term (make-term 3 0)
      (the-empty-termlist)))))

Polynomial Package

(define (install-polynomial-package)
  ;; Internal Rep: RepPoly = Variable X TermList
  ;; internal procedures
  (define (make-poly variable term-list) ...)
  (define (add-poly p1 p2) ...)
  (define (mul-poly p1 p2) ...)
  ;; representations used for terms and term-lists
  ;; External Rep: Polynomial = 'polynomial X RepPoly
  (define (tag x) (attach-tag 'polynomial x))
  (put 'add '(polynomial polynomial) (lambda (p1 p2) (tag (add-poly p1 p2))))
  (put 'mul '(polynomial polynomial) (lambda (p1 p2) (tag (mul-poly p1 p2))))
  (put 'make 'polynomial (lambda (var terms) (tag (make-poly var terms))))
  'done)

Addition of Polynomials

;; add-poly: RepPoly, RepPoly -> RepPoly
(define (add-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
        (add-termlists (term-list p1) (term-list p2)))
      (error "Polys not in same var -- ADD-POLY" (list p1 p2))))

;;; addition of term-lists
(define (add-termlists 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-termlists (rest-terms L1) L2)))
                 (< (order t1) (order t2))
                 (adjoin-term t2 (add-termlists L1 (rest-terms L2))))
                  (else
                   (adjoin-term
                     (make-term (order t1) (add (coeff t1) (coeff t2)))
                     (add-termlists (rest-terms L1) (rest-terms L2))))))))