

## Data-Directed Programming

Consider

```
(define (real-part z)
  (cond ((rectangular? z) (car (contents z)))
        ((polar? z)
         (* (car (contents z))
            (cos (cdr (contents z)))))
        (else
         (error "unknown D-T in REAL-PART"
                z))))
```

## Problems with that code

- As more representation types are added, the procedure has to be rewritten
- Procedure gets longer
- Procedure gets more complicated
- Procedure is one big conditional statement

## The data-directed programming technique

- Replaces the conditional statement with a lookup table
- The chunks of code that were in the branches of the conditional are now indexed by the two dimensions of the lookup table (procedure X representation type)

## Lookup table for complex numbers

Operations	Representation types	
	polar	rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular
	(named procedures or lambda expressions)	

## A prototype implementation of lookup table

- Basic operations are put and get
- Book's implementation isn't until the next chapter
- Our implementation is simple, suitable for rapid prototyping, but not the most efficient
- Can be replaced later by the book's better implementation without changing code built on top of it

## Basic idea of our implementation

- Lookup table is just a list of triples of the form (operation type action)
- Simulates a sparse array of infinite size
- Put adds a triple to the list
- Get searches the list for the right triple and returns the action part
- The type is a list of the representation types of the operation's arguments

## And now for the implementation

```
(define operation-table ())  
(define (put operation type action)  
  (set! operation-table  
    (cons (list operation type action)  
          operation-table)))
```

(Forget you saw set!, the reassignment operator, until the next chapter.)

## The get procedure

```
(define (get operator type)  
  (define (get-aux list)  
    (cond ((null? list) #f)  
          ((and (equal? operator (caar list))  
                (equal? type (cadar list)))  
           (caddr list))  
          (else (get-aux (cdr list)))))  
  (get-aux operation-table))
```

## Each representation type in its own package

```
(define (install-rectangular-package)
  (put 'real-part '(rectangular) car)
  (put 'imag-part '(rectangular) cdr)
  (put 'magnitude
       '(rectangular)
       (lambda (z)
         (sqrt (+ (square (car z))
                   (square (cdr z)))))))
```

## Rectangular package continued

```
(put 'angle
     '(rectangular)
     (lambda (z) (atan (cdr z) (car z))))
(put 'make-from-real-imag
     'rectangular
     (lambda (x y)
       (attach-tag 'rectangular
                    (cons x y))))
```

## part 3

```
(put 'make-from-mag-ang
     'rectangular
     (lambda (r a)
       (attach-tag 'rectangular
                    (cons (* r (cos a))
                          (* r (sin a)))))))
'done
```

## Polar package

```
(define (install-polar-package)
  (put 'magnitude '(polar) car)
  (put 'angle '(polar) cdr)
  (put 'real-part
       '(polar)
       (lambda (z)
         (* (car z) (cos (cdr z)))))
```

## Polar package continued

```
(put 'imag-part
    '(polar)
    (lambda (z)
      (* (car z) (sin (cdr z)))))
(put 'make-from-mag-ang
    'polar
    (lambda (r a)
      (attach-tag 'polar (cons r a))))
```

## Part 3

```
(put 'make-from-real-imag
    'polar
    (lambda (x y)
      (attach-tag
        'polar
        (cons (sqrt (+ (square x)
                       (square y)))
              (atan y x)))))
'done)
```

## The constructors

```
(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))
```

## Generic operation call

```
(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
            "APPLY-GENERIC failed"
            (list op type-tags))))))
```

(note: operations can have more than one argument)

## The selectors

```
(define (real-part z)
  (apply-generic 'real-part z))
(define (imag-part z)
  (apply-generic 'imag-part z))
(define (magnitude z)
  (apply-generic 'magnitude z))
(define (angle z)
  (apply-generic 'angle z))
```

## Three programming styles

- 1) Data-directed programming, where code for each operator and representation type combination is stored in a 2-dimensional operation table

## Continued

- 2) Conventional-style programming, where each operator decides what code to run by testing the representation types of its arguments

In effect, each operator has one row of operation table built into it.

## Continued again

- 3) Message passing, where each data object decides what code to run by testing the name of the operation it is being asked to perform

In effect, each data object has one column of the operation table built into it.

## A message passing version

```
(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 "MAKE-FROM-REAL-MAG failed"
                  op))))
  dispatch)
```

## Uses a different apply-generic

```
(define (real-part z)
  (apply-generic 'real-part z))
etc.
(define (apply-generic op arg)
  (arg op))
or more directly,
(define (real-part z)
  (z 'real-part))
```

## Advantage

The main advantage of the message passing programming style is that it more effectively hides the internal structure of data objects so that programmers are not tempted to access the insides of the data objects with `cars` and `cdrs`.

The programming style for classes in C++ most closely resembles message passing.

## Generic operations

- We want to do arithmetic with any combination of ordinary numbers, rational numbers and complex numbers
- We'll use the data-directed programming style
- Ordinary Scheme numbers will have to be represented and tagged like all the rest

## Basic arithmetic operations

```
(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))
```

## Scheme number package

```
(define (install-scheme-number-package)
  (define (tag x)
    (attach-tag 'scheme-number x))
  (put 'add
       '(scheme-number scheme-number)
       (lambda (x y) (tag (+ x y))))
  (put 'sub
       '(scheme-number scheme-number)
       (lambda (x y) (tag (- x y))))
```

## continued

```
(put 'mul
     '(scheme-number scheme-number)
     (lambda (x y) (tag (* x y))))
(put 'div
     '(scheme-number scheme-number)
     (lambda (x y) (tag (/ x y))))
(put 'make
     'scheme-number
     (lambda (x) (tag x)))
'done)
```

## Scheme-number constructor

```
(define (make-scheme-number n)
  ((get 'make 'scheme-number) n))
```

## Rational number package

```
(define (install-rational-package)
  (define numer car)
  (define denom cdr)
  (define (make-rat n d)
    (let ((g (gcd n d)))
      (cons (/ n g) (/ d g))))
  (define (tag x) (attach-tag 'rational x))
```

## rational package continued

```
(put 'add
     '(rational rational)
     (lambda (x y)
       (tag (make-rat
              (+ (* (numer x) (denom y))
                 (* (numer y) (denom x))))
              (* (denom x) (denom y))))))
```

## subtraction

```
(put 'sub
     '(rational rational)
     (lambda (x y)
       (tag (make-rat
              (- (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))))
```

## multiplication

```
(put 'mul
     '(rational rational)
     (lambda (x y)
       (tag (make-rat
              (* (numer x) (numer y))
              (* (denom x) (denom y))))))
```

## division

```
(put 'div
  '(rational rational)
  (lambda (x y)
    (tag (make-rat
          (* (numer x) (denom y))
          (* (denom x) (numer y)))))))
```

## constructor code

```
(put 'make
  'rational
  (lambda (n d) (tag (make-rat n d))))

'done)

(define (make-rational n d)
  ((get 'make 'rational) n d))
```

## Complex number package

```
(define (install-complex-package)
  (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))
  (define (tag z) (attach-tag 'complex z)))
```

## Addition

```
(put 'add
  '(complex complex)
  (lambda (z1 z2)
    (tag (make-from-real-imag
          (+ (real-part z1)
            (real-part z2))
          (+ (imag-part z1)
            (imag-part z2)))))))
```

## Subtraction

```
(put 'sub
  '(complex complex)
  (lambda (z1 z2)
    (tag (make-from-real-imag
      (- (real-part z1)
         (real-part z2))
      (- (imag-part z1)
         (imag-part z2)))))))
```

## Multiplication

```
(put 'mul
  '(complex complex)
  (lambda (z1 z2)
    (tag (make-from-mag-ang
      (* (magnitude z1)
         (magnitude z2))
      (+ (angle z1) (angle z2)))))))
```

## Division

```
(put 'div
  '(complex complex)
  (lambda (z1 z2)
    (tag (make-from-mag-ang
      (/ (magnitude z1)
         (magnitude z2))
      (- (angle z1) (angle z2)))))))
```

## Constructors

```
(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)
```

## continued

```
(define (make-complex-from-real-imag x y)
  ((get 'make-from-real-imag 'complex) x y))
(define (make-complex-from-mag-ang r a)
  ((get 'make-from-mag-ang 'complex) r a))
```

## Installing it all

```
(install-scheme-number-package)
(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
```

## Coercion

One approach:

```
(put 'add
     '(complex scheme-number)
     (lambda (z x)
       (tag (make-from-real-imag
                (+ (real-part z) x)
                (imag-part z))))))
```

Awkward when there are many combinations

## A better way

```
(put 'coerce
     'scheme-number
     (lambda (n)
       (make-rational (contents n) 1)))
(put 'coerce
     'rational
     (lambda (r)
       (make-complex-from-real-imag
        (/ (car (contents r))
           (cdr (contents r))))))
```

## Precedence info to control coercion

```
(put 'scheme-number 'rational 'precedence)
(put 'scheme-number 'complex 'precedence)
(put 'rational 'complex 'precedence)
```

## Revised apply-generic

```
(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)))
                (a1 (car args)
                    (a2 (cadr args))))
              (error "no method"
                     (list op
                           type-tags)))))))
```

## continued

```
(let ((t1up (get 'coerce t1))
      (t2up (get 'coerce t2))
      (p1 (get t1 t2))
      (p2 (get t2 t1)))
  (cond (p1
         (apply-generic
          op
          (t1up a1)
          a2))
        (p2
         (apply-generic
          op
          a1
          (t2up a2))))))
```

## again

```
(p2
 (apply-generic
  op
  a1
  (t2up a2)))
(else
 (error
  "no method"
  (list
   op
   type-tags))))))
```

**last line**

```
(error "no method"  
      (list of type-tags))))))
```