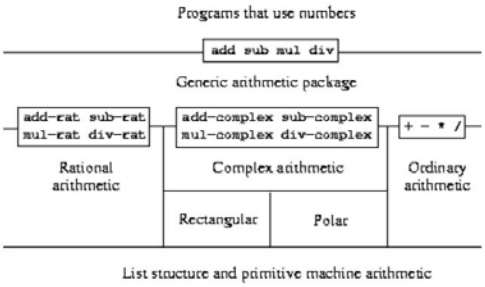
Topic 15
Generic Arithmetic Operations

Section 2.5.1 & 2.5.2

Systems with Generic Operations

- Previous section: designed systems in which data objects can be represented in more than one way
- Key idea: link the code that specifies the data operations to the several representations via generic interface procedures.
- Extend this idea: define operations that are generic over different representations AND over differet kinds of arguments.
- Several different arithmetic packages: regular, rational, complex – let's put them all together

Generic Use of Numbers



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

```
; generic operations definitions to use we would need to
; attach a tag to each kind of number and cause the
; generic procedure to dispatch the appropriate package
; for the data types of its arugments
(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

```
; package for handling ordinary scheme numbers
; note the key is (scheme-number scheme-number)
; since each takes two arguments, both of which
; are ordinary scheme-numbers
(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))))
```

continued

```
(put 'sub
   '(scheme-number scheme-number)
   (lambda (x y) (tag (- x y))))
(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

```
; user of the scheme-number package will create
; tagged ordinary numbers by means of the make
; procedure
(define (make-scheme-number n)
  ((get 'make 'scheme-number) n))
```


Rational number package

```
; package for performing rational arithmetic
(define (install-rational-package)
  ; internal procedures
  (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

```
; interface to rest of system
(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

```
; package for handling complex numbers
(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))
; interface to rest of the system
  (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

```
; exporting complex numbers to outside world

(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))
```


Complex Numbers – two levels of export

- Notice that complex numbers for a two-level tag system. A typical complex number e.g., 1 + 2i in rectangular form will be represented as
  (complex (rectangular 1 . 2))

First tag directs to complex number package, once there, second tag directs to rectangular package.

Installing it all

```
; run the procedures to set up the operations table
; operation-table will hold the triples of
; operations, their type, and the associated action
(install-scheme-number-package)
(install-rational-package)
(install-rectangular-package)
(install-polar-package)
(install-complex-package)
```


Combining Data of Different Types

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 – transform objects of one type into another type

```
;; to allow operations between mixed types
;; must allow coercion between types

; puts a procedure for coercing a scheme-number
; into a rational number
(put 'coerce
    'scheme-number
    (lambda (n)
     (make-rational (contents n) 1)))
```


Coerce a rational number into a complex one

```
; puts a procedure for coercing a rational
; number into a complex number
(put 'coerce
    'rational
    (lambda (r)
     (make-complex-from-real-imag
       (/ (car (contents r))
          (cdr (contents r)))
```


Precedence info to control coercion

```
When a mixed procedure is encountered,
  must try to coerce arguments.  Thus,
  must know which way coercion can occur.
  To do this, put precedence information
  in the operations table.

;; must put precedence information in the operation
;; table to control coercion
(put 'scheme-number 'rational 'precedence)
(put 'scheme-number 'complex 'precedence)
(put 'rational 'complex 'precedence)
```


Revised apply-generic

```
; new apply-generic procedure attempts to
; do type coercion if no operator with the
; appropriate type is defined in the operation
; table
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
```


continued

```
(if proc
    ; if procedure of appropraite type
    ; exists apply that procedure
    (apply proc (map contents args))
```



```
        ; otherwise, attempt to do coercion if
        ; there are two arguments
        (if (= (length args) 2)
          (let ((t1 (car type-tags))
                (t2 (cadr type-tags))
                (a1 (car args))
                (a2 (cadr args)))
            (let ((t1up (get 'coerce t1))
                  (t2up (get 'coerce t2))
                  (p1 (get t1 t2))
                  (p2 (get t2 t1)))
```

```
; t1up and t2up contain coercion
; procedures if they exist
; p1 and p2 tell whether precedence is correct
(cond (p1
        (apply-generic
        op
        (t1up a1)
        a2))
       (p2
        (apply-generic
        op
        a1
       (t2up a2)))
```



```
      (else
        (error
         "no method"
         (list
          op
          type-tags))))))
(error "no method"
       (list op type-tags)))))))
```