「アルゴリズムとデータ構造入門」 第 12 回 資料 (*Jan. 11, 2005,* 奥乃)
http://winnie.kuis.kyoto-u.ac.jp/∼okuno/Lecture/04/IntroAlgDS/

1 よきプログラミングスタイル — 例 微分係数を求めるだけでなく, 一般の数値微分も

```
(define dx 0.0038)

(define (ddx f x)
  (/ (- (f (+ x dx)) (f x)) dx) )

(ddx square 3)
```

この手続きの signature

$$\underbrace{(\underbrace{(number \Longrightarrow number)}_{\texttt{f}}, \underbrace{number}_{\texttt{x}}) \Longrightarrow number}_{\texttt{ddx}}$$

```
(define (ddx f x)
  ((deriv f) x) )
((deriv (deriv square)) 3)
((deriv (deriv (deriv square))) 3)
```

```
(define (deriv f)
  (lambda (x)
     (/ (- (f (+ x dx)) (f x)) dx) ))

((deriv square) 3)
```

この手続きの signature

$$\underbrace{\underbrace{(number \Longrightarrow number)}_{\texttt{f}} \Longrightarrow \underbrace{(number \Longrightarrow number)}_{\texttt{lambda式}}}_{\texttt{deriv}}$$

```
(define (compose f g)
   (lambda (x) (f (g x))) )
(define 2nd-deriv (compose deriv deriv))
((2nd-deriv square) 8))
((compose squere sqrt) 7)
```

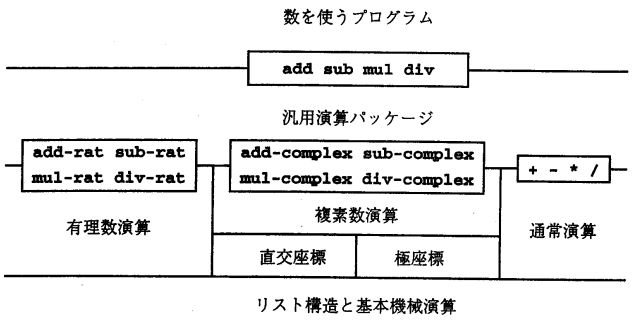
2.5 Systems with Generic Operations

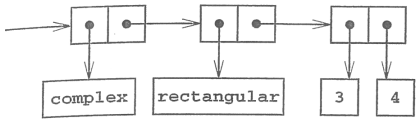
2.5.1 Generic Arithmetic Operations (Fig.2.23)

`(apply + (list 1 2 3 4))` **のアナロジー**

```
(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))

(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))) )
  (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)

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

(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)) )) 右上に続く.
```

```
  (define (add-rat x y)
    (make-rat (+ (* (numer x) (denom y))
                 (* (numer y) (denom x)) )
              (* (denom x) (denom y)) ))
  (define (sub-rat x y)
    (make-rat (- (* (numer x) (denom y))
                 (* (numer y) (denom x)) )
              (* (denom x) (denom y)) ))
  (define (mul-rat x y)
    (make-rat (* (numer x) (numer y))
              (* (denom x) (denom y)) ))
  (define (div-rat x y)
    (make-rat (* (numer x) (denom y))
              (* (denom x) (numer y)) ))
  ;;  interface to rest of the system
  (define (tag x) (attach-tag 'rational x))
  (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))) )

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

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






```
(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)
    (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 the system
  (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)

(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))

(put 'real-part '(complex) real-part)
(put 'imag-part '(complex) imag-part)
(put 'magnitude '(complex) magnitude)
(put 'angle '(complex) angle)
```

2.5.2 Combining Data of Different Types

1. Add operations for every combination of types

```
(define (add-complex-to-schemenum z x)
  (make-from-real-imag (+ (real-part z) x) (imag-part z)) )

(put 'add '(complex scheme-number)
     (lambda (z x) (tag (add-complex-to-schemeumb z x))) )
```

2. Coersion (強制型変換)

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

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

```
(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 ((type1 (car type-tags)) (type2 (cadr type-tags))
                    (a1 (car args)) (a2 (cadr args)) )
                (let ((t1->t2 (get-coercion type1 type2))
                      (t2->t1 (get-coercion type2 type1)))
                  (cond (t1->t2 (apply-generic op (t1->t2 a1) a2))
                        (t2->t1 (apply-generic op a1 (t2->t1 a2)))
                        (else (error "No method for these types" (list op type-tags))) )))
              (error "No method for these types" (list op type-tags)) )))))
(define (scheme-number->scheme-number n) n)
(define (complex->complex z) z)
;: (put-coercion 'scheme-number 'scheme-number
;:               scheme-number->scheme-number)
;: (put-coercion 'complex 'complex complex->complex)

(define (exp x y) (apply-generic 'exp x y))
;: (put 'exp '(scheme-number scheme-number)
;:      (lambda (x y) (tag (expt x y))))
```

2.5.3 Symbolic Algebra

```
(define (add-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (add-terms (term-list p1) (term-list p2)) )
      (error "Polys not in same var -- ADD-POLY" (list p1 p2)) ))

(define (mul-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (mul-terms (term-list p1) (term-list p2)) )
      (error "Polys not in same var -- MUL-POLY" (list p1 p2)) ))

;;  *incomplete* skeleton of package
(define (install-polynomial-package)
  ;;  internal procedures
  ;;  representation of poly
  (define (make-poly variable term-list)
    (cons variable term-list))
  (define (variable p) (car p))
  (define (term-list p) (cdr p))
  (define (variable? x) (symbol? x))
  (define (same-variable? v1 v2)
    (and (variable? v1) (variable? v2) (eq? v1 v2)) )

  ;;  representation of terms and term lists
  ;;[ procedures adjoin-term ... coeff は下記参照]

  ;;(define (add-poly p1 p2) ... )   ;;[add-poly は上記]

  ;;(define (mul-poly p1 p2) ... )   ;;[mul-poly は上記]

  ;;  interface to rest of the system
  (define (tag p) (attach-tag 'polynomial p))
  (put 'add '(polynomial polynomial)
       (lambda (p1 p2) (tag (add-poly p1 p2))))
  (put 'mul '(polynomial polynomial)
       (lambda (p1 p2) (tag (mul-poly p1 p2))))
```

多項式の表現法

$x^5 + 2x^4 + 3x^2 - 2x - 5 \Longrightarrow$
`((5 1) (4 2) (2 3) (1 -2) (0 -5))`

$x^{100} + 2x^2 + 1 \Longrightarrow$
`((100 1) (2 2) (0 1))`

```
  (put 'make 'polynomial
       (lambda (var terms) (tag (make-poly var terms))))
  'done)

(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))
                  (adjoin-term t2 (add-terms L1 (rest-terms L2))) )
                 (else
                  (adjoin-term
                   (make-term (order t1) (add (coeff t1) (coeff t2)))
                   (add-terms (rest-terms L1) (rest-terms L2)) )))))))

(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) )))

(define (mul-term-by-all-terms t1 L)
  (if (empty-termlist? L)
      (the-empty-termlist)
      (let ((t2 (first-term L)))
        (adjoin-term
         (make-term (+ (order t1) (order t2)) (mul (coeff t1) (coeff t2)))
         (mul-term-by-all-terms t1 (rest-terms L)) ))))

;;  Representing term lists
(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons term term-list) ))

(define (the-empty-termlist) '())
(define (first-term term-list) (car term-list))
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))

(define (make-term order coeff) (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))

;;  Constructor
(define (make-polynomial var terms)
  ((get 'make 'polynomial) var terms) )
```

3 宿題 – 〆切: １月１７日 (月) 午後５時 事務室レポート箱

- **Exercise 2.77 ～ 2.97**
- **プログラムが動くことにより解を確認すること. (添削が試験までに間に合うかは?)**