アルゴリズムとデータ構造入門

1.手続きによる抽象の構築

1.3 Formulating Abstractions with Higher-Order Procedures
（高階手続きによる抽象化）

奥 乃 博

1. 3，5，7で割った時の余りが各々1，2，3という数は何か？

1

11月1日・本日のメニュー

- 1.2.6 Example: Testing for Primality
- 1.3.1 Procedures as Arguments
- 1.3.2 Constructing Procedures Using `Lambda'
- 1.3.3 Procedures as General Methods
- 1.3.4 Procedures as Returned Values

2

左上教科書表紙：http://mitpress.mit.edu/images/products/books/0262011530-f30.jpg

Greatest Common Divisors（最大公約数）

- ユークリッドの互除法
- GCD(*a*, *b*) = GCD(*b*, *a* mod *b*)

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b)) ))
```

3

Chinese Remainder Theorem

連立1次合同式

$$x \equiv b_1 \pmod{d_1}$$
$$x \equiv b_2 \pmod{d_2}$$
$$\dots$$
$$x \equiv b_t \pmod{d_t}$$

の場合、$d_1, d_2, \dots d_t$ が互いに素であれば、

$$n = d_1 d_2 \dots d_t$$

を法として、ただ一つの解がある。

まず、$n / d_i = n_i$ とおけば、d_i と n_i は互いに素であるから、

$$n_i x_i \equiv 1 \pmod{d_i}$$

の解 x_i を求めることができる。ここで、

$$x \equiv b_1 n_1 x_1 + b_2 n_2 x_2 + \dots + b_t n_t x_t \pmod{n}$$

とすれば、この x は明らかにもとの合同式をすべて満足する。

5

Chinese Remainder Theoremの例

x mod *105* は？

- 3 * 5 * 7 = 105
- $x \equiv 1 \pmod 3$
- $x \equiv 2 \pmod 5$
- $x \equiv 3 \pmod 7$
- 35*2 ≡1 (mod 3)
- 21*1 ≡1 (mod 5)
- 15*1 ≡1 (mod 7) より、
- x mod 105

  ≡1*35*2 +2*21*1+3*15*1 mod 105

  =157 mod 105 ≡52 mod 105

6

Lameの定理

- GCD(a, b)（ただし、$b < a$）の計算に k step 必要なら、$b \geq \mathrm{Fib}(k)$
- 例えば、GCD(m, n)（ただし、$n < m$）が k step かかるとすると、$n \geq \mathrm{Fib}(k) \fallingdotseq \Phi^k / \sqrt{5}$

$$\phi = \frac{1}{2}\left(1+\sqrt{5}\right) \qquad Fib(n) \cong \frac{\phi^n}{\sqrt{5}}$$

- つまり、ステップ数は、n の対数的に増加。
- $\Theta(\log n)$ steps

7

Order of Growth: Examples

手続き	ステップ数	スペース
factorial	$\Theta(n)$	$\Theta(n)$
fact-iter	$\Theta(n)$	$\Theta(1)$
テーブル参照型fact	$\Theta(1)$	$\Theta(n)$
fib	$\Theta(\phi^n)$	$\Theta(n)$
fib-iter	$\Theta(n)$	$\Theta(1)$
テーブル参照型fib	$\Theta(1)$	$\Theta(n)$

8

Testing for Primality

```
(define (smallest-divisor n)
  (find-divisor n 2)
```

```
(define (find-devisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n
                 (+ test-divisor 1) )) ))
```

```
(define (divides? a b)
  (= (remainder b a) 0) )
```

```
(define (prime? n)
  (= n (smallest-divisor n)) )
```

10

Improvement by HGO

```
(define (smallest-divisor n)
  (find-divisor n 3)

(define (find-devisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n
                 (+ test-divisor 2) )) ))

(define (divides? a b)
  (= (remainder b a) 0) )

(define (prime? n)
  (if (even? n)
      2
      (= n (smallest-divisor n)) ))
```

The Fermat Test

- $a^{p-1} \equiv 1 \bmod p$ if p が素数(prime)

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m)) m) )
        (else
         (remainder (* base (expmod base (- exp 1) m)) m) )
```

```
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a) )
  (try-it (+ 1 (random (- n 1)))))
```

```
(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false) ))
```


Probabilistic Algorithms(確率的アルゴリズム)

- Carmichael numbers: 561, 1105, 1072, 2465,
  $a^{560} = a^2\ a^{10}\ a^{16}$
  $a^2 \equiv 1 \bmod 3$, $a^{10} \equiv 1 \bmod 11$, $a^{16} \equiv 1 \bmod 17$
  $\Rightarrow\ a^{560} \equiv 1 \bmod 561 = 3 * 11 * 17$
- Fermat's testは、エラーの機会を任意に小さくできる。
  → ***probabilistic algorithm***
  必要条件のみ満足
- ***Algorithm***: Wilson's test
  p is a prime precisely
  when $(p-1)! \equiv -1 \bmod p$
  必要十分条件

Discussion: Fermat's or Wilson's?

1. 単純な素数判定:
2. Fermat's test: p が素数なら
   $\forall a < p,\ a^{(p-1)} \equiv 1 \bmod p$
3. Wilson's test: p が素数である必要十分条件は
   $(p-1)! \equiv -1 \bmod p$

   ちなみに

   $n! \sim (2\pi n)^{1/2} (n/e)^n$

1.3.1 Procedures as Arguments

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b)) ))
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b)) ))
(define (cube x) (* x x x))
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b)) ))
```

$$\sum_{i=a}^{b} i$$

$$\sum_{i=a}^{b} i^3$$

$$\sum_{i=a}^{b} \frac{1}{i(i+2)}$$

```
(define (<name> a b)
  (if (> a b)
      0
      (+ (<term> a)
         (<name> (<next> a) b)) ))
```

16

1.3.1 Procedures as Arguments

```
(define (<name> a b)
  (if (> a b)
      0
      (+ (<term> a)
         (<name> (<next> a) b)) ))
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a)next b)) ))
(define (inc n) (+ n 1))
(define (sum-cubes a b)
  (sum cube a inc b) )
(define (identity x) x)
(define (sum-integers a b)
  (sum identity a inc b) )
```

$$\sum_{i=a,next(i)}^{b} f(i)$$

$$\sum_{i=a,i+1}^{b} cube(i)$$

$$\sum_{i=a,i+1}^{b} i$$

17

Pi-Sum (Pi/8) の計算方法

$$\sum_{i=a,\pi next(i)}^{b} \pi term(i)$$

```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))) )
  (define (pi-next x)(+ x 4) )
  (sum pi-term a pi-next b) )
```

18

積分（integral）の計算方法

$$\int_a^b f = \left[f\left(a+\frac{dx}{2}\right)+f\left(a+dx+\frac{dx}{2}\right)+f\left(a+2dx+\frac{dx}{2}\right)+\cdots\right]dx$$

$$\left(\sum_{i=a,+\Delta x}^{b} f(i)\right)\Delta x$$

```
(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
      dx ))
```


Ex.1.31 Product

$$\prod_{i=a,next(i)}^{b} f(i)$$

```
(define (product term a next b)
  (if (> a b)
      1
      (* (term a)
         (product term (next a)next b))
  ))
(define (product-cubes a b)
  (product cube a inc b) )
```

$$\prod_{i=a,i+1}^{b} i^3$$

```
(define (product-integers a b)
  (product identity a inc b) )
```

$$\prod_{i=a,i+1}^{b} i$$

Ex.1.32 Accumulation

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b)) )
```

$$\sum_{i=a,next(i)}^{b} f(i)$$

```
(define (product term a next b)
  (if (> a b)
      1
      (* (term a)
         (product term (next a) next b)) )
```

$$\prod_{i=a,next(i)}^{b} f(i)$$

```
(define (<combiner> <name> <term> a <next> b)
  (if (> a b)
      <null-value>
      (<combiner> (<term> a)
         (<name> <term> (<next> a) <next> b))
  ))
```

Ex.1.32 Accumulation

```
(define (<combiner> <name> <term> a <next> b)
  (if (> a b)
      <null-value>
      (<combiner> (<term> a)
         (<name> <term> (<next> a) <next> b))
 ))

(define (accumulate combiner null-value
   term a next b)
  (if (> a b)
      null-value
      (combiner (term a)
         (accumulate combiner null-value
               term (next a) next b ))))
```

22

lambda: Anonymous procedure

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1))) ))
```

は次の式と等価

```
(define fact
 (lambda (n)
   (if (= n 0)
       1
       (* n (fact (- n 1))) )))
```

24

Lambda as anonymous procedure

```
(lambda (x) (+ x 4))
((lambda (x) (+ x 4)) 5)

(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))) )
  (define (pi-next x)(+ x 4) )
  (sum pi-term a pi-next b) )

(define (pi-sum a b)
  (sum (lambda (x) (/ 1.0 (* x (+ x 2))
    a
    (lambda (x) (+ x 4))
    b ))
```

Using let to create local variables

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

$$a = 1 + xy$$
$$b = 1 - y$$
$$f(x,y) = xa^2 + yb + ab$$

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b) ))
  (f-helper
    (+ 1 (* x y))
    (- 1 y) ))
```

26

1.3.2 Local Variables with `let`

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b) ))
  (f-helper
    (+ 1 (* x y))
    (- 1 y) ))

(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)) )
    (+ (* x (square a))
       (* y b)
       (* a b) )))
```

```
(define (f x y)
  ((lambda (a b)
    (+ (* x (square a))
       (* y b)
       (* a b) ))
   (+ 1 (* x y))
   (- 1 y) ))
```

(let ((<v_1> <e_1>)
       (<v_2> <e_2>)
       …
       (<v_n> <e_n>))
   <*body*>)

シンタックス・シュガー

27

scope of variables

```
(let ((x 7))
  (+ (let ((x 3))
       (+ x (* x 10)) )
     x) )
```

```
(let ((x 5))
  (let ((x 3)
        (y (+ x 2)) )
    (* x y) ))
```

Substition model

28

scope of variables

```
(let ((x 7))          x = 7
  (+ (let ((x 3))       x = 3
       (+ x (* x 10)) )   -> 33
     x) )             x = 7 -> 40

(let ((x 5))          x = 5
  (let ((x 3)           x = 3
        (y (+ x 2)) )   y = 7
    (* x y) ))          -> 21
```

29

1.3.3 Procedures as General Methods

Finding roots of equations by the half-interval method（区間二分法）

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                 (search f neg-point midpoint))
                ((negative? test-value)
                 (search f midpoint pos-point))
                (else midpoint))))))
```

32

Finding roots of equations by the half-interval method

```
(define (close-enough? x y)
  (< (abs (- x y)) 0.001))

(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (negative? b-value) (positive? a-value))
           (search f b a))
          (else
           (error "Values are not of opposite sign" a b))
    )))
```

L：開始時の区間長、T:誤差許容度、ステップ数：$\Theta(\log(L/T))$

33

Finding fixed points of functions(不動点)

Xが不動点 f(x) = x　　$f(x), f(f(x)), f(f(f(x))), \ldots$

```
(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

34

Finding fixed points of functions(不動点)

```
(fixed-point cos 1.0)

(fixed-point (lambda (y) (+ (sin y)
   (cos y)))
```

y*y=x y=x/y と書くと、

Looking for a fix-point of the function y |-> x/y

```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y))
               1.0))
```

35

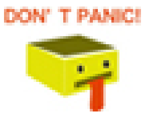
11月1日・本日のメニュー

- 1.2.6 Example: Testing for Primality
- 1.3.1 Procedures as Arguments
- **Intermission**
- 1.3.2 Constructing Procedures Using \`Lambda'
- 1.3.3 Procedures as General Methods
- 1.3.4 Procedures as Returned Values

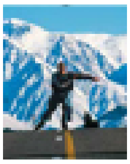
42

What is this instrument?

- A traditional roller-blader?
- A traditional inliner skate?
- Abacus
- 算盤 （そろばん）

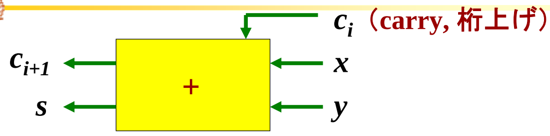


- そろわん



画像出所：http://www.newzealand.com/travel/library/847_3.jpg

Abacus & Binary Adder (2進加算器)




```
(define (adder x y c)
   (define (carry x y c)
      (if (or (and (= x 1) (= y 1))
              (and (= y 1) (= c 1))
              (and (= c 1) (= x 1)) )
          1 0 ))
   (define (sum x y c)
      (xor x y c) )
   (cons (sum x y c) (carry x y c)) )

(define (xor x y z)
  (if (= x 0)
      (if (= y 0) z (if (= z 0) 1 0))
      (if (= y 0) (if (= z 0) 1 0) z) ))
```


宿題：11月7日午後5時締切

- `lambda` を組合わせて手続きをくみ上げる
- 宿題は、次の10問：
- Ex.1.21, 1.23, 1.25, 1.29, 1.30, 1.31, 1.32, 1.33, 1.34, 1.35.
- 実行時間の測定は `(time (f a))`