アルゴリズムとデータ構造入門
2.データによる抽象の構築
2.1データ抽象化入門
2.2 階層データ構造と閉包性

奥 乃 博

大学院情報学研究科知能情報学専攻

http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/10/IntroAlgDs/

okuno@i.kyoto-u.ac.jp, okuno@nue.org

阿曽 慎平(M1) 奥乃研・音楽G (10号館4階奥乃1研)

平澤 恭治(M1) 奥乃研・ロボット聴覚G (10号館4階奥乃1研)

11月16日・本日のメニュー

2 Building Abstractions with Data

- 2.1 Introduction to Data Abstraction
- 2.1.3 What Is Meant by Data?
- 2.1.4 Interval Arithmetic
- 2.2. Hierarchical Data and the Closure Property(階層データ構造と閉包性)
- 2.2.1 Representing Sequences(並び)
- 2.3.1 Quote

12月21日 中間試験を実施

既約化を抽象化の壁から見ると

有理数を使ったプログラム

プログラム領域での有理数

add-rat sub-rat mul-等

分子と分母から構成される有理数

make-rat numer denom

ペアとして構成される有理数

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g)) ))
```

```
(define (make-rat n d)
  (cond n d))
(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g) ))
(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g) ))
```

cons car cdr

ペアの実装法

2.1.3 データって何？ ペア(対、pair)再考

(make-rat n d) の満足すべき条件は

$$\frac{\text{(numer x)}}{\text{(denom x)}} = \frac{n}{d}$$

1. 通常のセルで構築: 構築子 cons, 選択子 car, cdr
2. 次の手続きで構築

```
(define (cons x y)                    構築子
   (define (dispatch m)
      (cond ((= m 0) x)
            ((= m 1) y)
            (else (error "Argument not 0 or 1 -- CONS"
                         m ))))
   dispatch )
(define (car z) (z 0))   選択子        (z 0) ⇒ car
(define (cdr z) (z 1))                 (z 1) ⇒ cdr
(define foo (cons 10 25)) -> (car foo) は？
```


ペア(対、pair)を手続きで実現

```
(define (cons x y)                    構築子
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0
                  or 1 -- CONS" m ))))
  dispatch)
(define (car z) (z 0))                選択子
(define (cdr z) (z 1))
```

- (define foo (cons 10 25))
- (car foo) ⇒ (foo 0) ⇒ 10
- (cdr foo) ⇒ (foo 1) ⇒ 25

もっとかっこよくペア(pair)を手続きで実現

```
(define (cons x y)
  (lambda (m) (m x y)) )
(define (car z)
  (z (lambda (p q) p)) )
(define (cdr z)
  (z (lambda (p q) q)) )
```

観測したらデータが得られる⇒量子コンピュータ風の計算

- (define foo (cons 10 25))
- (car foo) ⇒
  ((lambda (m) (m 10 25)) (lambda (p q) p))
  ⇒ ((lambda (p q) p) 10 25)
  ⇒ 10
- (cdr foo) ⇒
  ((lambda (m) (m 10 25)) (lambda (p q) q))
  ⇒ ((lambda (p q) q) 10 25)
  ⇒ 25

ペアの実装法を抽象化の壁から見ると

有理数を使ったプログラム

プログラム領域での有理数

`add-rat sub-rat mul-等`

分子と分母から構成される有理数

`make-rat numer denom`

ペアとして構成される有理数

`cons car cdr`

ペアの実装法

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
```

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0
                       or 1 -- CONS" m ))))
  dispatch)
(define (car z) (z 0))
(define (cdr z) (z 1))
```


11月16日・本日のメニュー

2 Building Abstractions with Data

- 2.1 Introduction to Data Abstraction
- 2.1.3 What Is Meant by Data?
- **2.1.4 Interval Arithmetic**
- 2.2. Hierarchical Data and the Closure Property（階層データ構造と閉包性）
- 2.2.1 Representing Sequences（並び）
- 2.3.1 Quote

2.1.4 Interval Arithmetic

- **Constructor**

```
(define (make-interval a b) (cons a b))
```

- **Selectors, predicates 等**

```
(define (upper-bound x)
(define (lower-bound x)
(define (equal-interval? x y)
(define (sub-interval x y)
```

- **Interval arithmetic は単位系変換で重要.「公差」**

–1*in*≒2.54*cm*, 1*ft*≒30.48*cm*, 1*yd*≒0.914*m*, 1*mile*≒1.609*km*, 1*nautical mile*≒1.852*km*, 1*acre*≒4047m^2, 1 *UKgal*≒4.54*l*, 1*USgal*≒3.79*l*, 1*bbl*≒159*l*, **1*πsec*≒1 *nano-century* (10^{-7} *year*)**, 1 *light year*≒9.461×10^{12}km（**9.461T*km***）

–1*oz*≒28.3*g*, 1*lb*≒0.454*Kg*, 1*ct*≒0.2*g*

2.1.4 Interval Arithmetic

```
(define (add-interval x y)
   (make-interval
     (+ (lower-bound x) (lower-bound y))
     (+ (upper-bound x) (upper-bound y)) ))

(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
    (make-interval (min p1 p2 p3 p4)
                   (max p1 p2 p3 p4))))

(define (div-interval x y)
   (mul-interval x
      (make-interval (/ 1.0 (upper-bound y))
                     (/ 1.0 (lower-bound y)) )))
```

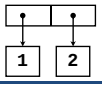
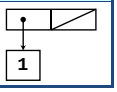

11月16日・本日のメニュー

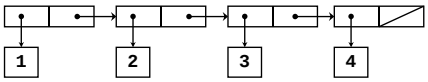
2 Building Abstractions with Data

- 2.1 Introduction to Data Abstraction
- 2.1.3 What Is Meant by Data?
- 2.1.4 Interval Arithmetic
- **2.2. Hierarchical Data and the Closure Property（階層データ構造と閉包性）**
- **2.2.1 Representing Sequences（並び）**
- 2.3.1 Quote

2.2 対（pairs）とシーケンス（sequences）

- 対（pair）
- **シーケンス（並び, sequence）**
- **`(cons 1 nil)`**
- **`(list 1)`と同じ**
- **`nil`は空リスト**

	`(cons 1 2)`	`(cons 1 nil)`
ドット記法（dotted notation）	`(1 . 2)`	`(1 . nil)` または `(1)`
箱ポインタ記法（box-and-pointer notation）		



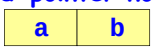
- `(list 1 2 3 4)`
- `(cons 1 (cons 2 (cons 3 (cons 4 nil) )))`

`(1 2 3 4)` あるいは

`(1 . (2 . (3 . (4 . nil))))`

2.2 Hierachical Data and the Closure Property (閉包性)

- Pair (cell) 対(セル)
  (cons a b)
- Box-and-pointer notation



- List structure(Backus-Naur Form, BNF記法)

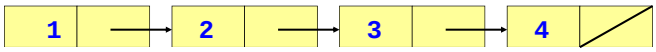
::= は定義　　| は代替

➢ <list> ::= <null> | (<element> . <element>)
➢ <element> ::= <name> | <number> | <list>

- Closure property (閉包性) of cons

2.2.1 Representing Sequences (シーケンス表現)

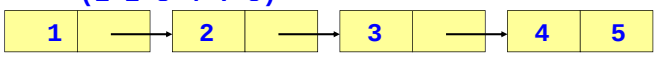
- Sequence (列・並び) 1, 2, 3, 4
  (1 2 3 4)



- (cons 1
    (cons 2
      (cons 3
        (cons 4 nil)
      )))
- (1 . (2 . (3 . (4 . nil))))
- (list 1 2 3 4)

Sequences表現の簡略化

- Sequence (列・並び) の表現の簡略化
  (1 2 3 4 . 5)



1. (xxx . nil) ⇒ (xxx)
2. (xxx . (yyy ···))⇒(xxx yyy ···)

(1 . (2 . (3 . (4 . 5))))
⇒ (1 2 . (3 . (4 . 5)))
⇒ (1 2 3 . (4 . 5))
⇒ (1 2 3 4 . 5)

2.2.1 List operations(リスト演算)

(define (list-ref items n) …)

- (list-ref 0 (list 0 1 2))
  0
- (list-ref 2 (list 0 1 2))
  2
- (list-ref 5 (list 0 1 2))
  ()

(define (length items) …)

- (length ())
  0
- (length (list 1 2 3))
  3

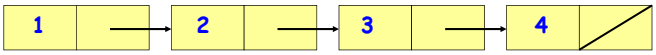
Lisp/Scheme Programming十戒

1. The First Commandment
   Always ask null? *as the first question in expressing any function.*
2. The Second Commandment
   Use cons *to build lists.*
3. The Third Commandment
   When building a list, describe the first typical element, and then cons it onto the natural recursion.

(Friedman, et al. "*The Little Schemer*", MIT Press)

List-ref by cdring down

(list-ref items n)
  if n=0, list-ref is car
  otherwise, (n-1)st item

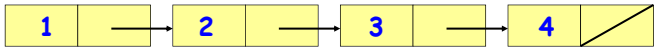



```
(define (list-ref items n)
    (if (= n 0)
        (car items)
        (list-ref (cdr items) (- n 1)) ))
```

- cdring down the list (cdr down)
- Tail recursion に注意(自動的に iteration に変換)

length by cdring down

```
(length items)
    if items is null?, length is 0
    otherwise, 1 + length of the rest
```




```
(define (length items)
        (if (null? items)
          0
          (+ 1 (length (cdr items)) ))
```

(+ (length (cdr items)) 1) との違い！

length : recursion and iteration versions

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)) ))

(define (length items)
  (define (iter a count)
     (if (null? a)
         count
         (iter (cdr a) (+ 1 count)) ))
  (iter items 0) )
```


Ex2.19 cc change of coins

```
(define us-coins (list 50 25 10 5 1))
(define uk-coins (list 100 50 20 10 5 2 1 0.5))
(define (cc amount coin-values)
   (cond ((= amount 0) 1)
         ((or (< amount 0) (no-more? coin-values))
          0 )
         (else
           (+ (cc amount
                  (except-first-demonination coin-values))
              (cc (- amount
                     (first-denomination coin-values) )
                  coin-values )))))
(define (except-first-denomination coins)
    (cdr coins) )
(define (first-denomination coins)
    (car coins) )
(define (no-more? coins)
    (null? coins) )
```


かっこよく自然数も手続きで実現

```
(define c0 (lambda (f) (lambda (x) x)))
(define (%succ c)
  (lambda (f) (lambda (x) (f ((c f) x))))
```

この自然数の表現を Church numerals （チャーチ数）という

```
(define c1 (%succ c0))
```

⇒ `(lambda (f) (lambda (x) (f ((c0 f) x)))`

⇒ `(lambda (f)`
`  (lambda (x)`
`    (f (((lambda (f) (lambda (x) x)) f) x)) ))`

⇒ `(lambda (f)`
`  (lambda (x)`
`    (f ((lambda (x) x) x)) ))`

⇒ `(lambda (f) (lambda (x) (f x)))`

自然数が0とf（後続関数）で定義

- `(define c1 (lambda (f) (lambda (x) (f x))))`
- `(define c2 (lambda (f) (lambda (x) (f (f x)))))`
- `(define c3 (lambda (f) (lambda (x) (f (f (f x))))))`

Church NumeralsとTAO

『老子』第42節 *Tao-te Ching, 42, Lao Tzu*

「道 (TAO) から一が生まれ，一から二が生まれ，二から三が生まれ，三から万物が生まれ，云々」

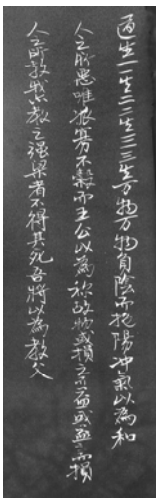
Tao produced the one.
The one produced the two.
The two produced the three.
And the three produced the ten thousand things.
The ten thousand things carry the yin and embrace the yang, and through the blending of the material force they achieve harmony.

と符合するものです．

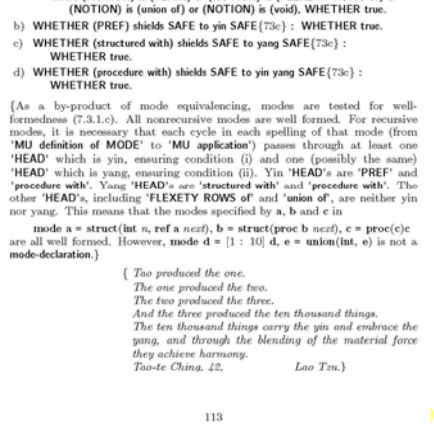
改良型Backus 記法が導入された Revised Report on the Algorithmic Language ALGOL 68 の113ページ にも上記の一節が引用されています．

INTREAL :: SIZETY integral ; SIZETY real.
SIZETY :: long LONGSETY ; short SHORTSETY ; EMPTY.

詳細は http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/04/IntroAlgDs/



Algol-68



7.4.1. Syntax

a) WHETHER (NOTION) shields SAFE to SAFE{73c} :
where (NOTION) is (PLAIN) or (NOTION) is (FLEXETY ROWS of) or (NOTION) is (union of) or (NOTION) is (void), WHETHER true.

b) WHETHER (PREF) shields SAFE to yin SAFE{73c} : WHETHER true.

c) WHETHER (structured with) shields SAFE to yang SAFE{73c} : WHETHER true.

d) WHETHER (procedure with) shields SAFE to yin yang SAFE{73c} : WHETHER true.

{As a by-product of mode equivalencing, modes are tested for well-formedness (7.3.1.c). All nonrecursive modes are well formed. For recursive modes, it is necessary that each cycle in each spelling of that mode (from 'MU definition of MODE' to 'MU application') passes through at least one 'HEAD' which is yin, ensuring condition (i) and one (possibly the same) 'HEAD' which is yang, ensuring condition (ii). Yin 'HEAD's are 'PREF' and 'procedure with'. Yang 'HEAD's are 'structured with' and 'procedure with'. The other 'HEAD's, including 'FLEXETY ROWS of' and 'union of', are neither yin nor yang. This means that the modes specified by a, b and c in

mode a = struct(int n, ref a next), b = struct(proc b next), c = proc(c)c

are all well formed. However, mode d = [1 : 10] d, e = union(int, e) is not a mode-declaration.}

{ *Tao produced the one.*
The one produced the two.
The two produced the three.
And the three produced the ten thousand things.
The ten thousand things carry the yin and embrace the yang, and through the blending of the material force they achieve harmony.
Tao-te China, 42. Lao Tzu.}

113

Operations on Church Numerals

```
(define c0 (lambda (f) (lambda (x) x)))
(define (%succ c)
  (lambda (f) (lambda (x) (f ((c f) x)))))
```

```
(define c1 (lambda (f) (lambda (x) (f x))))
(define c2 (lambda (f) (lambda (x) (f (f x)))))
(define c3 (lambda (f) (lambda (x) (f (f (f x))))))
```

1〜3の定義

```
(define (%add n m)
  (lambda (f) (lambda (x) ((m f) ((n f) x)))))
(define (%multiply n m)
    (lambda (f) (lambda (x) ((n (m f)) x))) )
(define (%power n m)
    (lambda (f) (lambda (x) (((m n) f) x))) )
```

比較・減算は難しい、再帰呼び出しが必要。
再帰呼び出しのために無名手続きをYオペレータで使用。

Church Numerals の出力

- 実際の動きを見るために，入出力の関数を定義しましょう．

```
(define (c->n c)  ; 出力
    ((c (lambda (x) (+ 1 x))) 0) )
(define (n->c n)  ; 入力
   (if (> n 0)
       (%succ (n->c (- n 1)))
        c0 ))
```

- 上記の c->n はメモリを大量に消費し遅い。高速版は:

```
(define (c->n c) ((c 1+) 0))
```

- では実験．

```
1.(c->n (%add (n->c 5) (n->c 3)))
2.(c->n (%multiply (n->c 5) (n->c 3)))
3.(c->n (%power (n->c 5) (n->c 3)))
4.(c->n (%add (%power c2 c3)
              (%multiply c3 (n->c 4)) ))
```


減算・比較・Fixed Point Operator F

- 減算は難しい。まず、大小比較を定義する。
- 再帰呼び出しに無名手続きを使う必要がある。
- Y オペレータを使う。

(Y F) = (F (Y F))

```
(define (Y F)
  ((lambda (s) (F (lambda (x) ((s s) x))))
   (lambda (s) (F (lambda (x) ((s s) x)))) ))

(define (fact0 f)
  (lambda (n) (if (= n 0) 1 (* n (f (- n 1))))) ))

((Y fact0) 10) => 3628800
```

詳細は Web ページにあります。
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/04/IntroAlgDs/

祝
京都大學
１１月祭

友達をつくろう

宿題はあります

KULAISISにログインし，

授業資料をダウンロードし，

確認フラグを立てること

DON'T PANIC!

11月16日・本日のメニュー

2 Building Abstractions with Data

- 2.1 Introduction to Data Abstraction
- 2.1.3 What Is Meant by Data?
- 2.1.4 Interval Arithmetic
- **2.2. Hierarchical Data and the Closure Property(階層データ構造と閉包性)**
- **2.2.1 Representing Sequences(並び)**
- 2.3.1 Quote

Lisp/Scheme Programming十戒

1. **The First Commandment**
   Always ask `null?` as the first question in expressing any function.
2. **The Second Commandment**
   Use `cons` to build lists.
3. **The Third Commandment**
   When building a list, describe the first typical element, and then cons it onto the natural recursion.

(Friedman, et al. "*The Little Schemer*", MIT Press)

数・cons (リスト) の生成関数

データ型	単位元	生成関数
自然数	`0` (零)	`successor`
リスト	`nil`, `()` (空リスト)	`cons`
文字列	`""` (空文字列)	`string-append`

Closure property (閉包性) of 自然数, cons, 文字列

データ型	単位元	演算	終了チェック
数	`0`	+	`zero?`
数	`1`	×	`zero?`
リスト	`nil`	`cons, car, cdr`	`null?`
文字列	`""`	`string-append`	`null-string?`

TUT 無, MIT Scheme有

The Fourth Commandment

Always change at least one argument while recurring.

When recurring on a list of atoms, `lat`, use `(cdr lat)`.

When recurring on a number, `n`, use `(sub1 n)`.
教科書では `(- n 1)`

When recurring on a list of S-expressions, `l`, use `(car l)` and `(cdr l)`, if neither `(null? l)` and `(atom? (car l))` are true.
教科書では `(not (pair? (car l)))`

It must be changed to be closer to termination. The changing argument must be tested in the termination condition:

- when using `cdr`, test termination with `null?` and
- when using `sub1`, test termination with `zero?`.

The Fifth Commandment

When building a value with `+`, always use `0` for the value of the terminating line, for adding `0` does not change the value of an addition.

When building a value with `*`, always use `1` for the value of the terminating line, for multiplying by `1` does not change the value of a multiplication.

When building a value with `cons`, always consider `()` for the value of terminating line.

Lisp/Scheme Programming十戒

6. **The Sixth Commandment**
   Simplify only after the function is correct.
7. **The Seventh Commandment**
   Recur on the subparts that are of the same nature
   - On the sublists of a list.
   - On the subexpressions of an arithmetic expression.
8. **The Eighth Commandment**
   Use help function to abstract from representations.
9. **The Ninth Commandment**
   Abstract common patterns with a new function.
10. **The Tenth Commandment**
    Build functions to collect more than one value at a time.

2.3.1 Quotation

- 定数データの表現: `quote` (引用) `'`
- `(define foo (list 'a 'b))`
  ⇒ `(a b)`
- `(define foo '(a b))` とほぼ同じ

append と reverse の例

```
(append (list 1 2 3) ())          (1 2 3)
(append (quote (1 2 3)) ())       (1 2 3)
(append '(1 2 3) ())              (1 2 3)
(append '(1 2 3) '(5 6 7))        (1 2 3 5 6 7)
(append () '(a b c))              (a b c)

(reverse '(1 2 3 4 5))            (5 4 3 2 1)
(reverse '(ni ku i shi  ku tsu u))
                                  (u tsu ku shi i ku ni)
(reverse '(に く い し  く つ う))
                                  (う つ く し い く に)
```


append by consing up while cdring down

```
(append list1 list2)
  if list1 is null?, append is list2
  otherwise, cons the 1st item of list1 and
   append of the rest of list1 and list2

(define (append list1 list2)
        (if (null? list1)
            list2
            (cons (car list1)
                  (append (cdr list1) list2)
            )))
```


reverse by consing up while cdring down

```
(reverse items)
  if items is null?, reverse is nil
  otherwise, append reverse of the rest of
  items and list of the 1st item of items

(define (reverse items)
        (if (null? items)
            nil
            (append (reverse (cdr items))
                    (list (car items)) )))
```