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
2.3 記号データ
2.4 抽象データの多重表現

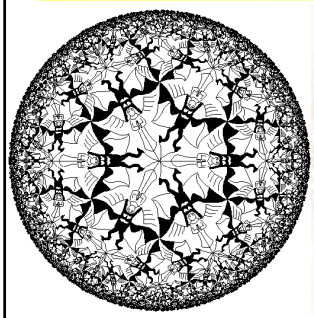
奥 乃 博
大学院情報学研究科
知能情報学専攻音声メディア分野
http://winnie.kuis.kyoto-u.ac.jp/
～okuno/Lecture/09/IntroAlgDs/
okuno@i.kyoto-u.ac.jp，okuno@nue.org

Escher's Square-Limit

H.G. Okuno

M.C. Escher

EscherのCircle-limit IV (1960)



Y.Y. Huang

M.C. Escher

1月7日・本日のメニュー

1. 2.3 Symbolic Data
2. 2.3.2 Symbolic Differentiation
3. 2.3.3 Representing Sets
4. 2.4 Multiple Representations for Abstract Data
5. 2.4.1 Representations for Complex Numbers

2.3 Symbolic Differentiation

本章で学ぶことは昔は世界トップの Computer Science で修士論文でした.

Hans Olsson, *Applications of automatic and symbolic differentiation in numerical computations,* Master's Thesis, June 1993,

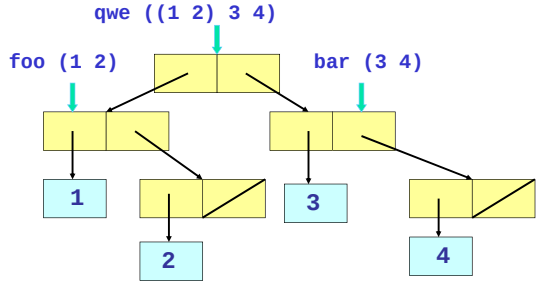
43

eq?

```
1. (eq? foo (list 1 2))
2. (eq? foo (car qwe))
3. (eq? bar (  ?  qwe))
```

```
(define foo (list 1 2))
(define bar '(3 4))
(define qwe (cons foo bar))
```

qwe ((1 2) 3 4)



45

微分を定義を思い出そう

- $\frac{dc}{dx} =$ 　　c が定数か x 以外の変数
- $\frac{dx}{dx} =$
- $\frac{d(u+v)}{dx} =$
- $\frac{d(uv)}{dx} =$

47

代数式の表現を考えてください

■次の代数式の表現法

1.和　$x+y$

2.積　ax

■代数式のための構築子･選択子･述語の設計

1.構築子(*constructor*)

```
(make-sum x y)
(make-product x y)
```

2.選択子(*selector*)

```
(addend s)
(augend s)
(multiplicant p)
(multiplier p)
```

3.述語(*predicate*)

```
(variable? x)
(same-variable? x y)
(sum? x)
(product? x)
```

49

代数式の構文

::= は定義

| は代替

<expression> ::= *<number>* | *<variable>* |
　(*<unary operator>*　*<expression>*) |
　(*< binary operator>* *< expression>*　*<expression>*) |
　(*<expression>*)

<unary operator> ::= + | - | *<function>*

<binary operator> ::= + | - | * | / | ^ | < | > | = | <= | >=

<function> ::= sin | cos | tan | log | …

50

代数式の表現法とその実装

■次の代数式の表現法　　表現法1　　表現法2

1. 和　$x+y$　　(+ x y)　　(+ (x y))
2. 積　xy　　(* x y)　　(* (x y))

■代数式のための構築子・選択子の設計

1. 構築子

```
(define (make-sum x y)
    (list '+ x y))
(define (make-product x y)
    (list '* x y) )
```

2. 選択子

```
(define (addend s) (cadr s))
(define (augend s) (caddr s))
(define (multiplicant p) (cadr p))
(define (multiplier p) (caddr p))
```

代数式表現の実装(続)

■代数式のための構築子・選択子・述語の設計

3. 述語

```
(define (variable? x) (symbol? x))
(define (same-variable? x y)
  (and (variable? x) (eq? x y) )
(define (sum? x)
  (and (pair? x)
       (eq? (car x) '+)) )
(define (product? x)
  (and (pair? x)
       (eq? (car x) '*)) )
```

52

代数式の表現法2を採用すると

■次の代数式の表現法　　表現法1　　表現法2

1. 和　$x+y$　　(+ x y)　　(+ (x y))
2. 積　ax　　(* x y)　　(* (x y))

■代数式のための構築子・選択子の設計

1. 構築子

```
(define (make-sum x y)
   (list '+ (list x y)) )
(define (make-product x y)
   (list '* (list x y)) )
```

2. 選択子

```
(define (addend s) (caadr s))
(define (augend s) (cadadr s))
(define (multiplicant p) (caadr p))
(define (multiplier p) (cadadr p))
```

代数式の表現法2を採用すると

■次の代数式の表現法

		表現法1	表現法2
1. 和	$x+y$	(+ x y)	(+ (x y))
2. 積	ax	(* x y)	(* (x y))

■代数式のための構築子・選択子の設計

1. 構築子

```
(define (make-sum x y)
   (list '+ (list x y)) )
(define (make-product x y)
   (list '* (list x y)) )
```

2. 選択子

```
(define (addend s) (caadr s))
(define (augend s) (cadadr s))
(define (multiplicant p) (caadr p))
(define (multiplier p) (cadadr p))
```


では微分手続きを定義してください

$$\frac{dc}{dx}=0$$
$$\frac{dx}{dx}=1$$
$$\frac{d(u+v)}{dx}=\frac{du}{dx}+\frac{dv}{dx}$$
$$\frac{d(uv)}{dx}=u\frac{dv}{dx}+v\frac{du}{dx}$$

```
(define (deriv exp var)
   (cond ((number? exp) 0)
         ((variable? exp)
          (if (same-variable? exp var) 1 0) )
         ((sum? exp)
           (make-sum (deriv (addend exp) var)
                     (deriv (augend exp) var)) )
         ((product? exp)
          (make-sum
             (make-product (multiplier exp)
                           (deriv (multiplicand exp)
                                  var ))
             (make-product (deriv (multiplier exp)
                                  var )
                           (multiplicand exp) )))
         (else
           (error "unknown expression type –
                   DERIV" exp ))))
```


代数式の表現・実装の問題点

1. (deriv '(+ x y) 'x)
   (+ 1 0)　　1
2. (deriv '(* x y) 'x)
   (+ (* y 1) (* 0 x))　　y
3. (deriv '(+ (* x y) (* 3 x)) 'x)
   (+ (+ (* y 1) (* 0 x))
      (+ (* x 0) (* 1 3)))
   y+3

何かおかしい

簡略化を忘れている。

微分結果の簡約化(その1)

- どの時点で簡略化をすればよいか.

```
(define (make-sum a1 a2)
   (cond ((=number? a1 0) a2)
         ((=number? a2 0) a1)
         ((and (number? a1) (number? a2))
          (+ a1 a2))
         (else (list '+ a1 a2)) ))

(define (=number? exp num)
     (and (number? exp) (= exp num)) )
```

59

微分結果の簡約化(その2)

```
(define (make-product m1 m2)
   (cond ((or (=number? m1 0)
              (=number? m2 0) )
          0 )
         ((=number? m1 1) m2)
         ((=number? m2 1) m1)
         ((and (number? m1) (number? m2))
          (* m1 m2))
         (else (list '* m1 m2)) ))
```

60

微分結果の簡略化の実験

1. `(deriv '(+ x y) 'x)`
   1
2. `(deriv '(* x y) 'x)`
   y
3. `(deriv '(+ (* x y) (* 3 x)) 'x)`
   (+ y 3)

今度は

簡略化成功

61

和と積に対する微分を拡張

1. 差、商に拡張
   (deriv '(- x y) 'x)
   (deriv '(/ 3 x) 'x)
2. 冪乗に拡張
   (deriv '(** x 3) 'x)
3. 2項演算子を多項演算子に拡張
   (deriv '(+ (* 3 x) y (* x y)) 'x)
   (deriv '(* x y (+ x 3)) 'x)
4. 任意の関数が自由に付加できる微分システム
   2.5.3 Data-Directed Programming and Additivity

62

記号微分の拡張法

```
(define (deriv exp var)
   (cond ((number? exp) 0)
         ((variable? exp)
          (if (same-variable? exp var) 1 0) )
         ((sum? exp)
           (make-sum (deriv (addend exp) var)
                     (deriv (augend exp) var)) )
         ((product? exp)
          (make-sum
             (make-product (multiplier exp)
                           (deriv (multiplicand exp)
                                  var ))
             (make-product (deriv (multiplier exp)
                                  var )
                           (multiplicand exp) )))
         (ここに微分ルールを追加)
         (else
           (error "unknown expression type - DERIV"
                  exp ))))
```

63

差・商に対する微分ルールを追加

1. 差は (* -1 <exp>) で表現
   (- x y) (+ x (* -1 y))
2. 商は(/ <exp1> <exp2>) で表現

```
(define (make-division d1 d2)
   (cond ((=number? d1 0) 0)
         ((=number? d2 1) d1)
         ((and (number? d1) (number? d2))
          (/ d1 d2))
         (else (list '/ d1 d2)) ))

 (define (divident d) (cadr d))
 (define (divisor d) (caddr d))
 (define (division? x)
     (and (pair? x) (eq? (car x) '/)) )
```

64

商に対する微分ルールを追加

$$\frac{d}{dx}\frac{u}{v} = -\frac{u}{v^2}\frac{dv}{dx} + \frac{1}{v}\frac{du}{dx}$$

```
((division? exp)
 (make-sum
   (make-product
      (make-division
         (make-product
            (make-product -1
               (divident exp) )
            (deriv (divisor exp) var) )
         (make-product
             (divisor exp)
             (divisor exp)  )))
   (make-product
      (make-division 1 (divisor exp))
      (deriv (divident exp) x) )))
```

65

冪乗に対する表現法と基本手続き

冪乗は (** <base> <exponent>) で表現

```
(define (make-exponentiation b e)
   (cond ((=number? e 0) 1)
         ((=number? e 1) b)
         ((=number? b 1) 1)
         ((and (number? b) (number? e))
          (** b e) )
         (else (list '** b e)) ))

(define (base x) (cadr x))
(define (exponent x) (caddr x))
(define (exponentiation? x)
   (and (pair? x) (eq? (car x) '**)) )
```

66

冪乗に対する微分ルールを追加

$$\frac{d}{dx}b^e = eb^{e-1}\frac{db}{dx}$$

```
((exponentiation? exp)
 (make-product
   (make-product
      (exponent exp)
      (deriv (base exp) var) )
   (make-exponentiation
      (base exp)
      (make-sum (exponent exp) -1) )))
```

67

三角関数に対する微分ルールを追加

関数は (*<func>* *<args>*) で表現

```
((sin? exp)
 (make-product
    (make-function
       'cos
       (argument exp) )
    (deriv (argument exp) var) ))

(define (make-function func . args)
   (cons func args) )
```

$$\frac{d}{dx}\sin(u) = \cos(u)\frac{du}{dx}$$

68

記号微分の拡張（1）

1. 差、商に拡張
   `(deriv '(- x y) 'x)`
   `(deriv '(/ 3 x) 'x)`
2. 冪乗に拡張
   `(deriv '(** x 3) 'x)`
3. 2項演算子を多項演算子に拡張
   `(deriv '(+ (* 3 x) y (* x y)) 'x)`
   `(deriv '(* x y (+ x 3)) 'x)`

69

記号微分の拡張(2)

4. 2項演算子を多項演算子に拡張
   augend, multiplierの定義を変更するだけで
   `(deriv '(+ x (* x y) (** x 3)) 'x)`
   に対応できる。
5. 多項式の整理
   - 多項式を降冪あるいは昇冪の順に整列
   - 多項式を簡略化により整理

   2.5.3 記号代数(Symbolic Algebra)
6. 任意の関数が自由に付加できる微分システム

   2.5.3 Data-Directed Programming and Additivity

70

1月7日・本日のメニュー

1. 2.3 Symbolic Data
2. 2.3.2 Symbolic Differentiation
3. **2.3.3 Representing Sets**
4. **2.4 Multiple Representations for Abstract Data**
5. **2.4.1 Representations for Complex Numbers**

集合(set)の表現

- 自然数の集合を定義してみよう

1. {0, 1, 2, 3, …}
   外延的記法(extensional notation)
2. *S = {n|0, n+1 if n ∈ S}*
   内延的記法(intentional notation)

- 外延的記法での課題
  次の定義のどちらがよいか？

1. {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, … }
2. {0, 10, 20, 30, 2, 12, 22, 24, 4, 14, 24, … }

72

集合(set)の手続きと表現法

- 集合の手続き
  1. union-set　S∪T
  2. intersection-set　S∩T
  3. element-of-set?　e∈T
  4. adjoin-set　{e}∪S
- 集合の表現法の実装(implementation)
  1. 順序なし表現(*unordered list*)
     {30, 0, 20, 10, 22, 2, 12, 24, 34, ... }
     `(30 0 20 10 22 2 12 24 34 … )`
  2. 順序付き表現(*ordered list*)
     {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, ... }
     `(0 2 4 6 8 10 12 14 16 18 … )`

73

集合(set)のUnordered List表現

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)) ))

(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set) )))

(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (cons (car s1)
                    (union-set (cdr s1) s2) ))))
```

74

union-set の両者の違いは

```
(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (cons (car s1)
                    (union-set (cdr s1) s2) )
        )))
(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (union-set (cdr s1)
                    (cons (car s1) s2)) ))))
```

(union-set '(1 2 1) '(a b c))の結果は？

75

集合のunordered list表現(続)

```
(define (intersection-set s1 s2)
  (cond ((or (null? s1) (null? s2)) ())
        ((element-of-set? (car s1) s2)
         (cons (car s1)
           (intersection-set
               (cdr s1) s2 )))
        (else (intersection-set
                  (cdr s1) s2 ))))
```

76

集合手続きの計算量 （#set=n）

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)) ))
```

Θ(n)

```
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set) )))
```

Θ(n)

#s1=n
#s2=m

```
(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (cons (car s1)
                    (union-set (cdr s1) s2) ))))
```

Θ(mn)

77

intersection-setの計算量

```
(define (intersection-set s1 s2)
  (cond ((or (null? s1) (null? s2)) ())
        ((element-of-set? (car s1) s2)
         (cons (car s1)
           (intersection-set
              (cdr s1) s2 )))
        (else (intersection-set
                (cdr s1) s2 ))))
```

計算のオーダは #s1=m1, #s2=m2 とすると、

Θ(n^2) n=max{m1,m2} （m1*m2 のオーダ）

78

集合(set)のOrdered List表現

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-set? x (cdr set)) ))
```

Θ(n) 平均的には n/2

```
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((= x (car set)) set)
        ((< x (car set)) (cons x set))
        (else (cons (car set)
                    (adjoin-set x (cdr set))
              ))))
```

Θ(n) 平均的には n/2

79

集合(set)のOrdered List表現

```
(define (union-set s1 s2)
  (if (null? s1)
      s2
      (let ((x1 (car s1)) (x2 (car s2)))
         (cond ((= x1 x2)
                (cons x1
                      (union-set (cdr s1) (cdr s2))) )
               ((< x1 x2)
                (cons x1 (union-set (cdr s1) s2)) )
               (else
                 (cons x2
                       (union-set s1 (cdr s2))
                 )))))))
```

計算のオーダは #s1=m1, #s2=m2 とすると、

Θ(n) n=max{m1,m2}(m1+m2 のオーダ)

80

集合のunordered list表現(続)

```
(define (intersection-set s1 s2)
  (if (or (null? s1) (null? s2))
      ()
      (let ((x1 (car s1)) (x2 (car s2)))
        (cond ((= x1 x2)
               (cons x1
                 (intersection-set (cdr s1) (cdr s2)) )
              ((< x1 x2)
               (intersection-set (cdr s1) s2) )
              (else
               (intersection-set s1 (cdr s2)) )))))
```

計算のオーダは #s1=m', #s2=m とすると、

Θ(n) n=max{m1,m2} (m+nのオーダ)

81

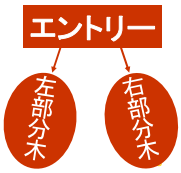
集合の二進木(binary tree)表現

- リスト構造(木)で集合を表現
- 設計方針
  - 順序付きリストのように制御してしないと, 木の高さをhとすると、Θ(h^2)の計算量がかかる
  - 左部分木のエントリーはノードのそれより大きくない
  - 右部分木のエントリーはノードのそれより大きい
- ノードの表現法
  - 次のリストでノードを表現
    (エントリー 左部分木 右部分木)

二進木(binary tree)表現の実装

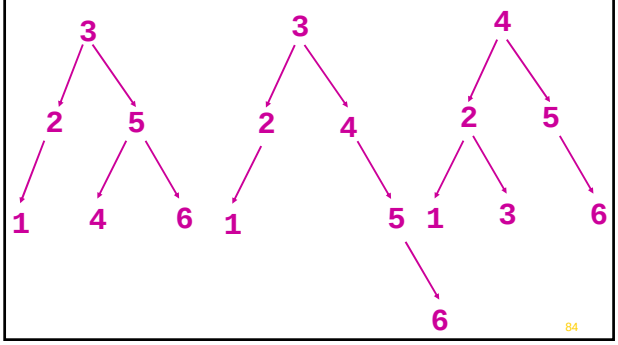
```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree)
   (caddr tree))

(define (make-tree entry left right)
   (list entry left right) )
```



二進木表現の曖昧性

集合{1, 2, 3, 4, 5, 6} の二進木表現



84

集合(set)のbinary tree表現

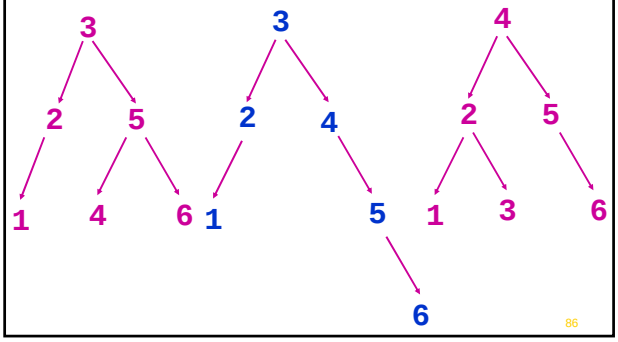
```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set))
         (element-of-set? x (left-branch set)) )
        (else
         (element-of-set? x (right-branch set)) )))

(define (adjoin-set x set)
  (cond ((null? set) (make-tree x () ()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                (adjoin-set x (left-branch set))
                (right-branch set) ))
        (else
          (make-tree (entry set)
                (left-branch set)
                (adjoin-set x (right-branch set)) ))))
```

85

adjoin-set の動き

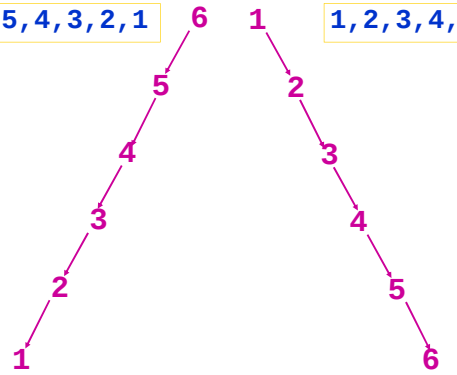
3,2,1,5,4,6 | 3,4,2,5,6,1 | 4,2,1,5,6,3



86

adjoin-set の動き

6,5,4,3,2,1 | 1,2,3,4,5,6



87

Tree⇒list 変換2種類

```
(define (tree->list-1 tree)
  (if (null? tree)
      ()
      (append (tree->list-1 (left-branch tree))
              (cons (entry tree)
                    (tree->list-1 (right-branch tree))))))

(define (tree->list-2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                      (cons (entry tree)
                            (copy-to-list (right-branch  tree)
                                          result-list )))))
  (copy-to-list tree '()) )
```

両者の違いは？

前順走査・間順走査・後順走査（第2回）

88

balanced binary tree表現

```
(define (list->tree elements)
   (car (partial-tree elements (length elements))) )

(define (partial-tree elts n)
  (if (= n 0)
      (cons () elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result (partial-tree elts left-size)))
           (let ((left-tree (car left-result))
                 (non-left-elts (cdr left-result))
                 (right-size (- n (+ left-size 1))) )
             (let ((this-entry (car non-left-elts))
                   (right-result (partial-tree
                                   (cdr non-left-elts)
                                   right-size )))
               (let ((right-tree (car right-result))
                     (remaining-elts (cdr right-result)) )
                 (cons (make-tree this-entry
                                  left-tree
                                  right-tree )
                       remaining-elts ))))))))
```

89

balanced binary tree表現(改)

```
(define (list->tree elements)
   (car (partial-tree elements (length elements))) )
(define (partial-tree elts n)
  (if (= n 0)
      (cons () elts)
      (let* ((left-size (quotient (- n 1) 2))
             (left-result (partial-tree elts left-size))
             (left-tree (car left-result))
             (non-left-elts (cdr left-result))
             (right-size (- n (+ left-size 1)))
             (this-entry (car non-left-elts))
             (right-result
                (partial-tree (cdr non-left-elts)
                              right-size )))
             (right-tree (car right-result))
             (remaining-elts (cdr right-result)) )
         (cons
            (make-tree this-entry left-tree right-tree)
            remaining-elts ))))
```

90

Sets & information retrieval

```
(define (lookup given-key set-of-records)
   (cond ((null? set-of-records) false)
         ((equal? given-key (key (car set-of-records)))
          (car  set-of-records) )
         (else (lookup given-key (cdr set-of-records)))
   ))
```

連想リスト(a-list, associative list)

((<属性> . <値のリスト>)
(<attribute> . <value-list>)
...)

```
(define (assoc given-key set-of-records)
   (cond ((null? set-of-records) false)
         ((equal? given-key (caar set-of-records))
          (cdar  set-of-records) )
         (else (lookup given-key (cdr set-of-records)))
   ))
```

91

簡単な情報検索

```
(define (lookup given-key set-of-records)
   (cond ((null? set-of-records) false)
         ((equal? given-key (key (car set-of-records)))
          (car  set-of-records) )
         (else (lookup given-key (cdr set-of-records)))))
(define population
 '((China 1285.0 660.5 624.5)
   (India 1025.1 528.5 496.6)
   (USA    285.9 141.0 144.9)
   (Indonesia 214.8 107.8 107.1)
   (Brazil 172.6 85.2 87.4)
   (Pakistan 145.0 74.5 70.5)
   (Russia   144.7 67.7 77.0)
   (Bangradesh 140.4 72.3 68.0)
   (Japan    127.1 62.2 65.0)
   (Nigeria  116.9 59.0 58.0)
   (Mexico   100.4 49.6 50.7) ))
(lookup 'Japan population)
(assoc 'Japan population) = (cdr (lookup 'Japan population))
```

連想リスト(a-list, associative list)

((<属性> . <値のリスト>)
(<attribute> . <value-list>)
...
)

key の順序

1. 数
   1. 昇順(increasing order, ascending order) <
   2. 降順(decreasing order, descending order) >
2. 辞書式順序(lexicographical order)
   1. (string=? "PIE" "pie")
   2. (string-ci=? "PIE" "pie")
   3. string<?, string<=?, …
   4. char=?, char-ci=?, char>?, char>=?, …
3. alphanumeric order

93

ソーティングの応用

1. 本1冊に出てくる単語の頻度を求めよ。
2. Unix の pipe で処理
   次の1行のコマンドでできる。
   tr '[\t,.;:]*' '\n' < file | 改行不可
   tr '[A-Z]' '[a-z]' | sort | 改行不可
   uniq –c | sort -r
3. www.gutenberg.org よりフルテキストを入手、
   Gulliver's Travel (Swift)
   the 2894, of 1844, and 1755, to 1557,
   i 1311, a 1177, in 984, my 768, was 625
   TAO (Lao-Tsu)
   the 675, and 373, to 345, of 335, is 290
   it 225, not 164, in 154, he 136, a 136

94

宿題:1月12日正午締切

宿題は、次の計3問:

Ex.2.57, 2.59, 2.60

Bugを最初に見つけたのは

- Grace Murray Hopper (Dec. 9, 1906 – Jan. 1, 1992)
- 1st women to receive a Ph.D in mathematics.
- Sep. 9, 1945. Mark II の Relay #70, Panel F で発見。
- bug はそれ以前から使用されていた。
- コンパイラ開発にも従事。
- NavyでCOBOL言語の検証ソフト開発

107

A Bug (moth) in the Mark-II

Photo #: NH 96566-KN (Color)

U.S. Naval Historical Center Photograph.

実験ノートをつけること。ルーズリーフはだめ

10大Bug（by Wired News）

1. **1962年7月22日　火星探査機『マリナー1号』**：マリナー1号は打ち上げ時に予定のコースを外れたが、これは飛行ソフトウェアのバグが原因だった。地上の管制センターは大西洋上でロケットを破壊した。事後 調査により、鉛筆で紙に書かれた数式（$\bar{F}_{n}$）をコンピューターのコードに置き換えるときにミス（¯を忘れた）が起き、これが原因でコンピューターが飛行コースの計算を誤ったこと が判明した。
2. **1982年　旧ソ連のガス・パイプライン**：シベリアを横断するガス・パイプラインの管理に旧ソ連が購入したカナダ製のコンピューターシステムに、米中央情報局(CIA)のスパイがバグを仕掛けたことがあるという。旧ソ連は当時、米国の機密技術を密かに購入しようと ―または盗み出そうと― しており、このシステムを入手したのもその一環だった。だが、計画を察知したCIAはこれを逆手にとり、旧ソ連の検査は問題なく通過するが、いったん運転に入ると機能しなくなるように仕組んだとされる。この結果起きたパイプライン事故は、核爆発以外では地球の歴史でも最大規模の爆発だったという。
3. **1985～1987年　セラック25**：複数の医療施設で放射線治療装置が誤作動し、過大な放射線を浴びた患者に死傷者が出た。セラック25は2 種類の放射線 ―低エネルギーの電子ビーム(ベータ粒子)とX線― を照射できるよう、既存の設計に「改良」を加えた治療装置だった。セラック25では電 子銃と患者の間に置かれた金属製のターゲットに高エネルギーの電子を打ち込み、X線を発生させていた。セラック25のもう1つの「改良」点は、旧モデル『セラック20』の電気機械式の安全保護装置をソフトウェア制御に置き換えたことだった。ソフトウェアの方が信頼性が高いとの考えに基づく判断だった。
   しかし、技術者たちも知らなかった事実があった ―セラック20およびセラック25に使われたOSは、正式な訓練も受けていないプログラマーが1人で作成したもので、バグが非常にわかりにくい構成になっていたのだ。「競合状態」と呼ばれる判明しにくいバグが原因で、操作コマンドを素早く打ち込んだ場合、セラック25ではX線用の金属製ターゲットをきちんと配置しないまま高エネルギーの放射線を照射する設定が可能になっていた。これにより少なくとも5人が死亡し、他にも重傷者が出た。

110

10大Bug（by Wired News）

4. **1988年――バークレー版UNIX(BSD)のフィンガーデーモンによるバッファー・オーバーフロー**：最初のインターネットワームとなった通称『モーリス・ワーム』は、バッファー・オーバーフローを悪用し、1日足らずで2000台から6000台のコンピューターに感染した。原因となったのは、標準入出力ライブラリー・ルーチン内の「gets()」という関数の コードだ。「gets()」関数はネットワーク越しにテキストを1行取得するように設計された。しかし、残念ながら「gets()」関数は入力を制限する ようには作られていない。そのため、あまりにも大きな入力があった場合には、接続可能なあらゆるマシンをワームが占拠する元凶になった。
   プログラマーは「gets()」関数を使用コードから排除することで問題に対処しているが、C言語の標準入出力ライブラリーからこれを削除することは拒否しており、この関数は現在も存在している。
5. **1988～1996年――『ケルベロス』の乱数生成アルゴリズム**：ケルベロスは暗号を使ったセキュリティーシステムだが、乱数発生器に与えるシード(種)が適切でなく、真にランダムな乱数が生成されていなかった。
   その結果、ケルベロスによる認証を用いているコンピューターについて、非常に簡単な方法で侵入可能な状態が8年間にわたって続いた。このバグが実際に悪用されたかどうかは、今も定かではない。
6. **1990年1月15日―― 米AT&T社のネットワーク停止**：米AT& T社の長距離電話用交換機『4ESS』を制御する最新版のソフトウェアにバグが入りこんだ。このため、4ESSは隣接するマシンの1つから、ある特定の メッセージを受け取るとクラッシュするようになってしまった――そしてそのメッセージとは、クラッシュした交換機が復帰した際に、隣接する交換機に送信するものだった。
   ある日、ニューヨークの交換機がクラッシュし再起動した。するとそれが原因で隣接する複数の交換機がクラッシュし、これらの交換機が再起動すると隣接する複数の交換機がさらにクラッシュし、この現象が延々と続いた。しばらくすると、114台の交換機が6秒ごとにクラッ シュと再起動を繰り返すようになった。この影響でおよそ6万人の人々が9時間にわたって長距離通話サービスを利用できなくなった。修復のため、技術者たち は1つ前のソフトウェアをロードした。

111

10大Bug（by Wired News）

7. **1993年――インテル社製『Pentium』(ペンティアム)による浮動小数点数の除算ミス**：米インテル社が大々的に売り出したPentiumチップが、特定の浮動小数点数の除算で誤りを引き起こした。たとえば、4195835.0／3145727.0を計算させると、正しい答えの1.33382ではなく1.33374となる。0.006%の違いだ。
   実際にこの問題の影響を受けるユーザーはごくわずかだったが、ユーザーへの対応から、同社にとって悪夢のような事態につながった。概算で300万～500 万個の欠陥チップが流通していた状況で、インテル社は当初、高精度のチップが必要だと証明できる顧客のみをPentiumチップの交換対象とした。しかし、最終的にインテル社は態度を改め、不満を訴えるすべてのユーザーのチップ交換に応じた。この欠陥は結局、インテル社に約4億7500万ドルの損害を与 えた。
8. **1995年／1996年――『Ping of Death』：[ピング・オブ・デス、不正なピングパケットによる攻撃]** 分割送信されたIPパケットの再構成を行なうコードのチェックとエラー処理が不十分だったため、インターネット上の好きな場所から不正な形式のピングパケットを飛ばすことで、さまざまなオペレーティング・システム(OS)をクラッシュさせることができた。 影響が最も顕著に現れたのはウィンドウズ搭載マシンで、この種のパケットを受け取ると、「死のブルー・スクリーン」と呼ばれる青い画面を表示して動作が停止してしまう。しかしこのバグを利用した攻撃は、ウィンドウズのみならず、マッキントッシュやUNIXを使ったシステムにも多くの被害をもたらした。
9. **1996年6月4日――『アリアン5』フライト501**：欧州宇宙機関の開発したロケット、アリアン5には、『アリアン4』で使われていたコードが再利用されていた。しかし、アリアン5ではより強力なロケットエン ジンを採用したことが引き金となり、ロケットに搭載された飛行コンピューター内の計算ルーチンにあったバグが問題を起こした。エラーは64ビットの浮動小 数点数を16ビットの符号付き整数に変換するコードの中で起こった。アリアン5では加速度が大きいため、64ビット浮動小数点で表現される数がアリアン4 のときよりも大きくなってオーバーフローが起こり、最終的には飛行コンピューターがクラッシュしてしまった。
   フライト501では、最初にバックアップ・コンピューターがクラッシュし、それから0.05秒後にメイン・コンピューターがクラッシュした。その結果、エンジンの出力が過剰になり、ロケットは打ち上げ40秒後に空中分解してしまった。

112

10大Bug（by Wired News）

10.2000年11月――パナマ国立ガン研究所：米マルチデータ・システムズ・インターナショナル社(本社ミズーリ州)が製作した治療計画作成用ソフトウェアを使っていたパナマの国立ガン研究所で、放射線治療で照射する放射線量の計算を誤る一連の事故が起きた。

マルチデータ社のソフトウェアでは、健康な組織を放射線から守るための「ブロック」と呼ばれる金属製のシールドの配置を、コンピューターの画面上に描いて決めるようになっていた。しかし、同社のソフトウェアではシールドが4個しか使えなかったにもかかわらず、パナマ人の技師たちはこれを5個使いたいと考え た。

技師たちは、真ん中に穴を持つ1個の大きなシールドとして、5個のシールドをまとめて表示させれば、ソフトウェアをだますことができることを発見した。だが、そうした配置にした場合、穴の描き方によってこのソフトウェアが返す計算結果が違ってくることにはまったく気づいていなかった。ある方向に向けて描くと正しい照射量が計算されるが、違う方向に描くと必要な照射量の最大2倍の量を推奨してきたのだ。

少なくとも8人の患者が死亡し、さらに20人が過剰照射によって深刻な健康被害を受けたとみられている。技師たちは、コンピュータによる計算結果を手作業で再チェックする法的義務を負っていたため、殺人罪で起訴されることになった。

11.2003年8月14日――米国北東部の大停電：3分間に21の発電所が脱落し、オハイオ州の系統で電圧異常が起こった.

12.2006年12月1日――首都圏のJR東日本，系統で自動改札システム停止：首都圏3分の1以上の駅（184/511）で改札ゲート閉鎖．12月1日に日付変更以降，日本信号の自動改札機でのSuica 定期券の状態チェックにバグ．

13.2007年5月15日――NTT東フレッツサービス大規模ダウン：1台のルータがダウンし，待機系に変更したところ，ルーティング情報（15K）が4000台のルータで書き換えられようとし，その書き換え負荷に耐えきれず．

113