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
2.3.2 記号微分 2.3.3 集合 2.4 表現

随意課題8
12月13日(木) 10:00～17:10
第10回情報学シンポジウム
時計台ホール

奥 乃 博 okuno@i.kyoto-u.ac.jp
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/07/IntroAlgDs/

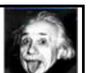
12月11日・本日のメニュー

データによる抽象化

- 図形言語の補足
  - `transform-painter`
- 2.3.2 Symbolic Differentiation
- 2.3.3 Representing Sets
- 2.4 Multiple Representations for Abstract Data
- 2.4.1 Representations for Complex Numbers

---

transform-painter



```
(define (half-vert painter)
  (transform-painter
     painter
     (make-vect 0.0 0.0)
     (make-vect 1.0 0.0)
     (make-vect 0.0 0.5)) )
```

```
(define (half-horiz painter)
  (transform-painter
     painter
     (make-vect 0.0 0.0)
     (make-vect 0.5 0.0)
     (make-vect 0.0 1.0)) )
```

```
(define (flip-vert painter)
  (transform-painter
     painter
     (make-vect 0.0 1.0
     (make-vect 1.0 1.0)
     (make-vect 0.0 0.0)) )
```

```
(define (shrink-to-upper-right painter)
   (transform-painter
     painter
      (make-vect 0.5 0.5)
      (make-vect 1.0 0.5)
      (make-vect 0.5 1.0)) )
```

```
(define (rotate90 painter)
  (transform-painter painter (make-vect 1.0 0.0)
                             (make-vect 1.0 1.0)
                             (make-vect 0.0 0.0) ))
```

12月11日・本日のメニュー

データによる抽象化

- 図形言語の補足
  - `transform-painter`
- **2.3.2 Symbolic Differentiation**
- 2.3.3 Representing Sets
- 2.4 Multiple Representations for Abstract Data
- 2.4.1 Representations for Complex Numbers

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
- $\frac{dx}{dx} =$
- $\frac{d(u+v)}{dx} =$
- $\frac{d(uv)}{dx} =$

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)`

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 | …

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	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 '+ 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) '*)) )
```

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

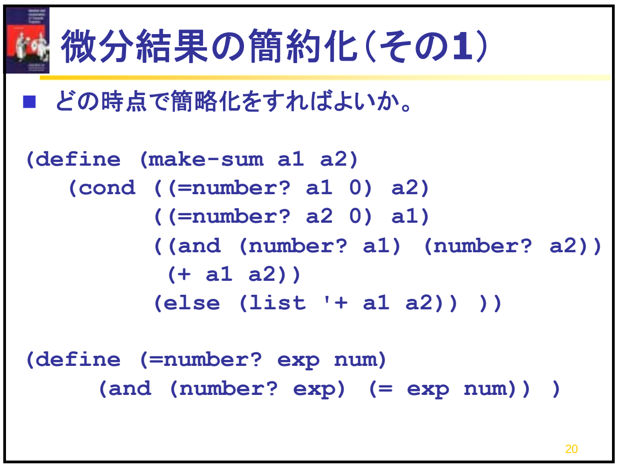
```
(define (deriv exp var)
   (cond ((number? exp) 0)
         ((variable? exp)
          (if (same-variable? exp var) 1 0) )
         ((sum? exp)
           (make-sum (deriv (addend exp) var)
                     (deriv (augend exp) var)) )
         ((product? exp)
          (make-sum
             (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

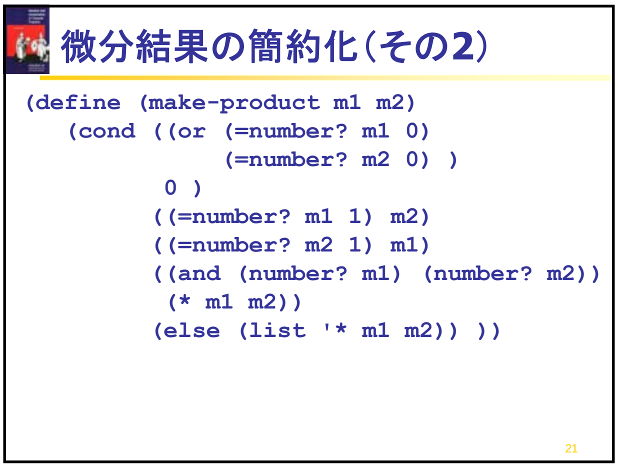
簡略化を忘れている。



微分結果の簡約化（その1）

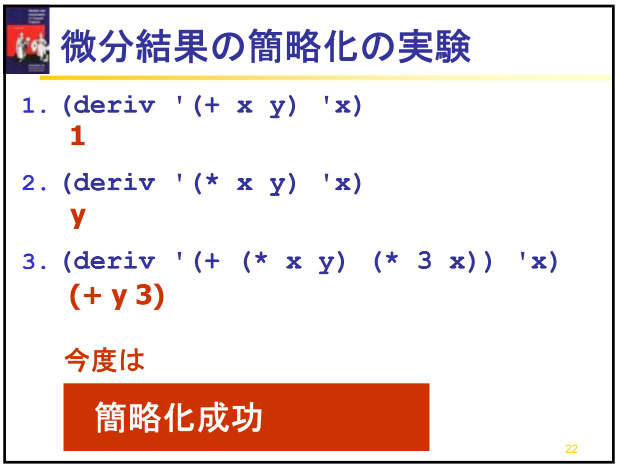
- どの時点で簡略化をすればよいか。

```
(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)) )
```



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)) ))
```

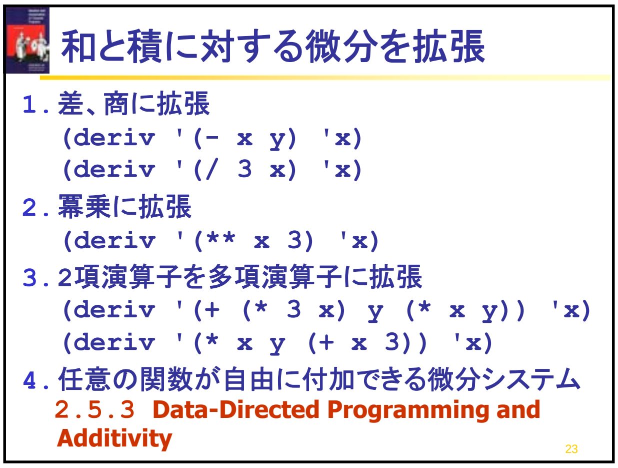


微分結果の簡略化の実験

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



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

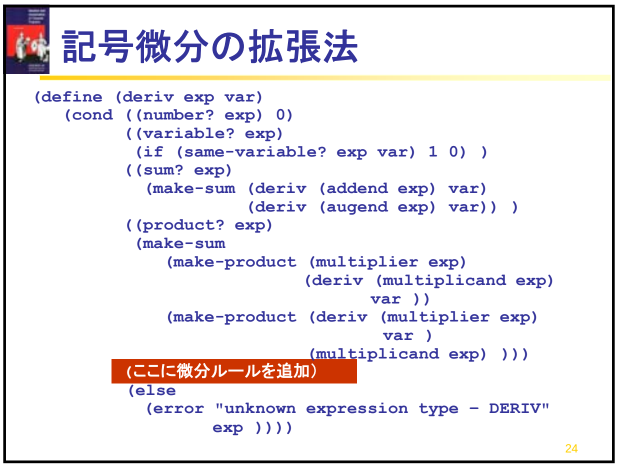
```
(deriv '(** x 3) 'x)
```

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)
```

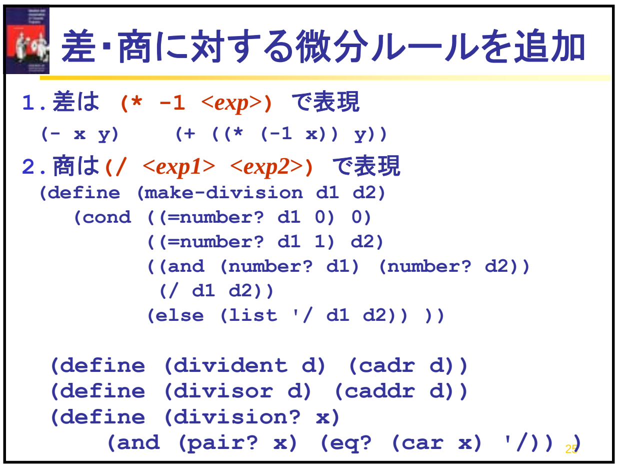
4．任意の関数が自由に付加できる微分システム

2.5.3 Data-Directed Programming and Additivity



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 ))))
```



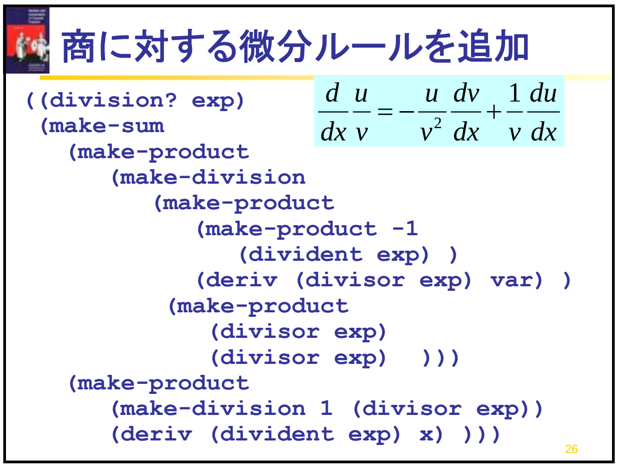
差・商に対する微分ルールを追加

1．差は (* -1 *<exp>*) で表現

```
(- x y)     (+ ((* (-1 x)) y))
```

2．商は (/ *<exp1>* *<exp2>*) で表現

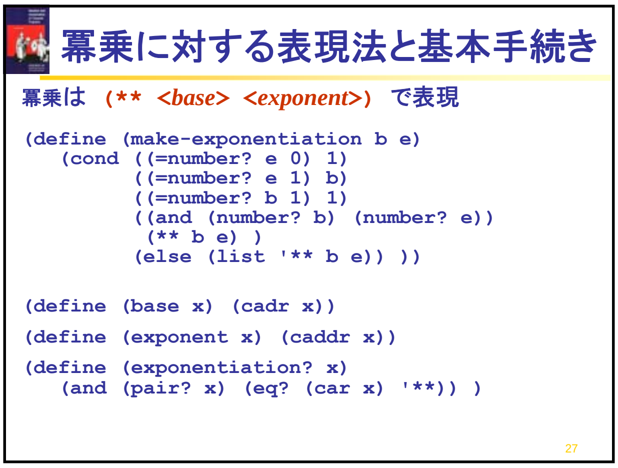
```
(define (make-division d1 d2)
   (cond ((=number? d1 0) 0)
         ((=number? d1 1) d2)
         ((and (number? d1) (number? d2))
          (/ d1 d2))
         (else (list '/ d1 d2)) ))

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



商に対する微分ルールを追加

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

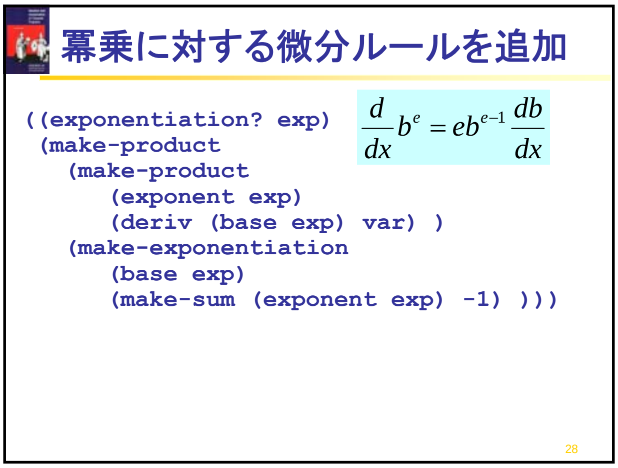
```
((division? exp)
 (make-sum
   (make-product
      (make-division
         (make-product
            (make-product -1
               (divident exp) )
            (deriv (divisor exp) var) )
          (make-product
             (divisor exp)
             (divisor exp)  )))
   (make-product
      (make-division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



冪乗に対する表現法と基本手続き

冪乗は (** *<base>* *<exponent>*) **で表現**

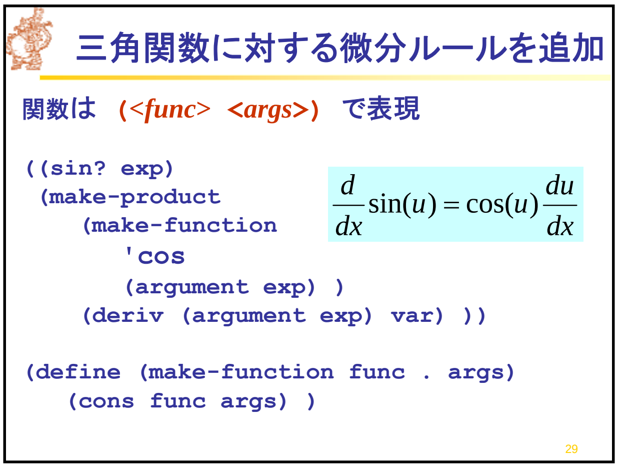
```
(define (make-exponentiation b e)
   (cond ((=number? e 0) 1)
         ((=number? e 1) b)
         ((=number? b 1) 1)
         ((and (number? b) (number? e))
          (** b e) )
         (else (list '** b e)) ))

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



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) )))
```

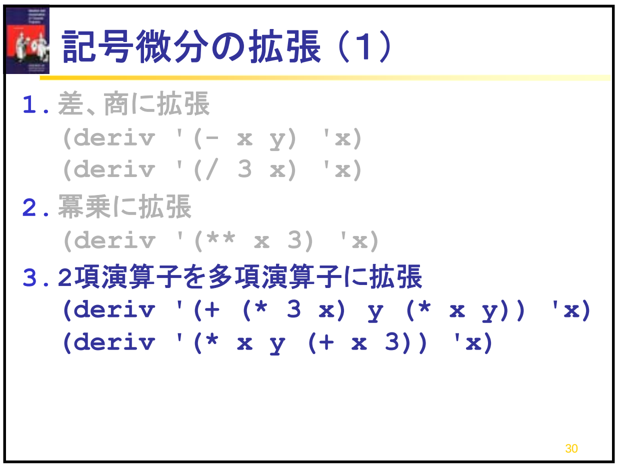


三角関数に対する微分ルールを追加

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) )
```

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



記号微分の拡張（1）

1．差、商に拡張

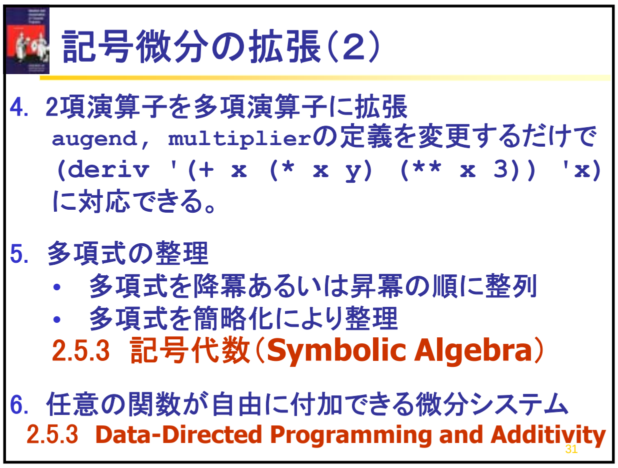
```
(deriv '(- x y) 'x)
(deriv '(/ 3 x) 'x)
```

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)
```



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

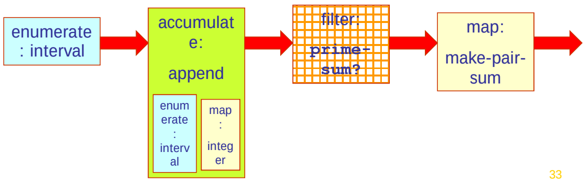
   2.5.3 **記号代数(Symbolic Algebra)**

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**

写像の入れ子（nesting of mapping）

1≤j<i≤n なる異なる正の整数 i,j に対して、i+jが素数となるものをすべて求める
n=6のとき

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
i+j	3	5	5	7	7	7	11



list of pairs of integers の作り方

```
(accumulate
   append
   nil
   (map
     (lambda (i)
       (map
         (lambda (j) (list i j))
         (enumerate-interval
                 1 (- i 1) )))
     (enumerate-interval 1 n) ))
```

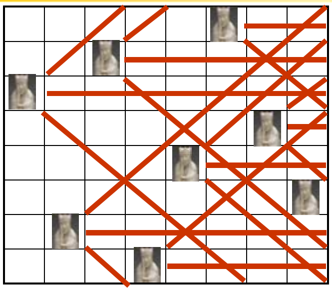
この呼び出しパターンを手続きとして定義

```
(define (flatmap proc seq)
  (accumulate append nil (map proc seq)) )
```

list of pairs of integers の作り方

```
(define (prime-sum? pair)
   (prime? (+ (car pair) (cadr pair)) )

(define (make-pair-sum pair)
   (list (car pair) (cadr pair)
         (+ (car pair) (cadr pair)) ))

(define (prime-sum-pairs n)
   (map make-pair-sum
        (filter prime-sum?
          (flatmap
             (lambda (i)
                (map (lambda (j) (list i j))
                     (enumerate-interval
                                1 (- i 1) )))
             (enumerate-interval 1 n) ))))
```

n-queens n人の女王の問題



8-queens puzzle

変種：すべての盤面をカバーする最小の女王の数は

- 女王は将棋の飛車角行
- お互いに取り合わないように配置

n-queensから呼ばれる手続き

```
(define (permutation s)
   (if (null? s)
       (list nil)
       (flatmap (lambda (x)
                   (map (lambda (p) (cons x p))
                        (permutation (remove x s)) ))
                s )))
(define (remove item sequence)
   (filter (lambda (x) (not (= x item))) sequence) )
(define (safe? k positions)
   (null?
     (filter
       (lambda (x)
          (not (or (= (cadr k) (cadr x))
                   (= (+ (car k) (cadr k))
                      (+ (car x) (cadr x)) )
                   (= (- (car k) (cadr k))
                      (- (car x) (cadr x)) ))))
       positions )))
(define empty-board '())
(define (adjoin-position new-row k rest-of-queens)
    (cons new-row rest-of-queens) )
```

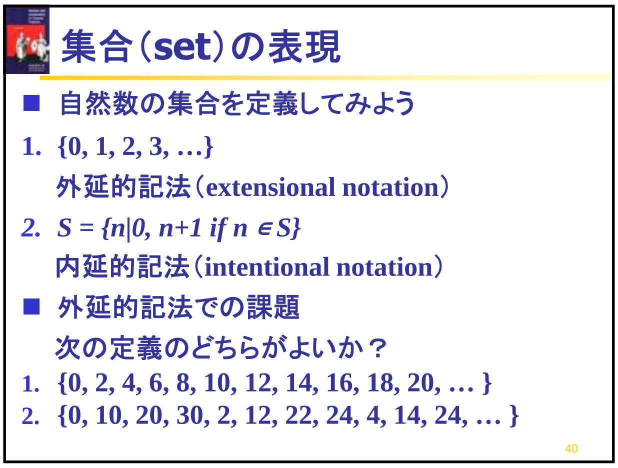
n-queens

```
(define (queens n)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter
          (lambda (positions)
            (safe? k positions) )
          (flatmap
            (lambda (rest-of-q)
              (map (lambda (new-row)
                      (adjoin-position new-row
                              k rest-of-q ))
                   (enumerate-interval 1 n) ))
            (queens-cols (- k 1)) ))))
  (queens-cols board-size) )
```

12月11日・本日のメニュー

データによる抽象化

- 図形言語の補足
  - `transform-painter`
- 2.3.2 Symbolic Differentiation
- **2.3.3 Representing Sets**
- 2.4 Multiple Representations for Abstract Data
- 2.4.1 Representations for Complex Numbers



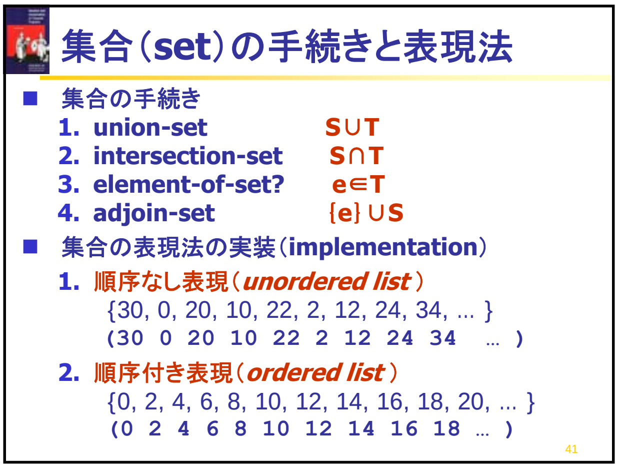
集合(set)の表現

- 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. $S = \{n|0, n+1 \text{ if } n \in S\}$
   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, … }



集合(set)の手続きと表現法

- 集合の手続き
  1. **union-set** **$S \cup T$**
  2. **intersection-set** **$S \cap T$**
  3. **element-of-set?** **$e \in T$**
  4. **adjoin-set** **$\{e\} \cup S$**
- 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 … )`

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) ))))
```

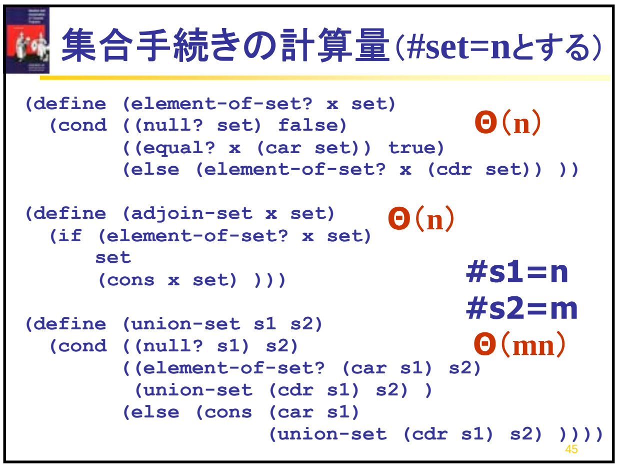
union-set の両者の違いは

```
(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？

集合(set)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 ))))
```



集合手続きの計算量(#set=nとする)

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

Θ(n)

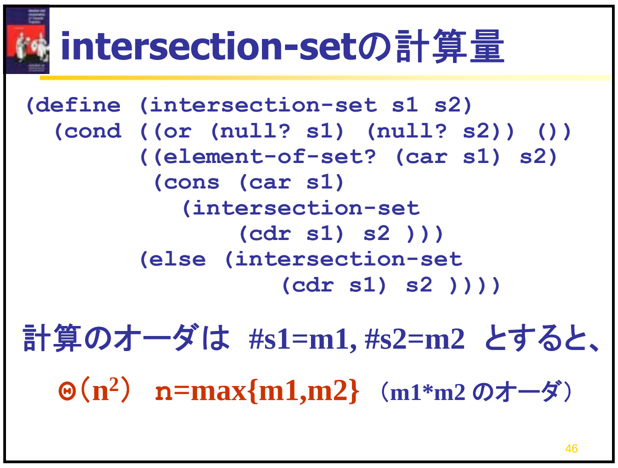
```
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set) )))
```

Θ(n)

#s1=n
#s2=m

Θ(mn)

```
(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) ))))
```

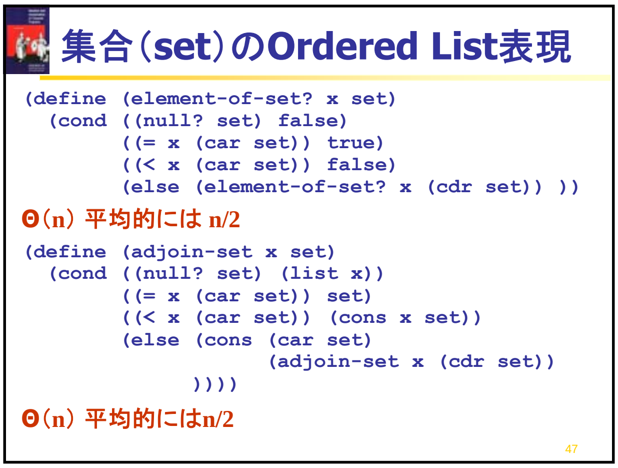


intersection-setの計算量

```
(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、

$\Theta(n^2)$ n=max{m1,m2} （m1*m2 のオーダ）



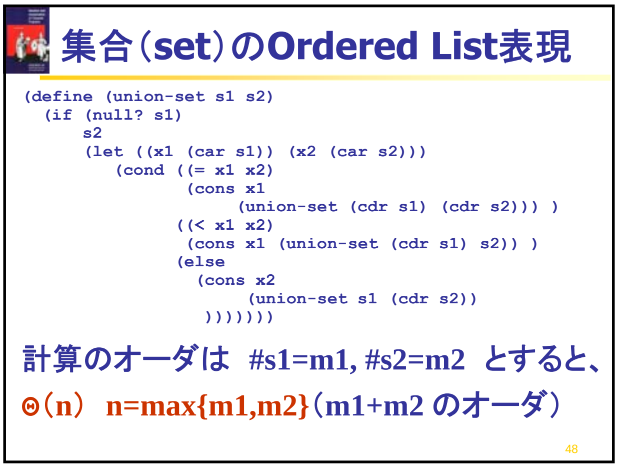
集合(set)のOrdered List表現

```
(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

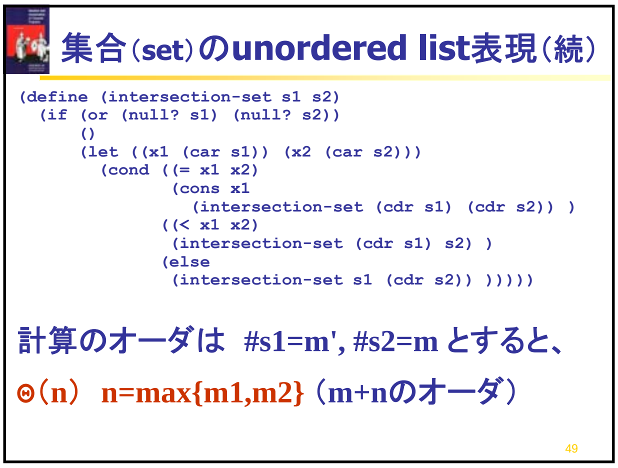


集合（set）のOrdered List表現

```
(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）

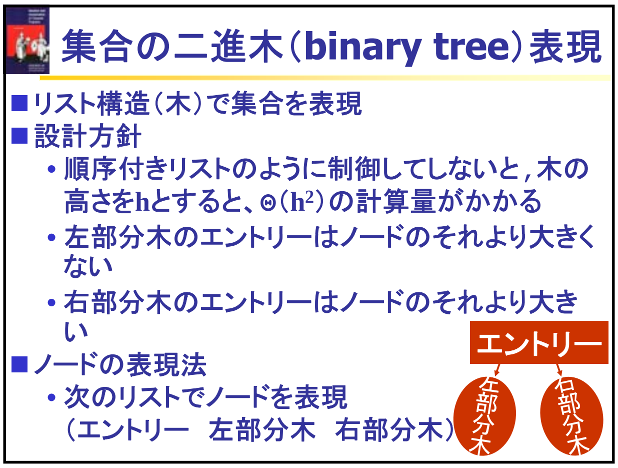


集合（set）のunordered list表現（続）

```
(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）

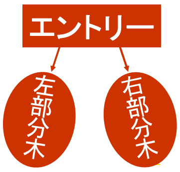


集合の二進木（binary tree）表現

- 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、$\Theta(h^2)$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）

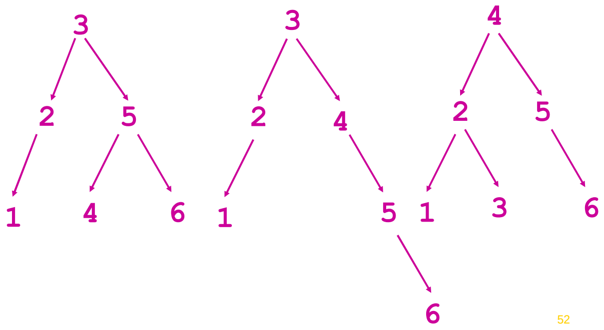
二進木（binary tree）表現の実装

```
(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



集合（set）のbinary tree表現

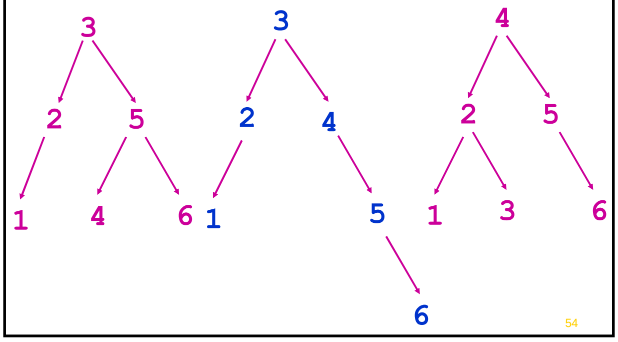
```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set))
         (element-of-set? x (left-branch set)) )
        (else
         (element-of-set? x (right-branch set)) )))

(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)) ))))
```

adjoin-set の動き

3,2,1,5,4,6

3,4,2,5,6,1

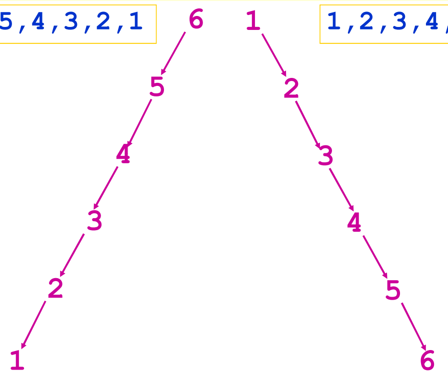
4,2,1,5,6,3



adjoin-set の動き

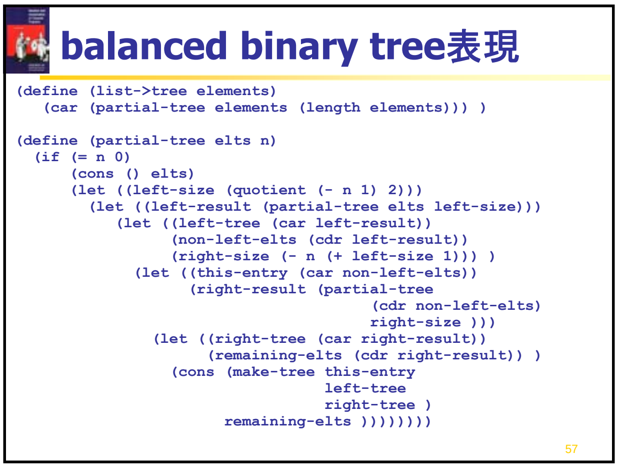
6,5,4,3,2,1

1,2,3,4,5,6



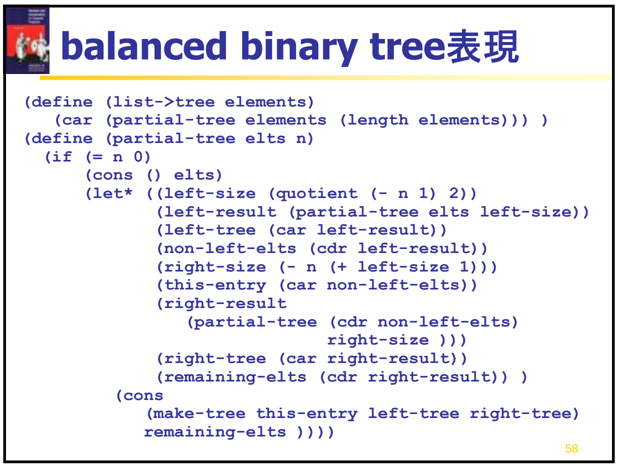
balanced binary tree表現

```
(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 ))))))
```



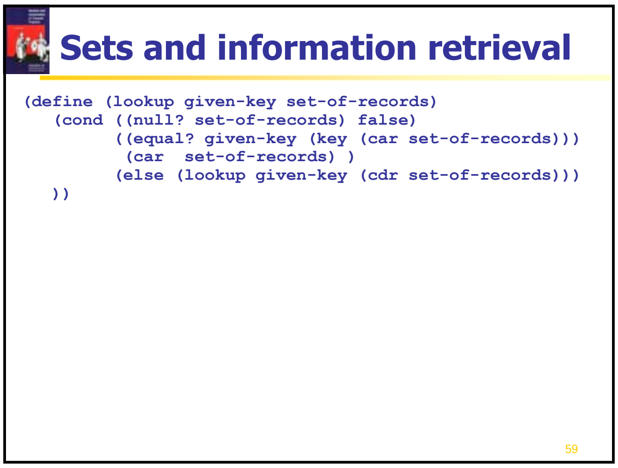
balanced binary tree表現

```
(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 ))))))))
```



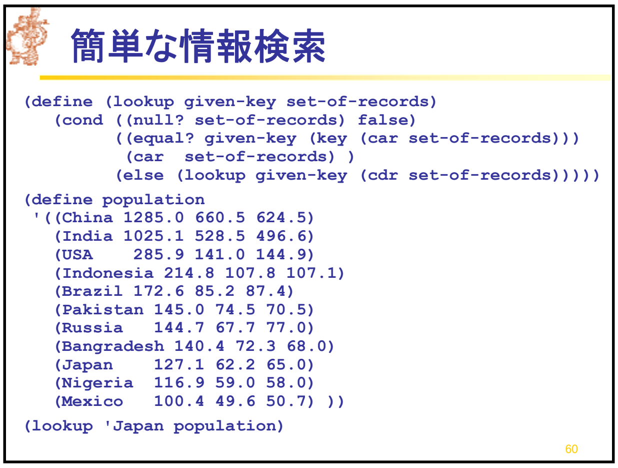
balanced binary tree表現

```
(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))
             (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 ))))
```



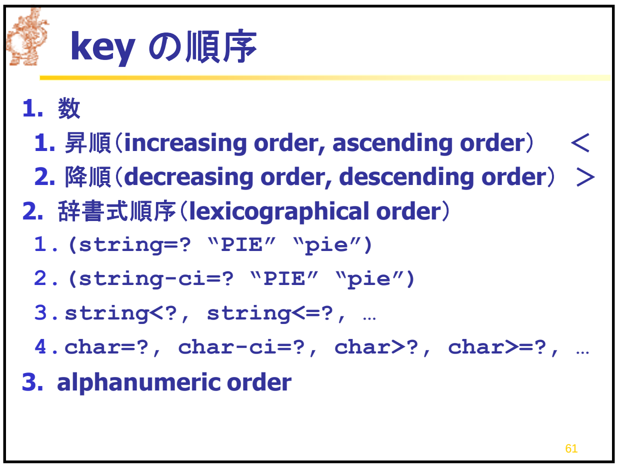
Sets and information retrieval

```
(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)))
   ))
```



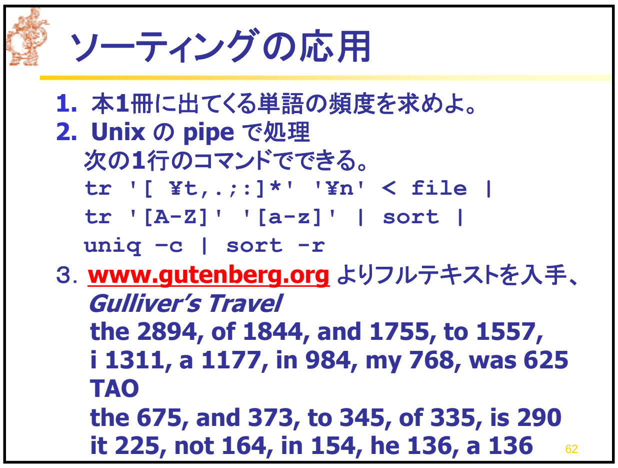
簡単な情報検索

```
(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)
```



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

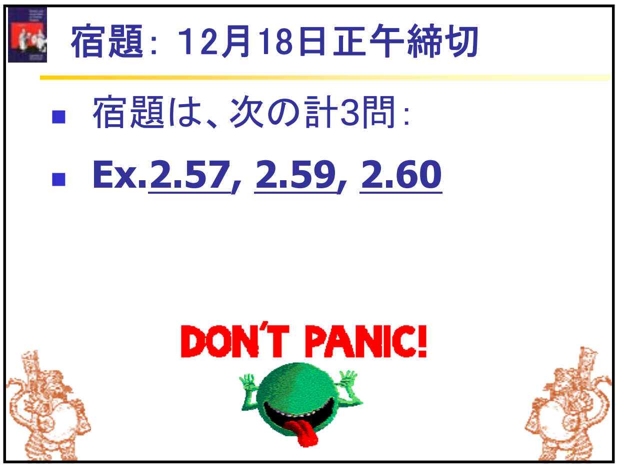


ソーティングの応用

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 |
tr '[A-Z]' '[a-z]' | sort |
uniq -c | sort -r
```

3. www.gutenberg.org よりフルテキストを入手、
   Gulliver's Travel
   the 2894, of 1844, and 1755, to 1557,
   i 1311, a 1177, in 984, my 768, was 625
   TAO
   the 675, and 373, to 345, of 335, is 290
   it 225, not 164, in 154, he 136, a 136



宿題：12月18日正午締切

- 宿題は、次の計3問：
- **Ex.2.57, 2.59, 2.60**

DON'T PANIC!