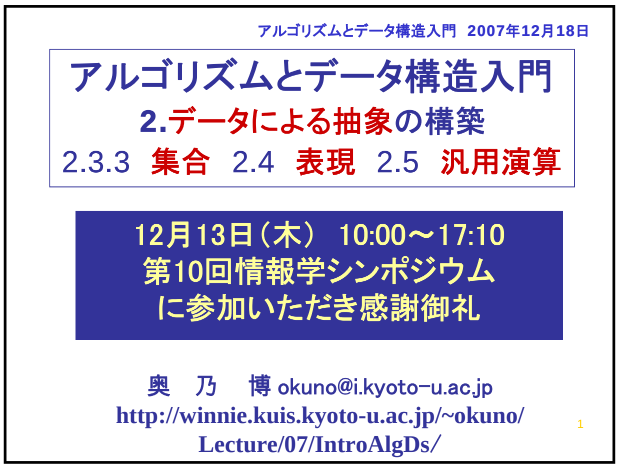


アルゴリズムとデータ構造入門

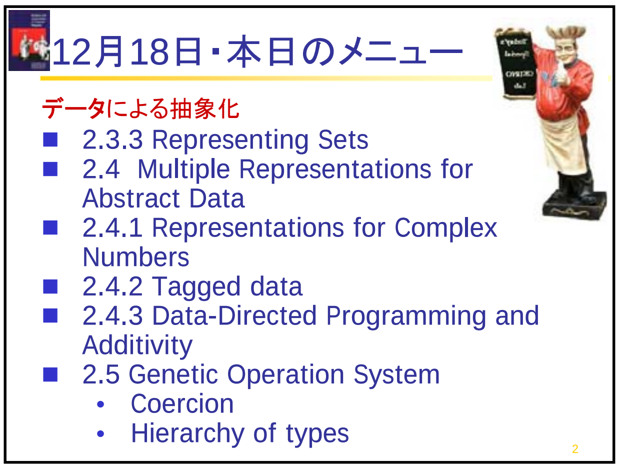
2.データによる抽象の構築

2.3.3 集合 2.4 表現 2.5 汎用演算

12月13日（木） 10:00～17:10
第10回情報学シンポジウム
に参加いただき感謝御礼

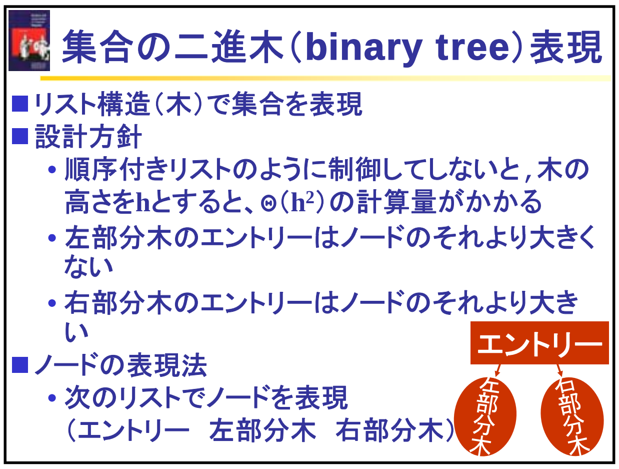
奥 乃 博 okuno@i.kyoto-u.ac.jp
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/07/IntroAlgDs/

1

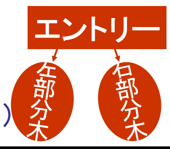

12月18日・本日のメニュー

データによる抽象化
- 2.3.3 Representing Sets
- 2.4 Multiple Representations for Abstract Data
- 2.4.1 Representations for Complex Numbers
- 2.4.2 Tagged data
- 2.4.3 Data-Directed Programming and Additivity
- 2.5 Genetic Operation System
  - Coercion
  - Hierarchy of types

2


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）

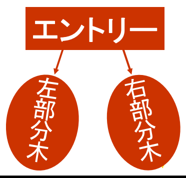


二進木(binary tree)表現の実装

```
(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) )
```

- ノードの表現法
  (エントリー 左部分木 右部分木)
- 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



集合(set)のbinary tree表現

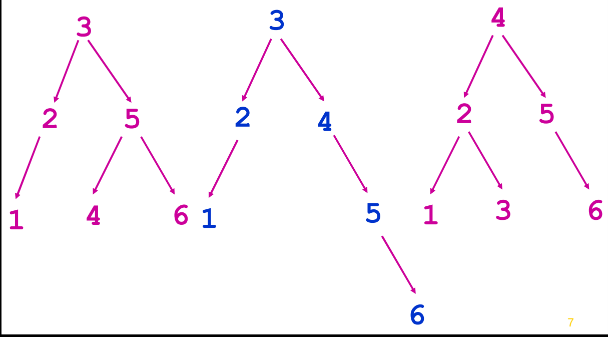
```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set))
         (element-of-set? x (left-branch set)) )
        (else
         (element-of-set? x (right-branch set)) )))

(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)) ))))
```

6

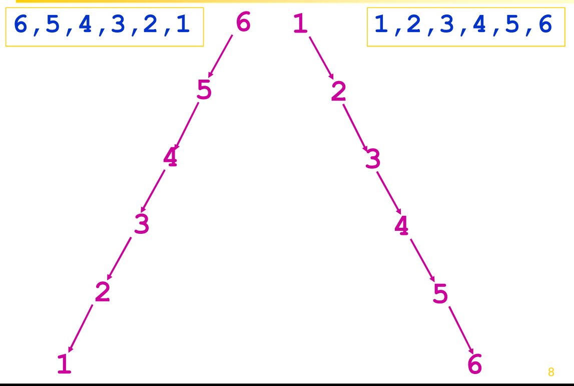
adjoin-set の動き

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



7

adjoin-set　の動き



8

Treeからlistへの変換法2種

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

両者の違いは？
　前順走査・間順走査・後順走査（第2回）

9

balanced binary tree表現

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

(define (partial-tree elts n)  ; 最初のn個の要素の平衡木作成
  (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 ))))))))
```

10

balanced binary tree表現(改)

```
(define (list->tree elements)
   (car (partial-tree elements (length elements))) )
(define (partial-tree elts n) ; 最初のn個の要素の平衡木作成
  (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 ))))
```

11

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)))
   ))
```

12

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)
```

13

簡単な情報検索 by 連想リスト

```
(define population
  '((China 1285.0 660.5 624.5)   これは(鍵 . データ)のリスト
    (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) ))

(assoc 'Japan population) ⇒  (127.1 62.2 65.0)

(define (assoc item a-list)
   (cond ((null a-list) ())
         ((eq item (caar a-list)) (cadr a-list))
         (else (assoc item (cdr a-list))) ))
```

14

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

15

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

16

12月18日・本日のメニュー

データによる抽象化

- 2.3.3 Representing Sets
- **2.4 Multiple Representations for Abstract Data**
- **2.4.1 Representations for Complex Numbers**
- 2.4.2 Tagged data
- 2.4.3 Data-Directed Programming and Additivity
- 2.5 Genetic Operation System
  - Coercion
  - Hierarchy of types

17

複素数システムのデータ抽象化の壁

複素数を使ったプログラム

汎用演算

プログラム領域での複素数

add-complex,sub-complex,mul等

複素数演算パッケージ

情報隠蔽

real-part imag-part magnitude angle

直交座標表現（Rectangular representation）

極座標表現（Polar representation）

cons car cdr

リスト構造と基本マシン算術

18

複素数の演算

1. 虚数（imaginary part）

$$z = x + iy \qquad i^2 = -1$$

2. 加算（addition）

$$\mathrm{Real-part}(z_1 + z_2) = \mathrm{Real-part}(z_1) + \mathrm{Real-part}(z_2)$$
$$\mathrm{Imaginary-part}(z_1 + z_2) = \mathrm{Imaginary-part}(z_1) + \mathrm{Imaginary-part}(z_2)$$

3. 乗算（multiplication）

$$\mathrm{Re}(z_1 \cdot z_2) = \mathrm{Re}(z_1)\cdot\mathrm{Re}(z_2) - \mathrm{Im}(z_1)\cdot\mathrm{Im}(z_2)$$
$$\mathrm{Im}(z_1 \cdot z_2) = \mathrm{Re}(z_1)\cdot\mathrm{Im}(z_2) + \mathrm{Im}(z_1)\cdot\mathrm{Re}(z_2)$$

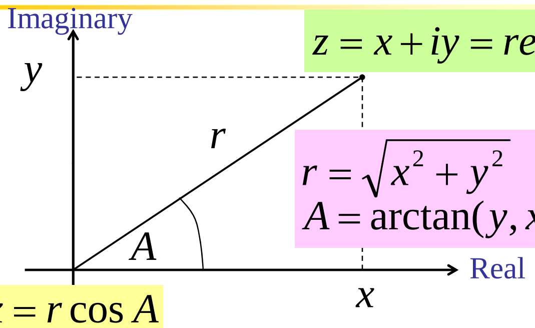
$$\mathrm{Magnitude}(z_1 \cdot z_2) = \mathrm{Magnitude}(z_1)\cdot\mathrm{Magnitude}(z_2)$$
$$\mathrm{Angle}(z_1 \cdot z_2) = \mathrm{Angle}(z_1) + \mathrm{Angle}(z_2)$$

複素数の四則演算

$z = x + iy = re^{iA}$

```
(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)) ))
```

複素数の表現法



$z = x + iy = re^{iA}$

$r = \sqrt{x^2 + y^2}$
$A = \arctan(y, x)$

$x = r\cos A$
$y = r\sin A$

複素数の表現法の実装

$z = x + iy = re^{iA}$

```
(make-from-real-imag
   (real-part z) (imag-part z) )
(make-from-mag-ang
   (magnitude z) (angle z) )
```

$x = r\cos A$
$y = r\sin A$

$r = \sqrt{x^2 + y^2}$
$A = \arctan(y, x)$

複素数の表現法

$z = x + iy = re^{iA}$

- 構築子(constructors)

```
(define (make-from-real-imag x y)
  (cons x y) )
(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (sin a))) )
```

- 選択子(selectors)

```
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (magnitude z)
  (sqrt (+ (square (real-part z)
           (square (imag-part z)) )))
(define (angle z)
  (atan (imag-part z) (real-part z)))
```

23

複素数の表現法(続)

$z = x + iy = re^{iA}$

- 構築子(constructors)

```
(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)) )
(define (make-from-mag-ang r a)
  (cons r a) )
```

- 選択子(selectors)

```
(define (real-part z)
  (* (magnitude z) (cos (angle z))) )
(define (imag-part z)
  (* (magnitude z) (sin (angle z))) )
(define (magnitude z) (car z))
(define (angle z) (cdr z))
```

24

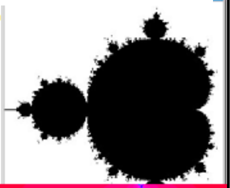
図形言語に複素数を導入

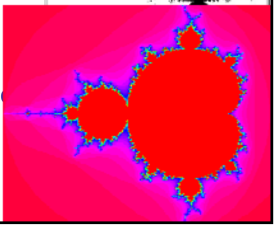
$$z_{n+1} = z_n^{\ 2} + C$$

$$z_0 = C$$

が収束する点 $C = (x, y)$

Mandelbrot Set





右上の図はframe coordinat
map未使用（直接点を描画）

http://mathworld.wolfram.com/MandelbrotSet.html

2通りの複素数表現

- どちらも動きことはわかった.
- ただし，混在できない.

- **どうしたら，両方の表現法を混ぜて使用することができるか.**

26

12月18日・本日のメニュー

データによる抽象化

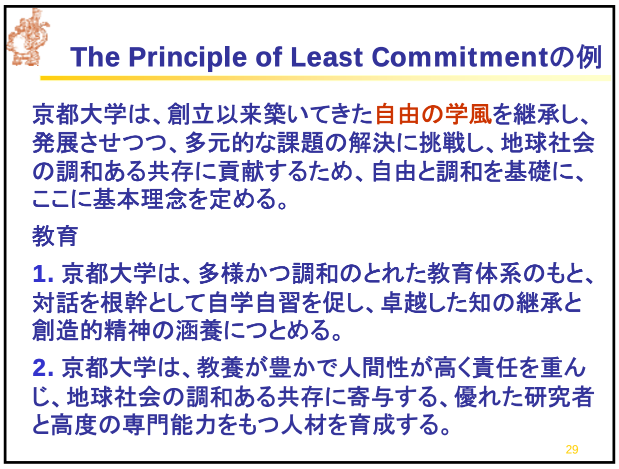
- 2.3.3 Representing Sets
- 2.4 Multiple Representations for Abstract Data
- 2.4.1 Representations for Complex Numbers
- **2.4.2 Tagged data**
- 2.4.3 Data-Directed Programming and Additivity
- 2.5 Genetic Operation System
  - Coercion
  - Hierarchy of types

27

2.4.2 Tagged data（タグ付きデータ）

- データ抽象化の1つの観点
- ***Principle of least commitment***
（最小責任の原則）
- 選択子と構築子を使用した抽象化の壁を使って、データオブジェクトの具体的な表現をできるだけ遅くし、システム設計における柔軟性を最大限にする。
- 本節ではさらに **principle of least commitment** を発展させる。

1. 表現法（選択子と構築子）の設計後でも、表現法の抽象化（曖昧性）を維持。
2. 直交座標と極座標が共用できる仕組みを考える。



The Principle of Least Commitmentの例

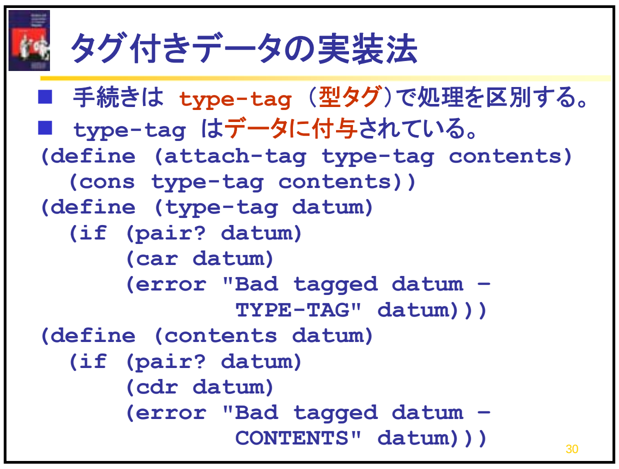
京都大学は、創立以来築いてきた自由の学風を継承し、発展させつつ、多元的な課題の解決に挑戦し、地球社会の調和ある共存に貢献するため、自由と調和を基礎に、ここに基本理念を定める。

教育

1. 京都大学は、多様かつ調和のとれた教育体系のもと、対話を根幹として自学自習を促し、卓越した知の継承と創造的精神の涵養につとめる。

2. 京都大学は、教養が豊かで人間性が高く責任を重んじ、地球社会の調和ある共存に寄与する、優れた研究者と高度の専門能力をもつ人材を育成する。

29

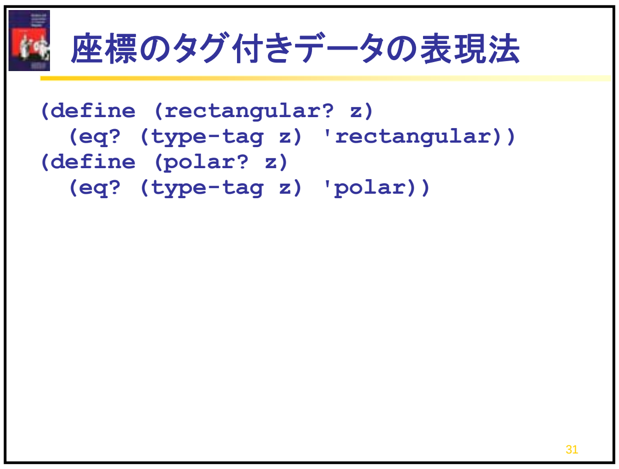


タグ付きデータの実装法

- 手続きは `type-tag` (型タグ)で処理を区別する。
- `type-tag` はデータに付与されている。

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum –
              TYPE-TAG" datum)))
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum –
              CONTENTS" datum)))
```

30



座標のタグ付きデータの表現法

```
(define (rectangular? z)
  (eq? (type-tag z) 'rectangular))
(define (polar? z)
  (eq? (type-tag z) 'polar))
```

31

直交座標のタグ付きデータの表現法

```
(define (real-part-rectangular z) (car z))
(define (imag-part-rectangular z) (cdr z))
(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z)
          (square (imag-part-rectangular z))
        )))
(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z) ))
(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)) )
(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
     (cons (* r (cos a)) (* r (sin a))) ))
```

32

直交座標のタグ付きデータの表現法

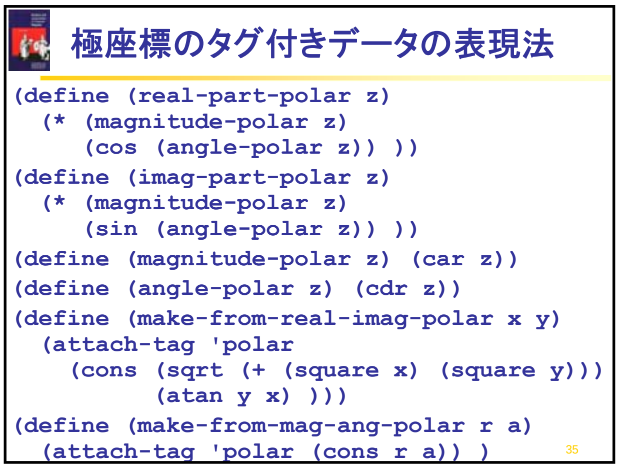
```
(define (real-part-rectangular z) (car z))
(define (imag-part-rectangular z) (cdr z))
(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z)
          (square (imag-part-rectangular z))
        )))
(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z) ))
(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)) )
(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
     (cons (* r (cos a)) (* r (sin a))) ))
```

33

極座標のタグ付きデータの表現法

```
(define (real-part-polar z)
  (* (magnitude-polar z)
     (cos (angle-polar z)) ))
(define (imag-part-polar z)
  (* (magnitude-polar z)
     (sin (angle-polar z)) ))
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))
(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
    (cons (sqrt (+ (square x) (square y)))
          (atan y x) )))
(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar(cons r a)) )
```

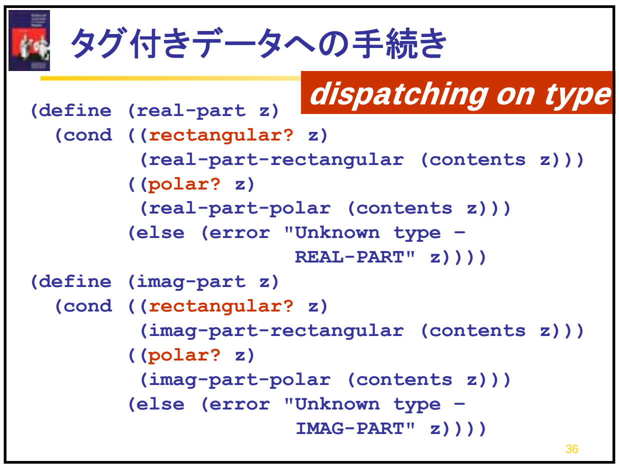
34



極座標のタグ付きデータの表現法

```
(define (real-part-polar z)
  (* (magnitude-polar z)
     (cos (angle-polar z)) ))
(define (imag-part-polar z)
  (* (magnitude-polar z)
     (sin (angle-polar z)) ))
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))
(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
    (cons (sqrt (+ (square x) (square y)))
          (atan y x) )))
(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)) )
```

35

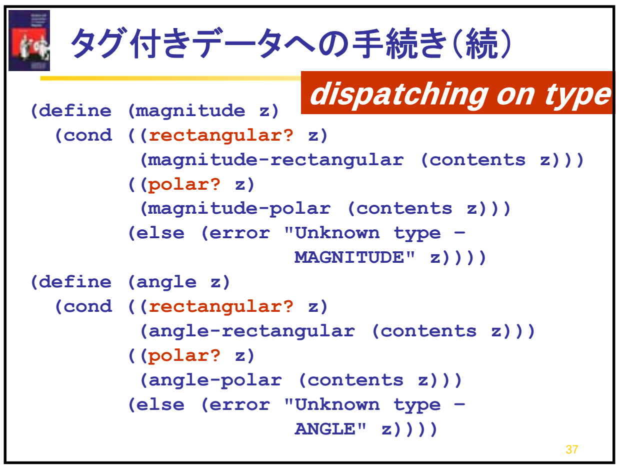


タグ付きデータへの手続き

dispatching on type

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type -
                      REAL-PART" z))))
(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))
        (else (error "Unknown type -
                      IMAG-PART" z))))
```

36



タグ付きデータへの手続き(続)

dispatching on type

```
(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
         (magnitude-polar (contents z)))
        (else (error "Unknown type -
                      MAGNITUDE" z))))
(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
         (angle-polar (contents z)))
        (else (error "Unknown type -
                      ANGLE" z))))
```

37

タグ付きデータへの手続き（続々）

- 複素数の演算は不変
- 複素数用汎用演算（generic operation）使用の為

```
(define (add-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))))
```

- 前と同じことに注意
- Principle of least commitmentによる遅延

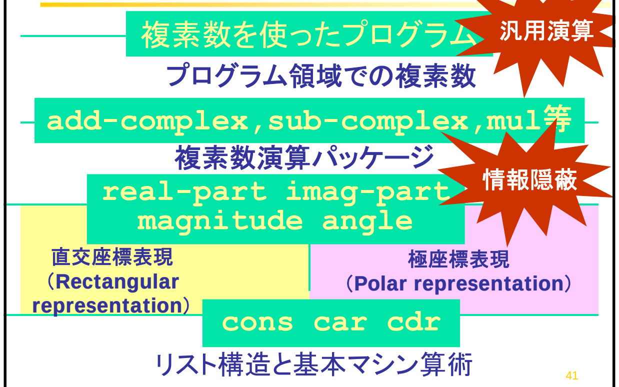
38

全システムの設計は

- 目的に合致した複素数表現を選ぶ
- 直交座標表現
  実数部と虚数部が分かっているとき
- 極座標表現
  半径と角度が分かっているとき
- 最終的に得られた複素数演算システムの構造は次のスライド
- type-tag の使用がポイント
- ***dispatching on type*** という技法

39

複素数システムのデータ抽象化の壁



41

12月18日・本日のメニュー

データによる抽象化

- 2.3.3 Representing Sets
- 2.4 Multiple Representations for Abstract Data
- 2.4.1 Representations for Complex Numbers
- 2.4.2 Tagged data
- **2.4.3 Data-Directed Programming and Additivity**
- 2.5 Genetic Operation System
  - Coercion
  - Hierarchy of types

42

2.4.3 Data-Directed Programming and Additivity

- 型タグ(type-tag)の問題点
- 汎用手続き(real-part, imag-part, maginitude, angle)は、異なる表現をすべて知っておく必要がある。
- 例えば、複素数の新表現を作成したら

1. (new-rep? z) を定義
2. 各手続きに new-rep? に関係する処理を追加

```
(define (real-part z)
   (cond ((rectangular? z) … )
         ((polar? z) … )
         ((new-rep? z) … )
         (else … )))
```

- **加法的(additivity)ではない。**

43

Data-Directed Programmingのポイント

- 加法的(additivity)なインタフェースとする為
- 表を行方向に分割:**type-tagでdispatch**

型 (type)

演算 operations	Polar	Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

44

表の操作

- 表に演算名・型(type)でその処理法をputで付加
- 表から演算名・型(type)で処理法をgetで検索
- (put *<op>* *<type>* *<item>*)
  表に*<op>* *<type>*で索引をつけて*<item>*を登録
- (get *<op>* *<type>*)
  表から*<op>* *<type>*の索引で検索し、あれば、
  *<item>*を抽出
- 演算に関連する情報*<item>*は、手続き(ラムダ式)
- *<type>*は、引数の型のリスト
- TUT-Scheme(tus2, tustk2)では、
- (define put putprop)
- (define get getprop)

45

put と get の動き

```
(put 'banana 'price 300)
(put 'banana 'color ' yellow)
(get 'banana 'price)
(put 'Kyoto 'Ja "kyouto")
(put 'University 'Ja "daigaku")
(get 'Kyoto 'Ja)
        -> "kyouto"
(map (lambda (x)  (get x 'Ja) )
    '(Kyoto University)  )
        -> ("kyouto" "daigaku")
(put 'University 'Ge "Universitate")
```

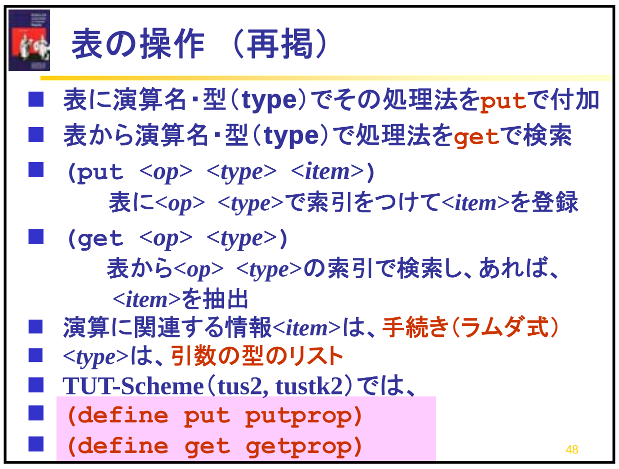
46

簡単な情報検索 by put & get

```
(define (lookup given-key set-of-records)
   (let ((result (get set-of-records given-key))
     (if (null? result) false result) ))
                           総人口   男性人口 女性人口
(put 'population 'China '(1285.0 660.5 624.5))
(put 'population 'India '(1025.1 528.5 496.6))
(put 'population 'USA '(285.9 141.0 144.9))
(put 'population 'Indonesia '(214.8 107.8
107.1))
(put 'population 'Brazil '(172.6 85.2 87.4))
(put 'population 'Pakistan '(145.0 74.5 70.5))
(put 'population 'Russia '(144.7 67.7 77.0))
(put 'population 'Bangradesh '(140.4 72.3 68.0))
(put 'population 'Japan '(127.1 62.2 65.0))
(put 'population 'Nigeria '(116.9 59.0 58.0))
(put 'population 'Mexico '(100.4 49.6 50.7))

(lookup 'Japan 'population)
```

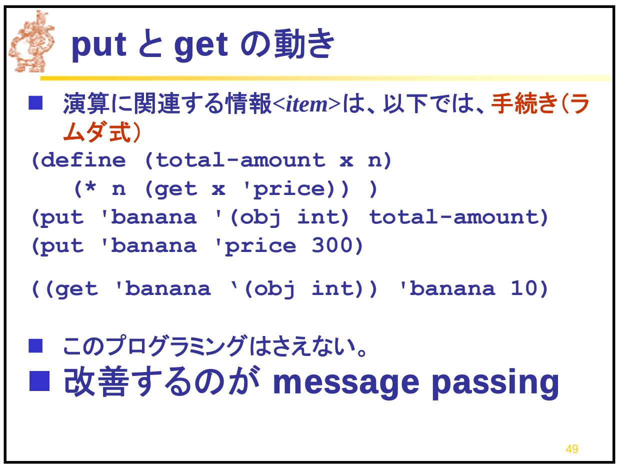
47



表の操作 （再掲）

- 表に演算名・型（type）でその処理法を`put`で付加
- 表から演算名・型（type）で処理法を`get`で検索
- (put *<op>* *<type>* *<item>*)
  表に*<op>* *<type>*で索引をつけて*<item>*を登録
- (get *<op>* *<type>*)
  表から*<op>* *<type>*の索引で検索し、あれば、*<item>*を抽出
- 演算に関連する情報*<item>*は、手続き（ラムダ式）
- *<type>*は、引数の型のリスト
- TUT-Scheme（tus2, tustk2）では、
- `(define put putprop)`
- `(define get getprop)`

48



put と get の動き

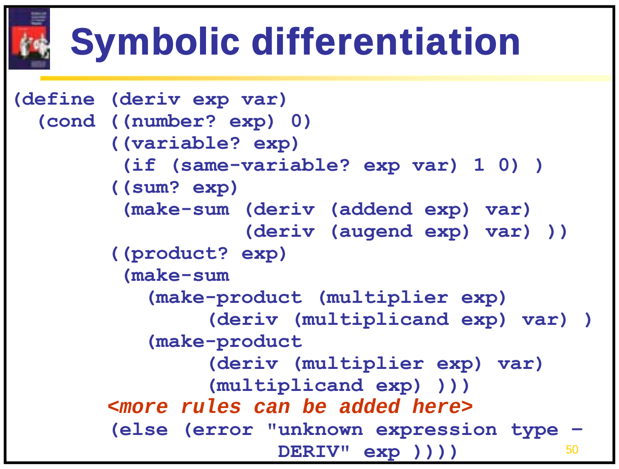
- 演算に関連する情報*<item>*は、以下では、手続き（ラムダ式）

```
(define (total-amount x n)
   (* n (get x 'price)) )
(put 'banana '(obj int) total-amount)
(put 'banana 'price 300)

((get 'banana '(obj int)) 'banana 10)
```

- このプログラミングはさえない。
- **改善するのが message passing**

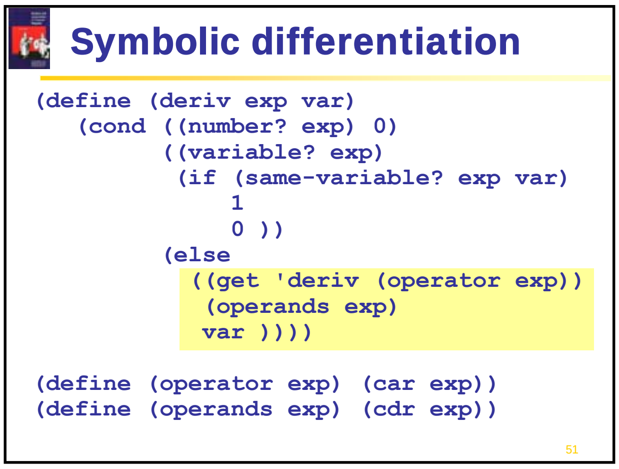
49



Symbolic differentiation

```
(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) )))
        <more rules can be added here>
        (else (error "unknown expression type -
                     DERIV" exp ))))
```

50

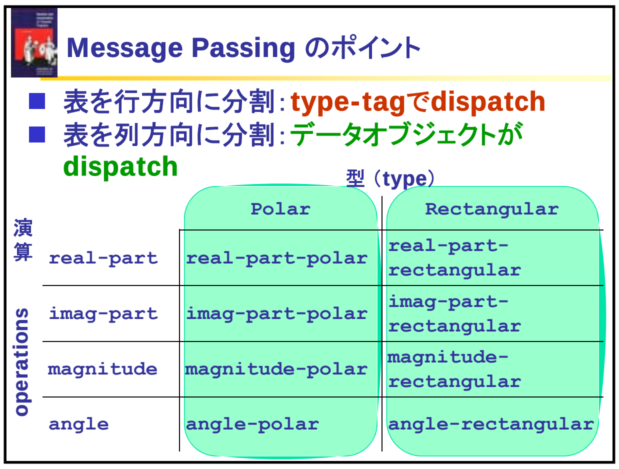


Symbolic differentiation

```
(define (deriv exp var)
   (cond ((number? exp) 0)
         ((variable? exp)
          (if (same-variable? exp var)
              1
              0 ))
         (else
           ((get 'deriv (operator exp))
            (operands exp)
           var ))))

(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
```

51

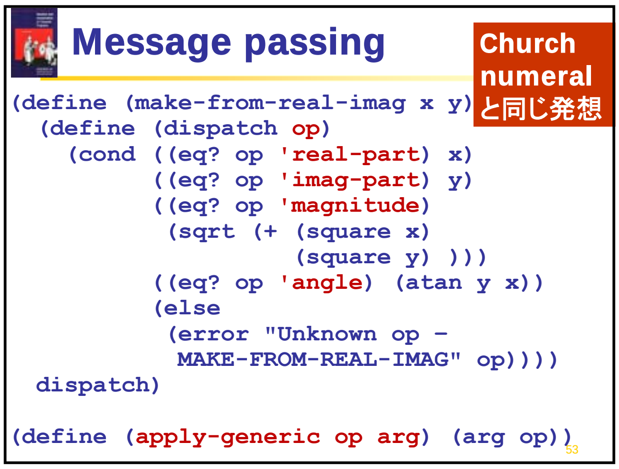


Message Passing のポイント

- 表を行方向に分割：type-tagでdispatch
- 表を列方向に分割：データオブジェクトが dispatch

型（type）

演算 operations	Polar	Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular



Message passing

Church numeral と同じ発想

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude)
           (sqrt (+ (square x)
                    (square y) )))
          ((eq? op 'angle) (atan y x))
          (else
           (error "Unknown op –
            MAKE-FROM-REAL-IMAG" op))))
  dispatch)

(define (apply-generic op arg) (arg op))
```

53

Merry Christmas and A Happy New Year

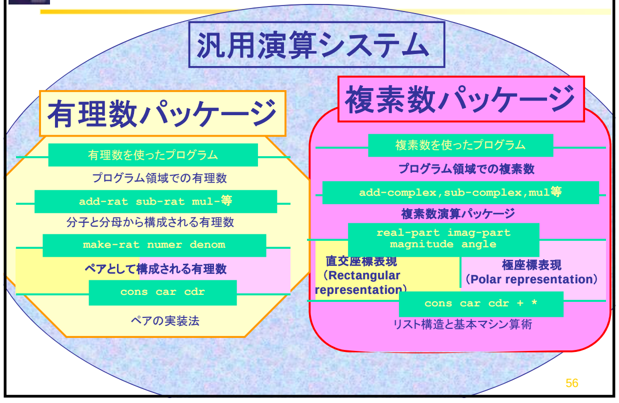
- 宿題は、次の計2問:
- **Ex.2.73, 2.76**
- 1月9日正午締切

12月18日・本日のメニュー

データによる抽象化
- 2.3.3 Representing Sets
- 2.4 Multiple Representations for Abstract Data
- 2.4.1 Representations for Complex Numbers
- 2.4.2 Tagged data
- 2.4.3 Data-Directed Programming and Additivity
- **2.5 Genetic Operation System**
  - **Coercion**
  - **Hierarchy of types**

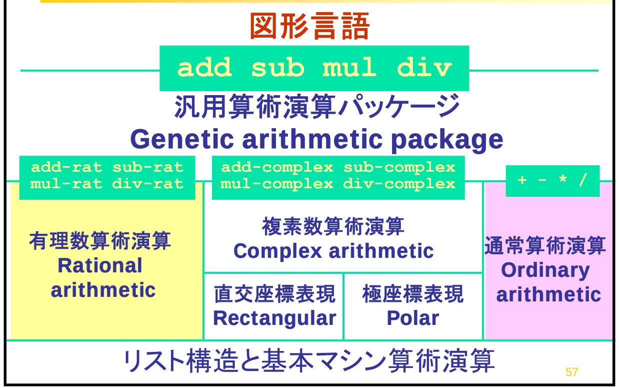
55

本節の目標: 統一システムの構築



56

汎用演算システムの構造



図形言語
add sub mul div
汎用算術演算パッケージ
Genetic arithmetic package
add-rat sub-rat mul-rat div-rat
add-complex sub-complex mul-complex div-complex
+ - * /
有理数算術演算 Rational arithmetic
複素数算術演算 Complex arithmetic
直交座標表現 Rectangular
極座標表現 Polar
通常算術演算 Ordinary arithmetic
リスト構造と基本マシン算術演算

57

2.5.1 汎用算術演算手続き

- add sub mul div だけで算術演算を記述する。
- 引数のタイプにより適切な演算を行う手続きを適用

```
(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) )
```

58

Ordinary number パッケージ

```
(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 )
```

59

Scheme number パッケージの使用法

```
(define (make-scheme-number n)
   ((get 'make 'scheme-number) n) )
(define foo (make-scheme-number 8))
```

scheme-number	8

```
(define bar (make-scheme-number 3))
```

scheme-number	3

```
(add foo bar)
((get 'add '(scheme-number scheme-number))
  (contents foo) (contents bar) )
(+ 8 3)
```

scheme-number	11

60

Rational number パッケージ

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

61

Rational number パッケージ(続)

```
(define (install-rational-package)
   ;; internal procedures

  (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)) ))
```

62

Rational number パッケージ(続々)

```
(define (install-rational-package)
  ;; 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 )
```

63

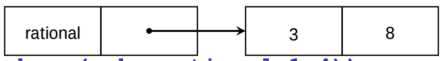
Rational number パッケージ

```
(define (install-rational-package)
  ;; 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 )
```

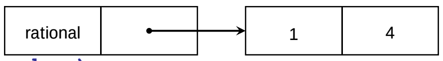
64

Rational number パッケージの使用法

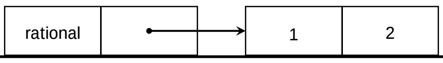
```
(define (make-rational n d)
   ((get 'make 'rational) n d)) )
(define foo (make-rational 3 8))
```



```
(define bar (make-rational 1 4))
```



```
(add foo bar)
((get 'add '(rational rational))
 (contents foo) (contents bar) )
(add-rat (contets foo) (contents bar))
```



65