プログラミング言語論

プログラミング言語の基礎

水野嘉明

目次

2

1. プログラミング言語とは

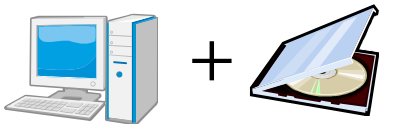
- プログラミング言語とは何か？

3

1. プログラミング言語とは

➢コンピュータ ＝

ハードウェア　　ソフトウェア



4

1. プログラミング言語とは

➢コンピュータで問題を解くためには、
ソフトウェア（プログラム）が必要

➢プログラムを記述するための言語

＝ *プログラミング言語*

5

1. プログラミング言語とは

●プログラミング言語の種類

➢*低級言語*（低水準言語）

●機械語、アセンブリ言語

➢*高級言語*（高水準言語）

●抽象度の高い言語

人にとって分かりやすい

ハードウェアを意識しなくてよい

6

2. 高級言語の必要性

- 高級言語は、なぜ必要なのか？

7

2. 高級言語の必要性

- *機械語*（machine language）
  - 各CPUは*命令セット*を持っている
  - 各命令には、バイナリのコードが定められている
    - CPUの種類によって、命令セットは異なる

8

2. 高級言語の必要性

- *機械語*によるプログラムの例

命令コードは、以下のような2進数（ビットパターン）の列

```
55 89 E5 53 56 57 39 DA 73 31
B9 00 00 00 00 BF 00 00 00 ・・・
```

（16進数表記）

9

2. 高級言語の必要性

- コンピュータは、この命令コードを解釈・実行する

```
55            ( BPをスタックにセーブ )
89E5          ( SPをBPにコピー )
53            ( BXをスタックにセーブ )
56            ( SIをスタックにセーブ)
57            ( DIをスタックにセーブ )
39DA          ( BXとDXを比較 )
7331          ( BX≦DXの時ジャンプ )
B900000000    ( CXに0を格納 )
```

10

2. 高級言語の必要性

- 機械語のままでは、人間には読み書きが大変

- 命令コードを、記号(mnemonic)を使って記述
  ＝ *アセンブリ言語*
  (assembly language)

11

2. 高級言語の必要性

- 機械語とアセンブリ言語の例

```
55          : pushl   %ebp
89E5        : movl    %esp, %ebp
53          : pushl   %ebx
56          : pushl   %esi
57          : pushl   %edi
39DA        : cmpl    %ebx, %edx
7331        : jae     ui_div_ovf
B900000000  : movl    $0, %ecx
BF00000080  : movl    $0x80000000,%edi
BE00000000  : movl    $0, %esi
D1EB        : shrl    $1, %ebx
```

12

2. 高級言語の必要性

- アセンブリ言語のままでは、まだまだ人間には大変

⇩

高級言語(high-level language)の登場
- ➢可読性、移植性
  - ●開発の手間
  - ●デバッグの効率
  - ●メンテナンス

13

2. 高級言語の必要性

- プログラムの目的・用途や、プログラムに対する思想・考え方により、様々な種類の高級言語が誕生

⇩

色々な言語と、その背後にある考え方を、以下の講義では見ていく

14

2. 高級言語の必要性

- 高級言語の実行方法は、大きく分けて二通り
  - ➢*コンパイラ*
    - ●ソースプログラムを、機械語による実行可能形式に変換する
  - ➢*インタプリタ*
    - ●ソースプログラムを逐次解釈しながら実行していく

15

2. 高級言語の必要性

- 多くの高級言語は、コンパイル方式
- インタプリタ方式は
  - LISP系の言語（コンパイラもあり）
  - BASIC　（　〃　）
  - スクリプト言語　など
- Javaは、いったん中間言語に変換された後、JVM（Java仮想マシン）にて実行される

16

3. 基礎知識

- 本題に入る前に必要となる、いくつかの基礎知識

⇨ 3.1 式の記述
3.2 関数
3.3 再帰
3.4 型システム
3.5 データ型の実際
3.6 反射的推移閉包

17

3.1 式の記述

- 2項演算子の記法
  - *中置記法*（infix notation）
    - 通常の記法
  - *前置記法*（prefix notation）
    - 演算子を前に置く書き方
  - *後置記法*（postfix notation）
    - 演算子を後ろに置く書き方

18

3.1 式の記述

- 中置記法
  - 通常の書き方を、*中置記法* という
    - 例1） a + b

      項と項の間に演算子を置いている
    - 例2） 3 + 5 * 2

19

3.1 式の記述

- 中置記法の特徴
  - 通常は演算子に優先順位がある
    - 例） +、- よりも *、/ が優先
      3 + 5 * 2 → 13
  - 括弧を用いて計算順序を指定する
    - 例）
      (3 + 5) * 2 → 16

20

3.1 式の記述

- 同一優先順位では、左から右へ、または右から左へグループ化
  - 1） 左から右へグループ化
    （*左結合*（left associative））
    例: 5-3-1 は (5-3)-1 の意
  - 2） 右から左へグループ化
    （*右結合*（right associative））

21

3.1 式の記述

注: 左結合か、右結合かは、演算子により決まっている

- 例1 [四則演算は左結合]
  x − y + z は x − (y + z) ではなく、(x − y) + z である
- 例2 [べきは右結合]

5^{3^2} は、$5^{(3^2)}$ の意

22

3.1 式の記述

- 前置記法
  - 「演算子 項1 項2」という書き方を *前置記法*（*ポーランド表記法*）という
    - 例1) + a b
  - 各項は、式でもよい
    - 例2) + a ＊ b c （a と *bc の和）

23

3.1 式の記述

- 前置記法の例
  - *(a + b) ＊ c*　（中置）

    ⇩

    ＊ + a b c　（前置）

例: (20 + 30) ＊ 60 = 3000

⇩

＊ + 20 30 60 = ＊ 50 60 = 3000

24

3.1 式の記述

- *a * (b + c)*　（中置）

** a + b c*　　（前置）

例　20 * (30 + 60) = 1800

* 20 + 30 60 = * 20 90 = 1800

25

3.1 式の記述

- 中置記法から前置記法への変換
  - 「X 演算子 Y」⇒ 「演算子 X Y」という変換が基本
  - 方法1:
    全体の構造を見て、外側から
  - 方法2:
    計算順序に従い、内側から

26

3.1 式の記述

- 方法1の例「(a + b) * c / d」

(a + b) * c / d

→ / (a + b) * c d

→ / * a + b c d

（アンダーラインは、まだ中置記法の部分）

27

3.1 式の記述

- 方法2の例 「(a + b)＊c / d」
  (a + b)＊c / d
  → + a b　　(aとbを加え)
  → ＊ + a b c　　(cを掛けて)
  → / ＊ + a b c d　　(dで割る)

28

3.1 式の記述

- 後置記法
  - 「項1 項2 演算子」という書き方が *後置記法* (*逆ポーランド表記法*)
    - 例1) a b +
    - 例2) a b c ＊ + (a と bc* の和)

29

3.1 式の記述

- 後置記法の例
  - *(a + b)＊c*　(中置)

    ⇩

    a b + c ＊　(後置)

  例: (20 + 30)＊60 = 3000

  ⇩

  20 30 + 60 ＊ = 50 60 ＊ = 3000

30

3.1 式の記述

- *a ＊ (b ＋ c)*　（中置）

a b c ＋ ＊　（後置）

例　20 ＊ (30 + 60) = 1800

20 30 60 + ＊ = 20 90 ＊ = 1800

31

3.1 式の記述

- 前置記法、後置記法の特徴
  - 計算順序は、明確
    ⇒ 括弧、優先順位は不要
  - 前置記法では、演算子はあたかも関数の適用のような形となる
    例) + a b ⇒ plus(a, b)
  - 後置記法は、コンピュータで計算するのに都合がよい

32

3.1 式の記述

- 演習1.1
  次の前置記法の式(1)、および後置記法の式(2)を、計算せよ

(1) `/ + 7 5 2`

(2) `10 3 - 25 5 / ＊`

33

3.1 式の記述

- 演習1.2
  次の式（中置記法）を、前置記法に直せ
  (1) x − y * z
  (2) b * b − 4 * a * c

34

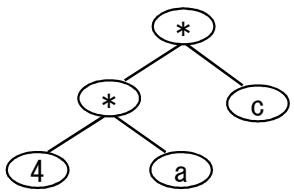
3.1 式の記述

- 演習1.3
  同じ式を、後置記法に直せ
  (1) x − y * z
  (2) b * b − 4 * a * c

35

3.1 式の記述

- 式の木表現 （構文木）

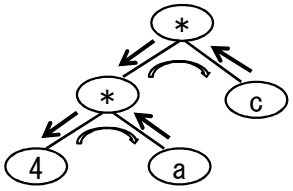


4 * a * c の 木表現

36

3.1 式の記述

- 構文木を順にたどることにより、前置／中置／後置の各記法の式を求めることができる



37

3.1 式の記述

- 先行順（preorder）でたどる
  ⇒ 前置記法 :「＊＊４ａｃ」
- 中間順（inorder）でたどる
  ⇒ 中置記法 :「４＊ａ＊ｃ」
- 後行順（postorder）でたどる
  ⇒ 後置記法 :「４ａ＊ｃ＊」

38

3.2 関数

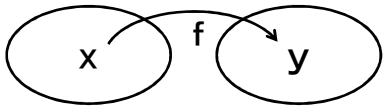
3.1 式の記述
⇨ 3.2 関数
3.3 再帰
3.4 型システム
3.5 データ型の実際
3.6 反射的推移閉包

39

3.2 関数

- 数学的な関数
  - 集合Aから集合Bへの写像
    （定義域）（値域）

y = f(x)



40

3.2 関数

- アルゴリズムとしての関数
  一連の命令群であり、以下のように動作するもの
  1. 引数と呼ばれるデータを受け取り
  2. 定められた処理を実行し
  3. 結果(値)を返す

41

3.2 関数

- アルゴリズムとしての関数の*宣言*
  - 次の3つの部分からなる
    1. 宣言された関数の名前
    2. 関数の引数
    3. 引数から結果を計算する規則

（構文の例）

```
fun  <name>(<para>) = <body>;
```

42

3.2 関数

- 式の中で関数を使うこと
  ＝ *適用* (application)
  - 関数の適用の規則には、前置記法を用いる

<name>(<para>)

43

3.2 関数

- 仮引数と実引数
  - 関数定義中の引数を、*仮引数* (formal parameters)という
  - 関数の適用時に、実際に与えられる引数並びを *実引数* (actual parameters)という
  - 実引数には、式を与える事もできる

44

3.2 関数

- 関数の評価 ＝ *駆動* (activation)
  - 実引数として与えられた式を評価
  - 関数本体内の仮引数を評価結果で置き換える
  - 関数本体を評価する
  - 評価した値を答としてリターンする

注: 引数の受け渡し方は、様々

45

3.2 関数

- 例
  - 関数宣言

    fun successor(n) = n + 1;

    仮引数
  - 適用

    x = successor(2 + 3);

    実引数

46

3.2 関数

- 駆動
  - 実引数 2 + 3 を評価→値5を得る
  - 関数本体 n + 1 の仮引数 n を値 5 で置き換える
  - その結果得られる式 5 + 1 を評価する
  - 答えの 6 をリターン

47

3.3 再帰

48

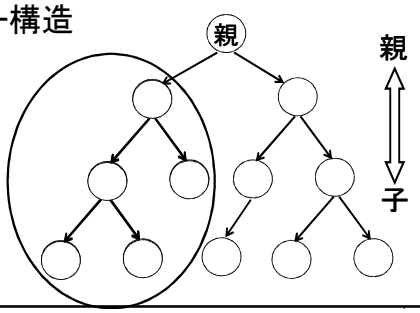
3.3 再帰

- *再帰的* (recursive)であるとは
  1. ある対象が、その一部分を取り出すと自分自身と同形であるような構造であること
  2. ある対象が、自分自身を用いて循環的に定義されていること
  3. プログラムの中から、自分自身を(直接または間接的に)コールしていること

49

3.3 再帰

- 再帰の例 (1) -- 再帰的な構造
  - ツリー構造



50

3.3 再帰

- 再帰の例(2) -- 再帰的な定義
  - fact(0) = 1
    fact(n) = n * fact(n-1)　　-- nの階乗
  - sum(0) = 0
    sum(n) = n + sum(n-1)　　-- Σn
  - fib(0) = 0, fib(1) = 1
    fib(n) = fib(n-1) + fib(n-2)
    -- フィボナッチ数

51

3.3 再帰

- 再帰の例（3） --再帰呼び出し
  - nの階乗を計算 （Java）

```
public static int  fact ( int  n )
{
    if (n <= 0)                  // 0の階乗は1
        return  1;
    else
        return  n * fact(n-1);
}
```

52

3.3 再帰

- C言語による同じプログラム例

```
int   fact ( int  n )
{
    if (n <= 0)                  // 0の階乗は1
        return  1;
    else
        return  n * fact(n-1);
}
```

53

3.3 再帰

- 本講義では、3番目の意味での再帰を見ていく
  - プログラムの中から、自分自身を直接または間接的にコールしている
    （関数 f がその関数本体の中に f の適用を含む）

54

3.3 再帰

- 演習1.4
  - フィボナッチ数を求めるメソッド（関数）fib を、Java（または C）にて作成せよ

    fib(0) = 0, fib(1) = 1
    fib(n) = fib(n−1) + fib(n−2)

    ※ 再帰を利用すること

55

3.3 再帰

- *線形再帰* (linear recursive)
  - 関数fの駆動f(a)が、最大でも1回しか新しいfを駆動しない
  - 例1

    fun fact(n) =
    if n=0 ***then*** 1 ***else*** n*fact(n−1);
  - 例2

    演習1.4の関数fibは線形再帰ではない

56

3.3 再帰

- *末端再帰* (tail recursive)
  - 関数fが再帰を必要とせず値をリターンするか、または単に再帰的駆動の結果をリターンする時、関数fは 末端再帰である
    （つまり、最後に再帰呼び出しをしてそのままリターン、ということ）

57

3.3 再帰

- 再帰の長所と短所
  - プログラムがシンプルになり、アルゴリズムが分かりやすくなる

    ⇧

    データ構造とアルゴリズムがマッチする
  - 実行の効率が悪くなりやすい

58

3.3 再帰

- 例1

  階乗のプログラムは、一般的には再帰を使わない方がよい（線形再帰はループにできる）
- 例2

  データ構造が再帰的な場合(ex. 木構造)は、プログラムも再帰が適している

59

3.4 型システム

3.1 式の記述
3.2 関数
3.3 再帰
⇨ 3.4 型システム
3.5 データ型の実際
3.6 反射的推移閉包

60

3.4 型システム

- 式の*型* (type)とは
  - ➢それがどのような値を取れるか
  - ➢それにどのような操作が適用できるか

  を示すもの

61

3.4 型システム

- ➢例えば、整数は加算できるが、論理値(true/false)はできない
  つまり、＋ は
  - ●二つの整数型の式には適用可能
  - ●二つの論理型の式には適用不可

  ○ 3 + 5

  X true + false

62

3.4 型システム

- プログラミング言語では

  すべての式は、唯一の型を持たなければならない

  (注:実行時に式の型が決まる言語もある)

63

3.4 型システム

- 全てのデータ(およびプログラム)はビットパターンで表現されている
  - IntelCPUでは、文字「@」、整数64、INC命令は皆同じビットパターン "0100 0000"
  - このパターンが何を表すかは、プログラムの解釈次第

64

3.4 型システム

- *型システム*
  - 言語が持つ、式の型を決定する規則の集合

（例）1　→ 整数
1.0 → 実数
整数 + 整数 → 整数
整数 + 実数 → 実数

65

3.4 型システム

- *型検査* (type checking)
  操作が正しく適用されることを確実にする
  - 静的な検査
    ＝ コンパイル時に検査
  - 動的な検査
    ＝ 実行時に検査
    （余分なコード → 効率の低下）

66

3.4 型システム

- 静的型付け と 動的型付け
  - プログラムの定義時点で、式や変数の型が決まるものを*静的型付け*と呼ぶ
  - 実行時に型が決定されるものを*動的型付け*という
    - LISP系言語、スクリプト言語など

67

3.4 型システム

- *強い型付け*
  ある処理・演算が間違った型の引数をとることを禁止する
  ＝ 安全な式だけを受け入れる
  （型エラー無く評価されることが保障される）
- *弱い型付け* ＝ 強くない

68

3.4 型システム

- 各言語の型付け
  - C言語
    - 弱い静的型付け
  - Java
    - 強い静的型付け
  - LISP
    - 強い動的型付け

69

3.5 データ型の実際

3.1 式の記述
3.2 関数
3.3 再帰
3.4 型システム
⇨3.5 データ型の実際
3.6 反射的推移閉包

70

3.5 データ型の実際

- プログラミング言語における型には2つのレベルがある
  - 基本データ型
  - 構造型

71

3.5 データ型の実際

- *基本データ型* とは
  - 自然で基本的な型であり、単一の値を持つもの
    例: 整数、実数、文字など
  - マシン命令により直接操作することができる
  - プリミティブ型、単純型などともいう

72

3.5 データ型の実際

- *構造型* とは
  - プログラマが記述して作成する型
    例: レコード、配列、クラスなど
  - 言語により、どのような型を作成できるかは異なる
  - 複数のデータを格納できる
  - ユーザ型、複合データ型ともいう

73

3.5.1 基本データ型

- 主な基本データ型
  - 整数
  - 実数
  - 文字
  - 論理型
  - 列挙型　など

74

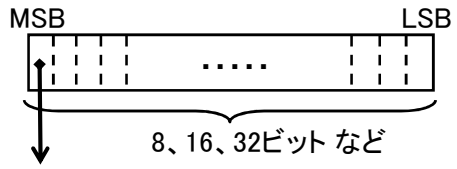
3.5.1 基本データ型（整数）

- 整数
  - 通常の（ノイマン型）計算機では、整数値は2進数で表現されている
  - 整数には、符号付きと符号なしの2種類がある
  - 符号付き整数は、負の数を2の補数表現にて表す

75

3.5.1 基本データ型（整数）

- 符号付整数の表現



符号ビット：このビットがオンのとき負
（符号なしの時はこのビットも数値）

76

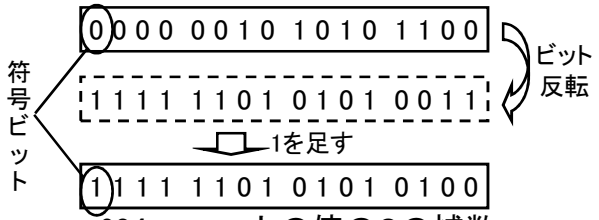
3.5.1 基本データ型（整数）

- 演習1.5
  - 整数を、16ビット2進数で表す。
    次の整数のビットパターンを求めよ
    (1) -1
    (2) 10
    (3) -10
    (4) 255
    (5) -255

77

3.5.1 基本データ型（整数）

- 演習のヒント ＝ 2の補数の求め方
  - $684_{(10)} = 02AC_{(16)}$



  - $-684_{(10)}$ ⇒ 上の値の2の補数

78

3.5.1 基本データ型（実数）

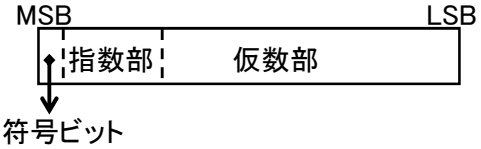
- 実数
  - ➢固定小数点数
    - ●ビット列中の小数点位置を固定
  - ➢浮動小数点数
    - ●科学表記（例 1.23×10^4）と同じ考え方

$$\pm(\text{仮数}) \times (\text{基数})^{\text{指数}}$$

79

3.5.1 基本データ型（実数）

➢浮動小数点数



基数は、通常″2″または″16″

$$\pm \text{仮数} \times 2^{\text{指数}}$$

80

3.5.1 基本データ型（文字）

- 文字
  - ➢文字は、1文字ずつコード（番号）をつけて表現する
  - ➢例: A -- $41_{(16)}$
    a -- $61_{(16)}$
    0 -- $30_{(16)}$
    字 -- 8E9A $_{(16)}$

81

3.5.1 基本データ型（文字）

- コードのつけ方には、何通りもある
- 大きく分けて、1バイト系と多バイト系がある
  - 1バイト系は漢字を表現できない（1バイトで表現できる数は256個まで）

82

3.5.1 基本データ型（文字）

- 演習1.6
  - 文字コードの種類には、どんなものがあるか。知っている文字コードを挙げよ

83

3.5.1 基本データ型（論理型）

- 論理型（ブーリアン型）
  - 真／偽の 論理値をとる
  - true/false、T/F、T/Nil 等と表わす
  - 多くの場合、真=1、偽=0 とされる（必ずそうなるわけではない）
  - C言語には、論理型がない

84

3.5.1 基本データ型（論理型）

- if文などの条件式は、論理型をとる式が求められる

```
if (x > 3.0)  ・・・
```

真または偽の値をとる

85

3.5.1 基本データ型

- 文字や論理型の値は、すべてコード（数値）で表現される

- コンピュータ内部では、整数と同様に扱われる

86

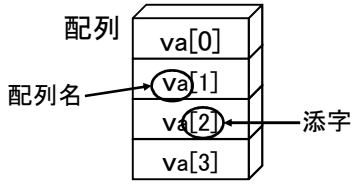
3.5.2 構造型

- *構造型* とは
  - 基本データ型を構成要素とする、複雑なデータ構造
  - 配列型、レコード型（構造体）、クラスなど

87

3.5.2 構造型（配列）

- 配列（array）
  - 同じ型の変数を集め、*添字*という番号で管理できるようにしたものが*配列*である



88

3.5.2 構造型（配列）

- 添字は、[2] のような定数のほかに変数や式でもOK
  例） va[x]　　array[2*y+3]
- 配列の特徴
  ＝添字を計算することにより、複数の要素の中から一つを番号で指定できる

89

3.5.2 構造型（配列）

- 一次元配列だけではなく、二次元、三次元、・・・が可能である

va[0][0]	va[0][1]	va[0][2]	va[0][3]
va[1][0]	va[1][1]	va[1][2]	va[1][3]
va[2][0]	va[2][1]	va[2][2]	va[2][3]

90

3.5.2 構造型（配列）

- 二次元配列のメモリ上での並び方には、二通り
  - *行優先配置*（row-major layout）
    va[0][0], va[0][1], va[0][2], va[0][3],
    va[1][0], va[1][1], ・・・
  - *列優先配置*（column-major layout）
    va[0][0], va[1][0], va[2][0], va[0][1],
    va[1][1], va[2][1], ・・・

91

3.5.2 構造型（配列）

- 添字は、言語により
  - 0から始まるもの（0 origin）
  - 1 から始まるもの（1 origin）
  - 範囲を指定できるもの

などがある

92

3.5.2 構造型（配列）

例1：C言語の配列は、
- 行優先配置
- 添字は 0から

例2：Fortran の配列は、
- 列優先配置
- 添字は1から

93

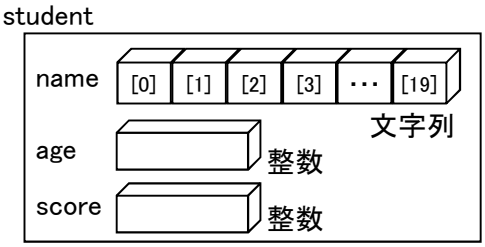
3.5.2 構造型（レコード）

- *レコード*(record)
  - 関連するデータを集め、ひとまとまりにしたもの
  - 各要素の型は異なっていてもよい
  - 実用的なプログラムには不可欠なデータ構造

94

3.5.2 構造型（レコード）

- レコードの例



ひとまとめにして、型としての名前をつける

95

3.5.2 構造型（レコード）

- 前記のレコード（構造体）を、C言語にて宣言した例

```
struct  student {
    char        name[20];       // 学生名
    int         age;            // 年齢
    int         score;          // 成績
} ;
struct student    student_data[200];
```

96

3.5.2 構造型（クラス）

- クラス
  - データと、そのデータを扱う手続き（メソッド）を ひとまとめにした構造

※ 詳細は、「オブジェクト指向」で解説する

97

3.5.3 言語とデータ型

- 言語によって、サポートするデータ型は異なる
  - COBOLには、文字列と数値の2種類のデータ型のみ
  - VBには、日付/時刻型や通貨型、オブジェクト型などもある
  - C言語には、文字型（char）はあるが文字列型は存在しない

98

3.5.3 言語とデータ型

【参考】C言語の型

<table>
<tr><td rowspan="5">スカラ型（基本データ型）</td><td rowspan="4">汎整数型</td><td>列挙型</td><td>enum</td><td rowspan="4">算術型</td></tr>
<tr><td>文字型</td><td>char 等</td></tr>
<tr><td>整数型</td><td>int 等</td></tr>
<tr><td colspan="2">浮動小数点型</td><td>double 等</td></tr>
<tr><td colspan="3">ポインタ型</td><td rowspan="4">派生型</td></tr>
<tr><td rowspan="3">集成体型（構造型）</td><td colspan="3">配列型</td></tr>
<tr><td colspan="3">構造体型</td></tr>
<tr><td colspan="3">共用体型</td></tr>
</table>

99

3.5.3 言語とデータ型

【参考】Javaの型

基本データ型	整数型	byte, short, int, long
	浮動小数点型	float, double
	文字型	char
	論理型	boolean
参照型（構造型）	クラス型	
	配列型	

100

3.6 反射的推移閉包

3.1 式の記述
3.2 関数
3.3 再帰
3.4 型システム
3.5 データ型の実際
⇨3.6 反射的推移閉包

101

3.6.1 閉包

- 集合 Σ, Γ に対して，その*連接*とは
  $\Sigma \cdot \Gamma = \{xy \mid x \in \Sigma, y \in \Gamma\}$
- $\Sigma \cdot \Sigma$ は、Σ^2と書く
  （例）要素 a, bからなる集合 $\Sigma = \{a, b\}$ について、
  $\Sigma \cdot \Sigma = \Sigma^2 = \{ xy \mid x, y \in \Sigma \}$
  $= \{ aa, ab, ba, bb \}$

102

3.6.1 閉包

- 集合 Γ に対し、

  $\Gamma^* = \{\varepsilon\} \cup \Gamma \cup \Gamma^2 \cup \Gamma^3 \cup \cdots$

  $= \{\varepsilon\} \cup (\cup_{i=1}^{\infty} \Gamma^i)$

  （つまり、Γ の要素を0個以上連ねることにより得られる要素列の集合）

  を、Γ の*閉包*（closure）という

  注：ε は長さ0の語（*空語*）

103

3.6.1 閉包

- 閉包の意味

  $\alpha \in \Gamma^*$ とは、

  > α は、集合 Γ の要素を 0個以上並べたものである

  という意味

104

3.6.1 閉包

- 参考：「閉じている」の意味

  $\forall x, y \in \Gamma^*$ に対し、$xy \in \Gamma^*$

  つまり、「Γ^* の外に出ない」

  ＝「閉じている」

105

3.6.2 関係

- 集合Sの要素 a, b の間で、関係Rが成り立っているとき、
  aRb
  と書く
  - 例
    - a＝b　a＜b　a≠b　など

106

3.6.2 関係

- 関係Rの定義
  - 集合S上の関係Rは、S×Sの部分集合である
    $R \subseteq S \times S$
  - a, b∈Sに対して (a, b)∈Rのとき aRb と書く

107

3.6.2 関係

- *反射的* (reflexive)な関係
  - 集合S上の関係Rが、*反射的* であるとは
    - どの元も、自分自身と関係Rが成り立つ
  - Rは反射的 ⇔ $^{\forall}a \in S:\ aRa$

108

3.6.2 関係

- *推移的*（transitive）な関係
  - 集合S上の関係Rが、*推移的*であるとは
    - 関係Rが、順々に伝わる
  - Rは推移的 ⇔ $^{\forall}$a, b, c∈S:
    aRb かつ bRc ならば aRc

109

3.6.2 関係

- *対称的*（symmetric）な関係
  - 集合S上の関係Rが、*対称的*であるとは
    - 左右交換しても、関係Rが成り立つ
  - Rは対称的 ⇔ $^{\forall}$a, b∈S:
    aRb ならば bRa

110

3.6.2 関係

- 関係の例
  - 「＝」は、反射的、推移的、対称的な関係（同値関係）
  - 「＜」は、推移的だが、反射的でも対称的でもない
  - 「≠」は、反射的でも推移的でもないが、対称的な関係

111

3.6.2 関係

- 演習1.7
  - 関係「 ≧ 」は、
    - 反射的か
    - 推移的か
    - 対称的か

112

3.6.3 反射的推移閉包

- 関係Rの *推移閉包* R^+ の定義
  - (1) $aRb \Rightarrow aR^+b$
  - (2) aR^+b かつ $bRc \Rightarrow aR^+c$
  - (3) (1)と(2)から導けないものは R^+の元でない

113

3.6.3 反射的推移閉包

- Rの *反射的推移閉包*

$$R^* = R^+ \cup \{(a, a) \mid a \in S\}$$

a R* b とは、「aから関係Rを0回以上繰り返せば bにたどり着ける」ということを表している

114

お疲れさまでした