プログラミング言語論

関数型プログラミング言語

水野嘉明

目次

2

1. 関数型プログラミングの特徴

(1) 純関数型プログラミング
(2) first-class object としての関数
(3) ラムダ式／ラムダ計算

3

1. 関数型プログラミングの特徴

(1) *純関数型プログラミング*

- 式の値があれば、それはその部分式の値にだけ依存する（つまり、副作用がない）
- 式は、評価されるたびに同じ値を持つ

参照の透明性（tranceparency）

4

1. 関数型プログラミングの特徴

- 純関数型プログラミングとは、変数や代入のないプログラミング
- 純関数型

内部状態を持たない

⇔

命令型

代入により内部状態が変化

5

1. 関数型プログラミングの特徴

- ほとんどの関数型言語は、代入操作を持つ

⇒ 純粋ではない

本質は、純粋な部分により支配されている

6

1. 関数型プログラミングの特徴

(2) 関数の地位 -- first-class object としての関数

- 関数は、他のすべての値と同じ地位を持つ
  - 関数は、式の値となり得る
  - 関数を引数として渡すことができる
  - 関数を、データ構造の中に収めることも出来る

7

1. 関数型プログラミングの特徴

注: first-class object

第一級オブジェクト

(一等値)

- プログラミング言語において、計算対象となるデータ(値)のこと

8

1. 関数型プログラミングの特徴

(3) ラムダ式／ラムダ計算

- 関数を定義し、関数適用を繰返すことにより、計算を行う
- ラムダ計算は、関数型プログラミングの理論的基盤である

9

2. ラムダ式とラムダ計算

2.1 ラムダ式
2.2 ラムダ計算
2.3 カリー化
2.4 チャーチ=ロッサーの定理

10

2. ラムダ式とラムダ計算

- λ式については、
  「3. プログラミング言語の特徴と分類」
  で、簡単に紹介した
- ここでは、復習の後 以下を見ていく
  - λ式の簡約
  - 「カリー化」の概念
  - チャーチ=ロッサーの定理

11

復習

2.1 ラムダ式

- 関数をλ式で表す

sq(x) = x * x

sq = λx.* x x

引数がxであることを示す

関数値の計算方法＝関数本体（body）

12

復習

2.1 ラムダ式

- 関数適用と関数抽象
  M、Nがλ式のとき
  - MN を *関数適用*（application)という
    - 関数の呼出しに相当する
  - λx.M を *関数抽象*（abstraction）、（または ラムダ抽象）という
    - 関数の定義に相当する

13

復習

2.1 ラムダ式

- λ式の定義 （M、Nはλ式とする）
  - 変数はλ式である
  - 関数適用（MN ） は λ式である
  - 関数抽象（λx.M) はλ式である

※ 紛らわしくない場合は、括弧は適宜省略できるものとする

14

復習

2.2 ラムダ計算

- λ計算とは
  - 関数の定義と実行を抽象化した計算体系
  - λ式の「簡約」により、計算を行う
  - λ算法ともいう

15

2.2 ラムダ計算

- *束縛変数* (bound variable)
  - 式Mに出現する変数 xは、抽象化 λx.M により*束縛* (bind)されるという

  （例）λx.+ x 1

  x が仮引数として宣言されたことを意味する

16

2.2 ラムダ計算

- *自由変数* (free variable)
  - 束縛変数ではない変数

  （例1） λx.+ x y

  xは束縛変数であるが、yは自由変数である

17

2.2 ラムダ計算

（例2）g(x,y) = a * x * y では、

a が自由変数で x,yが束縛変数

⇒ g = λx.λy.* * a x y

g(x) = a * x * y であるとすると、

yは aと同様の自由変数である

⇒ g = λx.* * a x y

18

2.2 ラムダ計算

- λ式の*簡約*（reduction）
  - α変換
  - β簡約
  - η変換

19

2.2 ラムダ計算

- α変換
  - ″f(x)=x*x″ と ″g(y)=y*y″ は同じ関数である
  - λ式 ″λx.*xx″と″λy.*yy″ は同じ関数 ＝ 書換え可能

    λx.*xx →$_\alpha$ λy.*yy と書く

20

2.2 ラムダ計算

- α変換は、「束縛変数の名前は、付け替えてもよい」ということを示している

21

2.2 ラムダ計算

- β 簡約
  - 関数適用
    $(\lambda x.*xx)z \rightarrow_\beta *zz$
  - それ以上 β 簡約できない λ 式を *正規形* (normal form)であるという

22

2.2 ラムダ計算

- α 変換、β 簡約では
  「変換前の自由変数が束縛変数に変化してはならない」
  例）
  $\lambda x.*xy \rightarrow_\alpha \lambda y.*yy$ ×
  $(\lambda x.\lambda y.xy)y \rightarrow_\beta \lambda y.yy$ ×

23

2.2 ラムダ計算

- 2番目の例は、
  $(\lambda x.\lambda y.xy)y \rightarrow_\alpha (\lambda x.\lambda z.xz)y$
  $\rightarrow_\beta \lambda z.yz$

24

2.2 ラムダ計算

- η 変換
  - 変数xが、λ 式M中に自由出現していない時、関数抽象 $\lambda x.Mx$ はMに変換できる

    $\lambda x.Mx \rightarrow_{\eta} M$ と書く

25

2.2 ラムダ計算

- xが Mの自由変数でない時、任意のλ 式Nについて

  $(\lambda x.Mx)N \rightarrow_{\beta} MN$

  ⇩

  $(\lambda x.Mx)$ は Mと同じ関数とみなせる

26

2.2 ラムダ計算

- P.24の例は、

$(\lambda x.\lambda y.xy)y \rightarrow_{\alpha} (\lambda x.\lambda z.xz)y$
$\rightarrow_{\beta} \lambda z.yz$
$\rightarrow_{\eta} y$

27

2.2 ラムダ計算

- λ計算の用途
  - ➢計算の意味論
  - ➢計算可能性理論
  - ➢型理論
- λ式/λ計算では、全ての「計算可能な関数」を表現し、計算することができる

28

2.2 ラムダ計算

- 演習7.1
  - ➢次のλ式を簡約せよ
    - ① (λx. λy. yx) ab
    - ② λx.λy.axy

ヒント:

29

2.3 カリー化

- *カリー化* (currying)
  - ➢"av(x,y) = (x+y)/2"という関数は、R×R→R (定義域R×R、値域R)の関係
  - ➢この関数をλ式で表すと

    av = λx. λy./+xy2

    これは λx.(λy./+xy2) の意

30

2.3 カリー化

- av = λ x. λ y./+xy2 とした時、
  ″av 3″ は、
  (λ x.(λ y./+xy2)) 3 → λ y./+3y2
  結果は1引数の関数となる
- つまり、λ 式で表した関数avは、
  R→(R→R) である
  (定義域R、値域は“R→Rの関数”)

31

2.3 カリー化

- λ 式に直すということは、
  複数引数の関数を、1引数の関数のネストに変換すること
  と、考えることが出来る

  このような変換を *カリー化* という
  (currying)

32

2.4 チャーチ=ロッサーの定理

- λ 式の簡約 (reduction)について
  - α 変換: $\rightarrow_\alpha$
  - β 簡約: $\rightarrow_\beta$
  - η 変換: $\rightarrow_\eta$

  $\rightarrow = \rightarrow_\alpha \cup \rightarrow_\beta \cup \rightarrow_\eta$ とし、
  $\rightarrow^*$ *を、その反射的推移閉包* とする

33

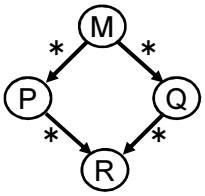
2.4 チャーチ=ロッサーの定理

- *チャーチ=ロッサーの定理*
  - λ式 M、P、Qについて、
    M →* P かつ M →* Q
    であれば、λ式 Rが存在し
    P →* R かつ Q →* R となる

34

2.4 チャーチ=ロッサーの定理

- チャーチ=ロッサーの定理の意味
  - Mが PとQに簡約されるならば、それらは共に共通のRへと到達する



35

2.4 チャーチ=ロッサーの定理

- チャーチ=ロッサーの定理から
  - 計算結果は、簡約を適用する順序に依存しない
  - 最終結果が存在するならば（つまり、計算が終了するならば）それは正規形であり、
    正規形は、一意である

36

3. Scheme

- 以降は、LISPの方言である Scheme を題材とする
  - Lispの中核部分を提供
  - 比較的小規模

37

3. Scheme

- なぜ LISPなのか
  - 最も早く開発された言語の一つ
    - 1958 McCarthyにより設計された
    - Fortranに次いで古い
  - 再帰、first-class objectとしての関数、ガベッジコレクション、形式的言語定義などを最初に実現
  - 統合プログラミング環境の先駆け

38

3. Scheme

- LISPの弱点
  - あまりに斬新な構文である
    - "Lots of Irritating, Silly Parentheses"
  - LISPの実現が非効率であった
    - (以前は)非常に計算が遅かった

39

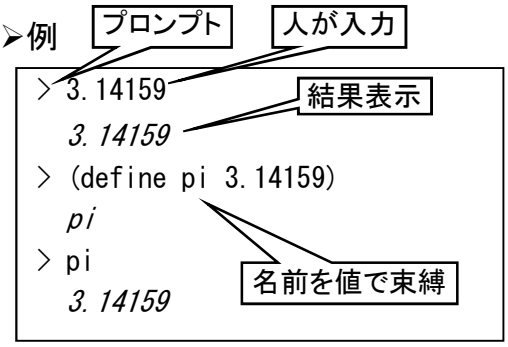
3. Scheme

- Schemeは、会話形式で動作する
  - Schemeに対して、二種類の対話を考える
    - 評価すべき式を与える
    - 名前を値で束縛する
  - Schemeは、評価した結果を表示する

40

3. Scheme

- 例



41

3. Scheme

- 式
  - 括弧の中に、演算子とオペランドを前置記法により記述する

$(E_{OP}\ E_1\ \cdots\ E_k)$

  - E_{OP}は、$E_1 \cdot\cdot E_k$に適用される演算子
  - $E_1 \cdot\cdot E_k$を評価してから、E_{OP}を適用する

42

3. Scheme

➢例

```
> (+ 2 3 5)
  10
> (+ 4 (* 5 7))
  39
> (acos -1)
  3.141592653589793
```

43

3. Scheme

- 引用
  - ➢式をデータとして扱う
  - ➢引用された項目を評価したら、その値はそれ自身となる
  - ➢次の2通りの書き方がある

```
(quote 〈item〉)
'〈item〉
```

44

3. Scheme

- ➢引用されていない名前は、ある値に束縛されている
  - 下の pi は 変数名である
- ➢引用すると、綴りとして扱える

```
> pi
  3.14159
> (quote pi)
  pi
```

45

3. Scheme

➢例

```
> (define f *)
f
> (f 2 3)
6
> (define f '*)
f
> (f 2 3)
ERROR: Bad procedure *
```

乗算関数

綴り「＊」を表す記号

46

3. Scheme

- 関数定義
  - ➢関数定義の構文

(define 〈*fname*〉 〈*lambda-expr*〉)

関数名　ラムダ式

  - ➢ラムダ式の構文は

(lambda (〈*param*〉) 〈*expr*〉)

仮引数　式(本体)

47

3. Scheme

➢例

```
> (define square
        (lambda(x) (* x x)) )
square
> (square 5)
25
```

48

3. Scheme

```
(define square
        (lambda(x) (* x x)) )
```

は、次の defineと同じ構造

```
(define  pi   3.14159)
```

λx.*xx というλ式(関数) と、
3.14159という実数が、同じ立場

49

3. Scheme

- 述語と論理値
  - 論理値 true / false は、
    #t / #f
    と表記する

50

3. Scheme

- *述語*(predicate)
  - ●何らかの関係や事実を述べる語
  - ●真(#t)か偽(#f)に評価される
  - ●Schemeの述語名は、？で終わるのが習慣

number?	引数は数か
symbol?	引数は記号か
equal?	引数同士は同値か

51

3. Scheme

➢述語の例

```
> (define a 1)
  a
> (number? a)
  #t
> (equal? a 2)
  #f
```

52

3. Scheme

- 条件式
  - ➢条件式の形式(1)

(if P E_1 E_2)

"if P then E_1 else E_2"

Pが真ならばE_1を評価し、偽ならばE_2を評価する

53

3. Scheme

➢条件式の形式(2)

(cond (P_1 E_1)
・・・
(P_k E_k)
(else E_{k+1}))

"if P_1 then E_1 else if ・・・
else if P_k then E_k else E_{k+1}"

54

3. Scheme

- 条件式の利用例 ＝ 再帰

```
(define fact
   (lambda (n)
     (if (<= n 1)  1
       (* n (fact (- n 1))) )))
```

55

3. Scheme

- 演習7.2

  フィボナッチ数を求める関数 fibを、定義せよ

  fib(0) = 0, fib(1) = 1

  fib(n) = fib(n−1) + fib(n−2)

  （ただし、n≧2）

56

4. リスト

- *リスト* (list) とは
  - ゼロ個以上の値の並び
  - どのような値もリストの要素となりえる
  - LISP系の言語では、*プログラムとデータは同じ形* をとる
    ⇒ リストとして表現される

57

4. リスト

- リストは、要素を括弧で囲み空白で区切ることにより表す
  - 例: (you like me)
- *空リスト* (empty list / null list)
  - ゼロ個の要素を持つリスト
    () と書く

58

4. リスト

- Schemeにおける、リストの要素
  - 論理値(ブール値)
  - 数
  - 記号
  - 関数
  - 文字、文字列
  - ベクトル
  - リスト

59

4. リスト

- リストの例

```
(it seems that you like me)
((it seems that) you (like) me)
(a ())
(a)
```

要素が6ヶ
要素が4ヶ
この二つは異なる

60

4. リスト

- *アトム* (atom) とは
  - 基本となるデータ
    それ以上分解できないデータ
  - アトムの種類
    - 記号アトム
    - 数アトム
    - 空リスト

61

4. リスト

- *ドット対* (dotted pair)
  - 二つのデータ S_1と S_2の対を、
    $(S_1 . S_2)$
    と書き、*ドット対* と呼ぶ

  （ドット．の両側には一つ以上の
  空白を入れなければならない）

62

4. リスト

- *S式* (Symbolic expression) の定義
  - アトムは S式である
  - S式同士のドット対は S式である

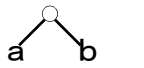
↑ 再帰的な定義になっている

例） a （a . b） （you . (like . me)) 等

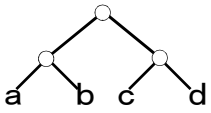
63

4. リスト

- S式は二分木であり、アトムが葉である



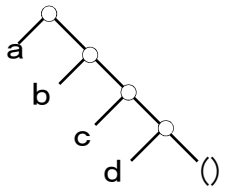
(a . b)



((a . b) . (c . d))

64

4. リスト



(a . (b . (c . (d . ( )))))

65

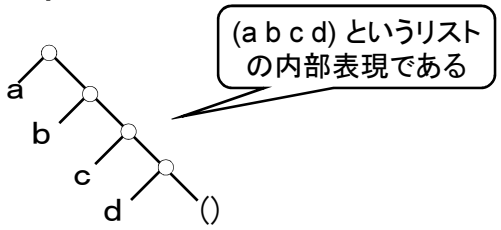
4. リスト

- S式
  (a . (b . (c . (d . ( )))))
  を
  (a b c d)
  と書き、リストと呼ぶ

リストのもう一つの定義

66

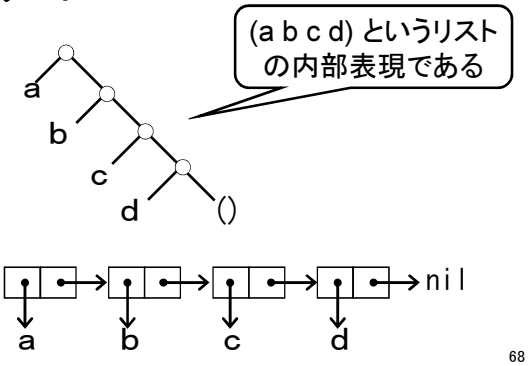
4. リスト



(a . (b . (c . (d . ( )))))
= (a b c d)

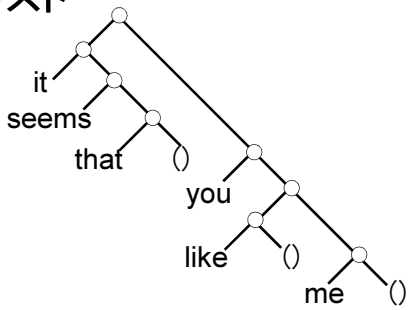
67

4. リスト



68

4. リスト



((it seems that) you (like) me)

69

4. リスト

- 演習7.3
  - 次のリストを、二分木で表わせ
  - また、それをS式で表現せよ

(1) `((i) love cats)`

(2) `(that is (the question))`

70

5. リストの操作

- リストに対する基本的な演算

(null? x) : xが空リストであれば真
さもなければ偽

(car x) : 非空リストxの最初の要素

(cdr x) : リストxから最初の要素を除いた残りのリスト

71

5. リストの操作

(cons a x) : carがaで cdrがxであるような値(リスト)を作成する

【例】 (cons a (b c)) = (a b c)

(car (cons a x)) = a

(cdr (cons a x)) = x

72

5. リストの操作

- car と cdr を連続して使用する場合は、省略形を使用できる

  (car (cdr x)) ⇒ (cadr x)

  (cdr (car x)) ⇒ (cdar x) など

73

5. リストの操作

- car と cdr の値 (1)

  (define x
  '((it seems that) you (like) me))
  と定義

式	値
x	((it seems that) you (like) me)
(car x)	(it seems that)
(cdr x)	(you (like) me)

74

5. リストの操作

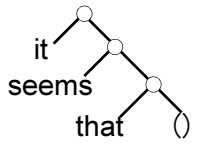
- car と cdr の値 (2)

式	省略形	値
(car (car x))	(caar x)	it
(cdr (car x))	(cdar x)	(seems that)
(car (cdr x))	(cadr x)	you
(cdr (cdr x))	(cddr x)	((like) me)
(car (cdr (cdr x)))	(caddr x)	(like)

75

5. リストの操作

- car / cdr の意味
  - car はドット対の左側（二分木の左の枝）、cdrはドット対の右側（二分木の右の枝）をとる

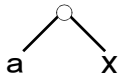


(it seems that)

76

5. リストの操作

- cons の意味
  cons演算子は、ドット対を生成する
  （cons a x） ⇒ (a . x)
  x がリストならば、(cons a x) もリストとなる



77

5. リストの操作

- 演習7.4
  x を次のように定義する

```
(define x  '((to be) or (not to be)
             that is (the question) ))
```

このとき、次の値を求めよ

(1) `(car (cdr x))`

(2) `(cdr (cdr (cdr x)))`

78

6. プログラム例

- いくつかの実例を見てみる

79

6. プログラム例

- リストの長さ(要素の数)を求める

(length ()) ≡ 0

(length (cons a x)) ≡ (+ 1 (length x))

```
(define length (lambda (x)
   (cond  ((null? x)  0)
          (else (+ 1 (length (cdr x)))) )) )
```

80

6. プログラム例

➢実行例

```
> (length  '((it seems that) you (like) me))
 4
```

81

6. プログラム例

- リストを逆順に並べ替える
  - 次のような等式による

(reverse x) ≡ (rev x ())
(rev () z) ≡ z
(rev (cons a y) z) ≡ (rev y (cons a z))

82

6. プログラム例

```
(define reverse (lambda (x) (rev x ())))

(define rev (lambda (x z)
  (cond ((null? x) z)
      (else (rev (cdr x) (cons (car x) z))) )))
```

83

6. プログラム例

- 実行の流れ

(reverse '(a b c d))
= (rev '(a b c d) ())
= (rev '(b c d) '(a))
= (rev '(c d) '(b a))
= (rev '(d) '(c b a))
= (rev () '(d c b a))
= '(d c b a)

84

6. プログラム例

- より大きなプログラムの例が、Webサイトにある

7.【資料】サンプルプログラム

85

【参考】Scheme 参考URL

- Racket
  http://racket-lang.org/
- 紫藤のページ
  http://www.shido.info/
- Functional Programming
  http://www.geocities.jp/m_hiroi/func/scheme.html

86

お疲れ様でした