プログラミング言語論

第7回
関数型言語

担当：犬塚

（第6回は欠番）

今日の講義

関数型プログラミング言語

- その特徴
- ラムダ計算
- LISP入門
- LISPプログラミング

ラムダ計算
λ（lambda）-calculus

- 論理学者（今から見れば、理論計算機科学者）のA.チャーチが考案した計算モデル。
- 計算について理論的に考えるときに使う標準体系の一つ。（チューリング機械もその1つ）
- 関数型言語の理論的基礎。

- 関数の合成と適用によって計算を表現。
- 関数と対象を区別しない。

λを使った関数の表現

- 通常用いる関数の表現：

  $f(x) = x^2 + x + 1$

  - 引数を $f(x)$ などと、括弧付きで表す。
  - 関数の名称 f などでつけて表す。名称が必要。
  - 関数を適用する場合はこの名称を用いて、$f(4)$などとする。
- λ計算では、上の関数を次のように書く。

  $\lambda x . x^2 + x + 1$

  あるいは、型付のλ計算では次のように書く。

  $\lambda x \in \mathrm{N} . x^2 + x + 1$

λを使った関数の表現

$\lambda x.\ x^2 + x + 1$

- 関数名を付けずに、関数を表現できる。
- そのまま適用できる。

  $((\lambda x.\ x^2 + x + 1)4) = 4^2 + 4 + 1 = 21$
- 名前をつけることもできる。

  $f \equiv \lambda x.\ x^2 + x + 1$

  $(f\ 4) = 21$
- 名前なしの関数を、無名関数という(LISPの用語)

(純粋) λ計算の構文

定義: λ式

1. 変数 $x, y, z, \ldots$ はλ式である。
2. Mがλ式, x が変数のとき $(\lambda x.\ M)$はλ式である。
3. M, Nがλ式のとき (MN)はλ式である。

括弧は適当に省略する。

2の構文を関数抽象(functional abstraction)、
3の構文を関数適用(functional application)という。

- 関数抽象は式Mに対し、変数xが引数だと教えること。
- 関数適用はまさに関数MにNを適用すること。

α変換とβ変換

M中の x をすべて y に置換える操作を $M\{y/x\}$と書く。

α変換: 引数と関数本体中の変数を同時に置換えても式の意味は変わらない。

$(\lambda x.\ M) = (\lambda y.\ M\{y/x\})$,

ただしyはMに現れない変数。

β変換: 関数適用の形式で、実際に適用を行っても意味は変わらない。

$((\lambda x.\ M)N) = M\{N/x\}$

例1

純粋λ計算は、＋とか－はまったく出てこないけれど、直感的な例を示す。

- M ≡ $(\lambda x.\ x + x)$ のとき、

  $(M\ 2) = ((\lambda x.\ \boxed{x} + \boxed{x})\ 2)$

  $= 2 + 2$ (β変換)

  $= 4$ ←

※) ← の箇所は、α変換でもβ変換でもない。
つまり、＋や＊などの組込み計算。
これをδ変換ということもある。

- M ≡ $(\lambda x.\ x + x)$ 、N ≡ $(\lambda x.\ 5 * x)$のとき、

  $(M(N\ 3)) = ((\lambda x.\ x + x)\ ((\lambda x.\ 5 * \boxed{x})\ 3))$

  $= ((\lambda x.\ x + x)\ (5 * 3))$ (β変換)

  $= ((\lambda x.\ \boxed{x} + \boxed{x})\ 15)$ ←

  $= 15+15$ (β変換)

  $= 30$ ←

例1　つづき

前の例は、別のやり方もある。

$(M(N\ 3)) = ((\lambda x.\ x + x)((\lambda x.\ 5 * x)\ 3))$

$= ((\lambda x.\ \boxed{x} + \boxed{x})((\lambda y.\ 5 * y)\ 3))$（$\alpha$ 変換）

$= \boxed{((\lambda y.\ 5 * y)\ 3)} + \boxed{((\lambda y.\ 5 * y)\ 3)}$（$\beta$ 変換）

$= 15 + 15$（δ 変換）

（※ このように、どの部分から計算しても結果が同じという性質を合流性（チャーチ＝ロッサー性）という。）

例2（多引数の関数）

純粋λ計算では引数は1つだが、多引数も省略法として許す。

- $M \equiv \lambda xy.\ x + y$ のとき、

$(M\ 2\ 5) = ((\lambda xy.\ x + x)\ 2\ \ 5)$

$= 2 + 5 = 7$

つまり、次の形式の省略。

- $M \equiv (\lambda x.(\lambda y.\ x + y))$ のとき、

$(M\ 2\ 5) = ((M\ 2)\ 5)$

$= (((\lambda x.(\lambda y.\ x + y))\ 2)\ \ 5)$

$= ((\lambda y.\ 2 + y)\ 5)$

$= 2 + 5 = 7$

例3（関数引数）

- $M \equiv \lambda f.(\lambda x.(f\ (f\ x)))$
- $N \equiv \lambda y.\ y * y$

のとき、

$((M\ N)\ 3)$

$= (((\lambda f.(\lambda x.(\boxed{f}(\boxed{f}\ x))))(\lambda y.\ y * y))3)$

$= ((\lambda x.((\lambda y.\ y * y)((\lambda y.\ y * y)x)))3)$

$= ((\lambda y.\ y * y)((\lambda y.\ y * y)\ 3))$

$= ((\lambda y.\ y * y)(\lambda y.\ 3 * 3))$

$= ((\lambda y.\ \boxed{y} * \boxed{y})9)$

$= 9 * 9$

$= 81$

一級市民としての関数、汎関数

- λ計算では値も関数も区別なく、なんでも関数に適用できる。
  これを関数も一級市民（first-class citizen）であるという。
- 関数は、値だけでなく関数を値として取れる。
  つまり、前の例では

  N ： 自然数の集合→自然数の集合

  M ： {（自然数の集合→自然数の集合）
  のタイプの関数の集合
  ×　自然数の集合}→自然数の集合

- Mは、関数を受取って動作する関数である。
  ＝こうしたものを汎関数（functional）
  または高階の関数（higher order function）という。

多引数の関数と汎関数

- 2引数の関数 $f: N \times N \to N$
  は、λ式では例えば次のような定義になる。
  $\lambda x.(\lambda y.x + y)$
- この式は、1つの値、例えば3、を受取ると、
  $((\lambda x.(\lambda y.x + y))\ 3)$
  $= \lambda y.3 + y$
  となる。1つの値を適用することで1引数関数となった。
- つまり、2引数関数 $f: N \times N \to N$ は、
  1引数の汎関数 $f: N \to (N \to N)$ と見なせる。

※このような扱いを、関数のカリー(Curry)化という。

練習

- L $\equiv \lambda f g x.(g\ (f\ x)\ (f\ x))$
- M $\equiv \lambda y.y * y$
- N $\equiv \lambda xy.x + y$

のとき、

(L M N 3) = $((\lambda f g x.(g\ (f\ x)\ (f\ x)))$ M N 3)
= (N(M 3) (M 3))
= $((\lambda xy.x + y)\ ((\lambda y.y * y)\ 3)\ ((\lambda y.y * y)\ 3))$
= $((\lambda xy.x + y)\ 9\ 9)$
= 18

練習

- L $\equiv \lambda f g x.(g\ (f\ x)\ (f\ x))$
- M $\equiv \lambda y.y * y$
- N $\equiv \lambda xy.x + y$

のとき、

(L M N 3)
$= ((\lambda fgx.(g(fx)(fx)))(\lambda y.y * y)(\lambda xy.x+y)3)$
$= ((\lambda xy.x+y)\ ((\lambda y.y * y)\ 3)((\lambda y.y * y)\ 3)))$
$= ((\lambda xy.x+y)\ ((\lambda z.z * z)\ 3)((\lambda z.z * z)\ 3)))$
$= ((\lambda z.z * z)\ 3) + ((\lambda z.z * z)\ 3)$
$= (3 * 3) + (3 * 3) = 18$

LISP

- 関数型プログラミング言語
- J. McCarthyの設計, 1959。実装1962。
- λ計算を基礎理論とする。

- Lisp : List Processor
- 記号処理、人工知能用。

- 純粋にλ計算に基づく関数型言語 純LISP
- 実用的な実際のLISP: 関数以外の要素を持つ
  副作用

LISP入門

LISPでは関数の適用は、次の形式を取る: 前置き記法
　（関数名　引数1　引数2 … ）
算術関数は用意されている。
　（＋　2　7）　→　9
関数の定義
　(defun f (x) (+ x 2))　⇒　f　＝　（λx. (+ x 2))
　式本体 x+2 を変数で関数抽象した λx.x+2に
　fという名をつけている。
関数の適用
　(f 5) →　7
LISPでは、式を計算する（評価する）ことを、eval(エバル)するという。

LISP入門

関数の定義
　（defun f (x) (+ x 2))
　式本体 x+2 を変数で関数抽象した λx.x+2に
　fという名をつけている。
関数の適用
　(f 5) →　7
LISPでは式を計算（評価）することを、eval(エバル)するという。
(f 5)をevalして、7となった。

二引数も同様
　（defun sum (x y) (+ x y))

練習

（1）(defun dbl (x) (+ x x))
　　(defun sqr (x) (＊ x x))
　　(defun sum (x y) (+ x y))
　のとき、次の式をエバルせよ。
　① (dbl 4)
　② (sum (dbl 5) (sqr 3))

（2） 関数 $g(x, y)=x^3+2xy-4$ を定義せよ。

(defun g(x y) (- (+ (* x x x) (* 2 x y)) 4))

LISPでも関数は一級市民

(defun dbl (x) (+ x x))
(defun sqr (x) (＊ x x))

(dbl 3) →　6　　(sqr 3) →　9
これは次の形式でも同じ。
(funcall 'dbl　3) →　6　(funcall 'sqr　3) →　9
「'」はクォートといい、これをつけるとdblの値（束縛された関数）でなく、dblという変数そのものを意味する。

(defun twice (f x) (funcall (f (f x))))
(twice 'dbl 3) →　12
(twice 'sqr 3) →　81

lambda 式

λ式そのものもLISPでは使える。
M= λ *x y*. x+2y に対して、(Ｍ３４)を計算するには、

```
(funcall
   #'(lambda (x y) (+ x (* 2 y)))
   3 4)
```

とかく。

つぎの式はどうなるか？
(twice #'(lambda (x) (＊ (+ x 1) 2)) 4)

練習

(distribute f g x)とあたえると、(f (g x) (g x))を計算する汎関数 distributeを定義してみよ。

```
(defun distribute (f g x)
  (funcall f (funcall g x) (funcall g x))
)

(distribute #'(lambda (x y) (+  x y))
            #'(lamnda (x) (* x x)
            3)
```


まとめ

- λ計算は関数を中心にいた計算モデル。
  - 関数抽象、関数適用
- 関数と値を区別しない ― 関数も一級市民
- その場合、関数を引数にとる関数となる＝汎関数
- LISPはλ計算に基づいた関数型言語
  - 関数定義 ― defun式
  - 関数抽象 ― lambda式
  - 関数の適用 ― funcall関数
- LISPでも、関数は一級市民