

プログラミング言語論

第8回 LISP

担当: 犬塚

1

今日の講義

LISP入門

- リスト構造とリストの操作
- 基本演算
- 基本的な制御構造
- 純粹関数型に反する機能

LISPは非常に多数の方言があるが、common lispとして統一規格が出来以来、これが主流。

2

LISPのデータ

- 変数ではなくデータ(オブジェクト)に型がある。
(実際には効率等の面から変数にも型を与えられる)
- LISPのデータ型: 表現によって決まる。
 - 数型(整数型、分数型、浮動小数点型、複素型)
 - 文字型 #¥a #¥newline
 - シンボル型 abc (= aBc)
 - コンス型
 - 関数型
 - その他コンス以外をアトムという。

3

リスト

LISPのリストは

- ()で要素を括った形式、
- 要素は空白で区切る。
- 要素にはLISPのどんなデータも入れることができる。
- リスト自身を要素としてもよい。
例 (1 2 3 4) (a (b c) d (e (f)))
- 空リストは、()あるいはnilと表す。
- リストは用途は決まっていない。好きにデータ構造を表せる。
例 (taro
 (fullname suzuki taro)
 (hight 172)
 (weight 67))

4

リストの基本演算 car, cdr, cons

- LISPの基本演算に car, cdr cons がある。
- car(カー), cdr(クダー)は当時の計算機のレジスタに由来する名称、cons(コンス)はconstruction。

(car '(a b c d)) a

: 先頭要素を返す

(cdr '(a b c d)) (b c d)

: 先頭要素を取り除いたリストを返す

(cons 'a '(b c d)) (a b c d)

: 第2引数のリストの先頭要素として、第1引数の要素を加えたリストを返す。

5

練習

- 以下の式のopに適切にリスト演算を入れて、aからeの各記号が答えになるようにせよ。

(op '(a ((b) c) (d (e))))

7

car, cdr の組合せ

- 2つ目の要素を取り出すには、
(car (cdr '(a b c d))) b
- 1つ目の要素であるリストの第2要素は、
(car (cdr (car '((a b) c d)))) b

- こうした演算はよく使うので、省略形がある：

(cadr '(a b c d)) b カードゥル

(cadar '((a b) c d)) b カダール

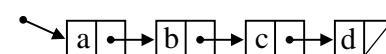
cとrの間にa,dを適当に並べるとこれらの演算を意味する
(たいてい4つ程度までサポート)

6

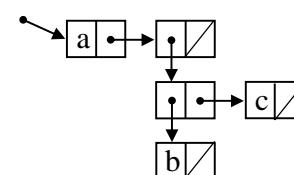
リストは連結リスト

- LISPのリストは連結リストで構成されている。

(a b c d e)



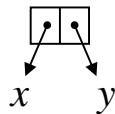
(a ((b) c))



8

コンス、点対

- LISPでは `c` をセルまたはコンスという。
- `(cons x y)` でコンスが1つ作られる。



- コンスの左をたどるのがcar、右をたどるのがcdr。
- consによってリストを表現できる。

例 `(cons 'a (cons 'b (cons 'c nil)))`
`(a b c)`

`(cons 'a (cons (cons (cons 'b nil) (cons 'c nil)) nil))`
`(a ((b) c))`

9

コンス、点対

- コンスを中置の「.」で表す形式もある。
- $(\text{cons } x \text{ } y) = (x \text{ . } y)$
 $'(a \text{ . } (b \text{ . } (c \text{ . } \text{nil}))) \quad (a \text{ } b \text{ } c)$
 $'(a \text{ . } (((b \text{ . } \text{nil}) \text{ . } (c \text{ . } \text{nil})) \text{ . } \text{nil})) \quad (a \text{ } ((b) \text{ } c))$

- この形式を点対(dot pair)という。
- 最後がnilでないリストは点対で表す。

$'(a \text{ . } (b \text{ . } (c \text{ . } d))) \quad (a \text{ } b \text{ } c \text{ . } d)$

10

練習

- アトム(aや1のように記号のみからなるデータ)からcons を用いて、次のリストを作れ。また点対で表現せよ。
`(a (b) c)`

`(1 (2 (3 (4 . 5))))`

11

LISPの基本的制御構造

- マッカーシーの条件式
`(cond (条件 値) ... (条件 値))`
左から順に調べ最初に真となったときの値が返る。
真となるものが無い場合はnil。
- 条件には任意の式が使える。式が値nilのとき偽、それ以外では真を表す。
- 条件に用いる関数 = 述語: 末尾にpがつくものが多い。
`(zerop x)` xがゼロであれば真。
`(null x)` xが空リストのとき真。
`t` いつも真であることを表す定数に束縛されている。

12

condを用いたプログラム例

例 つぎの関数absは絶対値を求める。

```
(defun myabs (x)
  (cond ((plusp x) x) ; x>0 なら x
        (t (* x -1) ; それ以外は -x
    ))
```

□ 再帰的な定義と組合わせると多様な関数が定義できる。

```
(defun fact (x)
  (cond ((zerop x) 1)
        (t (* x (fact (- x 1))))
```

$$x! = \begin{cases} 1 & ; x=0 \\ x(x-1)! & ; x>0 \end{cases}$$

13

再帰定義を用いたリスト操作

□ 再帰によるリスト操作はLISPの基本的技法である。

□ リストXとYの連結 appendは次のように書ける。

```
(defun myappend (x y)
  (cond ((null x) y)
        (t (cons (car x)
                  (myappend (cdr x) y))))
```

```
(myappend '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
```

14

Myappendのトレース

□ (defun myappend (x y)
 (cond ((null x) y)
 (t (cons (car x)
 (myappend (cdr x) y))))))

実行例:

```
(myappend '(a b c) '(d e f))
(null '(a b c)) は偽 (cons 'a (myappend '(b c) '(d e f)))
(myappend '(b c) '(d e f))
(null '(b c)) は偽 (cons 'b (myappend '(c) '(d e f)))
(myappend '(c) '(d e f))
(null '(c)) は偽 (cons 'c (myappend '() '(d e f)))
(myappend '() '(d e f))
(null '()) は真 '(d e f)
(cons 'c (myappend '() '(d e f))) = (cons 'c '(d e f)) = (c d e f)
(cons 'b (myappend '(c) '(d e f))) = (cons 'b '(c d e f)) = (b c d e f)
(cons 'a (myappend '(b c) '(d e f))) = (cons 'a '(b c d e f)) = (a b c d e f)
```

15

練習 次のリスト操作関数を定義せよ

□ リストxの最後の要素を取り出す (mylast x)

□ 数のリストxの要素の合計得る (mysum x)

16

LISPの関数呼出しと関数定義

- 関数適用は、前置き表現で行われる。

(関数 値 値 ...)

LISPは値には型があるので、関数を適用する際に型違反であれば、エラーとなる。

- 関数定義は次の形式で定義される。

(defun 関数名 (x y ...) 関数本体)

例 (defun myplus (x y) (+ x y))

17

LISPの関数呼出しとクオート

- LISPの式は、すべてリスト形式で表される。

- すると、(a b c d)としたとき次のものが区別できない。

- 4つの要素からなるリスト
- 関数aに対するb c dの適用する
- b c dもシンボルそのものか、その値か、

- オブジェクトxを評価せず、そのままのデータとしたいとき (quote x) とかく。'x はその省略記法。

したがって、

- (a b c d)とかけば、関数aに変数b c dの値を適用。
- (a 'b 'c 'd)とかけば、関数aにシンボルb c dを渡す。
- '(a b c d)と単にリスト。

18

その他の制御構造

- 関数定義において、途中結果を変数に置きたい場合がある。

$f(x) = \max(y, z)$, ただし $y=x^2$, $z=10x$

- こうした局所的変数は次の let 形式で扱える。

```
(let (  
    (変数 式) ... (変数 式)  
    )  
  本体式)
```

例 (defun func (x)
 (let ((y (* x x)) (z (* 10 x)))
 (max y z)
))

- letは変数の値を順に計算し、前の計算結果をその後の計算に反映させる。反映させないバージョンにlet*がある。

19

本来の関数型に反する機能

- 変数への代入:

- シンボルに属性を付加する形で、代入が可能 - setq
- それ以外にシンボルに様々な属性を与えることができる (plist)

- 手続き的プログラムの機能 - progn

- 破壊的リスト操作 - rplaca, rplacd

- 入出力関数など - print, format

20

set, setq

- setは、シンボルに値を割り当てる。
`(set 'x '(1 2 3))`
- 第1引数のシンボルはたいていいつもquoteされるので、これを含めた形式として次を許す。
`(setq x '(1 2 3))`
- setqでなくsetを使うのは、たとえば次の場合。
`(setq x 'a)`
`(set x '(1 2 3))` xにはaが、aに(1 2 3)が代入される。

21

progn形式

- LISPも手続き型言語と同様の形式を持つ。
`(progn 式 式 … 式)`
- 式を順に評価し、最後の式の値が全体の値になる。
- 最後の式以外の値は捨てられる。
- これが意味があるのは、途中の式がsetqのような副作用のある形式の場合のみ。
- さらに、ラベルとgoto文も持つ。
- let形式の本体式は、複数置くことができる。この場合の解釈はprognと同じ = implicit prognという。

22

破壊的関数:rplaca, rplacd

- rplaca (ルプラッカ、replace car), rplacd (ルプラクディ、replace cdr)は リストを破壊的に操作する
- 副作用を目的としている。

```
(setq x '(a b c d))
(rplaca x 1)      x=(1 b c d)
(rplacd x '(2 3)) x=(1 2 3)
```

23

メモリ構造

- LISPは関数型言語という抽象的性格の反面、メモリ使用を気にする必要がある。
`(setq x '(1 2 3))
(setq y (cons 100 x)) y=(100 1 2 3)
(rplaca x 99) x=(99 2 3) y=(100 99 2 3)
(setq x '(4 5)) x=(4 5) y=(100 99 2 3)`
- データの等価性も2種類(実際は更に多数)ある。
 - (eq x y) 同じメモリ領域または同じアトム
 - (equal x y) 印字名が同じ

24

練習

次を順に評価したとき、その式およびx,yの値どうなるか答よ？

```
(setq x '(nagoya inst tech))  
(setq y (cons x x))  
(eq x (cadr y))  
(rplacd x 'cs)  
(rplaca x 'nit)  
(eq x (car y))  
(eq (cons x x) y)
```

25

練習

□ リストを使って多数の人物のデータを記録している。
(データ1 データ2 …)

□ 各データは次の形式。
(人物名 (属性1 データ) (属性2 データ) …)

□ この形式のデータリストLに対し、次の関数を書け。
(getdata L 人物名 属性)

で人物のその属性を返す関数。

(putdata L 人物名 属性 値)

属性を値で書換える関数。属性がなければ追加する。

(putdataL 人物名 属性 値)

putdataと同様の機能を、グローバルな変数Lの内容を書換える方法で実現せよ。

26

まとめ

- LISPは代表的関数型言語。リスト処理を中心とし、記号処理が得意。
- 統一規格common lispは広範囲に用いられている。
- 関数の定義、適用が計算の中心。
- 純粋な関数型に反する仕組みを多数用意する。

追加

- LISPは変数宣言を意識する必要がない。そのため、メモリの回収の仕掛けが重要 = ガーベージコレクション。

27