Lisp プログラミング入門

田村直之

2016-12-14

目次

1 Lisp の概要

- 1957 年頃に米国 MIT の John McCarthy らにより開発された
- LISt Processor (リスト処理言語) の省略で，AI プログラム等の記述・開発に適している

- 多数の方言が存在するが，標準版として 1984 年に Guy L. Steele Jr. が中心となって Common Lisp を設計した
- Lisp 言語の一種である Scheme も広く用いられている
- Clojure という新しい言語も出ている

1.1 参考リンク

- An Introduction to Programming in Emacs Lisp
- On Lisp
- 初めての人のための LISP (Amazon, OPAC)
- Scheme
  - The Scheme Programming Language (Amazon, OPAC)
  - プログラミング言語 SCHEME (Amazon)
    The Scheme Programming Language の日本語訳
  - Structure and Interpretation of Computer Programs (SICP) (Amazon, OPAC)
    大変有名な教科書
  - 計算機プログラムの構造と解釈 (Amazon, OPAC)
    SICP の日本語訳 (HTML 版)
  - Gauche プログラミング (立読み版) (at Karetta)
- Clojure
  - プログラミング Clojure (Amazon, OPAC)
- プログラミング言語論
  - Scala プログラミング入門
- 神戸大学工学部システム工学科第 4 講座で開発された計算機
  - 神戸大学 Lisp マシン，Prolog マシンなどを紹介

2 Lisp の特徴

記号処理言語， リスト処理言語 データとして，記号 (シンボル) を取り扱うことができる．また，リストと呼ばれる可変長のデータの列を取り扱うことができる．

関数型言語 Lisp では新たな関数を定義することによってプログラムを作り上げていく．すなわち，Lisp のプログラムは関数定義の集まりである．Lisp や Prolog は，FORTRAN や BASIC などの手続き型言語とは異なり，非手続き型言語と呼ばれる．

対話的使用， 会話形使用 Lisp システムの使用形態は対話的である．すなわち，ユーザは Lisp システムを立ち上げたあと，システムと会話するような形で命令を与え，関数を定義したり実行したりできる．

3 記号

Lisp, Prolog などの言語では，**記号** (シンボル，symbol) をデータとして取り扱うことができる．記号を取り扱える言語は **記号処理言語** と呼ばれることがある．

記号処理言語はデータとして数値だけしか取り扱えない言語に比較して，より自然に問題を表現できる．

記号は英数字等を 1 文字以上続けた名前で表される．ただし，数値として見なされるもの (`123`, `+1` など) は省く．

```
X   coffee   B52   -   N-Tamura
```

英字の大文字と小文字は区別しない．いくつかの Lisp システムでは漢字も使用できる．
記号および数値は **アトム** (atom) と呼ばれる．

4 リスト

Lisp, Prolog などの AI 用言語では，記号や数値の列をデータとして取り扱うことができる．
この列は **リスト** (list) と呼ばれる．リストは，その **要素** (element) を空白で区切って並べたものをカッコでくくって表す．たとえば

```
(jan 31 1957 thu)
(pi 3.14)
```

であり，一般的には

$$(\text{要素}_1\ \text{要素}_2\ \ldots\ \text{要素}_n)$$

と書く．

- リストの要素の個数 n は，そのリストの **長さ** (length) と呼ばれる．前の例では，リストの長さはそれぞれ 4 と 2 である．
- 要素 $_i$ をリストの **第 i 要素** (i-th element) と呼ぶ．たとえば，最初の例での第 3 要素は `1957` である．
- 要素のないリスト (長さが 0 のリスト) は **空リスト** (empty list) と呼ばれる．空リストは記号 `nil` で表すこともできる．

  ```
  ()
  nil
  ```

- リストの要素が，またリストであってもよい．

  ```
  (coffee milk (orange juice))
  ((a (b) (c (d))) (e () f))
  ```

  上の例でのリストの長さはそれぞれ 3 と 2 である．

4.1 練習問題

1. リスト `(a () b)` の長さはいくつか．
   (解答例) 3
2. リスト `((a b) c)` の第 1 番目の要素は何か．
   (解答例) `(a b)`

5 Lisp を使ってみる

ここでは，Emacs Lisp を使って説明する．
Emacs 上で実行するには，Buffers メニューから `*scratch*` バッファを選択し，入力の最後に `Ctrl-J` をタイプすれば良い．

```
(+ 1 2) [Ctrl-J]
  3
```

このように，ユーザがシステムと対話しながらシステムを利用することから，このような使用形態は **対話的使用** と呼ばれる．

ユーザが入力する式は，一般的には

$$(\text{関数名 式}_1 \text{ 式}_2 \ldots \text{ 式}_n)$$

という **関数** (function) の形である．

- 式 $_i$ は **引数** (argument) と呼ばれる．引数が，また式であってもよい．たとえば，関数 log sin 1 の値は次のようにして計算できる．

```
(log (sin 1))
  -0.1726038
```

- リストを引数とする関数も使用できる．リストを引数とする場合には，リストの前に**クォート** (引用符，クォーテションマーク，quotation mark) をつける．
- たとえば，

```
(length '(coffee milk (orange juice)))
  3
(length '((a (b) (c (d))) (e () f)))
  2
```

  ここで， `length` はリストの長さを求める関数である．
- `+`, `*`, `log`, `length` などのように，Lisp システムであらかじめ用意されている関数は**組込関数** (built-in function) と呼ばれる．

5.1 練習問題

1. 与えられた 2 つのリストの長さの合計を求めるにはどうすればよいか．
   (解答例) たとえば以下のようにする．
   `(+ (length '(a b c)) (length '(d e)))`

6 変数

関数 `setq` を使うと，記号に値を代入できる．

```
(setq x (* 12 34))
  408
(+ x 56)
  464
(setq menu '(tea coffee milk))
  (tea coffee milk)
(length menu)
  3
(setq x (length menu))
  3
```

一般的には

(setq 変数 値)

と書く．変数としては任意の記号，値としては任意の式が書ける．

7 リスト処理関数

数多くあるリストを処理する組込関数のうち，いくつかを紹介する．

- append は 2 つのリストを連結したリストを求める関数である．なお空リスト () はクォートを省いてもよい．

```
(append '(a b) '(c d))
  (a b c d)
(append '(a (b c)) '(d () e))
  (a (b c) d () e)
(append '(a b c) '())
  (a b c)
(append () '(a b c))
  (a b c)
```

- reverse はリストの要素を逆順に並べ変えたリストを求める関数である．

```
(reverse '(a b c d))
  (d c b a)
(reverse '(a (b c) d))
  (d (b c) a)
```

- sort はリストを一定の順序 (昇順，降順，アルファベット順など) にしたがって並べ変えたリストを求める関数である．

```
(sort '(3 1 4 2) '<)
  (1 2 3 4)
(sort '(3 1 4 2) '>)
  (4 3 2 1)
(sort '(tamura banbara kumamoto) 'string<)
  (banbara kumamoto tamura)
```

これらの関数は，以下で述べるようなより基本的な関数の組合せで作られている．

8 基本リスト処理関数

- 先頭要素を求める関数 car
- 残りのリストを求める関数 cdr
- 先頭に要素を加えたリストを作る関数 cons

8.1 先頭要素を求める関数 car

関数 car はリストの先頭の要素 (第 1 要素) を求める．

```
(car '(a b c))
  a
(car '((a b) (c d)))
  (a b)
(setq menu '(tea coffee milk))
  (tea coffee milk)
(car menu)
  tea
```

8.2 残りのリストを求める関数 cdr

関数 cdr はリストの先頭の要素を除いた残りのリスト (第 2 要素以降のリスト) を求める.

```
(cdr menu)
  (coffee milk)
(cdr (cdr menu))
  (milk)
(cdr (cdr (cdr menu)))
  nil
(car (cdr menu))
  coffee
```

- (car (cdr menu)) で menu の第 2 要素を求めることができる.

8.3 先頭に要素を加えたリストを作る関数 cons

関数 cons は，第 1 引数を第 2 引数のリストの前につけたリストを求める.

```
(cons '(a b) '(c d))
  ((a b) c d)
(cons 'cocoa menu)
  (cocoa tea coffee milk)
(cons 'cocoa (cdr menu))
  (cocoa coffee milk)
```

- (cons '(a b) '(c d)) の結果は ((a b) c d) で，(append '(a b) '(c d)) の結果 (a b c d) とは異なることに注意する.
- 第 2 引数がリストでない場合も cons を実行することは可能だが，リストでないデータ構造が得られる．詳しくは Lisp のデータ構造 を参照.

リストに対する基本的な操作は，以上の car, cdr, cons の 3 つの関数で行える.

8.4 練習問題

1. menu の 3 番目の要素を求めるにはどうすればよいか.
   (解答例) (car (cdr (cdr menu)))

1. menu の第 1 要素を juice に変更したリストを作るにはどうすればよいか.

(解答例) (cons 'juice (cdr menu))

1. menu の第 1 要素と第 2 要素の間に juice を挿入したリストを作るにはどうすればよいか.
   (解答例) (cons (car menu) (cons 'juice (cdr menu)))

1. menu の第 2 要素を取り除いたリストを作るにはどうすればよいか.
   (解答例) (cons (car menu) (cdr (cdr menu)))

1. menu の第 1 要素と第 2 要素を交換したリストを作るにはどうすればよいか.
   (解答例) (cons (car (cdr menu)) (cons (car menu) (cdr (cdr menu))))

9 関数定義

Lisp では既存の関数を用いて，新たな関数を自分で定義することができる.

たとえば，引数として与えられたリストの 2 番目の要素を求める関数 nibanme は，関数 defun を使って次のようにして定義する.

```
(defun nibanme (x) (car (cdr x)))
  nibanme
(nibanme '(a b c))
  b
```

一般的には

$$(\text{defun 関数名 (仮引数}_1 \ldots \text{仮引数}_n) \text{ 関数本体})$$

と書く.

このように定義した関数は，Lisp システムにすでにある組込関数と同様に使用できる.

以下では，リストの 2 番目の要素を取り除いたリストを求める関数 del2 を定義し，それと関数 nibanme を用いて，リストの 1 番目の要素と 2 番目の要素を交換したリストを求める関数 ex12 を定義する.

```
(defun del2 (x) (cons (car x) (cdr (cdr x))))
(defun ex12 (x) (cons (nibanme x) (del2 x)))
(del2 '(a b c))
  (a c)
(ex12 '(a b c))
  (b a c)
```

実は，Lisp では以下のような組込関数があらかじめ用意されている.

```
(defun caar (x) (car (car x)))
(defun cadr (x) (car (cdr x)))
(defun cdar (x) (cdr (car x)))
(defun cddr (x) (cdr (cdr x)))
```

9.1 練習問題

1. リストの 2 番目の要素と 3 番目の要素を交換したリストを求める関数 ex23 を定義せよ.
   ヒント: ex12 を利用することを考える.

(解答例) `(defun ex23 (x) (cons (car x) (ex12 (cdr x))))`

1. リストを左へ 1 つ回転させたリストを求める関数 `rotate` を定義せよ．たとえば `(rotate '(a b c d))` の結果は `(b c d a)` である．与えられるリストの長さは 1 以上としてよい．
   ヒント: `(a b c d)` から `(b c d a)` を得るには，`(append '(b c d) '(a))` と考える．`append` の第 2 引数はリストでなければならない．`(cons 'a ())` とすれば長さ 1 のリスト `(a)` が得られることを用いる．`(append '(b c d) '(a))` ではなく `(cons '(b c d) '(a))` とすると，結果が `((b c d) a)` となり `(b c d a)` にはならないことに注意する．
   (解答例) `(defun rotate (x) (append (cdr x) (cons (car x) ())))`

10 述語と条件

Lisp のある種の関数は条件判断を行うためのもので，特に **述語** (predicate) と呼ばれる．述語は判断の結果が真ならば `t` を，偽ならば `nil` を返す．

- `null` は引数が空リストかどうか調べる述語である．

```
(null ())
  t
(null 'a)
  nil
(null nil)
  t
(null '(a))
  nil
```

- `=` は 2 つの引数が同一の数値であるかどうか調べる述語である．

```
(= 1 2)
  nil
(= 2 (+ 1 1))
  t
(= 'a 'b)
  ERROR
```

- `equal` は 2 つの引数が同一の記号やリストであるかどうか調べる述語である．数値の場合も利用できる．

```
(equal 'a 'b)
  nil
(equal 'a (car '(a b)))
  t
(equal '(b) (cdr '(a b)))
  t
(equal 2 (+ 1 1))
  t
```

`(null x)` と `(equal x ())` は同じ意味である．

- `/=` は 2 つの数値が等しくない時に真になる `(/= x y)` と `(not (= x y))` は同じ意味である．

- <, >, <=, >= は 2 つの数値の大小関係を調べる．string<, string= は 2 つの記号の大小関係 (アルファベット順) を調べる．

```
(< 1 2)
  t
(string< 'a 'b)
  t
```

- atom は引数がアトム (記号あるいは数値) かどうかを調べる．

```
(atom 1)
  t
(atom 'a)
  t
(atom '(a))
  nil
(atom ())
  t
```

述語の判断結果によって，すなわち条件によって異なる値を返す関数を作るには，関数 if を用いる．

(if 条件式 式 $_1$ 式 $_2$)

- 条件式の結果が nil 以外なら 式 $_1$ の計算結果を，nil の時は 式 $_2$ の計算結果を値とする

```
(if (null 'a) 1 2)
  2
(setq x '(a b))
  (a b)
(if (>= (length x) 2) (car (cdr x)) x)
  b
```

- if を使うと，2 つの数値の大きい方を返す関数 ookiihou を定義できる．

```
(defun ookiihou (x y) (if (> x y) x y))

(ookiihou 3 7)
  7
```

組込関数 max を利用すると同じことが行える．
and, or, not で，複数の述語の結果を組合せる論理演算を行える．

10.1 練習問題

1. 2 つの数値を要素とする長さ 2 のリストを昇順に並べ変えたリストを求める関数 sort2 を定義せよ．たとえば (sort2 '(3 2)) の結果は (2 3)，(sort2 '(2 3)) の結果は (2 3) である．
(解答例) 複数の解答例を示す．

```
(defun sort2 (x)
  (if (< (car x) (cadr x))
```

```
        x
        (ex12 x)))
(defun sort2 (x)
  (if (< (car x) (cadr x))
      x
      (cons (cadr x) (cons (car x) nil))))
(defun sort2 (x)
  (cons (min (car x) (cadr x))
  (cons (max (car x) (cadr x))
        nil)))
```

1. 与えられた西暦が グレゴリオ暦 でうるう年になるかどうかを判定する関数 leap_year を定義せよ．たとえば (leap_year 2000) の結果は t，(leap_year 2100) の結果は nil である．なお y を x で割った余りは (% y x) で求められる．

   (解答例) 複数の解答例を示す．

```
(defun leap_year (y)
  (if (= (% y 4) 0)
      (if (= (% y 100) 0)
          (if (= (% y 400) 0) t nil)
          t)
      nil))
(defun leap_year (y)
  (if (= (% y 400) 0)
      t
      (if (= (% y 100) 0)
          nil
          (if (= (% y 4) 0) t nil))))
(defun leap_year (y)
  (if (= (% y 400) 0)
      t
      (if (= (% y 100) 0)
          nil
          (= (% y 4) 0))))
```

if ではなく and, or を利用して定義すればさらに簡潔になる．その場合，以下のような真理値表を元に考えると良い．

y%4=0	y%100=0	y%400=0	うるう年
nil	nil	nil	nil
t	nil	nil	t
t	t	nil	nil
t	t	t	t

```
(defun leap_year (y)
  (or (and (= (% y 4) 0) (> (% y 100) 0)) (= (% y 400) 0)))
```

ある受講生は equal を使う方法を考えてくれた．その場合も簡潔に定義できる．素晴らしい!

1. (and) の結果は何になると思うか．

(解答例) `t`

and の単位元は t とするのが自然である.

1. `(or)` の結果は何になると思うか.
   (解答例) `nil`

or の単位元は nil とするのが自然である. ORG-LIST-END-MARKER

11 再帰的定義

数学では, 定義の中に自分自身が現れることがある. たとえば n の階乗は次のように定義される.

$$n! = \begin{cases} 1 & (n = 0) \\ n \times (n-1)! & (n \geq 1) \end{cases}$$

このように定義の中に自分自身が現れる定義を, Lisp では **再帰的定義** (recursive definition) と呼ぶ.
たとえば, 階乗を求める関数 `fact` は次のように定義できる.

```
1:  (defun fact (n)
2:    (if (= n 0) 1 (* n (fact (- n 1)))))
```

```
(fact 10)
  3628800
```

リストの要素の合計を求める関数 `sum` は次のように定義できる.

```
1:  (defun sum (l)
2:    (if (null l) 0 (+ (car l) (sum (cdr l)))))
```

```
(sum '(1 9 8 9))
  27
```

2 つのリストを連結したリストを求める関数 `app` は次のように定義できる.

```
1:  (defun app (x y)
2:    (if (null x)
3:       y
4:       (cons (car x) (app (cdr x) y))))
```

```
(app '(a b) '(c d))
  (a b c d)
```

実は, この関数は `append` と同じことを行う.

11.1 練習問題

1. 関数 `fact` を変更して, 1 から n の和 1+2+...+n を求める関数 `sigma` を定義せよ.
   (解答例) 解答例を示す.

```
(defun sigma (n)
  (if (= n 0) 0 (+ n (sigma (- n 1)))))
```

1. 関数 `fact` を変更して, 1 から n の平方和 $1^2+2^2+...+n^2$ を求める関数 `sigma2` を定義せよ.
   (解答例) 解答例を示す.

```
(defun sigma2 (n)
  (if (= n 0) 0 (+ (* n n) (sigma2 (- n 1)))))
```

1. 次の漸化式で定義されるフィボナッチ数を計算する関数 fib を定義せよ.

$$fib(n) = \begin{cases} n & (n < 2) \\ fib(n-1) + fib(n-2) & (n \geq 2) \end{cases}$$

(解答例) 解答例を示す.

```
(defun fib (n)
  (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
```

1. リストの全要素の積を求める関数 prod を定義せよ.

(解答例) 解答例を示す.

```
(defun prod (x)
  (if (null x) 1 (* (car x) (prod (cdr x)))))
```

1. リストの中で一番大きい要素を求める関数 maxelem を定義せよ.

ヒント : リストの長さが 1 の時は car が最大要素である. 長さが 2 以上の時は cdr の最大要素と car との大きい方が最大要素である.

(解答例) 解答例を示す.

```
(defun maxelem (x)
  (if (null (cdr x))
      (car x)
      (max (car x) (maxelem (cdr x)))))
```

1. リストを逆順にしたリストを求める関数 rev を定義せよ.

ヒント : 空リストの逆順は空リストである. 空リストでないときは cdr の逆順と, car だけからなる長さ 1 のリストとを append したものが全体の逆順である.

(解答例) 解答例を示す.

```
(defun rev (x)
  (if (null x)
      ()
      (append (rev (cdr x)) (cons (car x) ()))))
```

1. 第 2 引数のリストの要素として第 1 引数と同じ記号が現れるかどうか調べる述語 mem を定義せよ. たとえば (mem 'a '(b a c)) の結果は t , (mem 'a '(b (a) c)) の結果は nil である.

ヒント : 第 2 引数が空リストなら結果は nil である. 第 2 引数が空リストでないときは, 第 2 引数の car が第 1 引数と equal なら結果は t , equal でないなら第 2 引数の cdr について調べる.

(解答例) 解答例を示す.

```
(defun mem (x y)
  (if (null y)
      nil
      (if (equal x (car y))
          t
          (mem x (cdr y)))))
(defun mem (x y)
  (if (null y)
      nil
```

```
        (or (equal x (car y)) (mem x (cdr y)))
    ) )
```

1. ユークリッドの互除法 を用いて，与えられた 2 つの正整数の最大公約数を求める関数 gcd を定義せよ．

   (解答例) 解答例を示す．

```
(defun gcd (m n)
  (if (= n 0)
      m
      (gcd n (% m n))))
```