計算機プログラミング'93　＃5

久野　靖*

1993.10.6

なぜか、前回はあんまり思ったように進みませんでした。今回はそのため前回資料にあった「しらみつぶし」はちょっと置いておいて、ラムダ式の方を先にやります。ただし、その前に loop の親戚というか高級版である do をやっておきます。

1　do: より構造化されたループ

loop はただの無限ループを提供するが、そのための変数の準備や各変数の更新はそれぞれ別個に用意しなければならなかった。それよりも、もう少し構造化された (つまり各部分の役割が予め用意された) ループ構造として do が用意されている。これは非常に複雑に見えるけれど、慣れてくれば役に立つ (でもやっぱり嫌いだから loop を使うということでも別に悪くはない)。do は次の形をしている。

```
(do ((局所変数 初期値 更新式)
     (局所変数 初期値 更新式)
      …
     (局所変数 初期値 更新式))
    (条件式 結果式…)
    本体式…)
```

do の実行が始まるとまず指定した個数の局所変数が用意され、それぞれに初期値が与えられる。次に、条件式が調べられ、もし結果が「いいえ」ならループ本体に入り、本体式が順番に実行される。その後今度は各更新式が計算され、それらの結果が各局所変数に入れられ、再び条件式を調べる所にいく。これが条件式が「はい」になるまで何回でも繰り返され、「はい」になると結果式を順に実行して最後のものが do の値になる。更新式はなくてもよく、その時はその局所変数は自動的には更新されない。本体式や結果式の数は 0 個以上であり、結果式が 0 個の場合には条件式の値が do の値になる。例えば do を使って repn を作ると次のようになる。

```
(defun repn (x n)
  (do ((i 1 (+ i 1))
       (r nil (cons x r)))
      ((> i n) r)))
```

練習 14b　テキストの問題 10.1〜10.10。

*筑波大学大学院経営システム科学専攻

2 関数引数

これまで、Lisp の中で扱うデータはすべて S 式ということになっていたが、実はこれに加えて「関数」そのものもデータとして扱える。具体的にはどういう時に嬉しいだろうか? 第 1 回のレポートで 2 分探索をやっていただいた。この 2 分探索そのものは特定の条件を満たす関数であればどんなものでも根を求めることができる。そこで、関数を引数として受け取るような getroot を書けばどんな関数にでも使えて便利そうである。引数として渡された関数を呼び出すには

```
(funcall 関数 引数 …)
```

を使用する。例えば以下の通り。

```
(defun getroot (f a b e &aux x y)  ; f を引数として渡して貰う
  (setq x (* 0.5 (+ a b)))
  (setq y (funcall f x))           ; ここで関数 f を呼ぶ
  (cond ((< (- b a) e) x)
        ((< y 0) (getroot f x b e))
        (t       (getroot f a x e))))
```

これを呼び出す時には例えば次のようにする。

```
>(defun f (x) (- (* x x) 2.0))
F
>(getroot #'f 0.0 2.0 0.0000001)
1.414213567972183
```

このように、関数を渡す時には「#' 関数名」という書き方を用いる。(実はこれは「(function 関数名)」の短縮形である。) このように、「関数を引数として渡す」ことで 1 つの関数を様々な目的に汎用的に使用できるようになる。

練習 15 次の関数を書け。

- 1 引数の関数 F と任意の S 式 X を受け取り、X に F を 2 回適用した結果を返す関数 twice。
- 1 引数の関数 F と任意の S 式 X と正の数 N を受け取り、X に F を N 回適用した結果を返す関数 repeatapp。
- 1 引数の関数 F と任意のリスト L を受け取り、L の各要素に対して F を適用した結果をリストとして返す関数 xmapcar。これを用いて、数値のリストを与えてその各要素を 1 つ増したリストを計算させてみよ。また任意のリストを与えてその各要素を () に入れたリストを計算させてみよ。
- 1 引数の関数 F(ただし F は値としてリストまたは nil を返す) と任意のリスト L を受け取り、L の各要素に対して F を適用した結果を連結して返す関数 xmapcan。これを用いて、数値のリストを与えて各要素をその回数ぶん反復させたリストを計算させてみよ。
- 1 引数の述語 P と任意のリスト L を受け取り、P が「はい」を返すような要素を L から取り除いたリストを返す関数 xremove-if。これを用いて、数値のリストを与えて負の数を取り除いたリストを計算させてみよ。

実は最後の 3 つは mapcar、mapcan、remove-if という標準関数として予め用意されている。remove-if の条件を反転したものとして、remove-if-not もある。

3　ラムダ式

ところで、関数を引数として渡す時にごく簡単な関数までいちいち名前をつけて defun で定義するのはやっかいである。そこで、名前をつけてその名前を指定する代わりに、「ラムダ式」と呼ばれる形式で関数本体を直接指定することもできる。その形は次の通り。

```
#'(lambda (引数 …) 式 …)
```

ラムダ式とは要するに「defun 関数名」の代わりに lambda と書いたものだと思えばよい。これを使って 2 の平方根を求めるには次のようにする。

```
>(getroot #'(lambda (x) (- (* x x) 2.0)) 0.0 2.0 0.0000001)
1.414213567972183
```

これを利用すると、一般の数の平方根を求める関数は次のようにして作れる。

```
(defun sqroot (n)
  (getroot #'(lambda (x) (- (* x x) n)) 0.0 n 0.0000001))
```

なお、ラムダ式の中に使われている「n」は sqroot の引数の「n」で、その値がずっと覚えられていて getroot の中からこのラムダ式 (に対応する関数) が呼ばれた時に使われる。これはちょっと不思議な感じがするけれど、なかなか便利である。

4　マップ関数

mapcar、mapcan などの関数をマップ関数と呼ぶ。これは、リストの各要素に引数として与えた関数を適用した写像 (マップ) を集めて結果にするからである。しかし、単にそういう意味だけでなく、マップ関数は「リストの各要素を順に処理する」という意味合いで使うこともできる。例えば次の例を見ていただく。

```
>(defun accum (l &aux a)
    (setq a 0)
    (mapcar #'(lambda (x) (setq a (+ a x)) a) l))
ACCUM
>(accum '(1 2 3 4 5))
(1 3 6 10 15)
```

つまり、mapcar に渡す関数本体の中で、予め用意した a という変数に渡す合計を順次累計していくわけである。ところで、これを利用するともっと簡単に次の関数が定義できることがお分かりだろうか? なお、この種の副作用だけが必要な時は結果のリストを組み立てる手間は余分であり、mapcar の代わりに結果としては nil を返す関数 mapc を使う方が効率が通い。

練習 16 次の関数をマップ関数を利用して書け。

- 数値のリスト L を受け取り、その合計を返す関数 goukei。
- 数値のリスト L を受け取り、その長さを返す関数 nagasa。
- リスト L を受け取り、逆順にしたものを返す関数 mreverse。

また、数値に関する反復についてはこの種のマップ関数は用意されていないが、自分で書くことは自由である。

練習 17 次の関数を作れ。

- 1 引数の関数 F と数値 a、b(ただし a `<=` b) を受け取り、F を a、$a+1$、$a+2$、…、b に対して順次適用して、その結果をリストとして返す関数 `mapnum`。
- `mapnum` と同様だが、結果としてはいつも `nil` を返す関数 `mapn`。
- 正の数 N を受け取り、1 から N までの数を順にならべたリストを返す関数 `intlist`。`mapn` を使うとよい。
- 正の数 N を受け取り、N^2、$(N-1)^2$、…、1 を順に並べたリストを返す関数 `sqdeclist`。`mapn` を使うとよい。

A　これまでに出た関数や特殊形式のまとめ

A.1　関数定義、制御構造

- (defun 関数名（引数…） 式…): 関数の定義
- (lambda（引数…） 式…): 名前を持たない関数本体
- (cond（条件 式…）（条件 式…）…): 条件分岐
- (let（変数…） 式…): 局所変数の使用
- (prog（変数…） 式…): 局所変数の使用+return の使用
- (loop 式…): 無限ループ
- (do ((変数 初期値 更新）…）（終了判定 . 結果）式…): より一般的なループ
- (return 式): prog やループからの抜け出し
- (return-from 関数名 式): 関数からの抜け出し
- (read): キーボードから S 式を読み込み値として返す
- (print X): X を画面に表示する
- (prin1 X): 改行を行なわない print
- (princ X): print と類似するが表示が簡潔
- (terpri): 出力を改行する

A.2　S 式とリストの基本関数

- (car L): L の先頭要素を取り出す
- (cdr L): L の先頭を取り除いた残りのリストを返す
- (cons X L): L の先頭に X を追加したリストを返す
- (list X_1 … X_n): X_1〜X_n から成るリストを返す
- (append L_1 … L_n): L_1〜L_n を連結したリストを返す
- (reverse L): L を逆順にしたリストを返す
- (last L): L の最後の要素のみから成るリストを返す
- (length L): L の長さを返す
- (nth N L): L の N 番目の要素を返す (先頭を 0 とする)
- (remove X L): L から X を除いたリストを返す
- (remove-if F L): L から F が「はい」を返すものを除いたリストを返す
- (remove-if-not F L): L から F が「いいえ」を返すものを除いたリストを返す
- (mapcar F L): L の各要素に F を適用した結果のリストを返す
- (mapc F L): L の各要素に F を適用する
- (mapcan F L): L の各要素に F を適用した結果を連結したリストを返す

A.3　主要な述語

- (atom X): X がアトムなら「はい」
- (numberp X): X が数値アトムなら「はい」
- (symbolp X): X が記号アトムなら「はい」
- (eq X Y): X と Y が同じ記号なら「はい」
- (listp X): X がリストなら「はい」
- (consp X): X が空でないリストなら「はい」
- (null X): X が空リストなら「はい」
- (equal X Y): X と Y が同じ内容の S 式なら「はい」
- (member X L): L 中に X があれば「はい」
- (and 条件 1 … 条件 n): 条件 1〜条件 n がすべて「はい」のとき「はい」
- (or 条件 1 … 条件 n): 条件のどれかが「はい」のとき「はい」
- (not 条件): 条件の反転

A.4　数値関係の関数と述語

- (1+ 引数): 引数+1 を返す
- (1- 引数): 引数-1 を返す
- (+ 引数 1 … 引数 n): 引数の和を返す
- (- 引数 1 引数 2): 引数 1 と引数 2 の差を返す
- (* 引数 1 … 引数 n): 引数の積を返す
- (/ 引数 1 引数 2): 引数 1 を引数 2 で割った商を返す
- (floor 引数 1): -∞へ向かって切捨て
- (truncate 引数 1): 0 へ向かって切捨て
- (ceiling 引数 1): +∞へ向かって切上げ
- (round 引数 1): 丸め
- (rem 引数 1 引数 2): 剰余
- (mod 引数 1 引数 2): 剰余
- (= 引数 1 引数 2): 引数 1 が引数 2 と数値的に等しいとき「はい」
- (> 引数 1 引数 2): 引数 1 が引数 2 より大きいとき「はい」
- (< 引数 1 引数 2): 引数 1 が引数 2 より小さいとき「はい」
- (>= 引数 1 引数 2): 引数 1 が引数 2 以上のとき「はい」
- (<= 引数 1 引数 2): 引数 1 が引数 2 以下のとき「はい」
- (zerop 数値): 数が 0 なら「はい」
- (plusp 数値): 数が正なら「はい」
- (minusp 数値): 数が負なら「はい」