計算機プログラミング'93　＃7

久野　靖*

1993.10.20

1　eval、apply

ごく最初の方で「Lisp ではプログラムを実行することを『評価する』という」などという説明をしたことをご記憶だろうか。例えば

```
(+ 1 2 3) →評価→ 6
```

のようになる。これは要するに、左側の S 式を KCL に向かって打ち込むと右側の式が返ってくる、というのに等しい。ところで、Lisp には評価を強制する (余計に行なわせる) 関数 eval というのが備わっている。すなわち、

```
'(+ 1 2 3) →評価→ (+ 1 2 3)
```

ですよね? この右側はさらに評価すると先に書いたように 6 になる。従って、

```
(eval '(+ 1 2 3)) →評価→ 6
```

となるわけである。これを利用して、次のようなことを考えてみる。例えば「(1 2 3 4 5)」のような数のリストを受け取って、その合計を計算するのに、これまでだと再帰的関数を使って計算していた。しかし、次のように考えたらどうか?

- そのリストの頭に'+をつけ加える。
- できあがったリスト ((+ 数値…)) を評価する。

```
(defun listsum1 (l) (eval (cons '+ l)))
```

ちょっと面白いでしょう? ところで、こういうことは時々必要になるので、いちいち cons で関数名をくっつける代わりに apply というのが用意されている。

```
(apply 関数 引数のリスト)
```

これを使うと上の例は次のように書ける。

```
(defun listsum2 (l) (apply #'+ l))
```

ところで、前回やった funcall と apply の違いがお分かりか?

```
(apply 関数 引数1 引数2 引数3 …)
```

*筑波大学大学院経営システム科学専攻

このように、引数を 1 つのリストにまとめて渡すのが apply、ばらばらに与えるのが funcall ということになる。

練習 21 次のような関数を作れ。

- 変数のリストを渡すと、それぞれの変数に入っている値のリストを返す関数 varvalues。
- Lisp の式の形をした S 式を渡すと、その S 式の内容を 3 回実行する関数 exec3times。
- 数が入った変数の名前と数 N を渡すと、その変数の内容を N ふやす関数 addvars。
- 変数のリストを渡すと、それぞれの変数に値 nil を設定する関数 nilvars。

2 マクロ

さて、eval のもう 1 つの応用として、制御構造の拡張を考えてみる。例えば、これまで条件分岐には cond のみを使って来たが、Pascal や C のように if が使えたらいいと思いません? つまり

```
(xif 条件 式1 式2)
```

というのを用意し、条件が「はい」なら式 1、「いいえ」なら式 2 を実行させようというわけである。これを例えば次のように書いたらだめだろうか?

```
>(defun xif (c e1 e2)
    (cond (c e1) (t e2)))
XIF
>(setq x 5)
5
>(xif (plusp x) 1 -1)
1
```

よさそうだと思うが…

```
>(xif (plusp x) (print 'plus) (print 'munus))
PLUS
MINUS
PLUS
```

つまり、実は式 1 も式 2 も両方とも実行 (評価) されて、その後で条件に応じてどちらかの値を選んでいるわけなのだった。もちろんこれでは、cond の代わりに使うわけにはいかない。(なぜか?) そこで代わりに、先の listsum のように必要な形のものを組み立てたあと eval を使うという方法を考えてみる。

```
>(defun xif (c e1 e2)
    (eval (list 'cond (list c e1) (list 't e2))))
XIF
>(xif '(plusp x) '(print 'plus) '(print 'minus))
PLUS
PLUS
```

こんどはちゃんと、条件に合った方だけが計算されている。しかし、毎回条件や式 1、2 に' をつけるのでは見にくいし煩わしい。

実は、このような目的にぴったりの機能としてマクロというのが用意されている。マクロは defun の代わりに defmacro を使って定義する。そして、マクロの場合は関数定義と次の点が違う。

- 各引数は、評価されない。
- マクロは式を組み立てて返すことを期待されている。Lisp 処理系はそのマクロが返した式を評価してくれる。

従って上の例をマクロにすれば、余分な' もいらないし、`eval` は使わなくてよい (処理系が `eval` を適用してくれる)。

```
>(defmacro xif (c e1 e2)
    (list 'cond (list c e1) (list 't e2)))
XIF
>(xif (plusp x) (print 'plus) (print 'minus))
PLUS
PLUS
```

この方がずっとそれらしい。実は CommonLisp にはこうやって作られたマクロ `if` もちゃんとある。

3 バッククォート式

マクロの 1 つの問題点は、`list` などを使って組み立てるので「最終的に実行されるべきもの」が何であるか分かりにくいことである。そこで、これを解消するため Lisp にはバッククォート式というものが用意されている。これは次のようなものである。

- 普通の「'」のかわりに「`」ではじまる。
- あとは普通の S 式の形をしている。
- ただし、その中に「, 式」というものがあると、そこの場所には「式」を評価した結果が「埋め込まれる」。

例えば次の通り。

```
>(setq x 1)
1
>'(a b (c d e))
(A B (C D E))
>`(a b (,x d ,(+ x 3)) f)
(A B (1 D 4) F)
```

これを利用すると、先の `xif` は次のように書ける。

```
(defmacro xif (c e1 e2)
   `(cond (,c ,e1) (t ,e2)))
```

これならいくぶん分かりやすいでしょう? これを使って、様々な新しい制御構造や機能を Lisp につけ加えることができる。例えば次の練習問題。

練習 22 次のようなマクロを作れ。

- 変数を渡すと、その値を 1 増すマクロ `xincf` と 1 減らすマクロ `xdecf`。いずれも値は増やした/減らした後の値であるものとする。例えば次のようになるはず。

```
>(setq x 3)
3
>(xincf x)
4
>x
4
>(xdecf x)
3
>x
3
```

- リストが入っている変数を渡すと、その先頭要素を取り除き、値としては取り除いたものを返すマクロ xpop。例えば次のようになるはず。

```
>(setq z '(a b c d))
(A B C D)
>(xpop z)
A
>z
(B C D)
```

ヒント: 先頭の値を覚えておくために let か prog を使うとよい。

- リストが入っている変数と任意の S 式を渡すと、変数にリストに S 式を先頭要素として追加し、値としてはその S 式を返すマクロ xpush。例えば次のようになるはず。

```
>(setq z '(a b c d))
(A B C D)
>(xpush z 'e)
E
>z
(E A B C D)
```

- 変数と正の数 N と形式 (実行できる式)E を渡すと、変数を 0、1、…、N-1 と変化させながら繰り返し E を実行するマクロ xdotimes。最後の値としては nil を返す。例えば次のように使えるはず。

```
>(xdotimes i 10 (print (+ i i)))
0
2
4
…
18
NIL
```

ただし変数はローカル変数になること。そのためには let か prog か do が必要。

- 変数とリストと形式 (実行できる式)E を渡すと、変数をリストの各要素に次々に変化させながら繰り返し E を実行するマクロ xdolist。最後の値としては nil を返す。例えば次のように使えるはず。

```
>(xdolist i '(a b c) (print (cons i i)))
(A . A)
(B . B)
(C . C)
NIL
```

ただし変数はローカル変数になること。そのためには let か prog か do が必要。

ときに、バッククォート式中では「,] が「リスト中の特定の位置に埋め込む」のに対し、もう 1 つの書き方「,@」が使え、これによって「リストのある場所から後ろをそっくり置き換える」こともできる。例えば次の通り。

```
>(setq x '(a b c))
(A B C)
>`(1 2 3 ,x)
(1 2 3 (A B C))
>`(1 2 3 ,@x)
(1 2 3 A B C)
```

4 可変引数

これまでに作った関数ではすべて、引数の個数は一定に決まっていた。でも、組み込み関数の+などでは引数は何個にでも変えられましたね? それを自分でもやる方法を説明しておく。(ただしCommonLisp でないとできない。テキストにも載っていない。)

まず、引数の個数は「概ね」決まっているのだけれど、最後の方は省略してもよいという場合からやる。そのような場合には

```
(defun 関数名 (引数 … &optional 引数 …) 式 …)
```

という形を用いる。ここで&optional の後ろに書いた引数は、普通の引数と同じだけれど呼び出す時に値を省略してもよい。省略した場合には値は nil になる。例えば「数 X に N を足すが、ただし N を省略すると 1 足す」という関数を書いてみる。

```
>(defun incr (x &optional n)
    (cond ((null n) (+ x 1))
          (t        (+ x n))))
INCR
>(incr 3)
4
>(incr 3 5)
8
```

これだと受けとれる引数の最大個数は決まってしまうことになるので、もう 1 つの方法として「ここから後はリストで受け取る」というやり方も可能である。それには

```
(defun 関数名 (引数 … &rest 引数) 式 …)
```

という形を使う。この場合、&rest の前に書いた引数は通常のように扱われるが、残った引数はまとめてリストにして&rest の後に書いた引数に渡される。[1]これを使って、任意個数の引数を順にプリントして、最後に値として nil を返す例は次の通り。

```
>(defun printall (&res lis)
   (do ((l lis (cdr l)))
       ((null l))
       (print (car l))))
PRINTALL
>(printall 'a (+ 1 2) "this")
A
3
```

[1]なお、&optional と&rest を両方使ってもよいが、その場合は&optional が先にくること。また&aux も一緒に使ってよいが、その場合は一番最後に置く。

```
"this"
NIL
```

なお、これらの機構は `defmacro` でも同様に使える。

練習 23 次のようなマクロを作れ。

- 先のマクロ `xif` は else 部がないとエラーになった。しかし if 文では else がないのは普通である。else 部を省略してもよいように直せ。
- 先のマクロ `xincf`/`xdecf` は常に 1 増やす/減らすものだったが、2 番目の引数として増やす/減らす値を指定できるようにして、省略された場合には 1 とするように直せ。
- 先のマクロ `xdolist`/`xdotimes` で形式の数を 1 個でなく任意個許すようにしてみよ。`,@` を使うことになると思う。

既にご想像の通り、CommonLisp には `incf`、`decf`、`push`、`pop`、`dotimes`、`dolist` が備わっている。ただし最後の 2 つはちょっと練習問題とは形が違っていて、

```
(dotimes (変数 正の数) 式 …)
(dolist (変数 リスト) 式 …)
```

という形になっている。