ソフトウェア工学実験/計算機工学実験'94　課題2 (Lisp言語) ＃2

久野　靖　(筑波大学大学院経営システム科学専攻)

1994.5.11,1994.6.13

1　Lispと記号処理

前回は、Lispを使ったとはいえ内容は数値の計算ばかりだった。今回はもっとLispらしく、記号とリストを中心にすすめる。まずはその解説。

- 記号:　記号(symbol)とはおおむね、英字から始まる英数字記号等の並び。記号は「あるものと別のものを区別する」ために使う。なおCommonLispでは大文字と小文字をおおむね区別しない。
- nilとt:　nilというのも記号だけど、やや特別。Pascalのポインタとちょっと似ているが、「何もない」というのを表すのにnilという記号を使う。tも記号だが、これもちょっと特別で、「真である」ということを表すのに tという記号を使う。なお「偽である」というのを表すのにはnilを使う。そして混乱することに、場合によってはnil以外のものは全部「偽でないから真」として扱うこともある。
- 数値:　前回やった通り。整数、実数、有理数などがありましたね。
- アトム:　記号とか数値などは、「それ以上分けられないもの」という意味で「アトム」と呼ばれる。nilはniとlに分けられるって?　それはniという記号とlという記号なのであって、nilが分かれたものでは全然ない。
- リスト:　記号処理でもデータ構造は扱いたい。そこで、リストというものを使う。リストとは

  ```
  (要素 要素 要素 ... 要素)
  ```

  の形をしている。つまり、

  1. 複数の「もの」を「まとめる」ことをリストとして表現する。
  2. リストを書き表すには、要素を「(」と「)」で囲んで書き表す。

  ということですね。なお、要素とは何かって?　それは、アトム、またはリストです。(?!)　なお要素どうしは「,」などで区切ったりせず、空白(やタブや行かえ)で区切る。ところで、またまた非常に変なのですが、要素が0個のリスト、つまり「()」は「nil」と同じものを表す、ということになっている。
- S式:　実は、上のようにしてアトムやリストを書き表す表現方法を「S式」と呼ぶ。下記はそれぞれの行が1個のS式の例である。

  ```
  nil
  abcde
  ```

```
(t nil abcde)
(a b (c d e) f)
(((((x)))))
```

かっこの対応が正しくないと S 式として正しくないことに注意。

- 形式: S 式は何のために使われると思いますか? データを表現するため? 確かにそれもそうだけど、実は Lisp ではプログラムも S 式で表現される (プログラムとデータが同じ形!)。そういわれて見ると、プログラムも S 式の形になっているでしょう?
- 評価: 「プログラムを実行する」と言うのを Lisp では「形式を評価する」という。例えば

```
(+ 2 3)
```

という形式を評価すると、答えはもちろん「5」である。

- 記号の値: Pascal では「x」などという「式」は何を意味したか? Lisp でも記号は変数のようなもので、そこに値を入れておける。それには「(setq 変数 値)」という形式を使う。

```
> (setq x 7)
7
> x
7
> (- x 3)
4
```

なお、setq はセットした値そのものを結果の値ともするので、それで 7 が打ち出されている。ところで、まだ値を入れていない記号を「評価」するとどうなると思うか?

```
> y
Error: The variable Y is unbound.
Error signalled by EVAL.
Broken at EVAL.  Type :H for Help.
>>
```

エラーになる。Lisp では「記号に値をセットしたとたん、その記号が変数にもなる」と思えばいい。

- quote: 「x」とか「(a b c)」のような S 式そのものをデータとして打ち込みたいときはどうするか。単に「x」とか「(a b c)」そのまま打つと「変数未定義」とか「a という関数はない」と言われる。例えば Pascal で x という変数ではなく「x と言う字」を表したければどうしたっけ? 「'x'」ですね。同様に、Lisp では

```
(quote S 式)
```

という形式を使うことで引数の「S 式」そのもの、を表すことができる。例えば次のような具合である。

```
> (quote y)
Y
> (quote (a b c))
(A B C)
>
```

これはあまりにもよく使うので、

```
'S式
```

と書いてもいいようにシステムに細工がしてある。だから

```
> 'y
Y
> '(a b c)
(A B C)
>
```

これでもいい。ときに、大文字と小文字の区別をしないと書いたが、実は「大文字が標準」でして、だからシステムからの表示は全部大文字にされてしまう。ところで、これまで数値には quote をつけていなかったのに大丈夫だったのを不思議に思った人はいませんか? 実は、数値は評価するとまたその数値自身になってしまうようになっているのでした。

```
> 10
10
> '10
10
>
```

だからどっちでもいいというか普通は数値は quote しませんね。同様に、nil も評価したら同じ nil、そして t も評価したら同じ t である。

- 5 つの基本関数: ここでリスト構造を操作するための 5 つの基本関数を紹介しておく。
  - (eq X Y) --- X と Y がともに記号である場合、それらが同じ記号なら t、そうでないとき nil を返す。
  - (null L) --- L が空リスト (つまり nil) の時 t、そうでないとき nil を返す。
  - (car L) --- L が空でないリストのとき、その先頭要素を返す。
  - (cdr L) --- L が空でないリストのとき、その先頭要素を除いた残りを返す。
  - (cons X L) --- リスト L の頭に新しい要素 X をつけ加えたリストを返す。

  例えば次のような具合である。

```
> (eq 'a 'a)
T
> (eq 'a 'b)
NIL
> (eq nil '())
T
> (null '(a b c))
NIL
> (null '())
```

```
T
> (car '(a b c d))
A
> (cdr '(a b c d))
(B C D)
> (cons 'x '(a b c d))
(X A B C D)
>
```

- caar, cadr, cdar, cddr, ...: ところで、car と cdr は例えば

  ```
  (car (cdr (cdr z)))
  ```

  といった風に沢山組合せて使うことが多いので、その時かっこが多くて見にくいのを打破するため

  ```
  (caddr z)
  ```

  とも書けるような細工がされている。一般に

  ```
  (cXXr z)
  (cXXXr z)
  (cXXXXr z)
  ```

  (ただし X のところには「a」か「d」が入る) はそれぞれ

  ```
  (cXr (cXr z))
  (cXr (cXr (cXr z)))
  (cXr (cXr (cXr (cXr z))))
  ```

  に対応している。a と d が合計 5 個以上になるようなら (めったにないがたまにある) 自分で適当に分けること。
- list, append: 基本関数ではないのだけど、リストを操作するときは次の 2 つも使うと 便利かも知れない。
  - (list A B C ...) --- 引数全部を順に並べたリスト (A B C ...) を作る。んなら'(A B C ...) でいいじゃないかって? 練習問題をやってから言うように。
  - (append L1 L2) --- 2 つのリスト L1 と L2 を連結したリストを作る。

練習 1 KCL を立ちあげ、まず次のように打ち込んでください。

```
> (setq x '(a b (c d) (e f (g h i))))
(A B (C D) (E F (G H I)))
>
```

さて、練習問題ですが、変数 x に入っている値から次のものを取り出してみて下さい。

- A
- (B (C D) (E F (G H I)))
- B
- (C D)
- C
- (D)
- D
- NIL
- (H I)
- (E F)

くどいようだが、直接打ち込むのではなくて x に入っている値から取り出すこと。例えば最初の A ですが、これは x に入っているリストの第 1 要素だから

```
> (car x)
A
>
```

のようにしてできます。

練習 2 例によって KCL を立ち上げて次のように打ち込んで下さい。

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

さて、これらをもとに次のようなリストを組み立ててみよ。当然、' を使ってはいけない。

- (A B C D E)
- ((A B C) (D E))
- ((A B C) D E)
- (A B C (D E))
- ((A B C D E))
- (Z A B C)
- (Z A B C Z)
- (A B C Z D E)

しつこいようだが、直接打ち込むのではなくて x〜z に入っている値から取り出すこと。例えば最初の (A B C D E) ですが、これは x と y のリストの連結だから

```
> (append x y)
(A B C D E)
```

となる。

2 リスト処理の関数

では例によってまた関数を色々作ることになるが、今回は数値ではなくリスト扱うものである。特に難しいことはない。例えば 2 つの引数を取り、それをリストに入れる関数 list2 は次のようになる。

```
>(defun list2 (a b) (list a b))
LIST1
>(list2 'a (+ 2 3))
(A 5)
```

なお if とかの使い方も前回やった通りだけれど、数値の「等しい」は「=」で記号の「等しい」は「eq」というように違っているので注意されたい。その他、渡されたものが「何であるか」を判定することも必要になることがあるだろう。それらをまとめておこう。

- (atom X) --- X がアトム (数値か記号) のとき t。
- (numberp X) --- X が数値のとき t。
- (symbolp X) --- X が記号のとき t。
- (listp X) --- X がリスト (nil を含む) のとき t。
- (consp X) --- X が空でないリストのとき t。

練習 3 次のような関数を作ってください。

- 1 つの引数を取り、それが 3 つ並んだリストを作る関数 list3。例えば (list3 'a) → (A A A)。
- 1 つの引数を取り、それが 3 つ並んだリストが 3 つ並んだものを返す関数 list33。例えば (list33 'a) → ((A A A) (A A A) (A A A))。
- 数を要素とする長さ 3 のリストを取り、それらの数の合計を返す関数 addlist3。例えば (addlist3 '(1 2 3) → 6。
- 長さ 3(以上) のリストを取り、それ全体と、それから最初の要素を取り除いたものと、それから最初の 2 つの要素を取り除いたものの 3 つをリストにして返す関数 remlis3。例えば (remlis3 '(a b c)) → ((A B C) (B C) (C))。
- 引数が A だったら B にするが、そうでなければそのまま返す関数 atob。つまり (atob 'a) → B、(atob 'c) → C となる。
- 引数が数だったら 1 足すが、そうでなければ「nan」という記号に取り換える関数 add1。つまり (add1 3) → 4、(add1 'a) → nan となる。
- リストをもらい、その先頭を取り除くがリストの長さが 0 なら何もしない関数 remtop。

- リストをもらい、その先頭を取り除くが取り除いた長さが 0 になるようなら何もしない関数 remtop1。
- 長さ 3 のリストをもらい、その中に A があればそれを B に置き換える関数 atob3。

3 リスト処理の再帰関数

ここまでに出てきた関数はどれも再帰関数ではなかったけれど、一般に長さがいくつか分からないリストを処理するにはやっぱり再帰関数が必要である。その基本的な記述のしかたは次のパターンになる。

- リストの長さが 0 なら〜する
- そうでなければまずリストの先頭を処理して、残りの部分は自分自身を再帰的に呼び出すことで処理する。

場合によっては前者が「長さが 1 なら」になることもある。例えば、リストの長さを数える関数は次の通り。

```
>(defun listlen (l)
     (if (null 0) 0 (+ 1 (listlen (cdr l)))))
LISTLEN
>(listlen '(a b c d e))
5
```

これの動作は結果を「追跡」してみるとよくわかる。

```
>(trace listlen)  ←関数 listlen の実行を追跡モードに設定
(LISTLEN)
>(listlen '(a b c))
  1> (LISTLEN (A B C))
    2> (LISTLEN (B C))
      3> (LISTLEN (C))
        4> (LISTLEN NIL)
        <4 (LISTLEN 0)
      <3 (LISTLEN 1)
    <2 (LISTLEN 2)
  <1 (LISTLEN 3)
3
```

練習 4 次のような関数を定義せよ。

- 数値から成るリストを受けとり数値の和を返す関数 listsum。
- 長さ 1 以上のリストの最後の要素を取り出す関数 listtail。
- 引数 x を引数 n で指定した回数だけ繰り返し並べたリストを返す関数 repn。例えば (repn 'a 5) → (A A A A A)。

- リストを逆転する関数 listrev。
- リスト l と 1 引数の関数 f を受け取り、l の各要素に f を適用する関数 shazou。例えば (shazou '(1 2 3 4) #'-) → (-1 -2 -3 -4)。実はこの関数は (引数の順序等がちょっと違うが)mapcar という関数と同様である。
- リスト l と 2 引数の関数を受け取り、l の各要素を f で 1 つの数値にまとめる関数 matome。例えば (matome '(1 2 3 4) #'+) → 1 + 2 + 3 + 4 = 10。
- リストの第 1 レベルに含まれているアトムの数を数える関数 countatoms。例えば (countatoms '(a b (c (d)) e (f))) → 3。

4 多方向分岐と cond

上のパターンで再帰関数の処理を書いていると、「先頭の要素の処理」でさらに分岐が欲しいことがある。例えば countatoms がそのはずである。その時 if の中の if を使うとかなり見にくい。ここは Pascal の if〜else if〜else if…のようなものが欲しいところである。そのような場合には Lisp では cond というのを使う。その一般的な形式は:

```
(cond (式1 式)
      (式2 式)
      ...
      (式n 式))
```

これは、まず式 1 を評価し、それが真ならそれに続く式を値とする。偽なら式 2 を評価し、それが真ならそれに続く式を値とする。偽なら式 3 を... 以下同様、というものである。式 n まで来てそれも偽だと全体の値が nil になるが、最後は「どれでもなければ... をする」になることが多いので、式 n として t を書くことが多い。これを用いて、さっきの countatoms を書き直してみる。

```
(defun countallatoms (l)
    (cond ((null l) 0)
          ((atom (car l)) (+ (countatoms (cdr l)) 1))
          (t (countatoms (cdr l)))))
```

練習 5 次のような関数を定義せよ。

- リストの中に含まれているすべてのアトムの数を数える関数 countallatoms。例えば (countallatoms '(a b (c (d)) e (f))) → 6。
- 引数 l(リスト) の中の引数 x と同じものを引数 y で置き換えたリストを返す関数 listrepl。例えば (listrepl '(e v e n) 'e 'x) → (X V X N)。
- 引数としてリストをもらい、それ全体、最初の要素を取り除いたもの、最初の 2 つの要素を取り除いたもの、...、最後の要素のみのリスト、から成るリストを返す関数 remlisn。例えば (remlisn '(a b c d) → ((A B C D) (B C D) (C D) (D))。
- 引数 l(リスト) から引数 x に等しい要素を取り除いたリストを返す関数 remove。例えば (remove '(e v e n) 'e) → (V N)。

- 引数 l(リスト) から、引数 x(リスト) に含まれている要素をすべて取り除いたリストを返す関数 remset。例えば (remset '(e v e r y t h i n g) '(e t n)) → (V R H I G)。
- 任意のリスト構造 l とアトム x、y を引数とし、l の中の x と同じアトム全てを y に置き換える関数 replace。例えば (repl '(a (b c d) c e) 'c 'z) → (A (B Z D) Z E)。
- 二つのリストを受けとり、その対応する各要素をそれぞれ長さ 2 のリストにし、それを並べたリストを返す関数 pair。例えば (pair '(a b c) '(x y z)) → ((A X) (B Y) (C Z))。
- 二つのリストを受けとり、それらを連結したリストを返す関数 concat。実はこれは append と同じもんである。
- アトム x と「リストのリスト」l を受けとり、l の中で x と同じものから始まるリストを取り出して返す関数 lookup。例えば (lookup 'a '((x y z) (a b c) (u v w))) → (A B C)。実はこれは assoc という関数と (ほぼ) 同じもんである。

5 実習

前回と同様、資料の例題を打ち込んで動かし練習問題のところに来たら、(全部やってると大変でしょうから) 気にいったのを 3〜4 問選んでやってみてください。最後まで来たら本日の出席を兼ねてやった練習問題の定義と実行の記録をとり、emacs で見やすく修正して分量が多ければ簡単な問題は削って 2〜3 ページ程度におさめ、プリントアウトしてください。2 ページ以上にわたる場合にはホチキスでとめる。なおかつ、その裏側に (複数ページにわたる場合には見やすさのため最後のページの裏側に) 次のものを記入すること。

- 学籍番号、氏名、所属、本日の日付。
- 以下のアンケートに対する答え (簡単でいいです)。

  Q1. 前回と今回で Lisp の関数が書けるようになりましたか。他の言語とくらべてどうですか。

  Q2. 本日やったことのうち特に難しいと思った部分はどこですか。面白いと思ったところはありますか。

  Q3. 本日の感想、今後の要望などお書きください。

出席として認定されるためには、本日 5 時までに事務室のレポートボックスに提出のこと。時間切れで練習問題が最後までできなかった場合には途中までででも結構です。(が、もともと全部やる必要はないので適宜飛ばしながら時間中に最後まで到達するほうがいいですけどね。)