計算機プログラミング'93　＃4

久野　靖*

1993.9.29

1　入出力と反復の復習

またまた前回の復習から。まず局所変数を使用するためには、

```
(let (局所変数名 …) 式 …)
(prog (局所変数名 …) 式 …)
(defun 関数名 (引数 … &aux 局所変数名 …) 式 …)
```

のいずれかを使用する。複数の式の実行の途中から抜け出すには

```
(return 式)                … prog の中から
(return-from 関数名 式)  … 関数の途中から
```

のいずれかを使う。ループを構成するには

```
(loop 式 …)
```

を使用する。実行の途中でS式の入力や表示を行なうには

```
(read)       … S式を読み込む
(print 式)  … S式を表示する
(princ 式)  … 〃、ただし改行せず、""などもつけない
(terpri)     … 出力の改行
```

などを利用すれば良い。変数に値を設定するには

```
(setq 変数名 式)
```

を使用する。さて、これらに関して次の練習をやってみる。

練習 12 次の関数を書け。

- リストLを与えると、まずそれ自身、次にその先頭を取り除いた残り、次にそのまた先頭を取り除いた残り…を nil に至るまで次々に表示する関数 removepr。値としては nil を返す。
- リストLを与えると、まず nil、次にLの先頭のみから成るリスト、次にLの最初の2要素の逆順のリスト…と進み、最後にL全体の逆順のリストを表示する関数 reversepr。値としては nil を返す。
- リストLの逆順のリストを返す関数 zreverse。ただしループ構文を利用して作ること。
- S式Xと数値Nを受け取り、SをN個並べたリストを返す関数 repn。ただしループ構文版と再帰版の両方を作れ。
- 空でない数値のリストLを受け取り、その中の最大値を求める関数 maxlist。ただしループ構文版と再帰版の両方を作れ。

*筑波大学大学院経営システム科学専攻

これらを見ると、ループでも再帰でも同等の仕事ができること、ただしどちらの方が使いやすいかは問題の種類にもよることがお分かりいただけると思う。

ところで、後になったけれどリストをループで処理する時には次の操作も知っておくと便利だと思う。

- (push 式 変数): (setq 変数 (cons 式 変数)) と同じ。つまり変数に入っているリストに先頭要素をつけ加える。
- (pop 変数): (prog (x) (setq x (car 変数)) (setq 変数 (cdr 変数)) x) と同じ。つまり変数に入っているリストから先頭要素を取り除き、なおかつその取り除いた要素を値として返す。

2 様々な再帰

前回までに学んだ再帰は cdr 再帰 (リストの各要素について順にたどっていく) だけだったが、今回はもう少し多様なバリエーションを学んで頂きたい。

2.1 構造再帰

構造を持ったデータを処理するのに再帰はとても有用な道具となる。例えば次の問題を考える。

> 普通の中置記法による、ただし完全にかっこづけされた数式は S 式でもある。例えば「(5 - ((2 * 3) + 1))」のようなものがこれにあたる。このような数式の値を計算する関数 compute を作れ。

この問題では各部分式は数値または長さ 3 のリストであり、リストの場合にはその中央が演算、両端がまた式である。(このような再帰的定義は S 式の時にも出てきた。) そこでこれを計算するには次のようにすればよい。

```
(defun compute (exp)
  (cond ((numberp exp) exp)  ; 数値ならそのままその値
        ((eq (cadr exp) '+) (+ (compute (car exp)) (compute (cdr exp))))
        ((eq (cadr exp) '-) (- (compute (car exp)) (compute (cdr exp))))
        ((eq (cadr exp) '*) (* (compute (car exp)) (compute (cdr exp))))
        ((eq (cadr exp) '/) (/ (compute (car exp)) (compute (cdr exp))))))
```

一般に S 式は最初から構造を持っているので、それに応じて様々な処理を行なうプログラムが考えられる。

練習 13 次の関数を書け。

- S 式 X を与えると、その中にある数値の合計を返す関数 ssum。
- S 式 X を与えると、その中にあるアトムの数を返す関数 countatoms。
- S 式 X を与えると、その中の数値をすべて取り除いたリストを返す関数 removenums。
- S 式 X を与えると、その中のすべてのリストを逆向きにしたものを返す関数 reverseall。
- S 式 X を与えると、その中のすべてのアトムを現れた順に並べたリストを返す関数 flatten。

2.2 しらみつぶし

「〜のあらゆる場合を生成する」というような問題、およびその変形として「〜のうちで一番〜なものを探す」というような問題、つまり簡単にいえば「しらみつぶし」を行うのには再帰がとても有用である。例えば次の問題を考えてみる。

> あるリストを集合だと考えて、その集合のすべての部分集合を列挙したリストを返す関数 powerset を作れ。

例えば (a b c) → ((a b c) (a b) (a c) (a) (b c) (b) (c) nil) となる。まず、リストが空なら部分集合としては空集合 (つまり nil) のみしか存在しない。それ以外の場合は、とりあえず最初の要素を除外したものの powerset を計算する。その答は今計算しつつある答に当然すべて含まれている必要がある。加えて、その答の各要素の前にさっき除外した要素をつけ加えたものも含まれている必要がある。それ以外には含まれるべきものはないはずである。というわけで:

```
(defun powerset (s &aux s1)
  (cond ((null s) (list nil))
        (t (setq s1 (powerset (cdr s)))
           (append (prepend (car s) s1) s1))))

(defun prepend (x l)
  (cond ((null l) nil)
        (t (cons (cons x (car l)) (prepend x (cdr l))))))
```

このように、人間が手でやろうとするとかなり面倒な問題が簡単 (?) に書けるところが計算機のいいところなわけである。

練習 14 次の関数を書け。

- 記号のリスト L を受け取り、そのあらゆる並べかえのリストを返す関数 permut。
- 方向つきグラフの各辺を長さ 2 のリストで表すものとし、辺のリスト (つまりグラフの情報) と出発点、ゴール地点を受け取って、出発点からゴールに至る経路を 1 つ求める関数 path。

3 関数引数とラムダ式

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。

ところで、この例のよう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。

練習 15 次の関数を書け。

- 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。

4 小課題その 3 — 整列

課題 2 が結構大ものなので、それと並行してもう少し簡単なのを出します。適宜選んで下さい。整列というのは、要するに「数のリストを受け取り、それを昇順 (または降順) に並べ換える」ということです。その手順 (アルゴリズム) として 2 通りを説明しますので、両方とも作ってください。2 種類のうち最初のものは bsort といいます。これは次のような手順に従います。

- 結果のリストを最初は空にしておく。
- もとのリストから 1 つ要素を取り除き、
- 結果のリストの「正しい位置」に挿入したものを新しい結果のリストとする。
- これを、もとのリストが空っぽになるまで繰り返す。

なお、「正しい位置に挿入」は前にやりましたね? さて、もう 1 つは qsort といいます。これは全体が再帰的になっていて、次の手順によります。

- 与えられたリストが空なら結果は空。
- そうでなければ、先頭の要素をとりあえず X とする。

- もとのリストを X より小さいもの、X に等しいもの、X より大きいものの 3 つののリストによりわける。
- X より小さいもの/大きいものはそれぞれ、`qsort` を呼び出して整列する。
- 最後に正しい順に連結する。

簡単なデータでうまく動いたら、もっと大きなテストデータを用意しましょう。

- 正の数値 N を与えると、リスト ($-N$ N $-(N-1)$ $(N-1)$ ... 0) を返す関数 `testdata` を書け。

この関数で生成したデータ ($N = 100$ くらい) を先の 2 通りの整列プログラムで整列し、所要時間を腕時計で計って下さい。また、この時間の差はどんなところから来ていると思うか、自分なりに考えて説明をつけてみてください。

結果をレポートとして提出する場合には、必ず A4 版の紙を使用し、次のような順で構成し、全体を綴じて提出すること。

- 表紙。課題名 (1993 年度計算機プログラミング課題# 3)、学籍番号、氏名、提出日付) のみを記すこと。
- 方針。どのような考え方で課題を解こうと思ったか記す。
- 回答。作成したプログラムとその解説、実行例など。
- 考察。課題をやった結果どんなことがわかったか、何が問題か、など。(感想も入れてほしいが、感想ばかりでは内容として不足。)

〆切は一応次々回の授業 (10/13) の直前までとしますが、遅れても受理はします。小課題の提出は義務ではありません。