記 号 処 理 52－1
（1989. 9. 29）

Lispでの繰返しの記述について

大田 一久
日本ユニシス

Lispでは読みやすく書きやすいプログラムのための各種の機能を持っており、今日広く使われている手続き型言語に比べて一歩進んでいると言うことができる。しかし、繰返し処理を記述する方法は各種用意されているが、一長一短である。中でも列関数を用いる方法が抽象度が高く記述しやすいが、効率の面で難点がある。本稿では、繰返しが必要となる代表的な局面を考え、各種の方法での記述の比較を試みる。また、特定の場合には列関数を使用しても、効率を悪化させないことができることを示す。さらに、データの並びのすべてがあらかじめ与えられない問題への、同様の記述の適用の可能性を示す。

Iterative Program in Lisp

Kazuhisa Ohta
Nihon Unisys

Lisp has a number of advanced language constructs, which enables us to program in reasonablly high level of abstraction. In addition to functional nature of the language, Lisp has macro feature, structure definition mechanism, generalaized variable and non-local exit constructs. Some of these constructs are not found in other procedural languages widely used today. Though there are several ways to write an iterative program in Lisp, most of them are not sophisticated enough, in contrast with other advanced features. Sequence functions are considered to be attractive to write a iterative program in fairly abstract level. But this method is not widely accepted because of its inefficiency. These iterative constructs are compared in this report, writing a simple iterative program. And possibility to avoid the inefficiency using sequence functions is described. And an idea to extend such a programming style to wider variety of programming context.

1. はじめに

プログラムの生産性、保守性を高めるため、各種のプログラミング言語、プログラミング・パラダイムが工夫されているが、この工夫は人間にとっての読みやすさ書きやすさを改善することであると言ってよい。これは抽象化、すなわち計算機での処理の詳細をなるべく人間から直接見えないようにすることで達成されている。

この抽象化は、次の2つ点とらえることができる。一つは概念を名前、記号で表わすことである。計算機の演算過程を +、- と言った演算子であらわすこと、手続きに名前を付けて引用すること、データ構造の構成要素を名前で参照すること、などがこれにあたる。もう一点は、自由度を制約することでプログラムの各部分の間の関係を簡単にし、組み合わせを容易にすることである。飛越しのかわりに繰り返しを使用すること、手続き呼び出しでモジュール間のデータの受渡しを制限すること、データ構造の参照を決まった手続きで行なうことなどがそうである。

Lispでも、このような抽象化を実現するための各種の機能が用意されている。プログラムを関数の集りとして記述すること、およびマクロによって構文を拡張することにより、手続きの抽象化がなされている。構造体はデータの抽象化を可能にし、汎変数により構成要素に対する参照、更新を統一的に行なうことができる。さらにオブジェクト指向機能の導入により、データ構造とその操作を一括して抽象化することが可能になろうとしている。また、非局所脱出、コンディションなどで例外処理の抽象化も可能にしており、今日広く使われている手続き型言語に比べて一歩先を進んでいると言ってもよいだろう。

その中で繰り返しの記述に関しては、Lispは各種の繰り返し構文を持つものの他の機能に比べて見劣りする感じは否めないだろう。一方で列関数のように抽象度の高い記述を可能にする機能を持ってはいるが、効率の面から広くは使われていないのが実情である。本稿ではLispプログラムでの繰り返しの記述の読みやすさ書きやすさを改善することを目的とし、繰り返し構文、列関数などを例題を用いて比較する。さらに読みやすさ書きやすさの面から好ましいと思われる列関数のような方法に関して効率改善の可能性を検討する。

2. 繰り返し処理を必要とする問題

本節では、繰り返し処理をを必要とする問題としてはどのようなものがあるかを考える。典型的なものは処理の対象とするデータ構造が繰り返しを持つ場合である。繰り返しを持つデータ構造は要素の並びとしてとらえることができる。その表現としてはリスト、ベクタなどが考えられ、それぞれ参照、あるいは作成に関して特有の性質を持っているが、並びを扱う処理は表現とは独立に次の4つの場合に分類できる。一つは要素の並びを走査し、結果として並びではないデータを得る場合である。例えば、要素の並びから条件を満たすものを検索する場合、数列の総和を求める場合などがこれに相当する。2つめの場合は要素の集合から並びを生成する場合である。集合の要素はある定義によって順次作り出され、その順に並べられる。例えば、10以下の自然数の数列を作る場合が相当する。3つめは並びの各要素に関数を適用しその結果の並びを得るもの写像である。例えば数列の各要素の2乗を要素とする数列を作成する場合が相当する。4つめは並びの要素間の順序関係を変えるようなものである。例えば、ソート、逆順に並べ変えること、条件を満たす要素を取り除くことなどが相当する。一般にはこれらの4つの

場合を組み合わせて意味のある処理が記述される。
繰り返しを使用する問題のもう一つの場合として、イベントを処理するようなプログラムがある。コマンド・インタプリタ、ウインドウ・システムなどがその例である。この種のプログラムは、イベントを検知し、対応する処理を行うことを繰り返している。コマンド・インタプリタでは、ユーザから入力されるコマンドがイベントであり、そのコマンドを実行することが対応する処理を行うことになる。このようなプログラムはイベントの時間的系列に対して、対応する処理結果の時間的系列を作り出していることになる。
さらにもう一つの場合として、初期状態をもとに状態遷移を繰り返し、目的の状態になったところで終了するようなプログラムがある。例えば、数値近似を行うプログラム、ネットワークであるノードから到達可能なノードを求めるプログラムなどが相当する。数値近似では、例えば初期値を基によりよい近似値を求めることを繰り返し、近似値の変化が十分小さくなったところでその値を結果とする。この種のプログラムも、状態の系列を生成しその中から条件を満たす状態を見つけることであると考えることができる。また、木構造の探索のようなプログラムも節点をある順序で並べた系列から目的の節点を見つけることであると考えることができる。
処理の対象が有限のデータ構造である場合は、繰り返しの回数はそのデータ構造から明らかになるため、期日を省略できるはずである。コマンド・インタプリタ、状態遷移ループ、木構造探索の場合は繰り返しの回数は事前に明らかではないため、この点で困難さがある。

3. Lispでの繰り返しの記述の比較

次にLispで繰り返しがどのように記述されるかを例題を用いて比較してみる。例題としては、簡単なリスト処理の問題で、与えられた整数のリストから、正の整数を選びその2乗のリストを作成する場合、およびその総和を求める場合を考える。Lispで繰り返し処理を記述する方法としては、次の方法がある。

再帰呼び出しによる方法	線形の再帰呼び出し、末尾再帰呼び出し
繰り返し構文による方法	do、loopマクロ
列を使用する方法	列関数

このほかにtagbody、goでループを構築する方法があるが、省略する。
まず、Lispでは最も基本的な関数呼び出しを使った再帰的定義を考える。まず線形の再帰の深さを要するプログラムは次のようになる。

```
(defun plus-square-lr (l)
  (cond ((null l) nil)
        ((plusp (first l))
         (cons (square (first l)) (plus-square (rest l))))
        (t (plus-square (rest l)))))

(defun sum-plus-square-lr (l)
  (cond ((null l) 0)
        ((plusp (first l))
```

```
        (+ (square (first l)) (sum-plus-square-lr (rest l))))
       (t (sum-plus-square-lr (rest l)))))
```

このプログラムでは、2乗の計算は順に行われるが、結果のリストの生成、あるいは総和の計算は逆順に行われる。そのため、リストの長さに線形のスタックが必要である。これを避けるためには末尾再帰呼び出しによる方法が考えられる。

```
(defun plus-square-tr (l &optional r)
  (cond ((null l) r)
        ((plusp (first l))
         (if (null r)
             (plus-square-tr (rest l)
                             (cons (square (first l)) nil))
             (progn (setf (rest (last r))
                          (cons (square (first l)) nil))
                    (plus-square-tr (rest l) r))))
        (t (plus-square-tr (rest l) r))))

(defun sum-plus-square-tr (l &optional (r 0))
  (cond ((null l) r)
        ((plusp (first l))
         (sum-plus-square-tr (rest l)
                             (+ (square (first l)) r)))
        (t (sum-plus-square-tr (rest l) r))))
```

このプログラムでは、2乗の計算、結果の生成とも順に行われる。リストの生成を先頭から順に行うためにはトリックが必要である。これに対して加算の交換法則のおかげで総和の計算に関してはなんら困難はない。コンパイラの最適化によっては繰り返し構文を用いた場合と同等の効率を得ることができる。リスト生成のトリックを避けるために次のようなプログラムも使われている。

```
(defun plus-square-tr-alt (l &optional r)
  (cond ((null l) (nreverse r))
        ((plusp (first l))
         (plus-square-tr-alt (rest l)
                             (cons (square (first l)) r)))
        (t (plus-square-tr-alt (rest l) r))))
```

ただし、これは nreverse のために結果の長さに線形の時間が余分に必要である。
Lisp の繰り返し構文、do を用いると次のようなプログラムになる。

```
(defun plus-square-do (l)
  (do ((s l (rest s)) (r nil) (c nil))
      ((null s) r)
    (let ((n (cons (square (first s)) nil)))
      (if (null c) (setq r n)
```

```
            (setf (rest c) n))
          (setq c n))))

(defun sum-plus-square-do (l)
  (do ((s l (rest s))
       (r 0 (+ (squre (first s)) r)))
      ((null s) r)))
```

リストを逆順に作成して最後に並べかえるテクニックはこの場合にも使われる。問題によっては、dolist、dotimes など類似の構文が使われる。効率はおそらく今回比較する方法の中では最もよい。
loop マクロでは次のようなプログラムになる。

```
(defun plus-square-lp (l)
  (loop for e in l
        when (plusp e)
        collect (square e)))

(defun sum-plus-square-lp (l)
  (loop for e in l
        when (plusp e)
        sum (square e)))
```

終了条件の判定がなくなり、要素の選択、結果の生成に関しても loop マクロの提供するキーワードで表現できるため、抽象度の高いプログラムである。特にリストを生成するテクニックを使用する必要がない点がよい。実際にはマクロ展開で tagbody、go を使った形にされるため、繰り返し構文と同等の効率が得られる。また、手続き型言語の経験を持つプログラマにとってはなじみやすいという利点も持っている。
列関数を使用した場合は次の通りである。

```
(defun plus-square-sq (l)
  (map 'list #'square (remove-if-not #'plusp l)))

(defun sum-plus-square-sq (l)
  (reduce #'+ (map 'list #'square (remove-if-not #'plusp l))))
```

この方法も繰り返しの終了条件は陽に表われず、抽象度の高いプログラムである。この例題のように入力が並びとして与えられる場合は、loop マクロ、列関数が向いているといえる。また、リスト生成のテクニックが不要である点も loop マクロと同様に優れている。列関数が特に優れている点は、要素の除去、写像、縮約といった処理が基本的な操作として高階関数の形で提供され、その組合せでプログラムを書くことができることである。プログラムを構成する各部分が単独で意味を持ち、再利用が可能である。例えば、この例題の総和を求めるプログラムは2乗を求めるプログラムを用いて次のように書くことができる。

```
(defun sum-plus-square-sq-alt (l)
```

```
(reduce #'+ (plus-square-sq l))
```

loop マクロや繰り返し構文では、対象とするデータ構造が決まれば、プログラムのパターンはほぼ決まってくるものの、列関数の場合のように簡単に組合せを作る訳にはいかない。

一方、列関数の最大の難点は、最終的には不要な中間結果の並びが作成される点である。この例題の総和を求める場合でも、条件を満たさない要素を除去したリスト、要素の2乗のリストが作成されてしまう。また、このような関数型の記述は、手続き型言語の経験を持つプログラマにとっては理解しにくいかもしれない。

以上の評価結果をまとめると次のようになる。抽象度では、loop マクロ、列関数、組み合せのしやすさでは、列関数、効率では、繰り返し構文、loop マクロが優れている。

4. 効率を損わない方法

前節までに見たように、データの並びを処理するプログラムでは、列関数を用いることによりプログラムがわかりやすくなるが、不必要に記憶領域を消費してしまう欠点がある。効率の点を考慮すると loop マクロが最も優れている。列関数のプログラムを書いて、繰り返し構文を用いた形に自動的に変換できれば、効率を損うことなく読みやすさ書きやすさを高めることができ、最も望ましい。

この目的を達成するために、列関数と同等の関数の組合せに関して繰り返し構文への展開を試みる機能をコンパイラに最適化として組込む実験を行った[5][6]。この実験では、要素の並びを表現するデータ構造をリストに限定し、以下に示す列関数と同等の関数の直接の組合せを対象として展開を行った。

map-list、remove-if-list、find-if-list、position-if-list、count-if-list、reduce-list

この機能を組込んだコンパイラを用いると、3節で扱った問題について次のようなプログラムを与えると、do を使った場合と同様のプログラムにソース・プログラム・レベルで展開されてコンパイルされ、中間結果のリストは生成されない。

```
(defun sum-plus-square-op (l)
  (reduce-list #'+
     (map-list #'square
        (remove-if-list
           #'(lambda (x) (not (plusp x))) l))))
```

この最適化では、上記の関数の直接の組合せしか認識できないため、使用者がこれらの関数を用いて新たに定義した関数の組合せについては展開されない。また、2ヵ所以上で使用されている中間結果は除去できない。例えば、次のプログラムの中間結果は除去されない。

```
(defun squre-plus (l)
  (let ((s (remove-if-not #'plusp l)))
    (map 'list #'* s s)))
```

また、当然のことながら関数の値として返されるリスト、あるいは他のデータ構造の中へ格納されるリストを除去することはできない。さらに、ソート、リバースなど、入力の並びの要素を全部走査したのち初めて出力の並びを作り出すようなプログラムは、入力を格納するバッファが必要であり、列関数の組合せの中で用いられても入力の並びをいったん作成しなければならない。

同じ目的をマクロ展開で実現しようとしたものとして、OSSマクロがある[4]。中間結果として使われる並びをシリーズと呼ばれるデータ構造と考え、これを扱う列関数と同等のマクロを用意している。さらに、2ヵ所以上で使用される中間結果、および使用者が定義した関数の作り出す中間結果を除去するため、これらを明確に記述する構文を用意している。これを使うと3節の例題は次のように書かれる[4]。

```
(defun plus-square-oss (l)
  (Rlist (square (Tplusp (Elist l)))))

(defun sum-plus-square-sq (l)
  (Rsum (square (Tplusp (Elist l)))))
```

一般には、先に述べた場合も含めて並びを取扱う関数の組合せで書いたプログラムで全ての中間結果が除去できるわけではない。プログラマとしては中間結果が除去できる保証がなければ、繰り返し構文で書き下すほうを選ぶかもしれない。OSSでは、要素が処理されるタイミングに関しての制約を設け、これを満たす限りは中間結果の除去が保証されるというアプローチをとっている[4]。

5. 無限の並び

3節の例題では有限の並びを取り扱う場合のみを考えたが、無限の並びを考えることができれば、イベント・ループ、状態遷移ループ、木構造探索に関しても抽象度の高いプログラムを書くことができるだろう。

例えば、次のような仮想的に単純化したコマンド・ループは次のように書ける。

```
(defun command-loop-do ()
  (do ((input (get-input) (get-input)))
      ((member input '(end exit quit bye) :test #'eq) nil)
    (process-input input)))
```

無限の並びが扱えるとすると、次のようなプログラムを考えることができる。

```
(defun command-loop-sq ()
  (map nil #'process-input
       (trim-if #'(lambda (input)
                    (member input '(end exit quit bye)
                            :test #'eq))
                (generate-input-sequence))))
```

ここで、generate-input-stream は入力の無限の並びを生成する関数で、trim-if は条

件を満たす要素が見つかるまで並びをコピーする関数とする。
無限の並びを遅延評価を用いて取り扱う方法は知られており[1][6]、効率を考えなければこの種のプログラムを直接実行することが可能である。特に木構造探索を記述した例[6]では、節点の生成とその検査という形でプログラムを書くことができた。中間結果の除去がこのような場合にも適用できれば、実用的なプログラムの記述法として広く使われるようになることが期待できる。

6. おわりに

列関数を用いたプログラムで、ある制約を満たせば読みやすさ書きやすさと効率を両立させることができる場合がある。ただ、最初に見たように歴史的に見てもプログラムの自由度と引換えに、生産性、保守性を高めてきたわけで、自由度に関する制約はある程度受入れられるべきものと考える。その一方で、有用なプログラムがこの制約にかかるようではこのスタイルのプログラミングが広く受入れられるようにはならないだろう。その意味では、本稿、あるいは参考文献でとりあげられているような小さな例ではなく、実用的な規模でのシステムの記述での使用経験をもとに評価を行ってみたい。また、無限の並びを扱う場合について中間結果を除去する技法の拡張の可能性を検討したい。

参考文献

[1] H.Abelson, G.J.Sussman,
Structure and Interpretation of Computer Programs, MIT Press, 1985.
[2] G.L.Steele Jr., Common Lisp: The Language, Digital Press, 1984.
[3] P.Wadler, Listlessnes is Better than Laziness:
Lazy Evaluation and Garbage Collection at Compile Time,
Proc. ACM '84 Lisp and FP Conference, 1984.
[4] R.Waters, Efficient Interpretation of Synchronizable Series Expressions,
Proc. ACM SIGPLAN '87
Symposium on Interpreters and Interpretive Techniques,
ACM SIGPLAN Notices, 1987
[5] 大田、コンパイル時ガーベジ・コレクション、
- Common Lisp 列関数への応用 -、
日本ユニバック・テクニカル・シンポジウム'87、1987
[6] 大田、ストリームを用いた論理型言語インタプリタ、
UNIVAC Technology Review 第11号、1986
[7] Compiler Operations, Explorer System LISP Reference,
Texas Instruments Incorporated, 1987