プログラムと対称性

上　田　和　紀†

プログラム，特に書換えに基づくプログラムにおける対称性とその意義について例を通じて考える．対称性を意識したプログラムは，プログラム不変量や可逆性など，単に美しいという以上の価値を持つように思われ，また対称性はプログラミングおける思考にも大きな役割を演ずる．

Programs and Symmetry

Kazunori Ueda†

Discussed in this paper is symmetry in programs based on rewriting and its implications. Programs written bearing symmetry in mind exhibit important properties such as invariants and reversibility, which are something more than beauty, and symmetry plays important roles in our thought in programming.

1. 二種類の対称性

対称性 (symmetry) は，自然界や数学や芸術などあらゆる場面で観察され，重要な意味をもつ性質である．

対称性という言葉から誰でも連想するのは，紋様，漢字の字形，動物のからだ，AND ゲートの二つの入力などに見られる対称性，名付けて**表と表の対称性**である．しかし本論文で注目したいのは，別の種類の対称性，名付けて**表と裏の対称性**である．表と裏の対称性とは，以下のような場面でみられる対称性である．

(1) 織物や衣服の表裏 — 裏返しても鑑賞に堪える，あるいは着用できるリバーシブルものは 2 倍の価値があるように思える．

(2) 人生 — 老いて次第に子供に近づき，静かに完結する一生はひとつの理想形であろう．

(3) 登山と下山 — 登山という言葉は山行の半分しか表現していない．

(4) プログラムの実行 — あるクラスのプログラムの実行は完全に後戻りができる[1]．

表と裏の対称性は表と表の対称性ほど自明ではないが，それゆえに論考の対象として深いものがありそうである．本論文で考察するプログラムの対称性も，オペランドの可換性のような性質でなく，入力と出力，もしくは個々の書換え規則の左辺と右辺に見られる対称性である．

† 早稲田大学理工学部コンピュータ・ネットワーク工学科
Dept. of Computer Science, Waseda University

2. LMNtal によるリスト連結

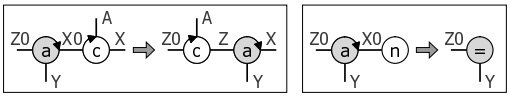
本論文では，2002 年の夏のシンポジウムでも紹介した LMNtal[3] のプログラムを扱うことにする．LMNtal は階層グラフ書換えモデルに基づく並行言語であり，プロセス構造もデータ構造も同じようにグラフ構造として扱うことを特徴としている．処理系も動き始めていて，`http://www.ueda.info.waseda.ac.jp/lmntal/` からダウンロードできる[2]．

たとえば，リストを連結するプログラムの二つの書換え規則は，LMNtal では

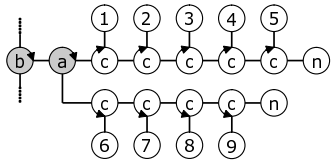
```
a(X0,Y,Z0), c(A,X,X0) :-
    c(A,Z,Z0), a(X,Y,Z)
a(X0,Y,Z0), n(X0) :- Y=Z0
```

と書ける．Prolog などと異なるのは，通常は手続きや関数とみなされる `append` アトム☆（プログラム中では `a` と略記）と，通常はデータと見なされる `cons` や `nil` アトム（プログラム中ではそれぞれ `c`, `n` と略記）とが同格である点である．`cons` や `nil` は通常より 1 個多い引数をもち，最後の引数がその値の利用者にリンクされる．**図 1** に，これらの規則の図形表現 (a) と実行例 (b)〜(d) を示す．(a) を見ると，再帰ステップが `append` と `cons` を交換するものであることがよくわかる．またこの規則からは対称性が見て取れる．つまり，左辺と右辺は同じアトムをもち，さらに

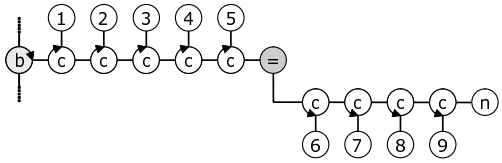
☆ LMNtal のアトムは，Lisp のアトムとは異なり，決まった本数（結合価）のリンクをもつ原子のことを指す．たとえば `append` は 3 価のアトムである．



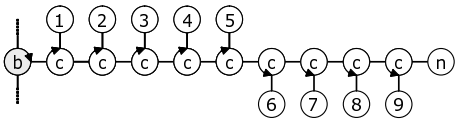
(a) 書換え規則



(b) 実行例：初期状態



(c) 実行例：最終状態 1



(d) 実行例：最終状態 2

図 1　リスト連結プログラム

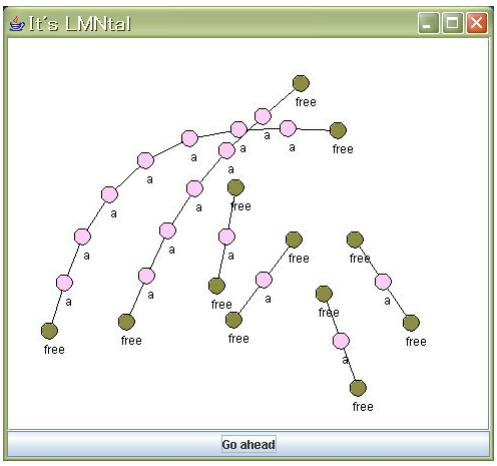
同じ自由変数をもつ．しかし完全に対称なわけではなくて，接続関係が変化する．完全に対称ならば左辺と右辺の両方を書く必要はないわけで，対称性は「少し崩れている」ことが重要であるように思える．

append の再帰規則は可逆である．つまり，計算を何ステップか進めたあとで再帰規則の左辺と右辺を入れ替えると，計算は逆向きに進んで出発点に戻る．もちろん，可逆性は対称性と密接な関係にある．append の計算が可逆性を失うのは最後の瞬間，つまり nil に遭遇して非再帰規則が適用されるときである．このとき，append の第 2 引数と第 3 引数はいったん =（コネクタと呼ぶ）で相互接続されるが（図 1(c)），コネクタは隣接する他のアトムが吸収することができる☆ので，結果としてできる（長い）リストのどこまでが第 1 引数から来たのかがわからなくなる（図 1(d)）．もし，自由に消滅できる = のかわりに別の適当な 2 価のアトムを使って

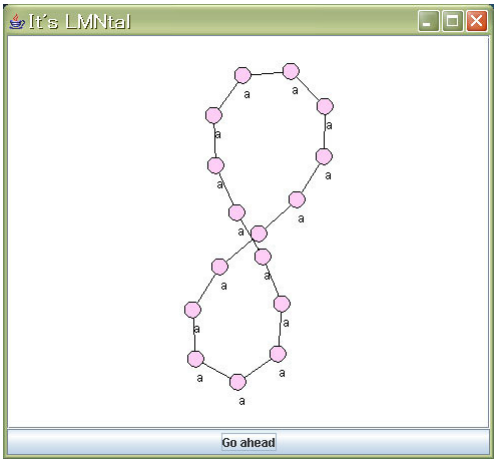
```
append(X0,Y,Z0),n(X0) :- m(Y,Z0)
```

などとすれば，結果のリストの中に m が残るものの，

☆ これは LMNtal の構造合同規則[3] によっている．



(a) 途中状態



(b) 最終状態

図 2　輪の形成

どこでつないだのかがわかるようになる．

3. LMNtal による自己組織プログラミング

LMNtal はまた，自己組織化原理による構造形成が容易に記述できることを特徴とする．たとえば，

```
a(A,B),free(B),free(C),a(C,D) :-
    a(A,X),a(X,D).
```

という 1 行プログラムを

```
free(L0),a(L0,R0),free(R0),
free(L1),a(L1,R1),free(R1),
...
free(Lf),a(Lf,Rf),free(Rf)
```

という初期分子集合を与えて上の処理系で走らせ，グラフィックモードで観察すると，個々の分子が手をつないで**図 2** のように大きな輪ができてゆく．

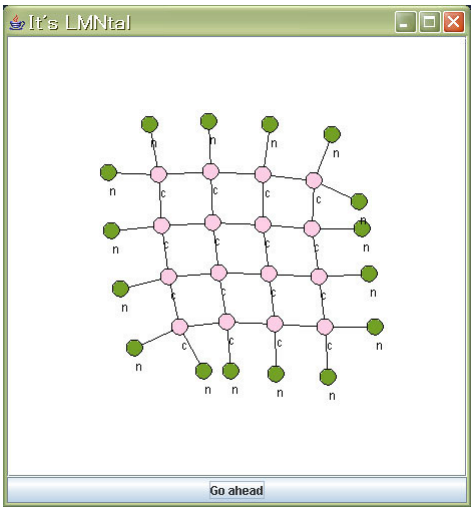


図 3 格子の形成

また，

```
c(X01o,Y01o,X00,Y00), c(X02,n,X01o,Y10),
c(n,   Y02, X10,Y01o) :-
  c(X01,Y01,X00,Y00), c(X02,Y11,X01,Y10),
  c(X11,Y02,X10,Y01), c(n,  n,  X11,Y11)
```

という 1 行プログラムと

```
c(A,D,n,n),c(B,n,A,n),c(C,n,B,n),c(n,n,C,n),
c(n,F,n,D),
c(n,G,n,F),
c(n,n,n,G)
```

という初期分子を与えると，**図 3** のような格子が形成される．つまり，上端の一辺と左端の一辺だけ与えておけば格子が補完されてゆく．ここで `c(A,D,n,n)` は `c(A,D,X,Y),n(X),n(Y)` の略記法である．(このように，アトムの最後の引数を省略したものを，省略した引数の相手方の出現位置に書き込んでもよい．この方法で，定数やリストや木構造の通常の表記が可能になる．略記法の正確な規則は文献 3) を参照してほしい.)

これらのプログラムも次の意味で可逆性をもつ．つまり規則の左辺と右辺を入れ替えて元のプログラムの最終状態に作用させると，元のプログラムの初期分子を復元することができる．

ボトムアップパーザも構造形成の一例である．**図 4** は簡単な演算子順位文法に基づく数式パーザである．規則集 (a) と演算子表 (c) とをトークンの線形リストに作用させると，結び付きの強い演算子の周辺から，ボトムアップかつ並行に木構造が形成されてゆく．たとえば分子

```
ans([begin,123,'+',45,'*',67,'+',8,end])
```

```
X0=[$opL,exp(E1),$op,exp(E2),$opR|X],
op($opL1,$pL),op($op1,$p),op($opR1,$pR) :-
    $opL1=$opL, $op1=$op, $opR1=$opR,
    $pL =< $p, $p >= $pR |
    X0=[$opL,exp(e($op,E1,E2)),$opR|X],
    op($opL1,$pL),op($op1,$p),op($opR1,$pR).

X0=[N|X] :- int(N) | X0=[exp(N)|X].
```

(a) パーザ

```
X0=[$opL,exp(e($op,E1,E2)),$opR|X],
op($opL1,$pL),op($op1,$p),op($opR1,$pR) :-
    $opL1=$opL, $op1=$op, $opR1=$opR,
    $pL =< $p, $p >= $pR |
    X0=[$opL,exp(E1),$op,exp(E2),$opR|X],
    op($opL1,$pL),op($op1,$p),op($opR1,$pR).

X0=[exp(N)|X] :- int(N) | X0=[N|X].
```

(b) アンパーザ

```
op('*',4).  op('*',4).  op('*',4).
op('+',3).  op('+',3).  op('+',3).
op(begin,0).  op(end,0).
```

(c) 演算子表

図 4 ボトムアップパーザとアンパーザ

をパーザに与えてみよう．この分子は

```
ans(X1), .(A1,X2,X1), .(A2,X3,X2), ...,
.(A9,X10,X9), [](X10),
begin(A1), 123(A2), '+'(A3), ..., 8(A8),
end(A9)
```

という分子を Prolog 風のリスト表記法を用いて略記したものであり，`begin` と `end` は式の両端を示す演算子である．反応が終わったとき，初期分子は

```
ans([begin,
  exp(e('+',123,e('+',e('*',45,67),8))),
  end])
```

に変形している．なお図 4(a) のパーザの記述では，数値やアトムどうしの比較をするために型付きプロセス文脈という記法（`$` から始まる名前）を使った．図 4(c) の演算子表に `'*'` と `'+'` が 3 回ずつ登録してあるのは，パーザの左辺で演算子の 3 個の出現を同時に検査するからである．

パーザプログラムもまた，トークンやリンクの総数を保存するプログラムであり，しかも逆実行によって

トークン列に戻すことが可能なプログラムである．実際，図 4(a) の各規則の左辺と右辺を機械的に入れ替えると (b) の規則集ができあがる．これはアンパーザとしてちゃんと動作して，元の

```
ans([begin,123,'+',45,'*',67,'+',8,end])
```

が復元される．

4. 対称性と zero-emission 計算

プログラムの対称性に注目するひとつの動機は，ゴミを出さない計算 (zero-emission 計算) との関連である．

プログラミング言語とその処理系の研究において，自動記憶管理は輝かしい歴史を誇ってきた．プログラマやアルゴリズムは求解に集中し，記憶管理は実行系に任せ，実行系はできる限り上手に記憶域の割当てと回収を行うのがスマートとされてきた．この方針によって実現されてきた計算を 20 世紀型計算と呼ぶことにしよう．C と Java，Java と関数型言語を互いに比べてみれば，逐次計算機上のプログラミングに関する限り，この王道にはかなりの説得力がある．

しかし，我々のプログラムが走る計算環境は，物理スケールの点でどんどん多様化してきている．並列分散計算環境，実時間環境，組込み環境などでは，好むと好まざるとにかかわらず，計算の論理的側面と同時に，物理的・資源的側面をアルゴリズム設計レベルから考えなければならなくなってくる．

また逐次型計算においても，計算の上流工程（求解）だけでなくて下流工程（使った資源のリサイクル）まで陽に考えたプログラミングはどのようなものになるかは，大変興味のあるところである．このように，下流工程をきちんと考慮してアルゴリズムに組み込んだ計算を，**21 世紀型計算**と呼びたい．

21 世紀型計算には，

- 実行系が超軽量ですむ
- プログラムの挙動の予測・検証の可能性が増す

などの利点がある．しかし，21 世紀型計算が 20 世紀型計算にとって代わるかどうかは明らかではない．とって代わるには

- リサイクルプログラミングを支援するプログラミング言語
- ごみのリサイクル，分別，持ち帰りの方法論

などが必要となるが，これらを確立することは，計算の本質を解明するための重要なステップであると考えている．

5. append の対称化

append の再帰規則は，アトムとリンクの数に関して対称である．つまり資源的には均衡していて，規則を適用するとアトムの接続関係だけが変更される．

しかし，非再帰規則はどう見ても対称はでない．適用すると 1 個の append と 1 個の nil が余ってしまう．append が余るのは連結作業が終わるからで，nil が余るのは 2 本のリストを 1 本にするからである．余ってもよいではないかという考えもあるが，ともかく資源的に均衡させてみよう．

均衡させる方法は一通りとは限らない．要は，アトムがすべて保存され，自由リンク（書換えの対象にならない部分とつながっているリンク）がすべて保存され，しかもアトムのすべての引数がどこかと 1 対 1 に接続されていればよい．

しかし append の場合は，まずは入出力端子の数を対称化するところから始めるのが筋が良さそうである．つまり，append の仕様を 2 入力 1 出力から 2 入力 2 出力に拡張してから均衡化に着手するのである．その方針で作ってみたのが下記のプログラムである．

```
append(X,Y,Z,U), n(X) :-
    Y=Z, append(V,W,V,W), n(U).
append(X,Y,Z,U), c(A,X1,X) :-
    c(A,Z1,Z), append(X1,Y,Z1,U).
```

このプログラムは，再帰規則に関しては，同一リンク U を持ち回る第 4 引数を追加しただけである．一方，非再帰規則については，余った nil を第 4 引数から返却する作業と，余った append を遊休資源として放出する作業とを追加している．放出する append の引数は，append の 2 本の規則が適用できないような形で適当に相互接続しておく．

上に示した 4 引数 append とオリジナルの 3 引数 append との間には重要な関係がある．それは，後者が前者の**プログラムスライス**になっているということである．もし 4 引数 append のようなものが誰にも容易に書けるのならば，3 引数版はそこからほぼ機械的に導くことができる．逆に，プログラマには 3 引数の append を書かせ，そこから機械的にリソースコンシャスな 4 引数版を構成したいと思う人もいるかもしれない．これがどれくらい自動的にできるかは，興味深い研究課題である．

6. マージソートの対称化

append よりも複雑な例として，リストのマージソートについても対称化を施してみよう．n 要素のリスト

のマージソートは，入力をまず n 個の 1 要素リストを要素とするリストに変換し，その隣接要素どうしをマージしてゆく．n 個の 1 要素リストを作るにはのべ n 個の cons とのべ n 個の nil が必要である．一方，2 本のリストのマージは 1 個の nil を返却するが，マージソートではそれらの 2 本のリストはばらばらにではなくて，cons でつながって来る．マージが終わるとリストが 1 本減るので，上述の 1 個の nil に加えて，リストどうしをつなげていた cons が 1 個返却される．きちんと収支が合っている．

さて，要素数 n のリストをマージソートするのに，補助資源として最初から n 個の cons と n 個の nil を投入すれば十分なのは明らかである．だがこれだけの数がどうしても必要というわけではない．最低何個くらいの cons と nil を投入すれば足りるのだろうか？

答は $O(\log_2 n)$ 個である．1 要素リストを作る作業 (cons と nil を消費) とマージ作業 (cons と nil を返却) を並行動作させ，マージが返却する cons と nil を直ちに 1 要素リスト作成のためにリサイクルすればよい．そうすれば，$O(\log_2 n)$ 個の補助資源で工場は操業を続けることができる．この補助資源量は，手続き型言語で書いたマージソートが消費するスタック量と合致する．

実際に，cons と nil を還流するマージソートプログラムを LMNtal で書いた（**図 5**）．この版は，cons と nil に関して完全に均衡化してあり，すべての規則がその個数を保存する．その他のアトム（手続き名に相当）に関しては均衡化していないが，前節の append にならって均衡化するのは容易（でおそらく自動化可能）な作業である．

得られたプログラムは一見煩雑だが，このプログラムは初期投入した cons と nil だけを補助資源として使い回して作業を行い，ソーティングが終わるとそれらをすべて第 4 引数から返却する．途中で追加資源を要求したり捨てたりしない，几帳面なプログラムである．

```
mergesort([3,1,4,1,5,9,2],[ [],[],[] ],
    result1,result2).

mergesort(Xs,InitNs,Ys,FinalNs) :-
    append(InitNs,ReturnedNs,Ns,N),
    wrapElem(Xs,Ns,Ws,FinalNs),
    mergeMany(Ws,Ys,ReturnedNs,N).

wrapElem([],Ns0,Ws,Ns) :- Ws=[], Ns=Ns0.
wrapElem([X|Xs],[N|Ns0],Ws0,Ns), N=[] :-
    Ws0=[W|Ws], W=[X], wrapElem(Xs,Ns0,Ws,Ns).

mergeMany([], Ys,Ns,N) :- Ys=[], Ns=N.
mergeMany([W],Ys,Ns,N) :- Ys=W,  Ns=[N0|N], N0=[].
mergeMany([W1,W2|Ws],Ys,Ns,N) :-
    mergeOneLevel([W1,W2|Ws],Zs,Ns0,N1),
    mergeMany(Zs,Ys,Ns1,N),
    merge(Ns0,Ns1,Ns,N1).

mergeOneLevel([], Ys,Ns,N) :- Ys=[], Ns=N.
mergeOneLevel([W],Ys,Ns,N) :- Ys=[W], Ns=N.
mergeOneLevel([W1,W2|Ws],Ys0,Ns0,N) :-
    mergeTwo(W1,W2,W,N0), Ys0=[W|Ys], Ns0=[N0|Ns],
    mergeOneLevel(Ws,Ys,Ns,N).

mergeTwo([],Ys,Zs,N) :- Zs=Ys, N=[].
mergeTwo(Xs,[],Zs,N) :- Zs=Xs, N=[].
mergeTwo([X|Xs],[Y|Ys],Zs0,N) :- X=<Y |
    Zs0=[X|Zs], mergeTwo(Xs,[Y|Ys],Zs,N).
mergeTwo([X|Xs],[Y|Ys],Zs0,N) :- X >Y |
    Zs0=[Y|Zs], mergeTwo([X|Xs],Ys,Zs,N).

merge([],Y,Z,N) :- Z=Y, N=[].
merge(X,[],Z,N) :- Z=X, N=[].
merge([A|X1],Y,Z,N) :- Z=[A|Z1], merge(X1,Y,Z1,N).
merge(X,[A|Y1],Z,N) :- Z=[A|Z1], merge(X,Y1,Z1,N).

append([],Y,Z,N) :- Z=Y, N=[].
append([A|X1],Y,Z,N) :-
    Z=[A|Z1], append(X1,Y,Z1,N).
```

図 5 マージソート

7. おわりに

書換えに基づくプログラムの対称性とその意味について考察した．対称性の第一歩は資源の完全均衡である．ガーベジコレクタに依存するのでなくリサイクルパスを明示することは，プログラミングの手間を増やすものではあるが，それによって計算の本質に迫ることができよう．また，プログラムに関する推論を容易にすることも期待される．

対称性の応用の次のステップは可逆性である．可逆計算の研究は長い歴史をもち，特に計算の消費するエネルギーの物理的限界の観点から注目されてきた．しかし，可逆性は論理的な観点からも興味深い．理論的には，どんな Turing 計算可能な関数も可逆 Turing マシンの計算（の射影）として表現可能なことがわかっている[1] が，より実用的な観点あるいはプログラムの美学の観点から，プログラムの中で可逆性がいかに保存され，いかに破られるか，また可逆性をいかに回復できるかを考えてみるのは面白いことであろう．

参考文献

1) Bennett, C. H.: Logical Reversibility of Computation, *IBM Journal of Research and Development*, Vol. 17, No. 6, pp. 525–532 (1973).

2) 原 耕司, 水野 謙, 矢島 伸吾, 永田 貴彦, 中島 求, 加藤 紀夫, 上田 和紀: LMNtal 処理系および他言語インタフェースの設計と実装, 情報処理学会第 50 回プログラミング研究会 (SWoPP2004), July 2004.
3) 上田 和紀, 加藤 紀夫: 言語モデル LMNtal, コンピュータソフトウェア, Vol.21, No.2, pp.44–60 (2004).
4) Ueda, K., Resource-Passing Concurrent Programming. In *Proc. TACS 2001*, LNCS 2215, Springer-Verlag, 2001, pp. 95–126.

付　　録

A.1 質 疑 応 答

- Q:（多田・電通大）オペレーティングシステムの観点からは，ゴミの分別はいらないような気もするが.
  A: 厳密なリサイクルパスが不要であれば，ヒープというデータ構造を一つ作って持ち回り，そこからアトムを取得開放する方法もある．しかしそれよりも精密な，たとえば用途別やサイズ別の記憶管理もプログラムできることを示した.
- Q:（宮原・上武大）アドレススペースが 64bit になろうとしているときに資源のリサイクルの意義が増大してゆくだろうか？
  A: 汎用機のアドレススペースが大きくなる一方で，組込みシステムなどのメモリ量の少ない計算環境もますます普及するかもしれない．また，アドレススペースが大きくなっても，メモリアクセスの局所性は依然として重要である．そのため，リサイクルのことを真剣に考えたいのである.
- Q:（寺田・電通大）append の非再帰節はどう読むのか？
  A: 第 1 引数が nil につながっている append を見つけたら，次の三つの作業をする：
  - その nil を，第 4 引数から出ているリンクの相手方につなげる.
  - 第 2 引数（から出ているリンク）と第 3 引数（から出ているリンク）とを相互接続する．言い換えると第 2 引数の相手方と第 3 引数の相手方とをリンクでつなげる.
  - これで，もらった append のすべての引数は使い終わったので，4 つの空き場所ができる．そこを適当にリンクでつなげる．ここでは第 1 引数と第 3 引数，第 2 引数と第 4 引数を互いにつなげてみた.（食べ終わった駅弁の箱をひもで縛るような気分である.）
- Q: cons はともかくとして，append を返してどうするの？
  A: 世の中全体の append の総数を管理したい場合を考える．この場合，append を使いたくなったら，遊休 append を入手して引数をセットアップして起動する．append の実行が終了したら，遊休 append の形に戻して返却する．この約束を守れば，初期状態で遊休 append を n 個用意したら、同時には最大 n 個の append しか起動されないことが保証される.
- Q:（多田・電通大）マージソートは $\log_2 n$ 個の cons と nil で足りるのか？
  A: 還流が滞らないようにすれば足りる．そのために図 5 のプログラムでは非決定的にアトムを還流するネットワークを構成している．なお，初期投入数が 1 個でも足りないと計算が途中で止まる.
- Q:（近山・東大）第 3 節のリング形成プログラムは，大きな一つの輪ができるとは限らないのではないか？
  A: 鋭い指摘である．現在の処理系では，書換え対象アトムの標準の選択方法では必ず大きな一つの輪ができるが，アトムの選択方法をランダムにするオプションを指定すると，小さな輪が複数できることもある.