B2　LISPのデータ構造についての2,3の考察

佐　藤　浩　史（お茶の水女子大学理学部）
後藤英一，野崎昭弘，野下浩下　（東京大学理学部）

はじめに

ＬＩＳＰは，非数値問題向きのプログラミング言語として広く使われていて，国内でも既に多くの処理系が作られているようである．また最近ＰＬＡＮＮＥＲ，ＲＥＤＵＣＥ，ＭＡＴＨＬＡＢ等のようにＬＩＳＰをその基礎においたプログラミングシステムがいくつか報告されている．我々もこのような情勢のもとでＬＩＳＰの処理系を作成中であるが，本稿では，その検討中に考察した問題や実現した部分の結果について，特にＬＩＳＰデータの計算機内での内部表現（データ構造）に焦点をあてて，報告する．

Ⅰでは，ＬＩＳＰで生成されるデータ構造，Ⅱでは，函数eq によるデータ構造の識別，Ⅲでは，入出力，Ⅳでは，データ構造の縮約について考察する．

本稿で，ＬＩＳＰとは もろもろのＬＩＳＰないしはその処理系の総称，ＬＩＳＰ1.5 とは，〔1〕 のこと，pure ＬＩＳＰとは，〔1〕 の第1章のものをそれぞれ指すものとする．ただし本稿では，専らデータ構造に注目しているので，Pure ＬＩＳＰといった場合，それがrplacaやrplacd等によってデータ構造を改変させる演算をふくまなければ，もっと広いＬＩＳＰと解釈することができるであろう．

記法として，□□ でcarとcdrポインタをもつ1つのセル（ cell ）を示す．セル中の，たとえば，ＡとかＢは，実際はatomＡとかＢへのポインタを示すものとする．また議論を簡単にするため，atomは，property list 等その内容にまで立入らないことにする．

Ⅰ　データ構造の生成

ＬＩＳＰでは，外部表現がＳ－表現を原則としているのに対して，内部表現としてのデータ構造が，acyclic directed graph（ loop をもたない方向付グラフ；特別の場合としてtree もふくむ）であり，さらに一般にはgeneral directed graph（ circulav list も許す）を取扱えるようになっている．

さて基本函数cons は，free storage より新しい1つのセルをもってくる．そこでたとえば，

d_1 ：〔λ〔〔x；y〕；cons〔x；y〕〕〕〔Ａ．Ｂ）；（Ａ．Ｂ）〕……(1)

では，tree が生成されるのに対して，

d_2 ：〔λ〔〔x〕；cons〔x；x〕〕〕〔（Ａ．Ｂ）〕 ……(2)

では，そうはならない（図2）

d_1，d_2 の一般化として，函数bt，ctを考えよう．

$bt〔n；x；y〕$ ::＝〔$n=0$ → cons〔x；y〕；

T → cons〔$bt〔n-1；x；y〕$；$bt〔n-1；x；y〕$〕〕 ……(3)

$ct〔n；x；y〕$ ::＝〔$n=0$ → cons〔x；y〕；

T →〔λ〔〔u〕；cons〔u；u〕〕〕〔$ct〔n-1；x；y〕$〕〕 ……(4)

なお(3)，(4)は，ＬＩＳＰの正記法ではない．正記法では，たとえば(3)は，

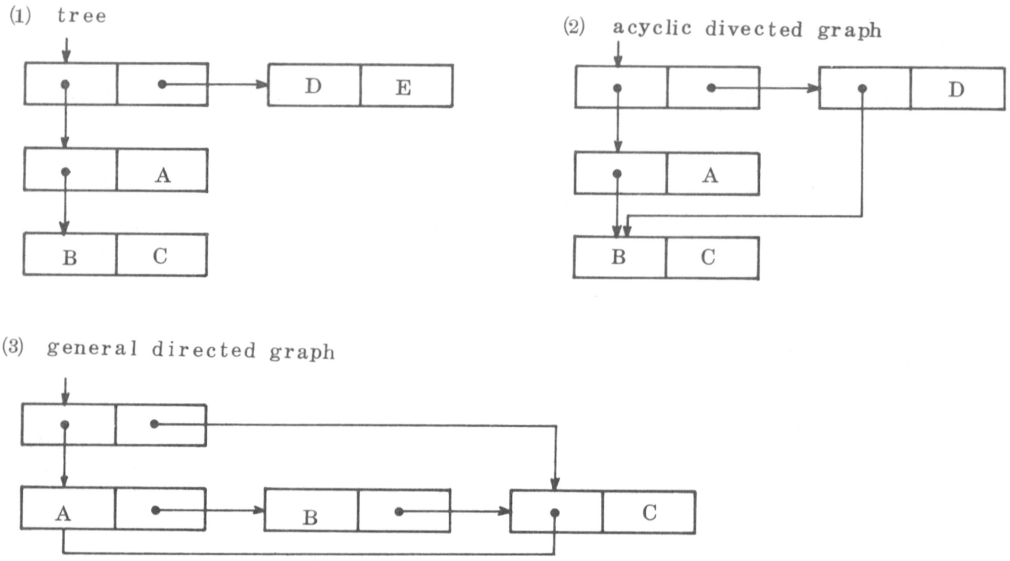


図1. データ構造の例

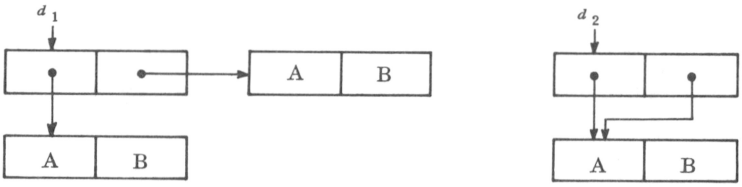


図2. d_1 と d_2

define〔bt；λ〔〔n；x；y〕；

〔zerop〔n〕→eons〔x；y〕；

T→〔cons〔bt〔sub1〔n〕；x；y〕；bt〔sub1〔n〕；x；y〕〕〕〕〕〕

となるが，記号 :: = や $n=0$ 等は，その意味が明らかであろう．以下ではこの記法を用いることが多い．

(3)および(4)でつくられるデータ構造は，図3のようになる．

d_1 と d_2 は，各々 bt〔1；A；B〕，ct〔1；A；B〕である．

次に集合 E を考えよう．

$E=\{\ l/\text{equal}〔l；bt〔2;A;B〕〕=T\ \}$．

ここで l_1，$l_2 \in E$ のとき，graph l_1 のnode で（equal の意味で）同等なgraph のうちいくつかを一致させることによって l_1 から l_2 が得られるならば，$l_1 > l_2$ とかく．集合 E がこの順序＞に関して（有限）lattice をなすことは，図4に示す通りである．

一般にこのように互に同等の関係にある loop をふくまないgraph すべての集合が，順序＞の F で有限lattice をなすことも証明できる．さらに一般に，1つ以上いくつかのroot をもつgraph（ＬＩＳＰで通常取扱うデータ構造はこれである）に対しても同様に有限lattice をなす．

この性質より，単一の最小元（最縮約元；上の例では ct によるもの）と単一の最大元（上の例では bt によるもの）の存在，ならびにデータ構造の縮約過程の実行順序に無関係に最縮約元に到達しうることが保証される（Ⅳ参照）．

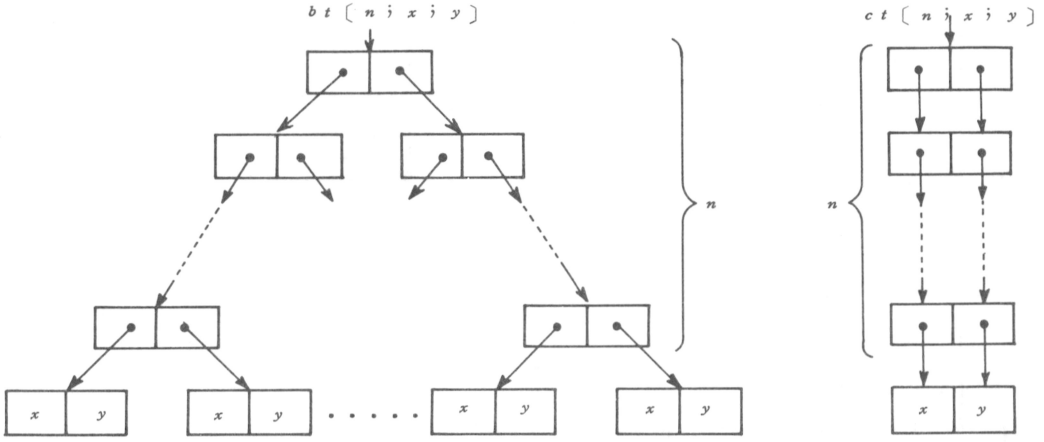


図3　bt と ct で生成されるデータ構造

Quiz.

集合 $\{ l / \mathrm{equal}〔l ; bt〔n ; A ; B〕〕 = T \}$ の要素の個数を n の函数としてあらわせ.

ＬＩＳＰ1.5においてgeneral directed graphが生成されることは，擬函数rplaca，rplacd等を考えれば，明らかである.

Ⅱ　データ構造の識別

ＬＩＳＰでは，rplacaやrplacdやそれに準じる演算を除けば，内部表現としてどういうデータ構造をとっているかは，使用者からみてそれほど明確でない．すなわちＩで述べたいろいろなacyclic directed graphのなかでどれをとっているかは，プログラムの意味に関係がない（部分が多い）．データ構造そのものを識別するには，函数eqやequal を用いてなされる．しかしpure ＬＩＳＰとＬＩＳＰ1.5では，eq の定義が異なるので，ここではpure ＬＩＳＰのeq をeqatom とよぶことにする．eqatom は，引数がatomに対してのみ定義されている．よってtree だけしか識別できない（正確には，Ｉで説明した順序≫で移れるものは，互に全く同じデータと見做してしまうということである）．たとえば d_1 と d_2 あるいは図1の17個のデータ構造は，pure ＬＩＳＰで区別することができない.

さて基本函数eq は，ポインタが同じものであるかを調べる函数であるから，acyclic directed graphはもちろんgeneral directed graphを識別することが可能である.

以上よりデータ構造の観点より，生成と識別の能力，取扱われるgraph の種類についてそれぞれ対照すると各ＬＩＳＰにおいてあまり統一のとれていないことが理解されるであろう.

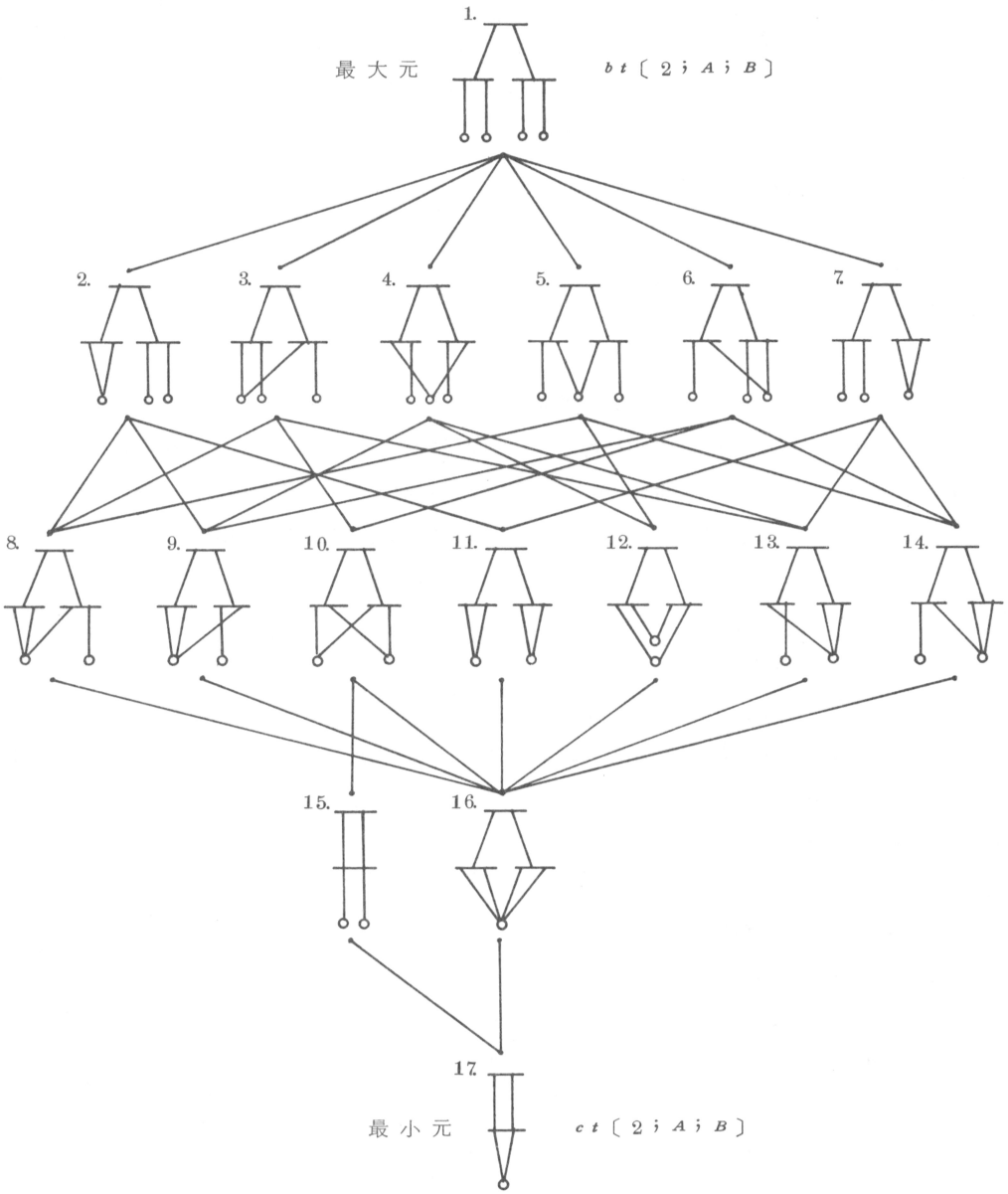
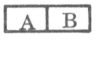


図4. Eの lattice diagram*

---

*  を  ,  を 0 と略記する．順序＞は，図で上方から下方への線で示される．

Ⅲ　データ構造の入出力

前章まで述べてきたようにＬＩＳＰがtree，acyclic directed graph，general directed graphを生成するのに対し，入出力擬函数readとprint は，tree（Ｓ－表現）だけを扱い，内部表現を入出力するには，非常に弱い．たとえば，acyclic directed graphをtreeとして出力するので，前章のgraph d_1 ,d_2 に対して，

$eq[car[d_1]; cdr[d_1]] = F$,

$eq[car[d_2]; cdr[d_2]] = T$

となるにもかかわらず，

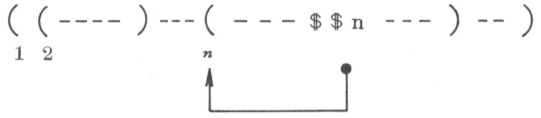
$print[d_1] = print[d_2]$

$= ((A \cdot B) \cdot (A \cdot B))$

となる．また図４の１７個のgraph に対してすべて同じ出力がえられる．general directed graphは，出力しようとすると，loop のあるところ（circular list ）でぐるぐる廻って，出力がとまらない．

このような例が示すとおり入出力でも問題点がみられる．また実際問題としてdebug にも不便であろう．

そこで我々は，ポインタも出力できるような擬函数print２を次のように作った．左括弧に左から順に番号を付しこの番号でポインタを表現することにする（この番号も印刷される）．



ここで$$は，ポインタを示す特別の記号で，数を表わすatomと区別するためのものである．この方法によれば，d_1 , d_2 は，

d_1 ：((A ・ B) ・ (A ・ B))

1 2　2 3　3 1

d_2 ：((A ・ B) ・ $$ 2)

1 2　2　1

と出力され，その区別がつく，図５は，図４のいくつかのデータ構造に対する出力を示すものである．

1. (((A ・ B) ・ (A ・ B)) ・ ((A ・ B) ・ (A ・ B)))
   1 2 3　3 4　4 2　5 6　6 7　7 5 1

4. (((A ・ B) ・ (A ・ B)) ・ ((A ・ B) ・ $$ 3))
   1 2 3　3 4　4 2　5 6　6　5 1

10. (((A ・ B) ・ (A ・ B)) ・ ($$ 3 ・ $$ 4))
    1 2 3　3 4　4 2　5　5 1

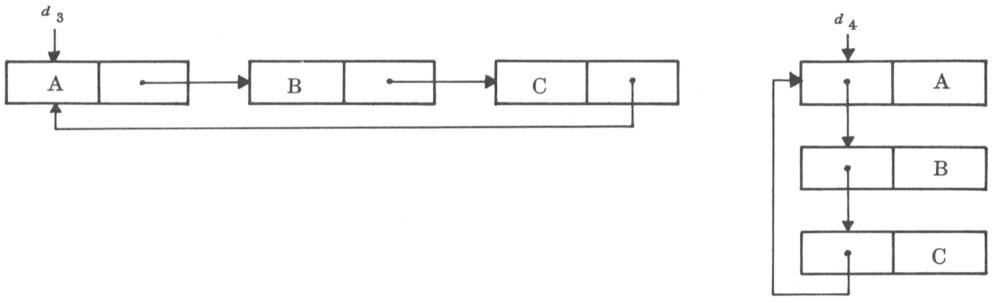
16. (((A ・ B) ・ $$ 3) ・ ($$ 3 ・ $$ 3))
    1 2 3　3　2 4　4 1

17. (((A ・ B) ・ $$ 3) ・ $$ 2)
    1 2 3　3　2　1

図５．図４のデータ構造の出力

（番号は，図４の対応する番号）

general directed graphもそのデータ構造を正しく出力できる．たとえば，



d_3 ：（A・（B・（C・$$1））））
1　2　3　3 2 1

d_4 ：（（（$$1・A）・B）・C）
1 2 3　3　2　1

という具合である．

このような出力函数は，ＬＩＳＰ1.5の範囲内でも作れるが，ＳＵＢＲとしてこのような函数を付加えた方が便利であろう．同様な考え方により入力擬函数read 2もポインタの入力を可能にすれば，print2とあわせてＬＩＳＰを一般のネットワーク構造を有する問題へも応用しやすくなるものと思われる．

Ⅳ　データ構造の縮約

ＬＩＳＰ1.5では，atomに対する内部表現としてのデータ構造は原則として1つしか作らないが，ポインタに関しては，そういう条件を考えないので，何度も同じデータを作りうる．そこで既にⅠで予告したように，“同じデータ構造のものは，1か所でしか作らない”ことを考えよう．このこと（またはそのための演算）を縮約という．縮約を考えるデータ構造の範囲としては，差当りacyclic directed graphに限定しておこう．

縮約により直ちに期待されるのは，記憶領域の節約である．たとえばⅠにおけるd_1とd_2では，d_1をd_2で表現すると，d_1では，3セル使用しているのに対して，d_2では，2セルでよい．一般に$bt〔n;A;B〕$と$ct〔n;A;B〕$では，$(2^{n+1}-1)$セルと$(n+1)$セルの差がでてくることになる．この例からも明らかなように，同じ意味をもつデータ構造は，最縮約形にしておけば，記憶領域を大幅に節約できる（場合がある）．

またデータ構造を縮約すれば，equal で同じポインタに一致することが多くなるから，equal 等の演算速度は著しく向上する．最縮約形のデータ構造内においては，函数equal と函数eq が全く一致することになる．

しかし演算速度の点に関しては，縮約のための手当ての余分な時間との関係できまるものであるから，全体の計算時間については，問題が残る．（そこで縮約のための手当てを高速に行う一方法を説明する．）

またＬＩＳＰ1.5でrplacaやrplacd等を用いるものに対しては，無条件には，縮約を行えるわけでもない．

さて縮約の扱い方として，次の2つの方法を考えた．

(A)　Packcons　による方法

(B)　reduceによる方法．

(A)　packcons　による方法は，記憶領域中（loop をふくまない）すべてのデータ構造を終止一貫して，最縮約形に保つようにすることを考えるものである．通常のＬＩＳＰでは，cons（に相当する演算）が，free storage の新しいセルを無条件に消費するが，これに対して今度は，cons〔$x;y$〕に相当する演算の実行に対して，（x・y）

という内容をもつセルが，記憶領域中に既に存在する時は，そのセルへのポインタをcons〔x；y〕の値とするのである．このように改変されたconsをpackconsと呼ぶことにする．ここで主眼とする点は，（x・y）と同一内容のセルを記憶領域全体の中で捜す手続きにある．記憶領域中を逐次捜すのは，実行速度の点で著しく不利であるので，この点を改善するのに次のようなhash codingを利用すればよい．＊

Packconsのアルゴリズムを ＬＩＳＰのprogram featuve（setqのかわりに：＝と書く）で記述すると次のようになる．

```
packcons〔x；y；m；n〕::=prog〔〔a〕；
    a:=hash〔x；y；m；n〕；
A   〔freep〔a〕→go〔B〕；
        car〔a〕=x∧cdr〔a〕=y→go〔C〕〕；
      a:=mod 3〔a+1；m；n〕；
      go〔A〕；
B   rplaca〔rplacd〔a；y〕；x〕；
C   return〔a〕〕
freep〔a〕::=〔atom〔car〔a〕〕→eq〔car〔a〕；FREE〕；T→F〕
mod3〔i；m；x〕::=m+remainder〔i；n〕
```

（説明）

ⅰ）carとcdr 対のセルのはいるポインタ空間$Sp〔m，n〕$は，m番地からはじまりn個の一連のアドレスを占有しているとする．未使用の自由（FREE）なセルは，chained listとはせず，セルがFREEであることを示す，他の目的には使用しない特別なatom FREEをcar部に書くことにする．理論函数freepは，アドレスaのセルがFREEか否かを判定する函数である．

ⅱ）Sapをatomとポインタをあわせたアドレス空間とする．x，$y\in Sap$（すなわちx，yはatomまたはポインタ）として，hash〔x；y；m；n〕は，$Sap\times Sap$から$Sp〔m，n〕$への適当なhash函数とする．

ⅲ）packconsの評価は，まずa＝hash〔x；y；m；n〕というアドレスを見てそれがＦＲＥＥならば，（x・y）は新しいのでaに（x・y）を書き，packcons の函数値をaとする．aがＦＲＥＥでないときには，aの内容が（x・y）ならば，（x・y）はすでに現れたものであるから，この場合もpachcons の値をaとする．上記以外の場合は，hashing に conflict があるので，ここでは最も簡単なrehash函数として$a+1$を作って，上記と同様な操作を反復する．ただし$a+1$が$Sp〔m，n〕$から外に出ないようにするために函数mod 3を使う．

このpackconsを正しく働かせるためには，ＬＩＳＰの通常のGarbage Collector(2)，(3)（ＧＣと書く）を次に示すＨＧＣに改造する必要がある．これは，通常のＧＣでセルがＦＲＥＥになるとhashing の検査プロセスがそこで不当に終了してしまう場合が生じるからである．

ＨＧＣ（Hash Garbage Collector）

ＨＧＣは，通常のＧＣと同様に使用中のセルに目印づけを行うとともに，packcons のアルゴリズムに適合するようにデータ構造を再配置する．このためにポインタ空間$Sp〔m；n〕$の容量nの1.5倍の容量のスタックを2次記憶（ドラム／ディスク）にとり，再配置後のアドレスa（a　$Sp〔m；n〕$）に内容（x・y）を書けというデータを順次スタックする．これを実行する擬函数をstack〔a；x；y〕とする．また各セルごとにgビットとfビットと

＊atomに対してhash codeを利用するのは，〔2〕等で指摘されている．

よぶ2種のビット表（計2^nビット）を使用する．gビットは，通常のGCの目印ビットに相当し，fビットは，hashing のconflictを示すために使用する．HGCの開始時にg，fビット表ともresetされているとし，setg〔a〕とsetf〔a〕は，それぞれアドレスaに対応するgとfビットをsetする擬函数，setgp〔a〕とresetfp〔a〕は，それぞれgビットのset，fビットのreset 状態を調べる論理函数とする．

HGCの中でrをroot とするデータ構造（ loop なしかつ最簡約形になっている）に目印をつける擬函数markのアルゴリズムは，次のように書ける．

```
mark〔r〕 :: = prog〔〔a ; x ; y〕;
  〔atom〔r〕 V setgp〔a〕→return〔 〕〕;
    mark〔car〔r〕〕;   mark〔cdr〔r〕〕;
    x : = new〔car〔r〕〕; y : = new〔cdr〔r〕〕;
    a : = newaddress〔x ; y〕; stack〔a ; x ; y〕;
    rplaca〔r ; a〕;
    setg〔r〕;  return〔 〕〕
```

この定義中で下線の3行を除くと普通のGCの目印づけアルゴリズムの一種（ Backpointer法でもよいが，ここでは定義式を短くするためmarkをrecursiveに用いた）となっているのが了解されよう．下線の部分のHGC用諸函数の定義は，次の通りである．

```
new〔x〕 :: =〔atom〔x〕→Z ; T→car〔Z〕〕
newaddress〔x ; y〕 :: = prog〔〔a〕;
  a : = hash〔x ; y ; m ; n〕;
A 〔resetfp〔a〕→go〔B〕〕;
    a : = mod 3〔a + 1 ; m ; n〕; go〔A〕;
B  seft〔a〕; return〔a〕〕
```

newは，atomまたはポインタの再配置後のアドレスを求める函数で，atomならそのまま，ポインタの場合は，そのポインタの指すセルは必ずgビットがsetされ，その新アドレスは，replaca 函数でcar部に書かれているはずだからそれを取りだす．

HGCが，いま述べた函数markによる目印づけをすべての必要なデータについて行うことは，通常のGCと同様である．目印づけが終了したらポインタ空間$Sp〔m, n〕$のcar 部を全部FREEに書換えてから，スタックの内容に従って新しいデータ構造$Sp〔m, n〕$に書くことによって完了する．ポインタ領域の容量と同じ容量の一連の領域を主記憶（コア）にとれば，インコアでHGCを実行できることは，次のreduce函数の説明からも，明らかである．余分のコア領域が全くなしに能率のよいpackconsのHGCが作れるかどうか，loop をふくむデータ構造をどのように取扱ったらよいか，は残された問題である．

（なお上記のpackcons のプログラム中，free storage がなくなった時の手当てについては，簡単のため省いた）

(B) reduceによる方法

reduceによる方法は，引数に与えられたloopのないデータ構造を最縮約形にする擬函数reduceを与えることによる．函数reduceは，使用者が自由に使えるとしても，次の問題が生じる．記憶領域内のいくつかのデータ構造は，一般にその端末部分を共有していることが多いので，1つのroot に関する縮約の結果他のroot のデータ構造に（見かけ上）影響を与えない工夫が必要である．（このあたりの事情は，コンパイラ等の話題でよく聞かれるあろう．）

そこで各セルは，他のデータ構造から指されていることを予想して，そのセルを縮約された先へのポインタを中継に保持しておく方法をとり，以後は，eq のようにポインタをたどる演算でこの中継点を無視するようにするか，あるいは他の生きているtreeを普通のGCのように調べ中継点を除いてしまうかすればよい．

さて以下ではこの手当てを施すものと仮定しておいて，reduceのアルゴリズムを検討しよう．

まず各atomは，目印がつけられたセルとよぶことにする．car，cdr部の両方にいま目印をつけられたばかり（片方は以前につけられていたかも知れない）のセルすべてについて作業用領域を用いて一致を調べて，car，cdr部が各々一致するセルは，1か所に縮約する．そしていま対象となっていた各セルを指しているセルのcar部またはcdr部に目印をつける．この操作を繰返して，与えられたrootのセルの両方に目印がつけられたらreduceは，完了する．

このアルゴリズムを能率よく行うために，次のような工夫をする．まず初期設定として，GCでよく使われるback pointesを用いる方法で，与えられたacyclic directed graphのrootの方向に逆にたどれるようなポインタ（のリスト）を各セルに付加しておく（treeならば，backpointerを各セルそのものを用いて貯えることができる）これで各段階で1つroot方向のセルの探索が無駄なく行える．また各段階で集めるセルの比較は，作業用領域への登録時にいろいろ高速化を工夫すればよい（作業用領域は，高々各段階で集められるセルの個数の大きさで十分であろう）

さてreduceのアルゴリズムに（A）で用いたhashcodingを用いる方法が，次のものである．上述のものは，完全にbottom up式であるのに対して，こちらはfrom left to rightのbottom up式にできる（上述のものでもこの方式でできるが，実行時間の点では，能率のよい方法はみつかっていない）．

与えられたroot rに連なるloopのない任意のデータ構造に対して，このreduceアルゴリズムは，（A）のHGCのmark〔r〕を次のように改変することにより得られる．rをrootとするデータ構造の細胞数よりやや多い個数n'の容量をもつ一連の領域$Sp〔m', n'〕$をポインタ空間$Sp〔m, n〕$とつなげてとり（$m'=m+n$），$Sp〔m', n〕$のcar部には全部FREEをいれる．またHGCのmark〔r〕の定義中の～～～～線部（newaddressとstack）をpackcons〔x, y, m', n'〕とする．次いで$Sp〔m, n〕$を適当に走査して，gビットのついているセルを指しているポインタを$Sp〔m', n'〕$内に指し換え（ここでもし中継のセルを使う方式だとこの部分は不要であり，そのセルに中継であるとの印を適当につけておけばよい），最後に$Sp〔m', n'〕$を$Sp〔m, n〕$に加える．

なお使用者にとってみれば，LISP 1.5中で函数reduceを自由に使えるためには，函数copyや分解（縮約の逆）演算が必要となろう。またここで考えたreduceは，生きているデータ構造すべてに対するものでないので，reduceの前後で実行時間が無関係な演算も存在する．そこで，reduceが余計なオーバーヘッドになってしまう可能性があることに注意しなければならない．

おわりに

LISPの処理系の作成中に考察したいろいろな問題点やその解決法等について述べてきた．LISPのデータ構造における生成，識別，入出力について議論し，記憶領域の節約とある種の演算の高速化を狙った縮約について詳しく説明した．これは，数式処理等で共通の因子を探したり，データ構造の等価性の判定を頻繁に行う場合などは，特に有効ではないかと思われる．

ここで議論した諸点について，一般のデータ構造（general directed graph）に対してどの程度のことがいえるのかは，多くの点で問題として残されている．

またLISPは，様々な方言が使われていて，standard化する提案もあるほどになっているが，ここで述べたいく

つかの考えは，使い方によっては，大いに有効性が期待される．差当っては，普通のＬＩＳＰと組合せるのが，現実的な活用法であろう．

引　用　文　献

〔1〕 J.McCarthy et al.,"ＬＩＳＰ1.5 Programmer's Manual."ＭＩＴ Press（１９６２）

〔2〕 D. G. Bobrow, "Storage Management in ＬＩＳＰ." in Symbol Manipulation Languages and Techniques, North-Holland（１９６８）, pp.291-301.

〔3〕 D.E.Knuth, The Art of Computer Programming. Vol.1, Addison-Wesley（１９６８）.

本 PDF ファイルは 1965 年発行の「第 6 回プログラミング―シンポジウム報告集」をスキャンし、項目ごとに整理して、情報処理学会電子図書館「情報学広場」に掲載するものです。

この出版物は情報処理学会への著作権譲渡がなされていませんが、情報処理学会公式 Web サイトの https://www.ipsj.or.jp/topics/Past_reports.html に下記「過去のプログラミング・シンポジウム報告集の利用許諾について」を掲載して、権利者の捜索をおこないました。そのうえで同意をいただいたもの、お申し出のなかったものを掲載しています。

過去のプログラミング・シンポジウム報告集の利用許諾について

情報処理学会発行の出版物著作権は平成 12 年から情報処理学会著作権規程に従い、学会に帰属することになっています。

プログラミング・シンポジウムの報告集は、情報処理学会と設立の事情が異なるため、この改訂がシンポジウム内部で徹底しておらず、情報処理学会の他の出版物が情報学広場 (=情報処理学会電子図書館) で公開されているにも拘らず、古い報告集には公開されていないものが少からずありました。

プログラミング・シンポジウムは昭和 59 年に情報処理学会の一部門になりましたが、それ以前の報告集も含め、この度学会の他の出版物と同様の扱いにしたいと考えます。過去のすべての報告集の論文について、著作権者（論文を執筆された故人の相続人）を探し出して利用許諾に関する同意を頂くことは困難ですので、一定期間の権利者捜索の努力をしたうえで、著作権者が見つからない場合も論文を情報学広場に掲載させていただきたいと思います。その後、著作権者が発見され、情報学広場への掲載の継続に同意が得られなかった場合には、当該論文については、掲載を停止致します。

この措置にご意見のある方は、プログラミング・シンポジウムの辻尚史運営委員長 (tsuji@math.s.chiba-u.ac.jp) までお申し出ください。

加えて、著作権者について情報をお持ちの方は事務局まで情報をお寄せくださいますようお願い申し上げます。

期間：2020 年 12 月 18 日 〜 2021 年 3 月 19 日
掲載日：2020 年 12 月 18 日

プログラミング・シンポジウム委員会

情報処理学会著作権規程
https://www.ipsj.or.jp/copyright/ronbun/copyright.html