解　説

LISPシステムにおける記憶管理†

山口喜教†† 元吉文男†††

1. まえがき

人工知能研究や記号処理の分野で広く用いられているLISP言語[1]は，リストというデータ構造とラムダ計算に基づく関数評価方式を結びつけることにより柔軟な制御およびデータ表現を可能としている．LISP言語の機能的な特徴は，リストデータの動的割付けや関数の再帰的定義，そしてパターンマッチングなどに使用される連想的処理であり，これらの機能を実現するシステムでのメモリアクセスの様子は通常の科学技術計算におけるものとは違った側面を持つ．例えばリスト構造を計算機上で表現するために通常はメモリのアドレスをポインタとして互いの接続関係を表わすが，このようなデータ表現によってデータの取り扱いが柔軟になる反面，メモリアクセスが分散化する可能性があるため，メモリ管理の面での対応処置が必要となる．特にこれはメモリアクセスの局所性を有効に利用する記憶階層をもつシステムで問題となる．

近年，LISP言語の特性やその応用システムにおける動的振舞いが明らかになり[2)~4),18)]，LISP言語を効率よくインプリメントするための技術やそのための専用マシンシステムの研究が進んでいる．これには記憶階層の持つ利点がLISPシステムでも生かされるようにすることの他に，LISP特有の機能に対するメモリアクセスの専用管理機構を設けることなどが含まれている．本稿ではLISP言語インプリメントにおける問題点や種々の研究成果をメモリ管理という側面からみることにする．

2. リスト処理とそのメモリ表現

2.1 リストの表現形式とその特性

リスト構造とはある情報の要素を並べる際に，その要素の他に次の要素へのリンクも情報として蓄えておくものである．これによって一次元の配列やスタックなどのように単に情報を線形に並べておくだけの構造に比べ要素の挿入や削除などを柔軟に処理することができる．LISP言語では，このリスト構造を図-1に示すようにcar部，cdr部という2つのポインタからなるセルを単位として表わす．セル内のポインタは通常メモリのアドレスを表示している．ここでLISPにおけるポインタの距離をポインタによって指されるセルのアドレスとポインタを含むセルのアドレスの差と定義する．リストの柔軟構造からみて，リストのセルはメモリ上の任意の場所に分散することが可能であるが，実際のLISPシステムでは図-2に示すようにリ

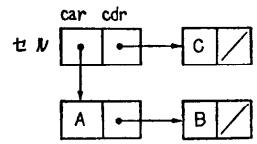

図-1 リスト［(A, B)C］の表現
(注) A, B, Cはリスト表現の基本単位でありアトムと呼ばれる

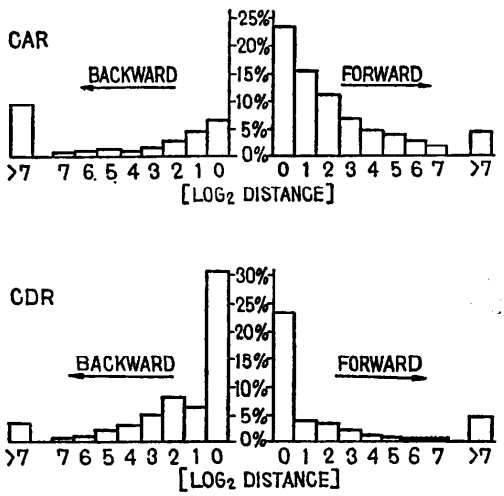

図-2 リストポインタ間距離のヒストグラム
((文献2)による)

† Memory Management of LISP Language Systems by Yoshinori YAMAGUCHI (Computer Division, Electrotechnical Laboratory) and Fumio MOTOYOSHI (Information Science Division, Electrotechnical Laboratory).

†† 電子技術総合研究所電子計算機部
††† 電子技術総合研究所パターン情報部

ストポインタ間の距離の分布はそれ程ランダムにならないことが測定されている[2]．このような特性を利用して次に述べるようにリストを線形化したりポインタをコード化することによりメモリ空間の有効利用やワーキングセットの軽減などを実現しようとしている．

2.2 リストの線形化とリスト圧縮

記憶階層を持ったシステムにおいて，リスト処理が効率よく実行されるためにはポインタによって互いに接続されたセル同志が物理的にも近くのアドレスに存在することが必要である．このために線形化 (linearization) という操作により，リスト構造における連続した cdr (または car) がメモリ番地においても連続的になるように構造を再配置することが行われる．これには**図-3** に示す如く car 方向と cdr 方向の線形化があり，実際のプログラムでの効果が測定されている[2]．これを**表-1** に示す．この表からもわかるように cdr 方向への線形化を行った場合 98% 以上の cdr ポインタは次のセル番地を指すことがわかる．これはリストの cdr 部のメモリ表現をコード化することによりセル表現の有効ビット長を軽減し得る根拠となっている．

リスト圧縮[13]はリスト構造データの統計的性格によりその最適表現が決まるものであり，この時圧縮した表現で処理不可能となった場合 (escape と呼ぶ) の回復機能の処理コストを考慮しなければならない．リスト圧縮の方式としては種々のものが考えられるが[14]，ここでは実際のシステムとして提案されているものを2つ紹介する．1つは XEROX の LISP マシンでの方式であり仮想記憶のページングを全面的に利用するために，32 ビットのセル表現において car 部に 24 ビットのアドレス値を持たせ，cdr 部の 8 ビットを**表-2** に示すようにコード化する．これにより大部分のデータはページ内相対アドレスで表現されることになり，実効的なセルのサイズは 35.9 ビットと計算される[14]．一方MIT の LISP マシン[15],[16]では特にページングを意識せずに，cdr を2ビットでコード化しており，次の4つの場合がある．1) cdr=NIL, 2) cdr は次のセルを指す．3) cdr ポインタは次のセルに置かれる．4) invisible ポインタ．この方式では cdr が直後のセルを指す場合にリスト圧縮が有効に働くためリストの線形化操作が特に有用となる．

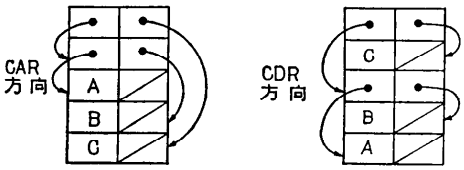

図-3 リスト線形化の例：(((A)B)C)

表-1 リスト線形化後にポインタがすぐ後ろのセルを指す割合〔%〕((文献 2) による)

プログラム名		NOAH	PIVOT	SPAR-SER	STR-GEN	WIRE
car 方向の線形化後	car	79.5	95.7	78.5	93.6	90.2
	cdr	77.4	75.5	79.3	71.2	77.1
cdr 方向の線形化後	car	27.3	41.7	33.3	24.3	19.1
	cdr	98.8	99.1	98.7	98.3	99.3

表-2 XEROX LISP マシンにおけるリスト圧縮表現[16]

cdr の内容	cdr の意味	car の意味	セルのサイズ
ϕ	NIL	標準的なポインタ	32 ビット
1-127	ページ内相対	標準的なポインタ	32 ビット
128	invisible セル	invisible ポインタ	96 ビット
129-255	間接ページ相対*	標準的なポインタ	64 ビット

* (指されたセルの car 部に実際の cdr 値がある)

invisible ポインタは一種の間接アドレッシングであり，このセルに対するアクセスはセル内に置かれたポインタへのアクセスとして自動的に変換される．このため rplacd などにより cdr ポインタを書き換えた際にこれを invisible ポインタにしておけば圧縮リストの構造をその都度書き直す必要がなくなり，オーバヘッドを減少し得ることとなる．

2.3 リストの動的割付けとメモリ管理

LISP では新たなリストセルの生成は cons 関数による．cons[x : y] を実行すると，未使用のリストセルがフェッチされ，その car 部が x を cdr 部が y を指すようになる．これらの操作は動的記憶割付けと呼ばれており，システム内部のメモリ管理ルーチンで処理されユーザやプログラマからは隠されている．このようなメモリ管理方式においては，

(a) リストを構成する要素としてのセルの集合体から未使用のセルを供給するための機能と，

(b) 不要になったセルを回収する機能

が必要である．

前者に関しては，未使用のセルをリストにしておき (これをフリーリストと呼ぶ)．新しいセルが要求されるとこのリストの先頭を切離して供給することが一般的であるが，Interlisp ではページング環境におけるリストアクセスのヒット率を高めることを目的として，各ページごとにフリーリストを設け cons 操作を次のアルゴリズムに基づいて行っている[5]．

cons[x ; y] のおかれるセル Z は，

(1) もしyと同一のページにあきがあるならそのページに，さもなくば，

(2) もしxと同一のページにあきがあるならそのページに，さもなくば，

(3) 最も最近に cons を行ったページにあきがあるならそのページに，さもなくば，

(4) 16 以上のあきのある任意のページにとられる．

Clark は上述のアルゴリズムと通常の cons アルゴリズムとを実際のプログラムで比較測定し，同一のページにリストポインタの存在する割合にほとんど違いのないことを示している[2]．この理由は Clark も指摘しているように，通常の場合でもフリーリストはアドレスの順にリスト化されているので，特別の操作なしでも関連のあるリストが物理的にも近いアドレスに置かれる可能性が高いためである．

後者の機能はガーベジ・コレクション(以下 GC と略す) と呼ばれており種々のアルゴリズムが提案されている[6]．最も一般的なものはマーク・トレース法であり，すべての使用中のセルをトレースする必要があるため，その処理時間が使用セル数に比例し，大規模プログラムでしかも実時間応答を必要とする応用には適さない．このため種々の並列 GC の提案がなされているが[7)~10]，並列プロセス交信の複雑さやオーバヘッド，記憶階層における処理の複雑さなどの問題点がある．インクリメンタルGCは，一定時間ごとに不必要なセルの一部を回収するものであり，実時間処理に適している．これにはリストをコピーする方式[11]とリファレンスカウントを用いる方式とがある．前者の方式ではページング環境でのワーキングセットが大きくなる傾向があり，後者ではリファレンスカウントの計算のオーバヘッドやカウント値が2次記憶にあった場合に，これを変更する手間がかかるなどの問題がある．ハッシュテーブルを利用することによりリファレンスカウントの欠点を取り除いた GC の提案もなされている[12]．

リスト処理における本質的な操作はメモリアクセスに帰着されるため，これを効率よく実行するためにはメモリサイクルの高速化，メモリバス幅の拡大，仮想記憶におけるヒット率の向上などが第一の要件であるが，上に述べてきたような動的記憶割り付けに伴う種種のオーバヘッドを最小限に抑えていく改良も見逃してはならない．

3. スタック記憶

3.1 LISP におけるスタック構造

スタックを有効に利用することにより，メモリ操作の負担を軽減できることは ALGOL や PASCAL などのインプリメンテーションでも明らかになっているが，LISP においてもスタックアクセスを独立した機能を持ったメモリアクセスとし，一種の機能メモリとすることによってパフォーマンスの向上が計られている*．

スタックに対する基本操作はプッシュとポップであり，これにはメモリアクセスの他にポインタの増減処理，さらにオーバ・アンダフローの検出などが含まれている．これらの操作をハードウェアまたはファームウェアで実行することにより，スタック操作に伴うオーバヘッドを最小限に抑えることが可能である．またメモリ上の一次元配列として表わされたスタックは線形化されたリストと考えることもできスタック内に置かれた要素へのアクセスも可能である．LISP においてスタック操作に最も影響のあるものは変数の結合(binding) 方式である[20]．ALGOL や PASCAL では自由変数の結合環境は静的でありテキスト上で固定されているが，LISP では関数の呼ばれた情況により動的に変化する．

変数結合の形態をAリスト[1] と呼ばれる結合リスト上に作成する方式を deep binding と呼んでおり，通常このリストをスタック上に置く．この方式においては変数の値を求める際に，変数名，変数値のテーブルをサーチすることが必要とされる．ただしコンパイルされた関数においては，各関数の持つ局所変数に対してはあらかじめスタック内の相対位置が知れているのでスタックメモリ内にインデックス参照することが可能である．一方 shallow binding と呼ばれる方式は，各変数名ごとに結合値をスタックしていくものであり，通常は最近の値のみを value セルという場所に置き旧結合情報をスタック上に格納しておくことが行われている．変数の値は value セルより直ちに求まるが関数への入出力時にこの値をスタックに退逃したり戻したりする操作を行わなければならない．また関数引数(Functional Argument) のインプリメントも複雑となる．通常のメモリ構成においては shallow の方が deep よりオーバヘッドが少ないと考えられておりこれを示すデータも提出されている[18),19)]．

* 文献 18) によれば，仮想的な LISP 計算機において全命令実行数に対して約 1.5 倍の回数のスタックアクセスがあることが測定されている．

スタックは関数のコール・リターンの制御情報や局所的な記憶場所として使用することもでき，これらは前に述べた結合情報と共に関数の実行環境を形成する．これを通常 Frame[21] と呼んでいるが，この構造は関数の制御構造などに密接に関連している．

3.2 スタックのインプリメンテーションと特性

主メモリ上にスタック内のデータを格納する場合でも，キャッシュを備えたシステムの場合には，スタックアクセスのワーキングセットはかなり小さいと考えられるので良いパフォーマンスを示すであろう．さらに MIT[15),16)] や神戸大[17)] の LISP マシンにみられるように，スタック領域を高速メモリとして独立させることが行われている．このような方式はキャッシュのような複雑な制御を必要としないので，小型の専用マシンに適していると考えられる．

4. 連想記憶

4.1 LISP における連想処理

汎用的な計算機処理では，表を作成したり検索をする場合に，鍵となるものは整数でしかもその範囲が限られている場合が多い．このために，インデックスレジスタなどを利用してランダムアクセスの記憶装置から，表を高速にアクセスすることが可能であった．

ところが，LISP などを用いて記号処理を行う場合には，鍵となるものが記号であったり，それの組み合わされたリストであることのほうがむしろ多く，単純なインデクシングは使えなくなる．また，データが構造をもっているので，二つのデータが同等であるかどうかを調べるときにも時間がかかってしまうことがある．またデータを検索するときに単純に鍵が一致するものを見つけるのではなく，鍵の部分構造があるデータと一致するものがほしくなったり，似たようなデータを鍵にするものが必要になったりする．

この後者のほうの問題は，前者のいわば「記号添字」のアクセスが高速に実行できれば解決する部分も多い[25)]．二つのデータの同等性を調べるには，データを作るときに同じ構造のデータは一つしか作られないようにしておけば，単に二つのデータを現わすポインタが等しいことを調べればよい．LISP の場合を例にとると，記号添字を行うものとして，アドレス関数 $ad(k_1, k_2)$ があるとする．点対 (dotted pair) を作るときに，k_1 と k_2 が共に唯一つしかないデータであれば，アドレス関数 $ad(k_1, k_2)$ によって，k_1, k_2 から唯一にアドレスが決まるのでそこに k_1, k_2 を記憶してそのアドレスを $(k_1 \cdot k_2)$ を現わすデータであるとする．ここで同じアトムは唯一であるとすれば，上の方法を繰り返し用いることによって同じ構造のデータは唯一つだけ作られることになる[22)]．ただしこの場合には RPLACA* や RPLACD* のようなポインタを変更する操作を行って k_1 あるいは k_2 の値を変えることは許されなくなる．

* RPLACA は目的セルの car 部を変更するものであり，RPLACD は cdr 部に変更を加える働きをする．

4.2 ハッシング・ハードウェア

前節の記号添字を実現する最も単純な方法は，鍵を前の方から順につめて表を作り，探索するときは同じ鍵にぶつかるまで順に調べていくやり方であるが，これは表の大きさに比例して時間がかかってしまう．

そこで，コンパイラなどで用いられているハッシングの方法[23)]を使うと，平均探索時間を短縮できることが知られている．これは，データは何らかの形でビットの列であるが，それを限られた範囲の整数に一様にちりばめる関数を利用するものである．異なったデータが同じ整数に一致した場合についての処理の仕方によりさまざまな方法が考えられている．また不用になった鍵を削除するときのことも考えておかねばならない．

この考えを進めて，一様にちりばめる動作をハードウェアで行い，さらに複数の記憶装置で同時に探索すれば，この記号添字は高速に行われ間接指定を行う時間程度で実行できる[24)]．実際に理化学研究所では，このハッシング・ハードウェアを持った計算機の提案や試作も報告されている．

4.3 問題点

連想処理について「記号添字」の方法を利用すれば多くの処理が高速に実行できることを示したが，この方法では，似たような構造をもったデータを検索するということは実現がむずかしいように思われる．

また以上の話は主記憶に対しての連想処理であったが，扱うデータの量が多くなり，二次記憶を利用したデータベースの連想処理の問題については今後の研究を待たなければならない．

5. むすび

LISP システムの特徴的な性質をメモリ管理との関連で概観してきたが，リスト処理が一般に考えられている程非能率でないこと，記憶階層を利用するためのインプリメンテーション技術が開発されていること，

高速メモリによるハードウェアスタックやハッシングハードウェアといった専用ハードウェアが開発されていることなどが明らかになった．LISP は言語というよりもシステムとして発展してきたものであり，今後もより改良が加えられていくであろう．

参 考 文 献

1) McCarthy, J., et al.: Lisp 1.5 Programmer's Manual, MIT Press (1962).
2) Clark, D. W. and Green, C. C.: An Empirical Study of List Structure in Lisp, Commun. ACM, Vol. 20, No. 2, pp. 78-87 (1977).
3) Clark, D. W.: Measurements of Dynamic List Structure Use in Lisp, IEEE Trans. Software Eng., Vol. SE-5 (1979).
4) Bobrow, D. G. and Murphy, D. L.: A Note on the Efficiency of a LISP Computation in a Paged Machine, Commun. ACM, Vol. 11, No. 3, pp. 558-560 (1968).
5) Bobrow, D. G. and Murphy, D. L.: Structure of a LISP System Using Two-Level Storage, Commun. ACM, Vol. 10, No. 3, pp. 155-159 (1967).
6) Knuth, D. E.: The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison Wesley (1968).
7) Steele, G. L. Jr.: Multiprocessing compactifying garbage collection, Commun. ACM, Vol. 18, No. 9, pp. 495-508 (1975).
8) Wadler, P. L.: Analysis of an Algorithm for Real Time Garbage Collection, Commun. ACM, Vol. 19, No. 9, pp. 491-500 (1976).
9) Dijkstra, E. W., et al.: On-the-fly Carbage Collection: An Exercise in Cooperation, Commun. ACM, Vol. 21, No. 11, pp. 966-975 (1978).
10) 日比野靖：並列ガーベジコレクションアルゴリズムと LISP への適用.
11) Baker, H. G. Jr.: List Processing in Real Time on a Serial Computer, Commun. ACM, Vol. 21, No. 4, pp. 280-294 (1978).
12) Deutsch, L. P. and Bobrow, D. G.: An Efficient, Incremental, Automatic Garbage Collector, Commun. ACM, Vol. 19, No. 9, pp. 522-526 (1976).
13) Hansen, W. J.: Compact List Representation: Definition, Garbage Collection, and System Implementation, Commun. ACM, Vol. 12, No. 9, pp. 499-507 (1969).
14) Bobrow, D. G. and Clark, D. W.: Compact Encodings at LIST Structure, ACM Tran. on Programming Languages and Systems, Vol. 1, No. 2, pp. 266-286 (1979).
15) Greenblatt, R., et al.: The LISP Machine, IEEE on Computer (to appear).
16) Steel, G. L. Jr., et al.: CADR, IEEE on Computer (to appear).
17) 瀧，金田，前川：LISP マシンの試作，情報処理学会論文誌，Vol. 20, No. 6, pp. 486-493 (1979).
18) 山口，島田：仮想計算機による LISP プログラムの動的特性，電子通信学会論文誌 D, Vol. 61-D, No. 8, pp. 517-524 (1978).
19) 金田，小林，前川，瀧：試作 LISP マシンの高速化について，電子通信学会技術研究報告，EC 79-58 (1980).
20) Allen, J.: Anatomy of LISP, McGRAW-HILL (1978).
21) Bobrow, D. and Wegbreit, B.: A Model and Stack Implementation of Multiple Environments, CACM, Vol. 16, No. 10, pp. 591-603 (1973).
22) Goto, E.: Monocopy and associative algorithms in an extended LISP, Information Science Lab. Tech. Rep. 74-03 (Apr. 1974).
23) Ida, T. and Goto, E.: Analysis of Parallel Hashing Algorithms with Key Deletion, J. of Infomation Processing, Vol. 1, No. 1, pp. 25-32 (Apr. 1978).
24) Ida, T. and Goto, E.: Performance of a parallel hash hardware with key deletion, Proc. of IFIP Congress 77 (1977).
25) Sassa, M. and Goto, E.: A hashing method for fast set operations, Information Processing Letters, Vol. 5, No. 2, pp. 31-34 (1976).

(昭和54年12月21日受付)