解　説

リスト処理とアーキテクチャ†

黒川利明††*　井田哲雄†††

はじめに

本稿の目的は二つある．一つは以下の各論の導入部として，リスト処理やそのためのソフトウェアとハードウェアの技法を述べることであり，今一つは「リスト処理アーキテクチャ」の概観を与えることである．

1. 基本事項

● リスト構造，リスト処理

一般的なデータとしてのリストは，「要素の重複を許した順序列」と考えることができる．「ブラック・リスト」も「ワイン・リスト」もこの意味での「リスト」の一種である．

しかし，計算機用語としてのリストは，上の一般的なリストの要素を「ポインタ」によりつなげたものを指す．すなわち，リストの開始，終了を「(」と「)」で表わした場合，THIS, IS, (A LIST) という要素をもつリストは**図-1** のように表現される．

ここで文字列表記は入出力に使われ，二分木表記はリストを概念的に木構造として扱う時に使われ，そして箱表記は（各箱にポインタをつけて）計算機内でのデータのあり様を示すのに用いられる．

ここで注意しなければいけないのは，①リストの要素として再帰的にリストが含まれており，②ポインタはリスト構成要素を結びつける何らかの仕組であり，必ずしもアドレスにこだわる必要*はなく，③ポインタの対手先は部分的なリスト構造か，リスト以外のデータ(一般的にアトムと呼び慣わす)かであり，④木の分岐は「2」に限らず一般には「n」で良いことである．

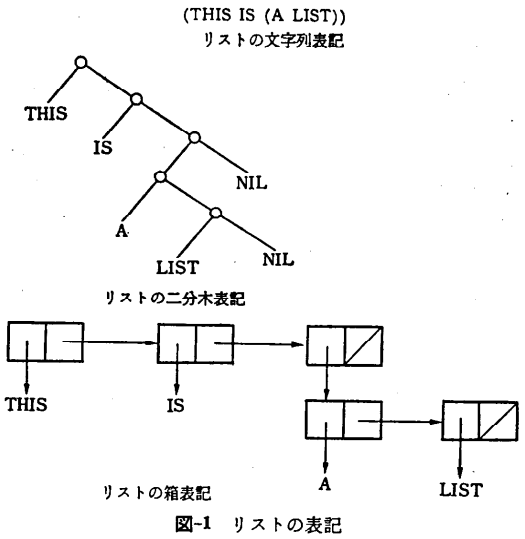


図-1　リストの表記

リスト処理には，リストの要素取り出し，生成，消去，挿入，削除，分割，合併，複写，走査などがある．

リスト処理として特に問題となるのは，**ポインタの取り扱い**と**リストの生成消去**である．

リストの生成消去はリスト要素をどのように生成し，不要な要素をどのように削除回収するかという問題である．リストは概念的にはひとまとまりであっても，計算機の中では各要素がバラバラに存在しているので，その生成消去は記憶空間の管理という問題を含む．

● 各種のリスト構造

リストを構成する各要素にポインタをどうつけるかにより，いくつかのリスト構造が存在する．簡単に紹介しよう．

① 終端データ・単一連絡リスト (singly-linked list with data at terminal nodes)——図-1 のもの．

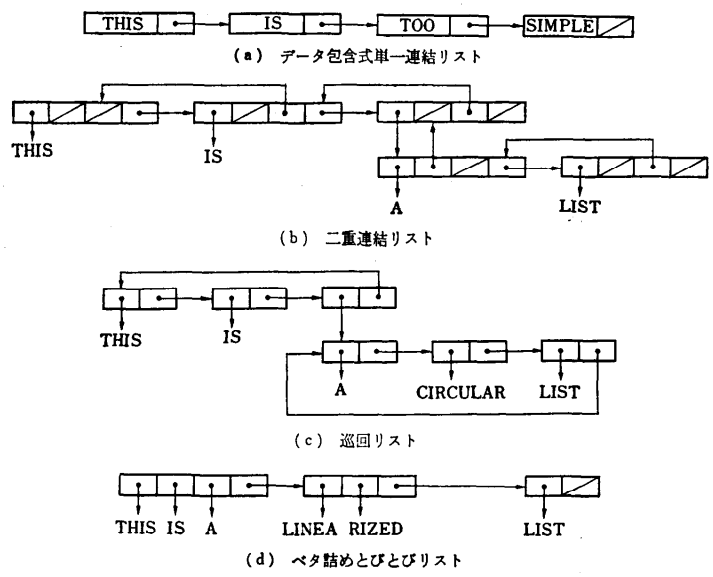
② データ包含・単一連結リスト (singly-linked list with data in it)——**図-2**(a)．
リスト連結用の箱にデータ要素を収める．

---

† List processing and architecture by Toshiaki KUROKAWA (Toshiba Reseach and Develpment Center) and Tetsuo IDA (Iustitute of Physical and Chemical Research).

†† 東京芝浦電気(株)総合研究所
††† 理化学研究所情報科学研究室
* 現在(財)新世代コンピュータ技術開発機構
Institute for New Generation Computer Technology.

* 従来はポインタすなわちアドレスと考えることが多かったが，リスト構造の内に埋めこまれるポインタと記憶空間上の位置を示すアドレスとは，概念的には一応別個のものと考えるが妥当であろう．



(a) データ包含式単一連結リスト

(b) 二重連結リスト

(c) 巡回リスト

(d) ベタ詰めとびとびリスト

図-2

③ 二重連結リスト (doubly linked list)——**図-2** (b).
ポインタが後だけでなく前へもついていて，リスト走査が前後どちらの方向にでも簡単にできる.

④ 巡回リスト (circular list)——**図-2** (c).
ポインタをたぐると元の位置に戻るもの.

⑤ ベタ詰めとびとびリスト (linearized list)——**図-2** (d).
リスト要素をアドレス順に並べておき，都合によってはポインタで連結するもの.

さらに一つの箱の中に n 個のポインタをもつものがある．上の基本的な構造を組み合わせて，たとえば**図-3** のような B-* ツリー[25] のような階層リストを実現することも可能である．しかし，n 分木は二分木で表現可能なので一般的には図-1 のような二分木で連結要素はポインタだけを含む単純なものが用いられる.

ここで述べるリスト構造に対比する意味で「配列」を取り上げるとその相違は**表-1** のようにまとめることができる．リスト構造はアドレスの連続性を犠牲にして，その代わりに可変な構造を実現しているといえる.

リスト構造は，通常は主記憶空間上に表現される構造を指す．高速ランダムアクセス性の前提の成立しない二次記憶でのポインタ操作には，本稿の視点とは異なった議論が必要であり，本稿では取り上げない.

● リスト構造の用途

リスト構造は主として記号処理に用いられているがその理由はリスト構造が実現する柔軟，可変な構造にある．狭義の記号処理（数式処理，人工知能など）に限らず，データベース，文章画像処理（編集操作），

表-1 リストと配列の比較

	リスト	配列
生成	部分的，動的 番地の連続性は保証されない.	全体的，静的 番地は連続的に取られる.
参照	間接アドレス，ポインタ属性をチェック.	インデックス修飾可能，インデックス範囲のチェック.
再利用	ガーベジ・コレクションにより自動的に.	ユーザが自分で行う.

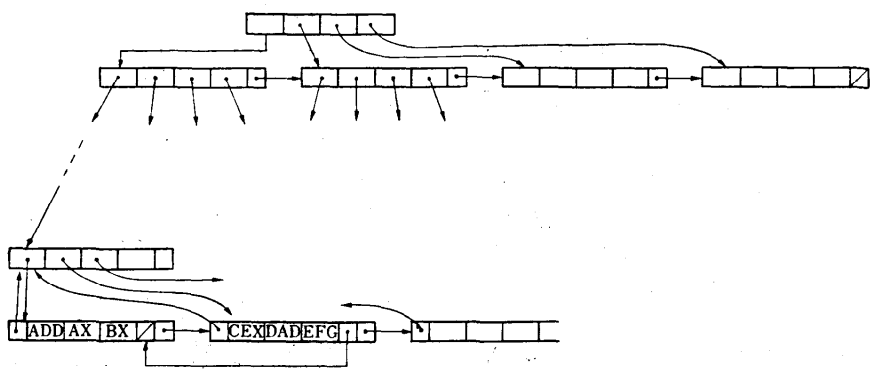


図-3 B*-ツリー

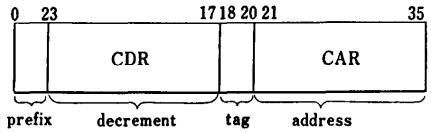


図-4 IBM 704 の場合

シミュレーション，ファイル・システム，オペレーティング・システムなどの非数値計算において幅広く用いられている．

● リスト処理用言語

リスト処理用言語としては Newell, Simon, Shaw による IPL (Information Processing Language の略) が最初であろう．Fortran ができる前のことである．続いて FLPL (Fortran List Processing Language) がある．それから LISP (List Processor) となるのだがこの間の事情については 文献 14) に詳しい．

LISP 以降，リスト処理は基本的な処理となった．それは PL/I, ALGOL 68, PASCAL, C といった汎用プログラミング言語についてリスト操作 (あるいはポインタ操作) が基本的に含まれていることからも明らかであろう．そして，人工知能や記号処理研究に使用されてきた著名なプログラミング言語であるスタンフォード大学の SAIL，エジンバラ大学の POP-2，ベル研究所の SNOBOL-4，ゼロックスのパロアルト研究所の Smalltalk，フランスで開発された PROLOG などでは，リスト処理は言語の提供する諸機能を実現する上で不可欠なものとなっている．LISP の上に構築された microPLANNER[6], CONNIVER[7] QA-4[8], KRL[5], REDUCE[16] などはいうまでもあるまい．

もちろんリスト処理はアセンブラ，FORTRAN, BASIC などでも実現できないことはない．リスト処理用のプログラミング言語の重要性はユーザが，計算機のアドレスのレベルにまで抽象化のレベルを落すことなくアルゴリズムを考えることのできる点にある．

2. 汎用計算機アーキテクチャとリスト処理

かつて (高々十数年前のことではあるが) 電子計算機は高価な品物であった．リスト処理だけでなくソフトウェア一般が既存のハードウェアに合せて作られるしかなかった．良く知られた例として FORTRAN の言語仕様が当時の IBM 704 のアーキテクチャを考慮した事実がある．

リスト処理においてもたとえば LISP の CAR, CDR という名前は IBM 704 の Content of Address Register, Content of Decrement Register からきている．IBM 704 では図-4 のように CDR が一語の前半，CAR が後半なので図-1 とは前後逆転していた経緯があり，後の LISP システムでも CAR, CDR を逆転していた例がある．

しかしながらリスト処理そのものは数値計算に比べると汎用計算機アーキテクチャとの折り合いがあまり良くなかった．それをどう解決しようとしたかをみてみよう．

まずリスト処理で普通必要とされる操作を並べあげてみよう．(以下 LISP の用語を用いる．)

① CAR, CDR——リストの要素 (ポインタ) を取り出す．

② CONS——リストの要素を一つ作り出す．

リスト処理言語 (LISP など) とリスト処理も可能な汎用言語 (PL/I, PASCAL, C など) との主たる相違は CONS が，リスト要素の箱を割りつける操作 (allocation) と，要素を代入する操作とを一体化しているところにある*．

③ GC (Garbage Collection)——CONS 操作により作られたリスト要素が不安になったとき，新しく利用可能なリスト要素を作成 (回収) する操作．このガーベジ・コレクションの詳しい話は，本特集中の日比野の記事を参照願いたい．

④ EQ——データの同一性判定．データの構成上の同値性 (equality check) とアドレスの同一性判定 (identity check) とがある．基本操作としては後者が取られる．

以上がごく基本的な操作であるが，LISP のようなリスト処理言語の場合には次のような操作が追加される．

⑤ TYPEP——データ型判定．リスト，アトム，数，ストリング等のデータの種別を判定する．

たとえば LISTP (リストか)，ATOM (アトムか)，NUMBERP (数か)，STRINGP (ストリングか) など．

⑥ NULL——空リスト (=NIL) かどうかの判定．LISP の場合にはリストの終わりとしての意味や，条件判定時の不成立 (false) の意味も持つ．

⑦ ポインタ変更——RPLACA (replace CAR),

* lazy evaluation や lenient cons の考え方に見られるように，割りつけ操作，要素の計算，計算結果の割りつけられた場所への格納を，各々分離した操作とみなして，リスト処理の効率を向上させる提案もある[27),28)]．リスト処理の並列化を今後考えていく上で，この点は重要になろう．

RPLACD (replace CDR) などポインタの行先を変えてリスト構造を変更する.

ポインタの変更には二つの目的がある. データ構造の共有 (sharing) とデータ領域の節約である. しかし, この操作は副作用を伴うので注意が必要である.

⑧ リスト以外のデータ型の作成, 削除——リスト以外のデータ型については自動的な領域管理をする場合としない場合とがある.

たとえばストリングなどは自動的に管理するが, 文字アトム (識別子) などは, ユーザが積極的に領域を解放する必要のある場合が多い.

これらの基本操作の上に LISP などでは, 変数の値束縛や関数呼び出しが作られる.

これらの基本操作は汎用計算機上では次のように実現されている

(i) CAR, CDR, CONS——これらはメモリの内容による間接修飾 (indirect access) あるいはレジスタ修飾によるメモリアクセスによる.

CONS の場合には供給可能なリスト要素がなくなると GC を呼び出す必要があるので, 閉じたサブルーチンで処理をする場合が多い*.

(ii) GC——かなり複雑な処理なので閉じたサブルーチンになっている.

(iii) EQ と NULL——EQ は通常のアドレス比較演算を用いる. NULL は (EQ x NIL) として実現する場合と, NIL の実現値を 0 にして, 0 比較を行う場合などがある.

(iv) データ型判定——データ型をどう表現するかによって実現方式が異なる[2]. いずれにせよ, 型判定はアドレス比較, ビットのマスク, ポインタ操作等を必要とするため, 2〜数ステップを要する. 数値演算の場合のように命令上のサポートがあるわけではない.

(v) 数値演算——普通の数値演算命令を用いる. ただし, ポインタと同形の整数値 (short integer) や大きな数 (big integer あるいは big float) を組み込んだ場合には数値演算の結果によって数値データを生成する領域管理が必要で, 四則演算がサブルーチン呼び出しとなる.

(vi) リスト以外のデータの生成削除——サブルーチン. ただし short integer や副作用によるストリング変形操作 (ストリングの中味を直接書き換える) などでは, 生成削除が不要となる.

(vii) 変数, 関数処理——スタックを用いるので, スタック上の操作が必要となる. ハードウェア・スタックがないアーキテクチャでは, レジスタによるインデックス修飾が必要.

3. リスト処理アーキテクチャ

リスト処理の高能率化には, 処理速度の向上と, 所要記憶容量の減少の二つの側面がある. 前者は, ポインタを高速に実記憶空間のアドレスに変換し, かつ, 目的とする箱を短かいアクセス時間でとり出してくる問題である. 後者は, ポインタをできるだけコード化して, 所要記憶容量を減少させる問題である. 両者は, トレードオフの関係にあるため, リスト処理専用アーキテクチャでは, 両者をいかに調和させて, 実現するかが, 一つのポイントになる.

リスト処理の実現手法を, 命令の実現法と, 記憶の構成法とに分けて, 考察してみよう.

• 命令の実現

専用マシンでは, ポインタをアドレスに変換して, 実メモリにアクセスをかける操作と, ポインタの指すデータ型の判定を同時に実行することが可能となる. さらに, 特殊なデータ型の場合には, 割り込みを起すようにして, リスト処理用命令を単純な形式で実現することができる.

2 章で述べた CAR, CDR, の操作は, 上のように考えた時, 通常の計算機命令のロード命令に対応させることができる. ストア命令に対応するものとして, LISP では RPLACA, RPLACD という基本操作がある. メモリへの書きこみは, データ型のチェックに先立って行われる必要があるため, CAR, CDR よりは一般には, 余分なマシンサイクルを必要とする.

• 記憶の構成法

第 1 は, リスト構造の実現法である. 1 章で述べたベタ詰めとびとび法は, 専用マシンでは, さらに, 効率良く表現する方法が考えられる. つまり, リストセル中に, リストの連結状態を示すタグをもたせて, 図-2(d) のようなコード化されたポインタを含むような圧縮リストセルを考える. このリストセルの読み出しは, 専用マシンでは, オーバヘッドなく実現されるはずである.

圧縮リスト表現は, 別名 CDR-coding[5] とも呼ばれる. タグは, ①リストの CDR が隣りにある (00), ②CDR が, 一回間接アドレスによって得られる (01),

* 特殊な割り込みを使って GC 起動をすればマクロ展開も不可能ではない. 1)参照.

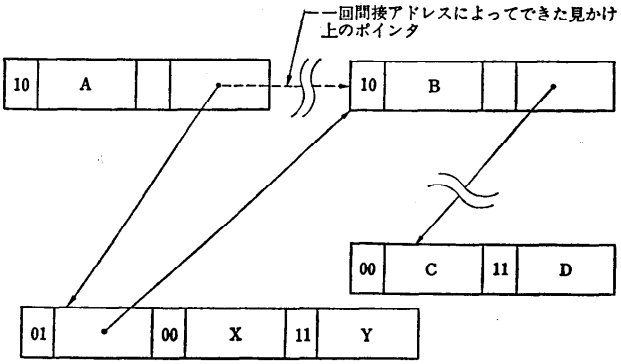


図-5 (A B C D) というリストを CDR-coding で表わした例 P≡(A B X Y) に (RPLACD (CDR P) (C D)) をした結果

③従来どおりのポインタ (10), ④CDR が NIL で, リストが終りである(11), といった状態を示す. (図-5 参照) この方式は CDR だけでなく CAR についても簡単に拡張できる[24]. 圧縮リストでは, 記憶の節約といった面のほか, 仮想記憶空間において, リスト・データの局所性を高めるといった効能がある.

第二はデータ領域管理, すなわちガーベジ・コレクションである. 大きなポイントとして実時間 GC と並列 GC とがある. アーキテクチャの面から今後の検討が期待されるのは, 並列 GC によるデータ領域管理である. これも, レファレンス・カウントやコピーなど, リスト構造のメモリ中にデータ領域管理用の情報を含ませて効率的に管理することができる.

第三は仮想記憶の問題である. 仮想記憶を用いたリスト処理自身は目新しい問題ではないが[19], 大容量の仮想記憶 (たとえば1ギガバイト) においては, 従来と異なる配慮をしないと, 処理速度の点でリスト処理が使い物にならない危険がある.

仮想空間上のメモリアクセスのふるまいをも含めて, リストセルの構造の検討が必要となってくる. また, 仮想空間と, 実メモリとの対応を従来のワーキングセットモデルを用いて, 考えればよいのか, 新しい視点を必要とするのか, 今後の検討課題である.

4. 将来の見通しについて

• 現状から

「リストというデータ構造」のもつ有用性は今日広く認められていると考えて良いであろう. LISP というプログラミング言語を中心にして, リスト処理をどう実現するかという経験はすでにかなりの量が積み重ねられている.

この結果, 大型計算機からマイクロコンピュータに至るまでの各種計算機に実用性の高いリスト処理システム (特に LISP) が実現されている.

一方において, LISP を中心とした専用マシン化の動きがある. (例 CADR, 1100SIP[26], FLATS, ELIS) これらのマシンについては, これまでに議論したアーキテクチャの視点からだけではなく, システム的な見地からも考察する必要がある. 専用マシン化の理由は, 以下のようなものであろう.

1) パーソナル化——同一の計算機を共同で利用する従来の TSS の進化した形態として, 各人が専用マシンを使うべきだとする考えによる. CADR, 1100 SIP などがその例.

2) 価格性能比——LSI 技術などを用いて, 汎用計算機アーキテクチャとリスト処理との不整合を比較的安価に克服する専用マシンを作ることが可能となった (ELIS がその例).

3) バックエンド化——汎用計算機アーキテクチャでリスト処理を行う不便さを解消するためハードウェア化して処理速度の飛躍的向上を狙う. FLATS, 富士通研の LISP マシン[29] などがその例.

• 応用分野から

世界的な規模では知識工学 (Knowledge Engineering) がリスト処理の最大の応用分野として注目を浴びている. この分野においては大量の知識ベースを表現する大容量記憶が必要となる.

従来からの応用例としては数式処理がある. この場合, FLATS の提案[11] にもあるように, 大容量記憶だけではなく数値計算の高速性とか, 既存の数値計算パッケージとの整合性も要求されるであろう.

一方, プログラミング・システムとしての LISP の対話操作性からは, ディスプレイやマウスなどの入出力機器などを含めた, 高度なマン・マシン・インタフェースが要求されるであろう.

応用分野として捉えるのはちょっとおかしいが, 教育上の見地からは低廉なマイクロプロセッサ上での LISP システムの実現も必要である.

• 技術的検討課題

最大の鍵はガーベジ・コレクションを含めた「大容量の記憶管理」技術であろう.

次に問題となるのは「マン・マシン・インタフェース」であるが, これは汎用計算機アーキテクチャの場

合より重要である.

高速性の問題については，これまでに知られた高速化技法の他には並列化しか残っていないだろうが，リスト処理における並列処理という議論はどちらかといえば始まったばかりで，応用分野からのフィード・バックを含めた検討は今後の課題であろう.

• リスト処理言語の見直し

リスト処理の代表的言語であるLISPは，ラムダ算法に基づいて設計され，さらにLISPの言語仕様を満足する高速マシンが考えられるといった進歩の過程をへてきた．しかし，ハードウェア（同一多数のプロセッサと大容量メモリ）が安価に得られるようになると，逆に，ハードウェアを高能率に駆動するような言語仕様が再度検討されるということになる．SussmanとSteeleによるLISPチップ[13]で稼動するSCHEMEと呼ぶリスト処理言語（LISPの方言）はその一例である．データフローの概念に基づくリスト処理言語の検討[12]なども，この流れに沿うものと考えることもできる.

他に例を挙げれば，抽象データ型や並列化，さらにはより厳密な意味での関数型化とか，第5世代の計算機で論じられているような論理型化といったテーマがある．あるいはsmalltalkのような，メッセージ伝播機能に基づいた言語体系も考えられる.

データ型化については「型つきLISP (typed LISP)」が以前から提案されており[21]，「並列処理LISPまたはconcurrent LISP」もある[22]．関数型プログラミングについてはリダクション・マシンのようにラムダ式を計算する手法があり，論理型化という点ではLISPの上のPROLOGとも言えるLOGLISP[23]の提案などがある.

以上みたようにリスト処理技術は今後大幅な進歩が予想される新たな応用分野の基礎となる技術である．これからもなお多くの研究活動を必要としよう.

参考文献

1) Kurokawa, T.: Fast Interpreter of Lisp 1.9, Jour. Inf. Proc., Vol. 2, No. 2, pp. 81-88 (1979).
2) 黒川：LISPのデータ表現——TOSBAC 5000 Lispを中心にして——，情報処理，17巻，2号，pp. 127-132 (1976).
3) Barbacci, M., Goldbarg, H. and Knudsen, M.: C. ai—A LISP Processor for C. ai, CMU-CS-71-103 (Aug. 1971).
4) Moses, J. and Winston, P. Proposal for Personal Computer Capable of Executing Large LISP Programs, ARPA (1975).
5) Knight, T. CONS, MIT AI Working Paper 80 (1974).
6) Sussman, G. J., et al.: Micro Planner Reference Manual, AI Memo 203a, MIT (1971).
7) McDermott, D. V. and Sussman, G. J.: The Conniver Reference Manual, AI Memo 259a, MIT (1974).
8) Rulifson, J. F: QA 4 Programming Concepts, SRI AI Tech. Note 62 (1971).
9) 「第5世代の電子計算機に関する調査研究報告書——アーキテクチャ研究分科会——」，54R-015，日本情報処理開発協会 (1980).
10) Weinreb, D. and Moon, D.: Lisp Machine Manual, Symbolics (1980).
11) 後藤英一：FLATSマシンの構想，理研シンポジウム，次世代の科学技術計算 (Mar. 1979).
12) 雨宮真人他：リスト処理向きデータフローマシンアーキテクチャとそのソフトウェアシミュレータ，電気通信学会技報，EC 80-69 (Feb. 1981).
13) Sussman G. J. Steele, G. L. Jr.: SCHEME An Interpreter for Extended Lambda Calculus, AI Memo 349, AI Lab., MIT (1975).
14) Wexelblett, ed.: History of Programming Languages, Academic Press, (1981).
15) Bobros, D. and Winopad, T.: An Overview of KRL, a Knowledge Representation Language, CSL-76-4, Xerox PARC (1976).
16) Hearn, A.: REDUCE 2 Users Manual, UCP-19, University of Utah (1973).
17) Steele, G. L. Jr. and Sussman. G. J.: Design of a LISP-based Microprocessor, Comm. of ACM, Vol. 23, No. 11, pp. 628-645 (1980).
18) Backus, J.: Can Programming Be Liberated from the von Neumann Style?——A Functional Style and Its Algebra of Programs, Comm. of ACM, Vol. 21, No. 8, pp. 613-641 (1978).
19) Bobrow, D. G. and Murphy, D. L.: Structure of a LISP system Using Two Level Storage, Comm. of ACM, Vol. 10, No. 3 (1967).
20) Clarke, T., Gladstone, P., Maclean, C., and Norman, A.: SKIM——The S, K, I Reduction Machine, Conference Record of the 1980 LISP Conference (1980).
21) Cartwright, R.: A Constructive Alternative to Axiomatic Data Type Definitions, Conf. Record of 1980 LISP Conf. (1980).
22) Sugimoto, S., Tabata, K., Agusa, K. and Ohno, Y.: CONCURRENT LISP ON A MULTI—MICRO—PROCESSOR SYSTEM, IJCAI-81, pp. 949-954 (1981).
23) Robinson, J. A. and Sibert, E.E.: THE LOGLISP USER'S MANUAL, Syracuse University (1981).

24) Bobrow, D. G., Clark, D. W.: Compact Encodings of List Structures, ACM TOPLAS, Vol. 1, No. 2, pp. 266-286 (Oct. 1979).
25) Comer, D.: The Ubiquitous B-Tree, ACM Computing Surveys, Vol. 11, No. 2, pp. 121-137 (Jun. 1979).
26) 富士ゼロックス: 1100 Scientific Information Processor (Jun. 1982).
27) Henderson, P. and Morris, J. H.: A lazy evaluator, Proc. 3rd Conf. on Princ. of Prog. Lang., pp. 95-103, ACM (1976).
28) Friedman, D. P. and Wise, D. S.: Cons should not evaluate its arguments, In Automata, Languages, and Programming, pp. 257-284, Edinburgh Univ. Press (1976).
29) 服部 彰, 篠木 剛, 品川明雄, 林 弘: 「高速リスト処理に適したアーキテクチャについて」, 情報処理学会記号処理研究会資料 (Mar. 1982).

(昭和57年3月25日受付)