記　号　処　理 39－5
（1986. 11. 10）

リスト構造の内部表現と処理系

和田良一　　青木豊　　本間真人　　江村里志
松下電器産業株式会社　　無線研究所

リスト構造を従来のポインタ表現ではなく、要素の表で表現した処理系を持つリスト処理マシンＡＴＯＭ（A Tabular List Representation Oriented Machine）の開発を行っている。

ＡＴＯＭにおけるリスト構造は、葉のノード位置を一次元ベクトルで表すADDRESS部と要素の内容を示すVALUE部との組から成る表形式で表現され、リストの要素アクセスやパターンマッチングを高速に処理することができる。

本稿では、ＡＴＯＭの処理系の構成、リスト構造の内部表現形式を紹介し、基本的なリスト演算およびマッチング処理の手法について述べる。

A LIST PROCESSING ARCHITECTURE BASED ON A TABULAR REPRESENTATION OF LISTS

Ryoichi Wada　Yutaka Aoki　Masato Homma　Satoshi Emura
Wireless Research Laboratory, Matsushita Electric Industrial Co., Ltd.
1006, Kadoma, Kadoma-shi, Osaka, 571 Japan

In this paper a list processing machine ATOM (A Tabular List Representation Oriented Machine) is described. The main difference between the ATOM and other architectures is that it has introduced a new way of representing list structures in tabular form. Each entry of a table corresponds to a leaf node and consists of the address and value parts. The address part of is a vector which identifies uniquely the leaf location and the value part contains a pointer to its content. This representation is suitable for fast list access and pattern matching.

1．はじめに

近年、リスト構造は主として人工知能分野を中心に幅広く使用されるようになり、その処理系の高速性の追求は計算機アーキテクチャ分野の重要な研究課題の一つである。従来からリスト構造を取り扱うことの出来る言語としてＬＩＳＰがあり、この分野の研究は直接的にはＬＩＳＰ言語を高速に実行する言語実行系を中心に展開されており、リスト構造データの処理自体の面からのアプローチは比較的少なかった。

伝統的にリストの計算機内部表現はポインタが用いられている。これはリストをノイマン型コンピュータへマッピングする際の最も素直で柔軟性に富んだ表現であるからだと考える。しかしながら一方で、その処理系については様々な問題点を抱える原因となった。代表的なものをＬＩＳＰを中心に以下に示す。

1．任意の要素へのアクセスがリストたぐりとなり効率が悪い。
2．リストマッチングはリストの分解操作を伴うため非効率である。
3．ガーベッジコレクションが困難である。
4．メモリ参照の局所性が悪く、キャッシュのヒット率が下がる。

また、基本的には共有構造をとるため以下の問題が生じた。

5．ＲＰＬＡＣＡ，ＲＰＬＡＣＤ等、直接リスト操作を行うと陰に他のデータも変更してしまうといった思いがけない副作用が生じる。
6．並列処理時、変数のロックが困難である。

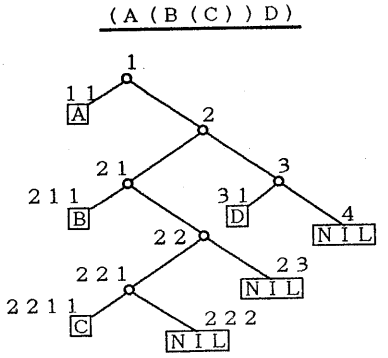
我々は直接的には言語ではなくリスト構造データの処理の観点からその計算機内部表現を変えることにより、上記問題点を解消した処理系を持つリスト処理マシン ＡＴＯＭ（ A Tabular List Representation Oriented Machine ）の開発を行っているのでその内容につき報告する。

2．リストデータの表形式表現

２進木リストは始点のノードから始まって順次左右に分岐して行き葉のノードでそれぞれの分岐が終了する形をとる。葉のノードにはアトムノードとＮＩＬノードの２種類がある。葉のノードでないノードは分岐が続行している事を示すリストノードである。このリストノードは葉のノードの位置を間接的にあらわすためものである。

ポインタ表現ではこの構造表現をそのままの形で全てのノードをアドレスで接続したセルで表現している。

しかしながら、葉のノードの位置を直接的にあらわすことができれば、リストノードの情報を持つ必要はない。したがって、葉の位置情報と葉自身の情報を順次並べた表で、等価なリストデータを表現することができる。我々はこの葉のノード位置を表現する方法としてＣＤＲ方向に順次番号を付け、ＣＡＲ方向に順次項目を割り当てた一次元ベクトル表現



（1）図式表現

1	1			.....	A
2	1	1		.....	B
2	2	1	1	.....	C
2	2	2		.....	NIL
2	3			.....	NIL
3	1			.....	D
4				.....	NIL

ADRESS部　　VALUE部

・ＡＤＤＲＥＳＳ部：　リストの葉の位置
・ＶＡＬＵＥ部：　　各アトムへのポインタ

（2）表形式表現

図１　リストの表形式表現

を採用した。従ってリストデータは葉の位置情報を示すベクトルと葉自身の情報を組としたデータの集合で表現される。

図１にリストデータの表現例を示す。これはＳ式で表記した場合（Ａ（Ｂ（Ｃ））Ｄ）となるリストデータの図式表現（１）、および、表形式表現（２）を示したものである。図式表現において丸印はリストノードを表し、四角で囲ったものは葉のノードを示している。また各ノードの上に付記した数字列は上記した方法に従って表したノード位置を示すものである。この葉の部分を抜きだして表の形で表現したものが表形式表現（２）であって、ＡＤＤＲＥＳＳ部にノード位置ベクトルが、ＶＡＬＵＥ部に葉の要素が入った表で構成されている。

表の制限

表形式表現においてはポインタ表現と異なり、ＡＤＤＲＥＳＳ部の大きさの制限から表現できるリストの大きさに制限が生じる。例えば、ＡＤＤＲＥＳＳ部の各項目のビット数を８ビットとした場合、表現できるリストの長さは２５６に制限される。また、項目の数は表現できるリストの深さを制限する。

処理系の設計時にはこの大きさの設定が重要である。設計時に想定した大きさを超えたリストが発生した場合、なんらかの例外処理を行う必要があるが、この場合効率低下は避けられず、実際の使用時における発生頻度が問題となる。

我々はいくつかのＬＩＳＰアプリケーションプログラムにつき予備実験による発生リストの大きさの調査を行ったが、深さで８レベル、長さで２５６を超えるリストの発生頻度はいずれも５％以下であった。したがって、この場合における表のＡＤＤＲＥＳＳ部の大きさは６４ビットとなる。

拡張ＬＩＮＧＯＬを使用して行ったリストの発生頻度の調査結果の一部を図２に示す。

基本的リスト演算

図３にＬＩＳＰの基本リスト操作関数における表操作の一例を示す。

図で明かなようにＣＡＲ関数はＡＤＤＲＥＳＳ部の先頭が１の要素を抜き出し、２番目以下のＡＤＤＲＥＳＳを先頭方向にシフトすることにより（これを以下ＣＡＲシフトと呼ぶ）実行することができる。ＣＤＲ関数はＣＡＲ関数とは逆に先頭ＡＤＤＲＥＳＳが１の要素を取り除き、残った要素の先頭ＡＤＤＲＥＳＳから１を減じることにより（以下これをＣＤＲシフトと呼ぶ）実行される。

<< LINGOLにおける解析の一例 >>

********** リストの長さ **********

<長さ>	<回数>	<割合%>	<累積%>
0	3018	68.33	[68.33]
1	279	6.32	[74.64]
2	289	6.54	[81.19]
3	301	6.81	[88.00]
4	126	2.85	[90.85]
5	334	7.56	[98.42]
6	12	0.27	[98.69]
7	2	0.05	[98.73]
8	2	0.05	[98.78]
9	2	0.05	[98.82]
10	2	0.05	[98.87]
11	6	0.14	[99.00]
12	2	0.05	[99.05]
13	9	0.20	[99.25]
14	2	0.05	[99.30]
15	2	0.05	[99.34]
16	2	0.05	[99.39]
17	2	0.05	[99.43]
18	2	0.05	[99.48]
19	2	0.05	[99.52]
20	2	0.05	[99.57]
21	2	0.05	[99.62]
22	2	0.05	[99.66]
23	2	0.05	[99.71]
24	2	0.05	[99.75]
25	2	0.05	[99.80]
26	2	0.05	[99.84]
27	2	0.05	[99.89]
28	2	0.05	[99.93]
29	2	0.05	[99.98]
30	1	0.02	[100.00]

********** リストの深さ **********

<深さ>	<回数>	<割合%>	<累積%>
0	3018	68.33	[68.33]
1	702	15.89	[84.22]
2	132	2.99	[87.21]
3	216	4.89	[92.10]
4	85	1.92	[94.02]
5	62	1.40	[95.43]
6	33	0.75	[96.17]
7	49	1.11	[97.28]
8	31	0.70	[97.99]
9	22	0.50	[98.48]
10	15	0.34	[98.82]
11	13	0.29	[99.12]
12	9	0.20	[99.32]
13	10	0.23	[99.55]
14	11	0.25	[99.80]
15	5	0.11	[99.91]
16	4	0.09	[100.00]

図２　リストの出現頻度

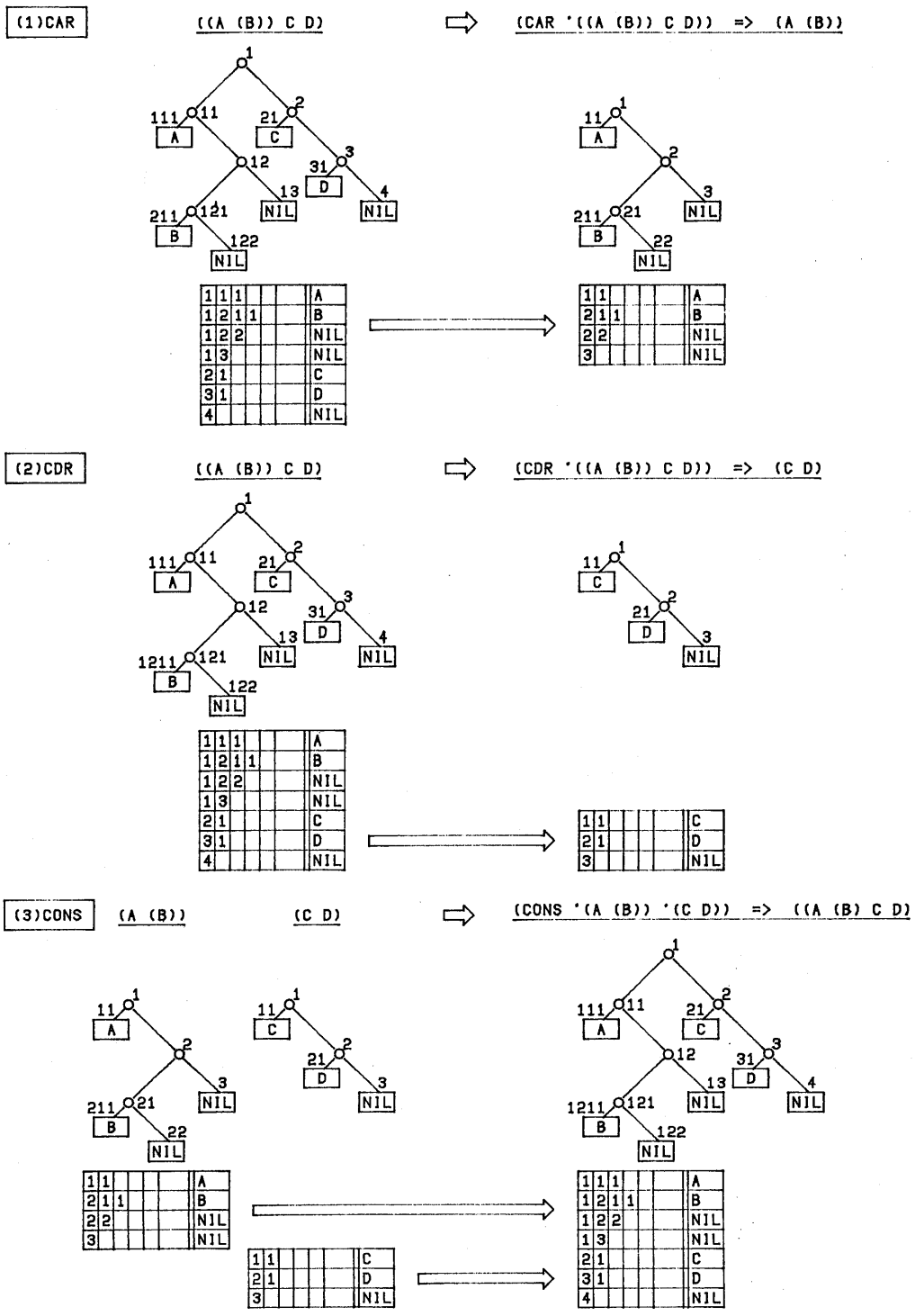

(1)CAR
((A (B)) C D)
(CAR '((A (B)) C D)) => (A (B))
(2)CDR
((A (B)) C D)
(CDR '((A (B)) C D)) => (C D)
(3)CONS
(A (B))
(C D)
(CONS '(A (B)) '(C D)) => ((A (B) C D)

図3　基本リスト操作

ＣＯＮＳ関数についてはもう少し複雑である。まずＣＯＮＳ関数の第一引数について、ＡＤＤＲＥＳＳ部を先頭方向とは逆の方向にシフトし、先頭ＡＤＤＲＥＳＳを１とする。これは前記したＣＡＲシフトとは逆の演算であり、以下ＲＣＡＲシフトと呼ぶ。第二引数については先頭ＡＤＤＲＥＳＳに１を加算する（以下これをＲＣＡＲと同じ理由でＲＣＤＲシフトと呼ぶ）。次にこの二つの引数を一つの表としてまとめればＣＯＮＳ関数は実行される。

以上述べてきたように、ポインタ表現では極めて容易に行えるＣＡＲ，ＣＤＲ，ＣＯＮＳといったＬＩＳＰの基本関数が表形式では全ての要素に対しての演算を必要とする。したがって、処理系のアーキテクチャはこの演算量の増加に対処した構成を持つことが必要である。

要素アクセスとパターンマッチング

一方、表形式表現ではポインタ表現と比較して得意な演算がある。一つは任意のリスト要素へのアクセスであって、ポインタ方式では始点から順次リストセルを手繰って行く必要があるのに対し、表形式では要素ＡＤＤＲＥＳＳから直接的にアクセスすることができる。特に後述するストラクチャレジスタでは、この演算を１サイクルで実行することができる。

もう一つはリスト構造データのマッチング処理である。ポインタ表現では両者のリストを同じ手順で要素まで分解し、それらの要素が等しいかどうかを検定することによりリスト構造データの同一性を調べている。表形式では、順次一方の要素を他方と比較して行き、すべての要素につきそのＡＤＤＲＥＳＳ部およびＶＡＬＵＥ部が等しければ、同一であることがわかる。

特に強力なのはＡＤＤＲＥＳＳの一部にワイルドナンバーを許すことにより部分リストのマッチングが容易に行えることである。ＡＳＳＯＣ関数の例を図４に示す。ここで＊はワイルドナンバーを示し、どのような数ともマッチングすることを示す。この方法を使用すると、ある構造データから特定の構造を持つ部分を抜き出すといった従来のポインタ方式では難しかった処理を比較的容易に行うことができる。

インプリメンテーション上の問題

表形式表現の現実的インプリメントに際し、いくつかの項目につき選択岐が存在する。まず、ＶＡＬ

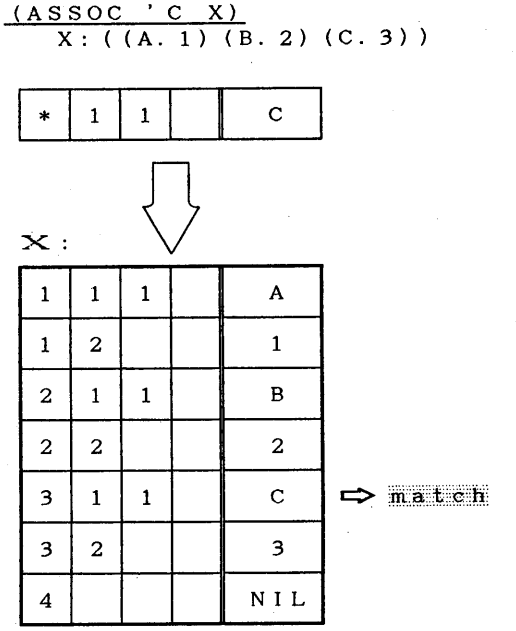


図４　ＡＳＳＯＣ関数の実行

ＵＥ部データは基本的にはアトムへのリファレンスであるが、ここに他の表へのリファレンスを許すかどうかということである。もし許さなければ循環リストが表現できないことになる。しかしながら循環リストは実用的には比較的重要度が低く、表形式がポインタ表現に近づくことにより従来の欠点を引きずることの方がむしろ問題であると考えるため、ＶＡＬＵＥ部に他の表へのリファレンスを許さないことにした。

また、表形式データそのものに対して複数の共有を許すかどうかということがある。前記説明で明らかなように、表形式表現においては途中のリストセルの共有は不可能である。しかしながら、表形式データそのものに対する共有の有無は別の選択岐である。この問題はＬＩＳＰのＥＱ関数の動作に影響するものであって、通常のＬＩＳＰインプリメンテーションではアドレスの一致を検定している。したがって対象はアトムでもリストでも良いことになるが、表形式において共有を許さないとするとリストの判定は不可能となる。しかしながら、もともとＥＱ関数の定義域はアトムであり、リストについてはインプリメンテーションとの関係で保証されたものではないことから、我々はここでも共有を許さない方を採用した。

したがって、我々が選択した表形式表現は通常手続き言語におけるデータの独立性に関して同じ性質を有することになる。この場合、リストのコピーが多く発生するおそれがあるが、もともと共有リストは２．５%程度しか発生しないとの報告もあり問題はないと考えられる。

また、ＲＰＬＡＣＡ，ＲＰＬＡＣＤ，ＳＥＴＦといった直接リスト操作を行う関数について、リストの共有構造を利用して陰の効果を期待したプログラムがあった時には、その動作が異なる。しかしながら、すべての効果はプログラムに明示的にあらわすべきだとの近代プログラミング技法の観点からみると、このサイドエフェクトはむしろ有害なものであり実用的には不用な効果であると考える。

表形式データのメモリ内表現

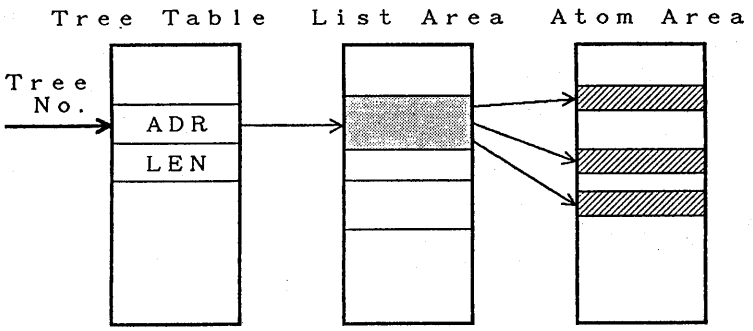
処理系における表形式データのメモリ内表現を図５に示す。表形式リストデータはメモリ内のリストエリアに連続して格納される。そして格納位置およびその大きさはツリーテーブルで管理される。すなわち、通常の可変長文字ストリングデータの格納と同じ形でメモリ内に配置され管理される。表形式リストデータへのアクセスはツリーテーブルのエントリーナンバーであるツリーナンバーを通して行われる。この様にすることにより、不要データエリアの回収、すなわち、ガーベッジコレクションを容易に行うことができる。変数に対して新しいデータを代入したときなどに発生する不要データは、必ずツリー単位で発生し、しかも前記したように共有構造を持たない。したがって、発生時に検出が可能であり、その時ツリーテーブルにマーキングをすることにより、任意の時点において回収が可能である。

３．処理系の概要

図６は処理系の一部で、表形式リストデータを処理する部分のブロック図である。表形式リストデータは処理の際、メインメモリからストラクチャレジスタＳＲにいったんロードされて、レジスタ演算として実行される。これは通常の計算機において数値や文字が処理される過程と同じである。この場合転送データ量が多くなるが、前記したようにリストデータは連続したブロックとして格納されているので、メモリ素子に高速連続転送が可能なものを採用することにより転送負担の軽減を図っている。表形式のリストデータは可変長であるが、ＳＲ内では６４の整数倍の固定長として扱われる。ＳＲは全部で２５６個あり１６個づつの１６グループで構成され、レジスタウィンドウを構成している。このため、変数へのリストデータ代入操作時にはＳＲからメインメモリへのデータ転送が行われるが、関数呼び出し時の引数の受渡しにはＳＲを使用することによりこの転送負担を軽減している。

条件レジスタＣＲは、ＳＲに蓄えられたリストデータの特定の要素にアクセスしたり、比較処理を行ったりする際に条件を指定するためのもので、任意のリストアドレスあるいはＶＡＬＵＥを設定できる。

データレジスタＤＲはリストの構成要素自身の演算を行うためのものである。たとえば、リストの構成要素に数値があってそれに対して加算等の演算を行う場合、ＳＲ内のリストデータのＶＡＬＵＥ部からリファレンスを取り出し、その値をＤＲにロードし、演算はＤＲ内で実行される。



ＡＤＲ：　Ｌｉｓｔ　Ａｒｅａ内のアドレス
ＬＥＮ：　リストの長さ

図５　リストのメモリ内表現

ストラクチャレジスタ

図７にストラクチャレジスタＳＲの構成を示す。
ＳＲは要素毎に２５６個の１２８ビットレジスタ、インプットシフト回路ＩＳ、ストラクチャ演算回路ＳＡＬＵによって構成され、６４要素まで同時に処理できるように同じ構成の回路が６４個で構成されている（以下この方向の単位をスライスと呼ぶ）。リストデータはメモリからロードされるとき、各スライス毎にレジスタ選択線ＲＳによって選択された番号のレジスタに一要素づつ入力される。したがってＳＡＬＵによる演算は各要素パラレルに行われる。ＩＳは前述したＣＡＲシフト、ＣＤＲシフト等の各種ＡＤＤＲＥＳＳ部のシフト操作を行う回路であり、同一のレジスタ内でのシフト操作、および異なったレジスタへのシフト結果の転送の両者が可能である。いずれにせよレジスタから読み出された要素データはいったんＳＡＬＵに入り、ＩＳの入力に戻されてシフト操作を受けた後、再びレジスタにストアされる。ＳＡＬＵはレジスタデータとＣＲのデータを比較し、その結果をＦＬＡＧとして出力したり、その比較情報に基づいてメモリデータバスやデータバスにレジスタデータの内容を出力するゲート回路を制御する働きを行う。

４．おわりに

現在、我々が開発を進めているリスト処理マシンＡＴＯＭで採用している表形式リスト内部表現、および、その処理系の概要につき述べた。
本方式によれば、リストの比較操作等、従来実行効率の悪かった演算について改善することができる反面、ＣＡＲ，ＣＤＲ，ＣＯＮＳ等、ＬＩＳＰにおける基本関数の実行に多くのデータ操作を必要とする問題がある。実用システムとしては、実際の応用局面におけるそれらの操作の出現頻度の加重平均でパフォーマンスの検定を行う必要があるが、現在存在するＬＩＳＰアプリケーションプログラムでは比較、要素の抜き出し等の演算も基本関数を組み合わせた形で表現されているため、そのままの形では検定が困難である。
我々はこれら新しいプリミティブを言語に反映させる方法で、現在いくつかのアプリケーションプログラムの検定を進めている。
また処理系においては、メモリとレジスタ間のデータトラフィックが増大するためメモリバンド幅がそのボトルネックとなる恐れがあり、これについても現在検証を進めている段階である。
最後にこの研究に対して御指導いただいた京大工学部情報工学科の大野豊教授、阿草清滋助教授、および、大野研のメンバーの方々に感謝します。

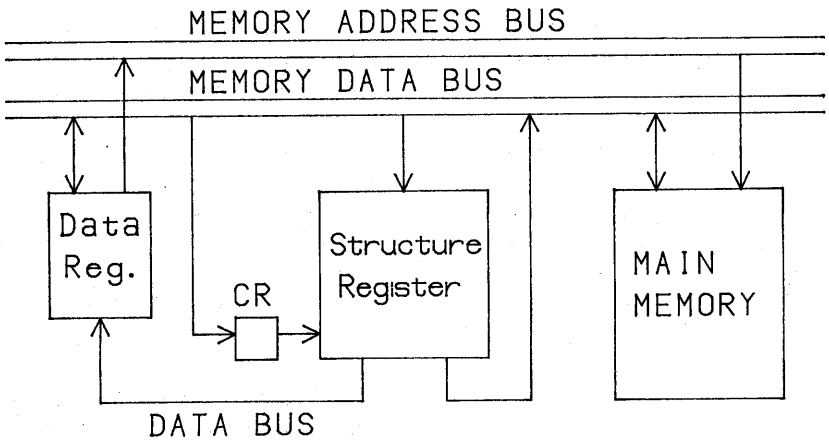


図６　リスト処理回路のブロック図

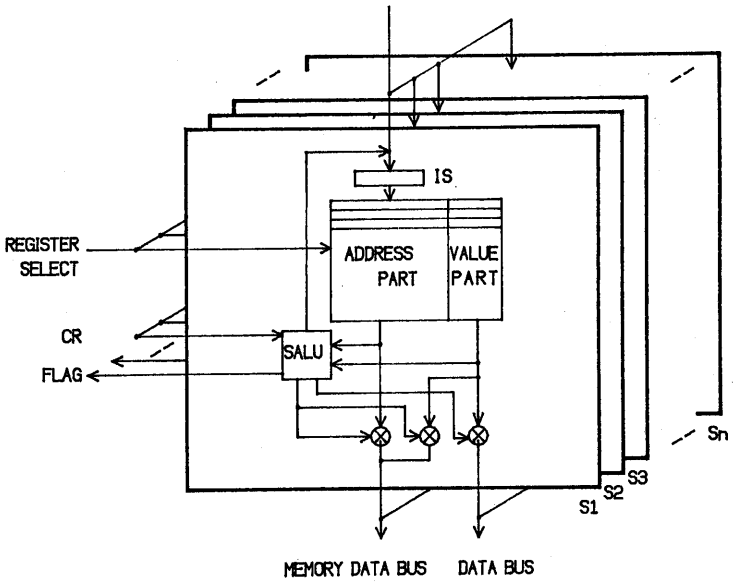


図７　ストラクチャレジスタ

参考文献

[1] John Allen: Anatomy of LISP, McGraw-Hill, 1978.

[2] Gurinder S. Sohi et al.: AN EFFICIENT LISP-EXECUTION ARCHITECTURE WITH A NEW REPRESENTATION FOR LIST STRUCTURE, Conf. Proc. Annu. Int. Symp. Compt. Archit., Vol.12, 1985.

[3] Tetsuo Ida et al.: Associative Descriptor Scheme - for the exlotation of address arithmetic in Lisp, J. Information Processing, Vol. 4, No.3, 1981.

[4] Bobrow, D. G. et al.: Compact Encoding of List Structures, ACM Trans. on Prog. Lang. and Sys., Vol.1, No.2, 1979.

[5] 太田他: LISP関数によって生成されるリスト構造の解析について，電子通信学会オートマトンと言語研究会資料，AL84-14，1984.

[6] Olsson, O.: The Memory Usage of a LISP system: The Belady Life-Time Function, SIGPLAN Notices, Vol.18, No.2, 1983.

[7] Padget, J. A.: The Ecology of LISP or The case for the preservation of the environment, Compt. J., Vol.25, No.2, 1982.

[8] Eick A. et al.: Inconsistencies of pure LISP, Lct. Notes Comput Sci, Vol.145, 1982.

[9] 丹羽他: 関数型言語の大域変数アクセスを高速化するキャッシュメモリ方式，情報処理学会第２４回全国大会，2L-7，1982.

[10] 黒川他: リスト処理とアーキテクチャ，情報処理，Vol.23, No.8, 1982.

[11] Harrison, P.G.: Efficient Storage Management for Functional Languages, Comput. J., Vol.25, No.2, 1982.

[12] Suzuki, M. et al.: A Primitive for Non-recursive List Processing, J. Information Processing, Vol.4, No.4, 1981.

[13] Goodwin, J.: Why Programming Environments Need Dynamic Data Types, IEEE Trans. on Software Eng., Vol.SE-7, No.5, 1981.

[14] 平木: Lispにおける記憶管理方式の評価，情報処理学会第２１回全国大会，5J-8，1980.

[15] 横内: パターン・マッチングを利用したリスト処理用言語とその応用例，情報処理学会第２１回全国大会，5B-5，1980.

[16] 服部他: 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，18-9，1982.

[17] 後藤他: 記号数式処理向き計算機ＦＬＡＴＳの構想，　情報処理学会記号処理研究会資料，　1-1，1977.

[18] 斉藤他: 高速ＬＩＳＰマシンとリスト処理プロセッサＥＶＡＬⅡ，情報処理学会論文誌，Vol.24, No.5, 1983.

[19] 和田他: 構造をもったデータの高速マッチング方式，情報処理学会第３３回全国大会，7D-2,1986.