計算機アーキテクチャ 65－1
（１９８７．３．１３）

# 記号処理マシンＡＴＯＭのアーキテクチャ

和田　良一　本間　真人　岡村　和男
松下電器産業株式会社　無線研究所

本報告では、リストデータを各要素の木構造上のアドレスと値の組の集合で表現し、処理する記号処理マシンＡＴＯＭ（A Tabular List Representation Oriented Machine）のアーキテクチャについて述べる。

ＡＴＯＭは通常のデータを処理するＤＯＵ（Data Operation Unit）と、リストデータを処理する複数のＳＯＵ（Structure Operation Unit）から構成されたＳＩＭＤ方式の計算機である。ＡＴＯＭではリストの各要素をそれぞれ別のＳＯＵに割り当て、並列に処理するため、従来のポインタ表現でボトルネックとなっていたリスト手繰りを伴うことなく、リスト処理が可能である。特にリスト処理において出現頻度の高いパターンマッチング演算の場合、特定要素の検索や構造体の比較を各要素毎に並列に行うことによって、高速な処理が実現できる。

# Architecture of ATOM: A Fast List Processing Machine

Ryoichi Wada Masato Homma Kazuo Okamura
Wireless Research Laboratory, Matsushita Electric Industrial Co., Ltd.
1006, Kadoma, Kadoma-shi, Osaka, 571 Japan

In this paper, an architecture of a list processing machine ATOM(A Tabular List Representation Oriented Machine) is described. ATOM is an SIMD machine constructed from Structure Operation Units (SOU), a Data Operation Unit (DOU). Alist structure is represented as a set of tuples which consist of the address and value part. Each tuple, which corresponds to a leaf in a list, is assigned to a different SOU and all tuples are processed by SOUs in parallel. Therefore ATOM performs list operations without sequential references to pointers that cause a major bottleneck in the conventional list processing machines.

## １．はじめに

近年、人工知能分野を中心にリスト等の構造を持ったデータを使用する機会が増大しており、その処理系の高速化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。

伝統的にデータ間の関係はポインタ表現の２進木リストで計算機内部表現され、また、それを操作することを前提に処理言語が組み立てられてきた。この表現は最も単純な構造の計算機内部表現であり、その柔軟性から記号処理分野へ広く使用されてきた。

しかしながら、記号処理の分野で最も出現頻度が高い演算はパターンマッチングと任意要素の取り出しであり、ポインタ表現ではこれらの演算はいわゆるリスト手繰りとなり、メモリアクセスの増大が処理速度上のボトルネックとなっている。

我々はそれぞれの要素に木構造上のアドレスを付加したデータ集合でリストを表現し、処理することにより主にパターンマッチング演算を高速にすることを目的としたリスト処理マシンＡＴＯＭ（A Tabular List Representation Oriented Machine）の開発を行っているのでその内容に付き報告する。

## ２．リストの内部表現

### ２．１．ポインタ表現

従来よりリスト構造は通常２進木で表現されている。

２進木リストは始点のノードから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ことを示すリストノードである。このリストノードはそれぞれの葉のノードの位置を間接的にあらわすためのものである。

ポインタ表現においてはこの構造表現をそのままの形で間接アドレスでメモリ内にマッピングする。したがって、任意の葉の要素にアクセスするためには順次リストノードを手繰って行く必要がある。（ＬＩＳＰ言語ではこの手繰り処理を直接的に表現する。）また、２つのリストが同値であるか調べるマッチング処理においては、構造を直接的に比較できないため、同一の手順で２つのリストを葉の要素まで分解しそれぞれの葉の要素が等しいかどうかを調べることにより判定する方法をとる。これらの処理は頻繁なメモリアクセスを要求し処理速度のボトルネックとなっている。

上記の問題は葉の位置をリストノードを使用して間接的に表現していることに起因している。したがって、自分自身の位置を示す情報を各葉の要素に持たせるようにできればリストノードは不要であり上記問題は解消する。

### ２．２．表形式表現

ＡＴＯＭにおいて構造は多進木表現であり、かつそれぞれの葉の位置情報と葉自身の情報を並べた表で内部表現されている。位置情報は以下に示す規則で作成された整数列で表現される。

（１）ルートノードにおける左からｎ番目の子ノードアドレスは［ｎ］。

（２）レベルｋのノードのアドレスは
［$P_1$, $P_2$, ・・・$P_K$］
で表され、そこにおいて左からｎ番目の子ノードアドレスは
［$P_1$, $P_2$, ・・・・, $P_K$, ｎ］
で示される。

ここで、Ｐｉ＞＝１（ｉ＞＝１），
ｎ＞＝１，ｋ＞＝１である。

Ｆｉｇ．２．１及びＦｉｇ．２．２に２進木表現と多進木表現のリストの関係、および、ノードの位置情報の例を示す。

（A （B （C）） D）

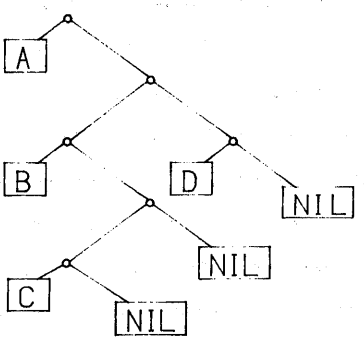



**Fig.2.1 An Example of binary tree**

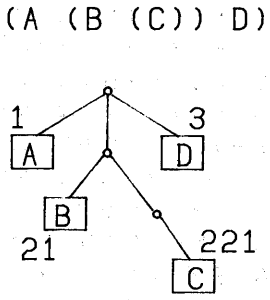



**Fig.2.2 Tree representation in ATOM**

リストの表形式表現は各々の葉の要素にその位置情報を付加し、このデータを並べた表でリストを表現するものである。Ｔａｂｌｅ２．１にリスト

（A（B（C））D）

の表形式表現の例を示す。

| | | | | | |
|---|---|---|---|---|---|
| 1 | | | | | A |
| 2 | 1 | | | | B |
| 2 | 2 | 1 | | | C |
| 3 | | | | | D |

**Table2.1 Tabular Representation of List**

表形式表現においてはポインタ表現とは異なり、インプリメント時の表の大きさの制限が表現できるリストを制限する。すなわち、リストのレベルは位置情報ベクトルの項目数で、リストの長さはそれぞれの項目に割り当てられたビット数で、葉の数は表のエントリ数でそれぞれ制限される。

我々はいくつかのＬＩＳＰのアプリケーションにつき解析を行ったが、レベルで８、長さで２５６、葉の数で３２を越えるリストの出現頻度はそれぞれ５％以下であった。この結果を参考にして、ＡＴＯＭでは位置情報の総ビット数を６４ビット、ＳＯＵ数（表のエントリ数）を６４とした。

## ３．ＡＴＯＭの構成

Ｆｉｇ．３．１にＡＴＯＭの構成図を示す。ＡＴＯＭは複数のＳＯＵ（Structure Operation Unit）と、一つのＤＯＵ（Data Operation Unit）と一つのコントローラとで構成されたＳＩＭＤ型のコンピュータである。

すべてのＳＯＵは同じ構成であって、表形式リストデータを蓄えるリーフメモリと葉の要素に対する各種演算を行うためのストラクチャレジスタとＡＬＵとで構成されている。

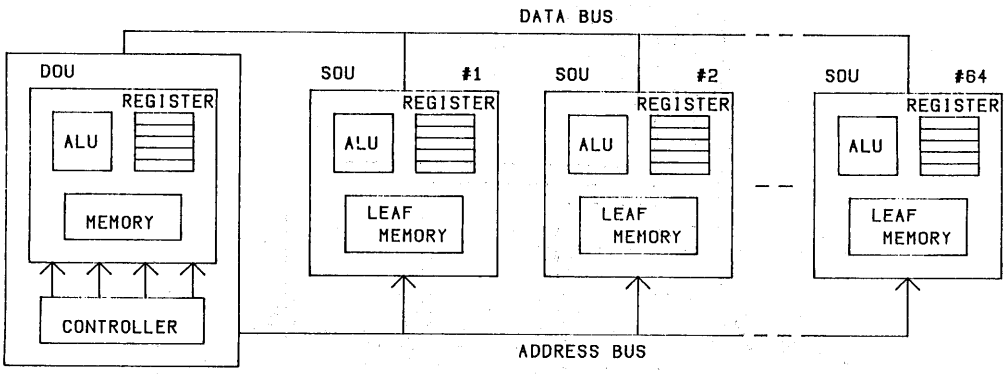



**Fig.3.1 Block diagram of ATOM**

ＤＯＵは２つの部分から構成されている。一つはアトムの値である数値や文字データを処理する部分で通常のコンピュータとほぼ同じ構成を有する。もう一つはＳＯＵと共同してリストデータを処理する部分である。ＳＯＵはＤＯＵのこの部分に同期して作動する構成になっている。

同一リストの個々の葉要素は異なったＳＯＵのリーフメモリの同一のアドレスの場所に格納される。この様子をＦｉｇ．３．２にしめす。したがって、通常は表のｎ番目の要素はｎ番目のＳＯＵに格納されている。ＤＯＵ内のメモリの対応した場所にはそのリストの識別子が格納されている。前述したように全てのＳＯＵはＤＯＵに同期して作動するため、全てのリストのデータはストラクチャレジスタへ同時にロードされ並列に処理される。これはＡＴＯＭにおいてはリストデータ全体をあたかもひとつのデータとして取り扱う事を可能とする。
事実、ＡＴＯＭの命令体系上存在するのはＤＯＵだけであって、直接ＳＯＵを操作する命令は存在せず、ＳＯＵは見えなくなっている。すなわち、命令アーキテクチャのレベルで既にこの抽象化が行われている。

例えば、リストデータをレジスタにロードするＭＯＶ命令では、そのオペランドはＤＯＵ内のレジスタである。しかしながら実行時にそのレジスタにロードされるのはリストデータの識別情報だけで、実際のリストデータはＤＯＵ内のレジスタに対応した複数のＳＯＵ内のレジスタに全ての葉の要素が同時にロードされる。他の命令も同じように動作する。したがって、ＡＴＯＭにおいては位置情報をもつ葉のデータの集合で表現されたリストデータ全体が一つのデータとして通常の数値データや文字データと同じように取り扱われる。

Ｆｉｇ．３．３にＤＯＵのブロック図を示す。ＤＯＵは文字や数値等、通常データを処理するためのＲレジスタ群（ＡＸ－ＨＸ）と前記リストデータを処理するためのＳレジスタ群（ＳＡ－ＳＨ）とで構成される。それぞれのビット幅は６４ビットである。またそれらを接続する内部バス幅も６４ビットである。レジスタＣＲＡ，ＣＲＶは葉の要素を個別に扱うため用意されたそれぞれ６４ビット幅のレジスタで、前者が葉のアドレスに、後者が葉の要素に対応する。このレジスタの出力はＳＯＵのデータバスに接続されている。Ｆｉｇ．３．４はＳＯＵのブロック図で、ＤＯＵのＳレジスタに対応するレジスタ群（Ｓ－ＳＡ　－　Ｓ－ＳＨ）とＡＬＵ、および、リーフメモリとで構成されている。内部バス幅は１２８ビットである。

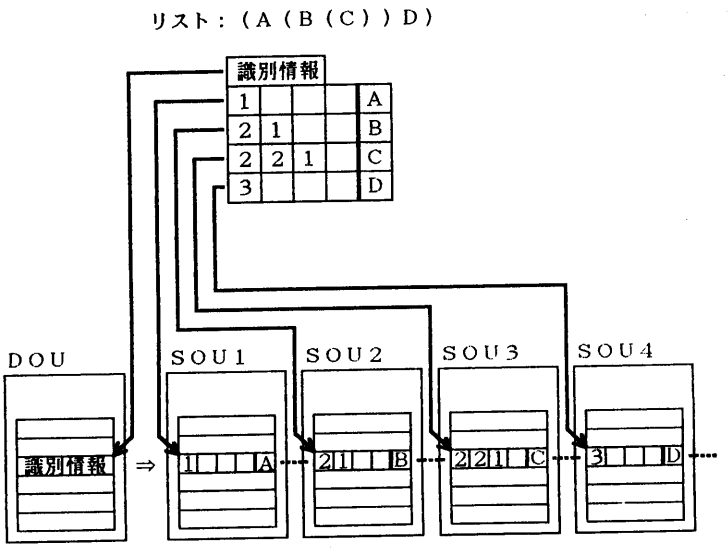



**Fig.3.2 Data Arrangement in ATOM**

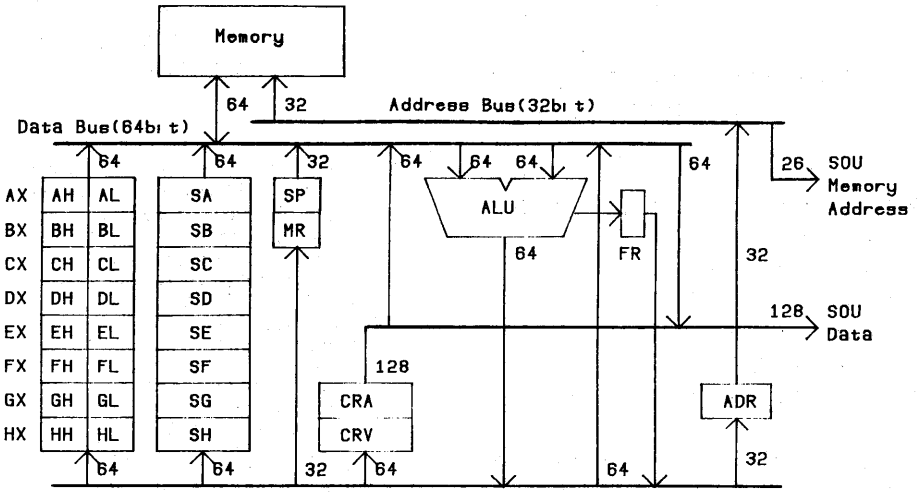



Fig.3.3 Block diagram of DOU

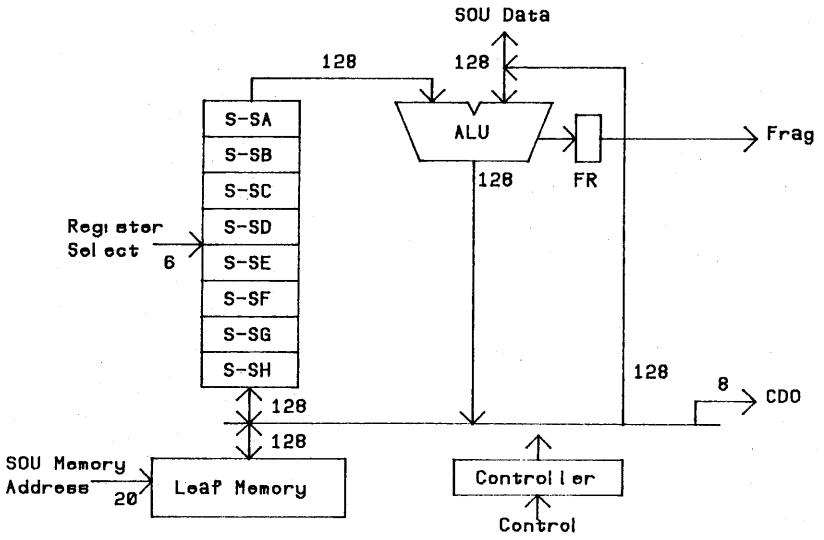



Fig.3.4 Block diagram of SOU

## ４．リストの操作体系

前述したようにリストデータはアドレスを持った葉のデータの集合で内部表現されている。リストの演算は直接的にはこのデータ集合に対する演算で定義されるためポインタ表現とは少し異なった体系となる。

Ｔａｂｌｅ４．１にＡＴＯＭの表形式リストに対する基本操作命令を示す。これらの命令の内容は以下の５つに要約される。

（１）リストデータの転送
（２）各要素のアドレス演算
（３）特定の要素へのマーク付け
（４）特定ノードの除去
（５）２つのリストのマージ

実際のリスト演算は上記の命令を組み合わせて実現される。たとえば、最初のノードを取り出すＬＩＳＰのＣＡＲ操作は最初のノードにマーク付けを行った後にマークされていないノードを除去すれば良いし、リストの要素を逆に並べるＲＥＶＥＲＳＥはトップアドレス（ルートノードにおける要素の位置アドレス）からリストの長さを減じて極性を逆にすれば良い。ポインタ表現においては明かに複雑度の異なるこの２つの例の操作の演算ステップは表形式リストにおいては殆ど同じである。このように、それぞれのリスト操作に対する実際の計算機の演算ステップはポインタ表現と比較してかなり異なったものとなる。特に次章で述べるパターンマッチングにおいてこの差は顕著である。

## ５．パターンマッチング処理

ここで言うパターンマッチングとは２つのリストの構造、各要素の照合操作のことで、ＰＯＰ－１１のシンタックスで matches や --> に対応する操作である。

以下の表示はこのＰＯＰ－１１のシンタックスを使用して行う。最も簡単な例は次式で示される操作である。

```
list matches [ == A == ]
```

これは list の中に要素 A があるかどうか調べるものである。ＡＴＯＭにおいては各ＳＯＵで並列に実行できるため簡単に１サイクルで実行できる。

もう少し複雑なマッチング操作に対応するためＡＴＯＭでは次に示す操作を用意している。

（１）正規化
これは葉のアドレスの順番に各要素をＳＯＵに割り当てる操作で、いわゆるソート操作である。

（２）変数要素
これはマッチングリストの中に変数がある場合に使用されるものでオープン変数とクローズ変数がある。

* add_all/add_match_top_address
* add_all/add_match_top_address(I)
* add_all_trap_top_address
* add_all_trap_top_address(I)
* add_match_trap_top_address
* add_match_trap_top_address(I)
* add_top_address
* add_top_address(I)
* append_all_tree
* append_match_tree
* clear_matching_flag
* compare_all_leaf
* compare_match_leaf
* complement_matching_flag
* delete_all_leaf
* delete_compare_all_leaf
* delete_compare_leaf
* delete_compare_value_leaf
* delete_match_leaf
* get_all_leaf
* get_match_leaf
* make_tree
* move_address
* move_all/move_match_address
* move_all/move_match_top_address
* move_all/move_match_top_address(I)
* move_all/move_match_value
* move_top_address
* move_top_address(I)
* move_tree
* move_value
* negate_all/negate_match_top_address
* negate_top_address
* search_all_leaf
* search_match_leaf
* set_eq
* set_matching_flag
* set_mr
* set_tag
* set_wild
* shift_down_address
* shift_down_all/shift_down_match_address
* shift_up_address
* shift_up_all/shift_up_match_address
* subtract_all/subtract_match_top_address
* subtract_all/subtract_match_top_address(I)
* subtract_all_trap_top_address
* subtract_all_trap_top_address(I)
* subtract_match_trap_top_address
* subtract_match_trap_top_address(I)
* subtract_top_address
* subtract_top_address(I)

**Table4.1 List operation in ATOM**

次に示す例は構造体間の比較である。

[ A [ B [ C ]] D ] matches [ A [ B [ C ]] D ]

この場合双方とも正規化されていれば前述の例と同じく並列に１サイクルで比較することができる。どちらか一方、あるいは双方共が正規化されていない場合は葉の要素毎の比較になるが、これは、マイクロプログラムで自動的に行われる。実際のプログラムにおいては一方は予め入力されている場合（辞書等）が多く、正規化されているので、比較データを陽に正規化して比較した方が一般的には高速になる。

次の例は変数を含む比較例である。

[ A [ B ] D ] matches [ ^X [ B ] ?Y ]

一般的には変数の束縛には順序が存在するのでこのマッチング処理をＳＯＵで並列に行うことは出来ない。しかしながら、変数を全てオーブン変数と見なして構造だけのマッチングは並列に実行することができる。一般にパターンマッチングの高速化には、マッチしないものを早く排除して行く機構を持つことが有効であり、この構造のマッチングはその目的に使用することができる。最終的なマッチング操作は構造の一致したリストについて順次、変数の処理を行うことによりなされる。

Ｆｉｇ．５．１は本章の最初の例であるリストの中に特定の要素が含まれているかどうかを調べる操作について、通常のＬＩＳＰマシンとＡＴＯＭを比較したものである。ＡはＬＩＳＰ言語による操作の定義の内容、ＢはＬＩＳＰマシンのマシンレベルの処理ステップ、ＣはＡＴＯＭの処理ステップ、Ｄは両者のマシンステップを比較したグラフである。この図から明かなように、通常のＬＩＳＰマシンにおいては対象リストの長さに比例して処理ステップが増大して行くのに対しＡＴＯＭにおいては一定であり、ＳＯＵの並列処理の効果が出ていることがわかる。

## ６．例外処理

出現リストの長さ、レベル、葉の要素数が前記制限を越えた場合、ＡＴＯＭは例外処理を行う。実際にはオーバフローした方向に新たに表を割り当てて行くことにより処理を行う。このオーバフロー情報はＤＯＵ内のリストの識別情報の中に格納されていて、処理時はマイクロ命令レベルで処理される。したがって命令上はオーバフローリストは存在しない。

```
(defun look (a X)
  (cond ((null X) nil)
        ((eq a (car x)) t)
        (t (look a (cdr X)))))
```

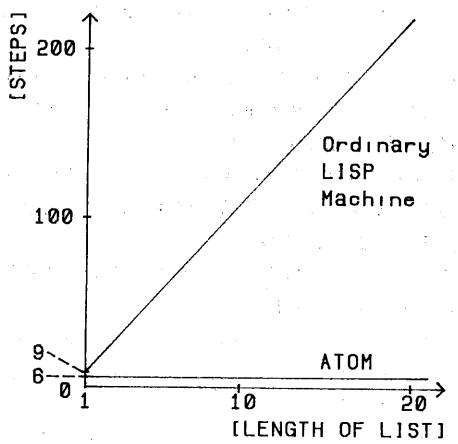
**A. A Sample Described by LISP**

```
 1  PUSH     (BP-1) ;X
 2  BRANCH-TRUE  5
 3  PUSH     NIL
 4  RETURN
 5  PUSH     (BP-0) ;a
 6  PUSH     (BP-1) ;X
 7  CAR      STACK
 8  EQ       STACK
 9  BRANCH-FALSE 12
10  PUSH     T
11  RETURN
12  PUSH     (BP-0) ;a
13  PUSH     (BP-1) ;X
14  CDR      STACK
15  CALL     LOOK
```

**B. Machine Instructions in an Ordinary LISP Machine**

```
1  MOVE     A,(BP-1) ;X
2  MOVE     cR,(BP-2) ;a
3  COMPARE  A
4  BRANCH-MATCH  7
5  MOVE     A,NIL
6  RETURN
7  MOVE     A,T
8  RETURN
```

**C. Machine Instructions in ATOM**




**D. Execution Steps**

**Fig.5.1 Comparison of Execution Steps**

## 7．おわりに

現在、我々が開発を進めている記号処理マシンＡＴＯＭのアーキテクチャについて述べた。

ＡＴＯＭにおいては従来のポインタ表現を前提とした処理系では効率の悪かったパターンマッチング等の処理を改善できる反面、そのリスト表現に起因してリストのセマンティックスがポインタ表現のリストと異なるという問題がある。具体的にはＬＩＳＰ言語における共用リストに起因する副作用や、循環リストの存在、ドットペアの存在等が実現できない。

このことはＡＴＯＭの処理言語として新しいリスト表現に基づいたものが要求されている事を示しており、現在処理系の開発と平行してこの言語の検討を進めている。

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

[4] 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.

[5] 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.

[6] 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.

[7] Harrison, P.G.: Efficient Storage Manage-ment for Functional Languages, Comput. J.,Vol.25, No.2, 1982.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 和田他: リスト構造の内部表現と処理系，情報処理学会記号処理研究会資料，39-5 1986.

[15] Barrett R. et al.: POP-11 A Practical Language for Artificial Intelligence , JOHN WILEY &SONS , 1985 .

[16] Barr A.他 編，田中他 監訳: 人工知能ハンドブック　第Ⅱ巻，共立出版　，1983