解　説

LISP マシンのアーキテクチャ†

井　田　哲　雄††

1. はじめに

LISP の初期の歴史を綴った論文[1]によると，現在使われている LISP の原型ができあがったのは，1958年から1959年にかけてである．この頃にはすでに LISP の特徴である次のような機能が実現され，インタプリタが稼動していたとのことである．

- S-expression (記号表現) によるプログラムおよびデータの一様な表現
- リスト構造による S-expression の実現
- QUOTE という特殊形式を用いた定値記号表現
- ラムダ計算法の表記を用いた関数表現
- 空リストを表す NIL，真偽値を表す T，F の表記
- 条件式を表す (COND (P_1 E_1) (P_2 E_2)‥(P_k E_k)) 表現

などである．

それから20年以上，LISP は存続してきた．生き長らえてきたというより，成長発展して，計算機科学の発展に多大な貢献をしてきた．表現手段を提供して発展を支えてきた人工知能などの応用分野は言うまでもなく，LISP 言語が提供してきた新しいアイデアがある．プログラミング環境，パーソナルワークステーションとしての LISP マシン，インタプリタとコンパイラの共存によって可能となるインクリメンタルプログラミング，リスト処理に関するアルゴリズム，たとえばガーベジコレクション，ポインタ操作，などである．

このうち本稿で議論したいのは，LISP 処理系の基本演算アルゴリズムとハードウェアに実現する上での考え方についてである．各種 LISP マシンについては，本学会誌に安井が広汎にわたるサーベイを発表しており[2]，具体的マシンの記述については，それに譲りたい．

なお本稿では LISP に関する基本的な知識を前提にしている．

2. LISP 処理の構造

LISP 処理系は大きく分けて (1)記号表現の入出力部，(2)記号表現の解釈部，(3)記憶領域の管理部，(4)記号表現を計算機固有の命令列へと翻訳するコンパイラ部よりなる．LISP 言語のセマンティックスは(1)，(3)を所与のものと考え，(2)を明確に規定することによって与えることができる．(4)は実行の効率を向上させることを目的とするもので，セマンティックスを与える上では一応は* 不要である．

2.1 LISP のセマンティックス

次の ＊EVAL がセマンティックスを操作的に与えるのに中心的な役割を演ずる関数である．ここでは議論の対象に COMMON LISP[3] を選んだ．＊EVAL のなかで用いられている関数の意味は文献[3]にある．またここでは説明簡略化のためにエラー処理およびマクロの取扱いを省略した．

```
(1)  (DEFUN ＊EVAL (FORM ENV)
(2)   (IF (CONSTANTP FORM) FORM
(3)    (IF (ATOM FORM)
(4)     (LET ((V (ASSOC FORM ENV)))
(5)          (IF V (CDR V) (SYMBOL-
                VALUE FORM)))
(6)     (LET＊ ((F (CAR FORM))
(7)             (FL (ASSOC F ENV)))
(8)       (APPLY
(9)          (IF FL (CDR FL)
             (SYMBOL-FUNCTION F))
(10)         (IF (SPECIAL-FORM-P F)
(11)             (CDR FORM)
(12)             (MAPCAR
(13)               #' (LAMBDA (X)
```

† Architectural Considerations for LISP Machines by Tetsuo Ida (Institute of Physical and Chemical Research).

†† 理化学研究所

＊ LISP 処理系によっては変数の取扱いにインタプリタとコンパイラとで相違があるので，これらシステムのセマンティックスを与えるにはコンパイラによる変数の扱いを追加する必要がある．

(＊EVAL X ENV))

(14) (CDR FORM)))))))

COMMON LISP で用いられる変数には静的変数と特別変数がある．前者は名前のスコープがプログラムテキスト上の位置できまるもの (lexical) である．後者は名前のスコープが動的にきまる変数で，前者との対応で言えば動的変数と言ったほうが適切であるが，歴史的な事情で普通は特別変数と言われる．この ＊EVAL では静的変数は ENV というリストで管理している．ここでは名前 Ai と名前が指示する値 Vi との対を次のような連想リストで保持するものとする．

$((A_1.V_1)(A_2.V_2)\cdots(A_n.V_n))$

FORM は ＊EVAL に渡される評価すべき記号表現である．＊EVAL の動作は自然言語で次のように言い表すことができる．

(1) もし FORM が定値ならば，その FORM をそのまま答えとせよ．

(2) もし FORM が記号ならば，記号名を変数とみなし ENV に探す．ENV に名前が存在すれば，対応する値を答えとせよ．

さもなくば，その名前を広域的 (global) なものとみなし，名前の指示する値を答えとせよ．

(3) さもなくば，FORM は，リストで表現された記号表現である．これを $(f\ V_1\ V_2\cdots V_n)$ としよう．

(4) もし，f が特殊形式 (special form) であるならば f の意味を定義している関数を得，それを引数リスト $(V_1V_2\cdots V_n)$ に作用 (apply) させよ．

(5) さもなくば，$V_1V_2\cdots V_n$ を評価してえられる引数リスト $(V_1'V_2'\cdots V_n')$，ただし V_i' は V_i を評価した結果に，f の指示する関数 f' を作用させよ．

例 ENV=((N. 100))

FORM=(IF (ODDP N) 1 0)

(＊EVAL FORM ENV)=>0

この FORM を評価する過程をトレースすると次のようになる．

IF → CONSTANTP → IF → ATOM →
LET＊ → CAR → ASSOC → IF →
SYMBOL-FUNCTION → IF → SPECIAL-F-
ORM-P → CDR → APPLY → (IF の評価)

ここで，IF は特殊形式である．特殊形式とは，実引数の評価を ＊EVAL に委ねずに，自分自身の定義の中に評価機構をもっているものである．IF の場合，第一引数を評価してから，その結果にしたがって第二または，第三引数を評価する必要があるので，(12)～(14)の MAPCAR を経由して評価することはできない．

なおここで，すべての関数の実引数の評価が，関数の作用に先立つことに注意してほしい．このような評価を applicative order の評価と呼んでいる．LISP では applicative order を用いているが，このほかにも，実引数の評価の順序は考えられる．たとえば，評価せずに実引数リストを作り，各実引数の評価を関数の側にまかせてしまう方法もある[4),5)]．

＊EVAL の処理はこのままでは，あまりに逐次的である．たとえば，(8)の APPLY に到達するのに，CONSTANTP, ATOM, SPECIAL-FORM-P といった述語関数による FORM のチェックを逐次的に行っていかなくてはならない．もし，FORM に目印がついていて，しかも，ハードウェアで目印をチェックして，ただちに，各目印に対応する処理を行うことができれば，LISP を高速化することができるのではないだろうか．

LISP マシン製作の動機のひとつは，＊EVAL にみられるような LISP 処理系の核を始めとして，各所に存在する不必要な逐次性を除去しようというものである．除去される逐次性の度合はこの例に見るように，大きいものではない．言い替えると，潜在的並列度は低い．しかしながら，このような逐次性をたんねんに除去し，価格性能比をあげようというのが，現在までに作られている LISP マシンの基本的設計理念である．

もう一つの例をあげよう．COMMON LISP では加算関数＋は総称関数 (Generic Function) である．(＋ 1 1.5 1.0) も (＋1/4 13/4) も正しい答えである 3.5 (あるいは 7/2) を計算しなくてはならない．したがって，インタプリタがよぶ関数＋は実引数の型をチェックし，必要ならば型変換を行い，型ごとに異なる加算のルーチンに処理を委ねなければならない．ここにも，＊EVAL の処理に類似した除去可能な逐次性が存在する．

2.2 コンパイラの役割

LISP コンパイラの重要な仕事のひとつは上の例にみられる逐次性の除去にある．

たとえば，

- 関数呼出しを，＊EVAL, APPLY を経由させずに，直接行うこと
- 入力となるデータ型をあらかじめチェックして，

データ型にあった処理コードを生成すること

● データ型が確定しないときには，もっとも使用頻度の高いデータ型から，最初にチェックするような処理コードを生成すること

である．

このほかに，静的変数の実行時の逐次的探索（2.1 では ASSOC で行っている）をなくし，直接アドレスできるよう静的変数に記憶語を割当てることもコンパイラの重要な仕事である．

したがって，LISP に適したアーキテクチャを考えるうえでは，コンパイラのできうる仕事の限界を見極め，投下ハードウェア資源とソフトウェアとの最適比率を検討する必要がある．

3. 新しいアーキテクチャの諸概念と LISP マシンの解

3.1 パーソナルマシン

アーキテクチャを与えるということは，ある尺度でもって既存の部品（論理素子，ソフトウェアなど）を選択し，システムを構成することを試みることである．この尺度は使用可能な技術によって変動する．最初の LISP マシンの構想が練られていた 1970 年の初頭には，LISP コミュニティの直面する最大の問題は記憶領域の不足であった．LISP のおもな応用分野である人工知能の処理がよりよい対話性と大容量の記憶を必要としていたのである．仮想記憶空間の不足の解消と，広い仮想空間の効率を保証する実記憶の確保が，当時の技術，将来の予測技術の水準を鑑みて，十分に実現可能なものであったにもかかわらず，当時の既存アーキテクチャではそれが不可能であった．MIT で開発された最初の LISP マシンの哲学は前節で述べた逐次性の解消はマイクロプログラミングでできる程度に抑え，記憶空間を十分広くとったパーソナルマシンを作るということであった[6]．まだマイクロコンピュータチップができて間もなく，しかも我が国においては，TSS システムでさえまだ十分に普及していない時期である．いまでこそ当り前になってしまったパーソナルマシンの概念は，当時は斬新なものであった．

3.2 命令体系のチューニングと並列処理の可能性

高級言語の基本処理を命令語として，備えようという考え方がある．たとえば，LISP の CAR，CDR，CONS などを基本命令語としようとするものである．この議論の主旨は，

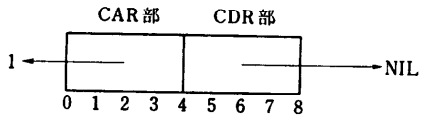

図-1 リストセル (1. NIL)≡(1) の表現法
（1語4バイトで，1リストセルに2語を用いている．バイトアドレスを仮定した．）

（1） 生成コードがコンパクトになること

（2） 高速に実行されうること

である．

（1）の点は窮極的には処理の高速化に結びつくのであるが，実記憶に裏づけされた広い仮想空間が安価にできるようになった今日では，従来ほどには重要でなくなっている．

（2）の点に関して，CDR を例にとって高速化の可能性を考えてみよう．議論の枠組として，ここでは IBM 360 アーキテクチャを使う．リストセルは連続した2語（8バイト）で実現し，図-1 のように CAR，CDR 部をとる．ポインタ P のさす先のビットによって，P がリストをさしているかアトムをさしているかが判別できるものとする*．すると，エラーチェックをきちんと行う CDR の操作は次のようにコーディングできる**．

```
TM LISTBIT, 0 (P)      リストかアトムか?
BZ ERROR               リストでなければ
                       エラー
L P, 4 (, P)           CDR 部を参照せよ
NEXT OP                次の命令
```

この一連の命令列を文字どうりの逐次的な実行であると解釈すれば，ここには並列性がないようにみえる．

CDR の操作を次のような基本操作に分解してみよう．

（1） TM 命令の実効アドレス a を計算する．

（2） a 番地の記憶語の読み出し要求をだす．

（3） a 番地の内容 Ma が読み出されたかどうかをチェックする．

読み出されていれば，次へとぶ．さもなければ，このステップを繰り返す．

（4） Ma にアトムをしめすビットがたっていれ

* ポインタ P にリストをさすかアトムをさすかを示すタグをもたすように LISP 処理系を設計することもできる．このときも，同様の考察をすることができる．

** COMMON LISP では (CDR NIL) は NIL なので，この前に

```
CR P, NIL          NIL かどうか?
BE NEXT            そうならば OK
```

といったコードがはいる．

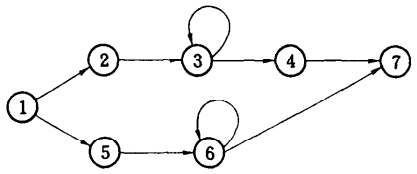

図-2 CDR 実行における基本操作のデータ依存関係グラフ

ば，エラー処理のステップへとぶ．

（5） a+4 番地の内容 Ma+4 の読み出し要求をだす．

（6） a+4 番地の内容 Ma+4 が読み出されたかをチェックする．読み出されていれば，次へとぶ．さもなくばこのステップを繰り返す．

（7） Ma+4 を答えとせよ．

この一連の命令ステップはデータの依存関係から**図-2** のグラフで表される．この図より，(5)〜(6)と(2)〜(4)は並列に実行できることがわかる．さらに，記憶からのデータの読み出し幅を8バイトとするならば，(5)〜(6)のステップは必要なく，Ma+4 をPへと転送し，Ma のビットテストの結果によって，Ma+4 をPにセットするか，エラー処理を行うかを決めればよい．正常の流れは Ma+4 をPにセットするようにハードウェアを設計しておき，エラー処理は「ゆっくりと」行うようにすればよい．

したがって，CDR という複雑な命令も，記憶の読み出し，ビットチェック，レジスタへのセットの制御といった操作に分解され，多少の並列性をとり出せる，つまりプログラムによる逐次処理より高速に実行されうることがわかる．このほかにも，LISP に限らず，よく使われる処理として，スタックのポップ，プッシュがあげられる．ポインタの増減，境界値のチェック，スタックからの読み出しあるいは書込みは，適切な同期をとって並列に行えることは明らかであろう．

3.3 VLSI 化と RISC (Reduced Instruction Set Computer)

CAR，CDR を前節で述べた意味でハードウェア化するのは一つの選択である．しかし，ビット操作や記憶の読み書きのほうが，より基本的操作であるから，この基本操作を高速化し，それを組合わせてできる操作はソフトウェアに任せようとする考え方もある．もし CAR，CDR などの高級命令を実現するのに，余計な回路をつけ，そのために演算の基本サイクルを遅くしてしまっては，意味がなくなってしまう．特に VLSI のように，回路の占有面積の増大が基本クロックサイクルの増大へとただちにつながるときはそうである．命令の実現頻度を 10% とするとき，その命令を付加することによって，基本サイクルが 10% のびてしまうならば，その命令を約10倍高速化しないとペイしないことになる．

それならば，命令は必要最少限の低レベルのものに絞り，基本サイクルを短縮し，そのかわりに命令の先取り機構をしっかり作ろうという考え方もできる．RISC[7] というのが，この発想である．

複雑な命令を実行する手法として，マイクロプログラミングが使われる．しかし，マイクロプログラムの記憶と命令コードを格納する記憶（普通はキャッシュメモリ）とが同じ素子で実現されうるようになってくると，マイクロプログラミングによる実現がかならずしも，経済性の高い方法ではなくなってくる．したがって CDR の逐次的実行を単純にマイクロプログラムで逐次的に実現するのであるならば，ハードウェア化の利点は少ないと RISC では考える．

さきほどの CDR を RISC 風に解釈しなおしてみよう．TM をより基本的と思われる記憶の読み出し操作と，ビット演算に分解する．

```
L W, 0 (, P)        タグを読み出す
N W, LISTBIT        チェックすべきビットを
                    マスクする
BZ ERROR
L P, 4 (, P)        CDR 部をとる
OP                  次の命令
```

こうすると，CDR の命令に余計なマシンサイクルを要することになる．

ここで，分岐命令に着目する．このような分岐命令は命令のフェッチの制御をしているだけであるから，ロード/算術，ビット演算には不要である．ここではエラー処理にさしつかえない限り L や次の OP を演算器に送り出してしまい，N の結果をみる．0でなければ，問題はないので，そのまま命令が流れる．エラーの時には，命令の流れが中断するが，その頻度は低いので，全体の効率の著しい低下にはつながらないはずである．CAR についても同様で，上記命令語列の 4 番目の命令語を L P, 0 (, P) とすればよい．したがって，RISC では CAR，CDR という特別な命令はいらないことになる．

以上は 360 アーキテクチャをベースに議論したが，

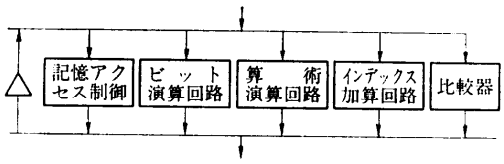

図-3 仮想的 RISC 用 CPU

一般には，並列に動作する演算器に対して，別個の命令を用意するという考え方がよいであろう．たとえば，**図-3** のような仮想的な CPU を考える．こうすると，命令制御部は各演算器に可能な限り命令を流しこんでしまうことができる．データの依存関係により逐次的に処理しなくてはならないものは，あらかじめ，わかるはずであるから，その同期はプログラムのコードでとれるはずである．RISC の場合，演算器にデータを流す順序はプログラムの制御下にある．

一般に，特定の RISC マシンにあわせて，データの依存関係をプログラマが指定するのはたいへんである．したがって，高級言語，たとえば LISP，C，でプログラムを書き，コンパイラが RISC 用最適コードを生成するようにする．

3.4 フォン・ノイマン・ボトルネックの解消

フォン・ノイマン・ボトルネックという言葉は J. バッカスがチューリング賞論文[8]で用いて以来，よく使われる．新しいアーキテクチャを考えるうえでフォン・ノイマン・ボトルネックの解消，あるいは非ノイマン化というのが一つの努力目的となっている．この語の含蓄するところはいろいろあるが，次の点に集約できよう．

本来並列性をもつアルゴリズムを逐次逐語処理を基本とするアーキテクチャで実行させるために，逐次逐語処理化していることの指摘と，逐次逐語処理に過度に依存することに対する警鐘と受け取ることができよう．

このような傾向は，ある特定の言語で記述したプログラムに限定されるものでもない．LISP においても，FORTRAN で書いた場合と同じく存在する．フォン・ノイマン・ボトルネック解消の提言は新しいプログラミングスタイルとそれを真に可能とする新しいアーキテクチャ研究のすすめと考えてよいであろう．LISP において，この新しいプログラミングの考え方が定着しているとは現時点ではいえないが，いくつかその影響がみられる．たとえば，COMMON LISP の列 (sequence) を扱う関数 (3)，pp. 247～261) は少なくともプログラマ側でフォン・ノイマン・ボトルネックを解消するのに役立つであろう．2.1 で従来 EVLIS と呼ばれている実引数を評価して実引数リストを作る関数を MAPCAR で書いたのはプログラマ側からのフォン・ノイマン・ボトルネックの解消の一例である．

アーキテクチャに関しては，LISP マシンよりも，より広い枠組で，プログラミングを関数的にとらえようとする関数型プログラムのアーキテクチャの研究にバッカスの提言の影響をみることができる[9),10)]．

4. LISP 処理系における基本演算

4.1 タグ付記憶とタグ処理

ここで言うタグはデータ型を識別するために設けられるデータに付加されたビット，および記憶語の属性を示すビットをさす．ここでは前者をデータタグ，後者を語タグとよぶことにする．

タグ付記憶の提案は古く，一部汎用商用機でも採用されている[11)]．商用化された LISP マシン[12)-14)]も，研究用 LISP マシン[15)]もタグ付記憶方式を踏襲している．第五世代プロジェクトの一環として開発された PROLOG マシンもタグ付記憶方式をとる[16)]．ハードウェアの一部としてタグ付記憶を実装するかタグをソフトウェアで処理するか，議論の余地はあろうが，次の理由からタグ自体は LISP の実現に不可欠である．

- インタプリタは実行時に型チェックを行う必要がある．
- 言語の仕様が型宣言なしでもよいように作られており，すべての型をコンパイル時に決定することはできない．
- ゴミ集めのために，記憶語が活性状態にあるか否かを示すビットが必要である．

データタグには自己規定タグと参照先規定タグがある．前者はタグ付けされたデータそのものの型（あるいは性質）を規定するものである．単精度整数，単精度浮動小数のようにデータが一語で納まる場合には，自己規定タグが用いられる．構造体データ，たとえば文字列，多倍長数，を示すには参照先規定タグが用いられる．タグの付け方も，分離タグビットによる方法とアドレス分割による論理的タグ付がある．

一般には，分離ビットをとって，自己規定，参照先規定タグの双方を用いるのが，実行時の効率はよいが分離ビットが多ければそれだけ記憶資源を必要とすることになる．

タグに関する処理は3.2でみたように，本来行うべき処理とタグチェックを並行して動作させ，最後に同期をとる方法がとられる．＋という関数を例にとれば，出現頻度の高い単精度整数どうしの演算と，単精度浮動小数どうしの演算を優先して実行させ，同時に被演算数の型を型チェック回路でチェックする．型チェック回路はビット演算回路と簡単なデコーダですむ．タイプチェックの結果，データ型エラーが生じた場合，命令で指定する番地にジャンプするか，あるいは割込みエラーをおこすようにすればよい．

4.2 オペランド指定方式

3.で議論の結果に基づいて，言語の基本処理に合致した命令語を設けたにせよ，RISC を考慮するにせよ，次の問題になるのは，その命令のオペランドをどのように指定して，フェッチしてくるかということである．これは LISP マシンに限らず，いわゆるコントロールフロー型の計算機一般にあてはまる議論である．

オペランドの指定方式には大きくわけて，汎用レジスタファイル方式とスタック方式がある．汎用レジスタ方式はあらかじめ決められた直接アドレス可能なレジスタを主たるオペランドとするもので，IBM 360 アーキテクチャがその代表的なものである．

一方，スタック方式は，スタックとして組織化された記憶をフレームという単位に分割し，フレーム内相対アドレスによってオペランドを指定するものである．両者のいずれがよいかという議論は昔から行われている．議論のポイントは次のようなものである．

- スタックは再帰呼び出しができる言語のインプリメンテーションに不可欠であり，オペランドもスタックに割当てたほうが都合がよい．
- 一方，スタックは主記憶に割つけられるために，アクセスに時間がかかる．
- スタックアクセスを効率良くするにはレジスタ方式よりも多くのハードウェアを必要とする．
- VLSI 化を考えると，レジスタファイル方式のほうがインプリメントが簡単である．
- キャシュメモリを備えた最近の計算機ではスタックもほかの記憶空間も同様に高速にアクセスすることができるので，スタック処理に伴うポインタ操作を基本操作としてハードウェアでサポートすれば，スタックは不要である．つまり，レジスタファイル方式でスタックを効率良く実現できる．

また，上の議論と直交して，オペランドをいくつにするかということも問題になる．2つあるいは3つが妥当であると考えられるが，このうちどちらを選ぶかは，ハードウェア資源の割当ての考えかたにかかっている．

4.3 領域管理とリスト処理

LISP のシステムはいくつかの概念的には分離した領域をもつ．リスト領域，フルワード領域，ハッシュ表領域，スタック領域，コード領域などである．リスト領域は以下に述べるリストを構築するための領域である．フルワード領域は自己規定タグとタグが示すデータが格納される領域である．コード領域はコンパイルされた関数のコードが格納される領域である．スタック領域，ハッシュ表領域は 4.4, 4.5 でふれる．

リストは図-1 のように通常は表現する．CAR, CDR がこの構造へのアクセス関数，CONS が構造生成関数である．

CONS は使用可能な領域（これを自由領域という）から一つのセルを得るものである．

(CONS X Y) のアルゴリズムは次のようになる．

（1） 自由領域から一つのセルを得よ．

もし自由領域が消費されてしまっていれば，ゴミ集めのプログラムを実行し，しかるのちに，自由領域から一つのセルを得よ．これによっても，自由領域を得ることができなければ，もはや実行継続不能である．

（2） 得られたセルPの CAR 部にXを，CDR 部にYを格納し，答えとしてPのアドレスを返せ．

この CONS のアルゴリズムは自由領域の構造には依存しないできまる．自由領域は，したがって，単純にポインタで接続（方法A）しても，連続領域となるよう空領域を作ってもよい．連続領域を作るときに活性セルをスライドして詰める方法（方法B）と，自由領域の他方の端から活性セルを移動して詰める方法（方法C）とがある．表-1 に方法A，B，Cの長短得失をまとめた．

CONS の場合記憶への書込みがあるので，CAR, CDR にみらるような並列性はとりだせないが，自由領域の境界のチェックとポインタ操作とは重複させることができる．

リストセルの表現方法には，CDR コーディングという手法を用いた線型リスト表現法が知られている（図-4 参照）．線型リストにするために，記憶語に論理的な接続関係を示す2ビットの語タグが必要となる．この方法では，セルのアクセスごとに，タグのチェックが必要になる，リストのつなぎかえに余計な

表-1 逐次処理型ゴミ集め法の性能比較

	リスト構造の局所性	CON の実行効率	自由領域の作成法
方法A	リスト生成時の局所性が保存．局所性高くない．	B，C より記憶アクセスが多い	最も簡単
方法B	局所性は最も高い．リロケーションにより，局所性は，高くなる．	高 速	リロケーションに時間がかかる．リロケーションのハードウェアが望まれる[15]
方法C	生成時の局所性は必ずしも保存されない．通常は局所性はBに次いで高い．	高 速	リロケーションはBよりも容易．特殊ハードウェア不要

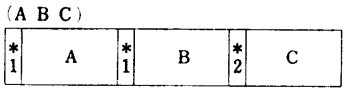

図-4 CDR コーディングによるリストの表現
*1 は次のリスト要素が隣接しているので，CDR を省略していることを示す語タグ
*2 は CDR が NIL であることを示す語タグ
この他に，図-1 のような通常リストセルを示している語タグが必要

処理が必要である，などの難点がある一方，記憶の節約がはかれるという利点もある．上記ふたつの難点は専用ハードウェアを構築することによって容易に取り除くことができる．言い替えれば，ハードウェア向き方法ということができる．CDR コーディングはいくつかの LISP マシンで実現されている[12)~15)]．CDR コーディングのアルゴリズムの紹介は文献 17)，18) にある．リストが完全に線型のとき，セルのアクセスにアドレス演算を用いて行える．このようなアクセス特性を利用して，ベクトル（一次元配列）やスタックを効率よくシミュレートすることもできる．

4.4 記号処理

LISP のもう一つの重要な特徴は記号を実行時においても記号のまま取り扱える点にある．これは，FORTRAN, PASCAL のようなコンパイルを基本とする言語にはみられない特徴である．これらの言語では記号は変数名として導入され，しかもコンパイルによって消去されてしまうのに対し，LISP の記号は変数名としても，また記号それ自体としても存在する．したがって，変数として値を保持させることもできるし，記号に属性を与えることもできる．

2.1 でLISP には2種類の変数が存在すると述べたが，これは変数が記号として実現されていることに由来する．一方において，このような実現法は変数のセマンティックスの混乱の原因ともなってきた．

まず，変数の実現法について考えてみよう．ハードウェア/ソフトウェアへの負担は特別変数のほうが重いので，ここでは特別変数に要求されるハードウェア機能について検討する．

次の例を考える．

```
(1)  (LET ((*EVALHOOK* 'USER-HOOK))
(2)       (DECLARE (SPECIAL
                    *EVALHOOK*))
(3)       (*EVAL EXPR NIL))
```

ここで，*EVALHOOK* は特別変数と宣言されている．したがって，*EVALHOOK* の値は(1)で USER-HOOK にセットされたのち，(3)の評価で *EVALHOOK* を参照する関数にみえることになる．しかし，この LET 形式の実行が終ったときに，*EVALHOOK* の値は LET 実行直前の値に復元されなければならない．この LET 実行における，変数 *EVALHOOK* へのアクセス操作は次のように実現する．

まず，制御スタックという特別変数の値格納用スタックを用意する．このスタックは **4.2** で述べたオペランドスタックで兼用することができるが，以下に述べる非正規制御を行うには，別個に用意したほうが都合がよい．制御スタックへの参照操作は CPOP, CPUSH とする．

記号は通常，記号名の格納領域に加えて，特別変数の変数値を格納する語，属性リストを格納する語，関数定義を格納する語よりなるアトム構造体で表現される．CPUSH P（ここでPは *EVALHOOK* のアトム構造体へのポインタである）は *EVALHOOK* のアトム構造体のアドレスと *EVALHOOK* の値を制御スタックにプッシュする．CPOP はアトム構造体の値を復元する．CPUSH, CPOP の命令は 3.2 で述べた理由によってハードウェアによる実現の価値がある．

静的変数は，解釈実行時には，変数名を示す記号と変数値の対からなる連想リストによって管理される．(2.1 参照) しかし，コンパイルすることによって，変数名を除き，オペランドスタックに値格納用記憶語を割付けることができる．したがって，これまでに述べた以上のハードウェアは必要ない．

LISP 特有の記号処理に属性リスト処理がある．属性リストの参照は CAR, CDR の組合せでできるの

で特別のハードウェアは必要としない．しかし，属性リストが長くなったときには，次に述べるハッシングのほうが能率がよい．

4.5 ハッシング

LISP では記号を記号のまま取扱うことができると指摘した．この考えをさらにすすめて，数だけではなく記号をもインデックスとする表を作り，データを管理したらどうであろう．

つまり，N 個のエントリをもち0から $N-1$ のインデックスによりアクセスされるベクトル $V(N)$ の代りに，$N'=N*(1+a)$，$a>0$ のエントリをもったハッシュ表を作る．（ハッシングの性能を保証するため，$1>a>0.25$ 程度は必要．）そして，記号名をインデックスとして表をアクセスする．見方を変えれば，記号名を与えてその記号に付随する値（属性値）を得ることができる．属性値を必要としないならば，ハッシュ表は記号の存在性をチェックする道具立てと考えることができる．ハードウェアハッシングのアルゴリズムについては文献 19)，20) を参照されたい．ハッシングの高速化には，2. で述べた並列化が期待できると同時に，表読み出しの並列化が期待できる．ハードウェアもすでに作られている[15),21)]．

5. 新しい記号処理

5.1 PROLOG

PROLOG の論理型言語としての特徴はさておいて，ここではインプリメンテーション上の，従来の LISP にない特徴について考えてみよう．それらは，単一化（ユニフィケーション）と非決定的処理である．非決定的処理は逐次型計算機においては，バックトラックによって支援される．バックトラックの主たる仕事は実行環境の復元であるが，これは前述の制御スタックを用いることによって，行うことができる．LISP との違いは，値の変化する変数すべてを，バックトラックに備えて制御スタックに CPUSH しておかなければならない点にある．

単一化は本質的に逐次的なところがあるので，2. で述べた並列化がとるべき方策であろう．単一化は前処理（コンパイル）によって，多くが LISP の関数呼び出しにおける変数束縛と同等の処理と，代入処理に分解できる．したがって，単一化の高速化は最適化コンパイラの製作と結びつけて考える必要があろう．また，引数の対応に値の代入の方向性に関する情報を付加することによって，コンパイラによる最適化がより容易になるであろう．このアプローチを取る言語に PARLOG がある[22)]．より一層の高速化のためには CAM (Content Addressable Memory) を用いる方法の提案がある[23)]．従来型の計算機にすぐにインプリメントできるものではないが，今後の研究の発展が期待される．

5.2 非正規制御

ここで，非正規制御とは THROW, CATCH によって実現される制御機構をさす．LISP 1.5 では非正規制御をひきおこす関数として ERRORSET, ERROR があったが，これら機能を合せもつ汎用制御構造として，CATCH, THROW が最近の LISP では取りいれられた．非正規制御はエラー処理に限定せずに，デバッガやトレーサのように制御をプログラマが自由に扱う必要があるプログラムの作成に用いることができる．

CATCH, THROW のハードウェアに対する要求は，制御スタックへの使用される特別変数の値の保存と CATCH のレベルまでの制御スタックのポップアップ (CPOP) である．これらの機能は 4.4 で述べてきた CPUSH, CPOP で十分実現できる．

6. おわりに

LISP の実現は，長い間，多くの研究者の研究テーマとなってきた．米国においても，大学の計算機科学科（たとえば，MIT，カーネギーメロン大学）や計算機を主たるビジネスとしない企業の研究所において長らく LISP 実現の研究が行われてきた．我が国においても，大学，研究所が LISP 研究の主な担い手であった．（たとえば，安井のサーベイを参照されたい）このことは，LISP の発展に長所と短所とを同時にもたらした．長所は LISP の仕様の未成熟なままでの固定化を防ぎ，たえず新たなアイデアの注入を可能としたこと，同時に，短所として，プロダクションレベルの最適化コンパイラの作成が遅れ，LISP は遅いといった神話を作り出したことである．

しかし，人工知能といわれる分野の研究が実用の域に達すると考えられるようになるとともに，企業による LISP のサポートが本格化してきている．MIT の CONS マシンの商品化が先鞭[12),13)]をつけ，ゼロックス社[19)]や富士通[24)]からも LISP マシンが商品として発売されるに至った．一方，COMMON LISP にみられるように LISP の仕様が，コンパイルすれば，汎用機でも早いコードが生成できるよう整理されつつあ

る．LISP は遅く非実用的であるという誤った認識は早晩解消されるようになると思われるが，次の段階として，LISP の応用における高速化が社会的にみて，切実なものとなり，「LISP をより高速にするには？」といったことが，再び問われることが予想されよう．

その時には，本稿で述べたアーキテクチャ上の検討に加え，マルチプロセッサによる並列処理，超高速素子による実現，並列処理向きリスト処理アルゴリズムの開発といった高速化への諸課題が総合的に検討されることが必要となろう．

参 考 文 献

1) Stoyan, H.: Early LISP History, Conference Record of LISP and Functional Programming, pp. 299-310 (1984).
2) 安井 裕：LISP マシン，情報処理，Vol. 23, No, 8, pp. 757-772 (1982).
3) Steele Jr, G.L. et al.: Common LISP The language, Digital Press (1984).
4) Henderson, P. and Morris, J.H.: A Lazy Evaluator, Conference Record of 3rd POPL (1976).
5) Friedman, D.P. and Wise, D.S.: CONS Should not Evaluate its Arguments, In Automata, Languages and Programming: Third International Colloquium, pp. 257-284 (1976).
6) Knight, T.: CONS, MIT AL Working Paper 80 (1974).
7) Patterson, D.A.: Reduced Instruction Set Computer, CACM, Vol. 28, No. 1, pp. 8-21 (1985).
8) Backus, J.: Can Programming be Liberated from von Neumann Style? A Functional Style and its Algebra of Programs, CACM, Vol. 21, No. 8, pp. 613-641 (1978).
9) Mago, G.: A Network of Microprocessors to Execute Reduction Languages, International Journal of Computer and Information Sciences, Vol. 8, No. 2, pp. 349-385 (1979), Vol. 3, No. 6, pp. 257-266 (1979).
10) Darlington, J. and Reeve, M.: ALICE A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages, Proc. 1981 ACM Conference on Functional Programming Languages and Computer Architecture, pp. 65-76 (1981).
11) Organick, E.I.: Computer System Organization The B5700/B6700 Series, Academic Press (1973).
12) Symbolics 3600 データシート．
13) LMI Lambda データシート．
14) 1100 Scientific Information Processor, Xerox データシート．
15) Goto, E. et al.: Design of a Lisp Machine-FLATS, Conference Record of LISP and Functional Programming, pp. 201-208 (1982).
16) Uchida, S. and Yokoi, T.: Sequential Inference Machine: SIM Progress Report, Proc. FGCS '84 pp. 37-58 (1984).
17) 黒川利明，井田哲雄：リスト処理とアーキテクチャ，情報処理，Vol. 23, No. 8, pp. 712-718 (1982).
18) Ida, T. and Itano, K.: Associative Descriptor Scheme-for the Exploitation of Address Arithmetic in Lisp JIP, Vol. 4, No. 3, pp. 147-151 (1981).
19) Ida, T. and Goto, E.: Performance of a Parallel Hash Hardware with Key Deletion, Proc. IFIP Congress, pp. 643-647 (1977).
20) Ida, T. and Goto, E.: Analysis of Parallel Hashing Algorithms with Key Deletion JIP, Vol. 1. No. 1, pp. 25-32 (1978).
21) Ida, T.: Hashing Hardware and its Application to Symbol Manipulation, Proc. International Workshop on High-level Language Computer Architecture, pp. 99-107 (1980).
22) Clark, K. and Gregory, S.: PARLOG Prallel Programming in Logic, Research Report, 84/4 (Apr. 1984).
23) 安浦寛人，大久保雅且，矢島脩三：論理型言語における単一化操作のアルゴリズム，京都大学工学部 (1984).
24) Hayashi, H., Hattori, A. and Aimoto, H.: LISP Machine "ALPHA", FUJITSU Scientific and Technical Journal, Vol. 20, No. 2, pp. 219-234 (1984).

(昭和 60 年 4 月 19 日受付)