解　説

LISP——その発展の方向†

寺　島　元　章††

1. はじめに

1950年代の終り頃，J. McCarthy を中心とした米国マサチューセッツ工科大学（MIT）の研究グループによって設計されたリスト処理用プログラミング言語 LISP は，リストの処理を基本とする記号処理や数式処理，人工知能などの研究に必要な道具として，また，数学的に定式化された言語としての厳密性や簡潔性を有し，拡張性にも富むことなどから，人工知能の研究者やそれに魅せられた熱狂的な愛好者の間で受け継がれ，育てられてきた．その一方で標準化に対する取り組みがこれまで一切行われなかったことから，数多くの方言や処理系の設計開発が行われ，○○ LISP とか LISP △△と言う名称が付けられた互に言語仕様を異にする「LISP」が巷にあふれることとなった．このような百花繚乱たる LISP の世界にも新しい流れが生れつつある．その1つは LISP マシンに代表される処理の高速化とパーソナル化であり，もう1つは Common LISP に代表される言語仕様の標準化への試みである．前者について言えば，半導体集積回路技術の進歩やマイクロプログラム化の可能な計算機の登場によりそれらが実現できるハードウェアの環境が整ったということであり，後者については，各 LISP 間の言語仕様の相違がプログラムの相互乗入れやソフトウェアの流通を阻害していることに対してそれを打開しようとする共通の認識が生れたことである．この2つの流れは一見異なる側面をもつように思われるが，これらは LISP が抱えた問題（その多くは LISP 誕生時からの宿命のようなもので，これらについては 3. 以降に述べる）に対するそれぞれの解答（対処法）でもある．なお，LISP マシンと Common LISP については本特集中にも解説があるので，詳細はそちらに委ねることにして，本稿では，LISP の基本的な特徴のなかで言語仕様や処理方式など「LISP」の変遷とともに変化した部分を主体にその背景をも含めて概説し，これまでの LISP の進展の方向を明らかにする．そして，個人的な好みに片寄った記述になるかも知れないが，LISP の置かれた現状とその問題点とを指摘し，その対処法を含めて LISP の今後の発展の方向について述べてみたい．

2. 歴史的経過

2.1 LISP の誕生

LISP が登場する以前のリスト処理言語としては，IPL[1),2)] と J. McCarthy もその開発に関与したと言われる FLPL[3)] とがあるが，この2つの言語はすでに歴史的な価値しかないと思われるので，紙面の制約上，ここではそれらについての説明を省くことにする．

J. McCarthy が今日の LISP の原型である LISP 1[4)] を考案したのは1958年のことである．それには，

（1） リストというデータ構造とそれを操作する基本演算（関数）の定義．

（2） リストを含む LISP データを表記するための S式の定義．

（3） ラムダ計算法[5)] に基づいた関数型の評価方式の導入，すなわち，ラムダ記法による関数の表記法とその評価関数 EVAL の存在．

（4） 条件式とそれを用いる帰納関数の定義．

（5） 関数引数の機能．

など，今日の LISP に共通する特徴を見い出すことができる．同年の秋には，IBM 704 計算機へのインプリメンテーションが開始された．その後，LISP 1 に（1）属性リスト（2）副作用をもつ関数（3）自由変数（大域変数）（4）数値（5）プログ形式（6）翻訳系などが追加されたものが LISP 1.5[6)] として1962年に公表された．LISP 1.5 は前述の IBM 704 計算機の次位機種である7090計算機（これらはともに1語36ビットであり，1語を分割して CAR 部と CDR 部に充てた）にインプリメントされたが，その処理方式は一括処理

† LISP —Its History and Future by Motoaki TERASHIMA (Department of Computer Science, The University of Electro-communications).

†† 電気通信大学計算機科学科

(バッチ) 方式であった．今日考えると，LISP に似合わない方式であったが，L. P. Deutsch が対話方式による最初の LISP 処理系[7] を PDP-1 計算機上に作成したのは翌年 1963 年のことである．LISP 1.5 のインプリメンテーションで生じた問題点を打開するために LISP 2 のプロジェクトが進められたが，参加機関の協力関係や経費，それに対象計算機の問題などが表面化して，このプロジェクトは立消えになってしまった．その後，LISP は PDP-1 の後継機種である PDP-6 や PDP-10 上に受け継がれることとなった．この間の事情については文献[8] に詳しい記述がある．

2.2 2つの LISP

DEC 社の PDP-6 計算機やその次位機種である PDP-10 は，半語命令やスタック命令の導入と時分割処理方式の採用などに見られるように LISP の作成に適したアーキテクチャを有している．事実，半語命令やスタック命令は LISP 開発での要求として出され，それを DEC が採用したという経緯がある．その LISP 開発とは，BBN-LISP[9] (後に Xerox PARC に引き継がれ，InterLISP[10] と名称が変わった) と MacLISP[11),12] のことである．この2つの LISP は，その言語仕様やエディタ・デバッガなどのプログラミング・ツールの機能にかなりの違いがあるものの，"良き" ライバルとしてその後の LISP 界の二大潮流を形成することとなった．

BBN-LISP は今やその仕様書が残されるだけであるが，InterLISPは PDP-10 (DEC-10 とも呼ばれているもので，DEC-20 との互換性がある) の標準的なOS である TOPS-10 上で稼動し，その仕様書や記録文書なども整備され，今なお多くのユーザを抱える "現役" の LISP である．InterLISP の特徴はその豊富なツールにあると言える．DWIM (Do What I Mean) と呼ばれるスペルの誤りを自動的に訂正する機能，対話的なデバッグを可能にするブレーク・パッケージ，システム内蔵の構造エディタ，言語仕様の拡張としての CLISP (手続き型言語に見られる **if…then**, **while…do** などの構文や中置表記 (infix notation) で LISP 関数が記述できる) などがそれである．InterLISP は，ラムダ式を用いた関数定義のための DEFINEQ，ラムダ式の仮引数リストをアトムに替えて表記する非展開ラムダ式 (non-spread lambda，実引数を展開しない評価法) の導入，MAP 関数族の引数が「データ」「関数」の順で並ぶこと，変数束縛が連想リストに対応するスパゲッティスタックを用いた "深い束縛 (deep binding)" であることなど，次に述べる MacLISP と比べると LISP 1.5 を色濃く反映していると言えよう*．最近 Xerox 社 が開発した LISP マシンの Dorad/1132 や Dolphin，Dandelion/1108 などの機種の上でも Inter LISP-D と呼ぶ，専用マシン向きに機能拡張を行った InterLISP が動いている．

InterLISP の言語仕様の安定性・持続性に比べて，MacLISP のそれは対照的である．すなわち，1970 年代の半ば頃になると Zeta LISP[13]，NIL (New Implementation of LISP)[14),15] Franz LISP[16] などの MacLISP の流れを汲む種々の「LISP」がさまざまな計算機上で動き始めるようになり，その言語仕様の相違などから MacLISP との互換性も危ぶまれる事態が生じたことや，MacLISP 自身も度重なる仕様変更が行われて仕様書の改訂版[12] が出るまでは一般のユーザにとってその細部の仕様 (の変更) は計り知れないと言われたことなどである．この原因としては，MacLISP がそのプロジェクトの特殊性から MIT の PDP-10，しかも ITS と名付けられた特殊な OS 下で動作するように設計され，そのユーザも MIT とその周辺の研究者に限られていたという "閉鎖的" なものであったことが挙げられる．また，その後の LISP の設計開発者の多くが PDP-10 の 18 ビット (約 256 K 語) のアドレス空間で MACSYMA (数式処理システム)[17] のような大規模 LISP プログラムを実行することに不満を感じていたことも事実で，このことが LISP マシンを含めて新たに登場した大きなアドレス空間をもつ計算機上での LISP 開発を生み出す原動力となった．

MacLISP の特徴はその機能性の高さや操作性の良さにあると言える．言語仕様でのマクロや readtable (構文規則の変更) の機能，データ構造での無限多倍長整数の導入，テキスト・エディタとファイル・システムと LISP 処理系の3者を有機的に結合させた Emacs と呼ばれるエディタの存在などがその例である．その言語仕様を前述の InterLISP と比較してみると，次のような違いがあることがわかる．構文について言うと，関数定義は (DEFUN…で始まり，FEXPR も定義できるが，その展開ラムダに相当するものがない．MAP 関数族の引数は「関数」「データ」の順に並ぶ．InterLISP の PUT に相当する PUTPROP の属性として設定する値は第2引数 (第2引数と第3引数とが入れ替る) となる．また，CLISP

*このことは仕様書[10] 中の M 式の記述からもうかがえる．

のような構文はマクロとして処理される．変数束縛では後述する“浅い束縛 (shallow binding)”の方式を採用している．

2.3 百花繚乱

LISP 界のことわざに「インプリメンテータが1人いればLISPが1つ生れる」と言われるほどLISPの作成や仕様の変更は容易に行うことができる．これを反映してか，1970年代に入ると種々の「LISP」が登場するようになった．MacLISP から分れた Stanford LISP 1.6[18] やこれに InterLISP の優れたツールを取り入れた“折衷”型の UCI-LISP[19] などがその先駆けである．前者は連想リストに替えてスタックによる“浅い束縛”法を最初に実現した処理系であり LEXPR や LSUBR の評価型非展開ラムダ式やマクロ機能の存在など MacLISP の原型をよく留めている．後者も PDP-10 上にインプリメントされたが，InterLISP 程には“重く”ないことから，その「代用品」として重宝がられて今日に至っている．

我が国でも1974年の夏に蓼科で記号処理に関するシンポジウム[20]が開かれ，KLISP, HLISP, EPICS-LISP, MELCOM-LISP, H 8350 LISP, LIPQ, OLISP, LISP-43 などの主として商用汎用計算機上にインプリメントされた LISP 処理系が報告された．現在でもこれらの子孫や新たに作られた「LISP」が多数実用に供されており，それはこの分野におけるレベルの高さを物語るものである．

2.4 LISP マシン

最近の LISP についての議論の多くは LISP マシンを除いては考えられないと言われる程に，LISP マシンの LISP（主に処理系）に及ぼした影響は計り知れないものがある．LISP マシンが重視されるのは，LISP のソフトウェア・インプリメンテーションにおける実行効率の低さを専用ハードウェアの力で補うことで実行速度の向上が図られたことと，プログラミング環境の整備されたパーソナル計算機が実現できたことにある．これについては，節を改めて述べることにする．

歴史的に見て，Wigingtonの論文[21]やカーネギー・メロン大学(CMU) の C. ai[22]，L. P. Deutsch の LISP マシンの設計[23]などが先駆的な LISP マシンの研究として挙げられるが，実質的には，L. P. Deutsch による ByteLISP[24] と R. Greenblatt の CONS[25),26] が最初であろう．これらのマシンには，ビットマップ・ディスプレイやマウスなどの入出力装置，Ethernet とか Chaosnet とか呼ばれるネットワークの機能が備わっており，このことからもこれらマシンの目標とするものがパーソナル化にあることは明瞭であった．LISP マシンはその開発の経緯から，InterLISP 系 (Xerox 社の LISP マシン) と ZetaLISP 系 (CONS の後継機種である CADR や LAMBDA (LMI 社)，3600 (Symbolics 社) など，それに，Explorer (TI 社)，4404 (Tektronix 社)がこの例である) に大別される．各 LISP マシンに対する評価は単にプログラムの実行速度だけでなく，機能性やプログラミング環境を含めた総合的な見地から行う必要があろうが，実行速度に関しては D. Gabriel の興味深い報告[27]がある．

我が国においては，前述の記号処理シンポジウム報告集[20]にある電総研の HP 2100 のマイクロプログラムを用いた Bobrow スタックモデルの LISP マシンの作製がその先駆けであろう．その頃，青山学院大ではインテル 8080 を用いた ALPS と言う LISP マシンの製作が進められていたが，LISP マシンの開発が各所で本格的に行われ始めたのは Xerox や MIT でマシンの試作が軌道に乗ってからの後のことである．その開発機関とマシン名称について主なところを列記すると，理化学研究所 (FLATS)，慶応大学中西研 (SYNAPSE)，同相磯・所研，NTT武蔵野通研 (ELIS)，京都大学 (NK 3)，大阪大学 (EVLIS)，神戸大学，東北大学 (AIM 1.0)，電総研 (ACE)，富士通 (ALPHA) などが挙げられる．紙面の関係でこれらの参考文献も省略せざるを得なかったが，詳しくは文献[28),29] が役に立つであろう．

2.5 統一への試み

LISP 1.5 が登場してから今や4半世紀になるが，これまで言語仕様の統一（標準化）についての積極的な取り組みがなんら行われなかった結果，LISP 界は百花繚乱の状態となり，このことが数式処理や人工知能のためのソフトウェアの普及の大きな障害になっていることは先に述べたとおりである．標準的な言語仕様を作る試みとしては Standard LISP[30] の例が挙げられよう．筆者もその使用経験があるが，言語仕様は LISP 1.6 に近いが，ほかの LISP との互換性が低く，その機能も必要最小限のものに留められていることから，現状では「標準」と見なされるまでには至っていない．

1980年は LISP 界にとって重要な年となった．それは，第1に LISP コンファレンス[31]が開催されたことであり，第2に，米国の人工知能関係のプロジェクト

をサポートする DARPA (Defence Advanced Research Project Agency) が LISP の標準化に向けて検討を始めたことである．そのLISPコンファレンス2日目のパネル討論会で各「LISP」の代表者がそれぞれに自慢話や批判話を行ったが，やはり大勢は「何らかの標準化案を作らなければならない」という雰囲気であったと思う．このような状況下で，標準化の作業がCMUのSpiceプロジェクト[32]のG. L. Steele Jr.を中心に開始され，その概要がCommon LISP[33),34)]として1982年に公表された．Common LISP は基本方針としてプログラムの移植性の高さ，言語自体の表現能力の豊富さ，翻訳系 (compiler) と解釈系 (interpreter) の一貫性などを追求したものであるが，言語仕様は¶Zeta LISP の影響が強く，変数の束縛戦略*は Steel 自身の SCHEME[35] の考えを流用しているとも言える．これがLISPの「標準規格」として定着するか否かは，その言語仕様や処理系の規模，普及度，蓄積されるソフトウェアの量などさまざまな要因を考えなければならず，これを見きわめるにはまだしばらくの年月が必要であろう．

3. LISP の基本的特徴

言語仕様を異にする「LISP」間でこれまで曲りながらもソフトウェアの流通が行われていたのは，それらの「LISP」が LISP の基本的特徴を共有していたからにほかならない．ここでは，その基本的特徴について，仕様や処理方式の変化した部分も含めて述べることにする．

3.1 データ型

LISP は型のない言語であると言われる．しかし，これには注釈がいるのであって，すべてのデータ(型)を含むS式という1つの型だけが存在する言語ということである．LISP 1.5 を例にとれば，S式を構成するアトム（文字アトムや数値），点対 (dotted pair, リストもこれに含まれる)，配列** はそれぞれ異なるデータ型とするのが型を強く主張する「LISP」の考え方である．これによれば，述語 ATOM はアトムという型を判定する関数のことであり，EXPLODE や IMPLODE は型変換関数ということになる．型の主張の究極は「型宣言」である．型宣言のある言語に共通することは，利点としてのプログラムの信頼性の向上とコンパイラの最適化による実行速度の向上とであり，欠点としては型の多様化に対するプログラミング上の煩雑さが挙げられる．Common LISP は型宣言を採用している数少ない LISP の1つである．LISP は伝統的にデータ型の増加を好まなかった．その理由がインプリメンテーション上の制約によるのか，S式という万能のデータ構造の存在のためなのかは定かでないが，LISP 1.5 以降，多くの LISP に新たに追加された型は文字列(string)位であろう．これは識別子として用いる非数値アトム(シンボルとも言う)と単なる文字の列記に用いるデータの文字列とを区別することで OBLIST の負担を軽減できるという記憶管理の効率化のために導入されただけにすぎない．MacLISP の配列の4つの型（成分が通常のS式，1語整数，1語実数，印付け* しないもの）や圧縮リスト表現に似た hunk と呼ばれる構造体なども PDP-10 上の LISP の宿命である1語1セル表現での妥協の産物でしかない．Common LISP では，前者は1つの配列という型に，後者はベクタとしてリストを含む“列 (sequence)”という型に統合されるのである．では，Common LISP のデータ型はどうかと言うと，その数はほかの「LISP」に比べてかなり多い．その中に注目すべきものがいくつかあるので，以下に述べることにしよう．

（1） 数 (number) ここでは，分数，複素数の導入と浮動小数点数の充実が注目される．

（2） 文字 (character) 1文字（シンボルではない）を表わす型が導入されたこと．

（3） 配列 (array) 一次元の配列をベクタとして分離したこと，文字列は文字を成分型とするベクタとすることが注目される．

（4） 構造体 (structure) データの抽象化を実現するための機能が備わったこと，これと類似のものが MacLISP にある．

（5） パッケージ (package) 関数名や変数名などで大域的に用いられる名前の衝突を回避するための1つの方法．要するに OBLIST が複数個実現されると考えればよいであろう．この機能を備えた LISP には，ZetaLISP や PSL (Standard LISP の改訂版) などがある．

型を強く意識しない LISP にとって，その最大の問題の1つは与えられたデータに対して各関数（述語を除いて)が行う実行時検査によるオーバヘッドである．CAR ではそのデータがアトムか否かを，算術関数で

* binding strategy

** 定義した配列名が関数の働きをするという仕様であった．

* marking のこと，ごみ集めで現役 (使用中) のデータに印を付けてごみにならないようにする作業のこと．

はそれが数値か否かを検査するといったことである．こうした負担を軽減するために，特定のデータ型のための関数を新たに作る（MacLISP の算術関数＋－などがこの例である）とか，コンパイラでの型検査省略機能（ユーザオプションによる）などが考案されているが，言語仕様が複雑になるだけで，抜本的な解決とまではいかないのが現状である．

3.2 関数の型

LISP 1.5 の関数の型は EXPR, SUBR, FEXPR, FSUBR の4個であり，EXPR と SUBR の違いは，前者の関数定義がS式で存在するのに対して，後者のそれは機械語であるという“形式的”なことである．EXPR と SUBR の前にFが付いたものはその実引数を評価しない関数であることを意味する．これは，LISP が call-by-value（実引数をすべて評価してから関数に渡す）と call-by-need（呼ばれた関数が必要に応じて実引数を評価する）という2種類の引数渡しの方式を有することである．

InterLISP では関数の型が12個に増えた．この数 $12=3\times2\times2$ は，関数にS式，翻訳（compile）されたもの，機械語で組み込まれたものの3つの“形式”があること，引数渡しの方式に前述の2つのほかに，ラムダ式に通常の展開型と非展開型の2つがあることによるものである．S式定義関数で，評価・展開型は EXPR に，評価・非展開型は LEXPR (LISP 1.6, MacLISP) に，非評価・非展開型は FEXPR (MacLISP) に対応し，非評価・展開型は LISP 1.6 の FEXPR の一部に見られるだけである．

FEXPR の利点は無駄な評価を省くことによる処理時間の短縮にある．よく知られている例として AND* がある．次の例

```
(AND T NIL···)
```

では，NIL 以降の形式を評価しなくても答は NIL になることが明らかである．したがって，形式の評価をこの NIL で止めてしまうのである．AND は DF** を用いて

```
(DF AND (X) (COND ((NULL X) T)
  ((EVAL (CAR X))
    (APPLY (FUNCTION AND) (CDR X)))))
```

と書くことができる．この定義式を見て気づくことは EVAL, APPLY という解釈系を背負っていることである．呼ばれた側 (AND) で実引数を評価する必要から使用されるものであるが，処理速度の向上という観点からは，“重い”解釈系を内在するのは好ましくないし，翻訳ともなれば一層深刻な問題となる．しかし，実際の処理系では，この例の AND のような使用頻度の高い FEXPR は機械語 (FSUBR) で組み込まれたり，翻訳でもそれが展開された形式で機械語が生成されるので，このような問題は起らないことも事実である．

この FEXPR に替わるものとして導入されたのがマクロ (macro) である．マクロは MacLISP や LISP 1.6 に現われており，InterLISP にも一部であるがマクロ機能（入力時や翻訳時に使用できるマクロ）がある．前述の AND をマクロを用いてどのように記述するかは，それが生成する形式で決まる．一例として

$(\mathrm{AND}\ e_1\ e_2\ e_3\cdots e_n)$

を，これと等価な次の条件式

$(\mathrm{COND}\ (e_1\ \mathrm{T})\ (e_2\ \mathrm{T})\ (e_3\ \mathrm{T})\cdots(e_n\ \mathrm{T}))$

に展開するならば，次のマクロ定義が考えられよう．

```
(DM AND (X) (LIST 'COND (MAPCAR
  (FUNCTION (LAMBDA (Y) (LIST Y 'T)))
  (CDR X))))
```

MacLISP に準拠すれば，次のように書ける．

```
(defmacro and (& rest forms) (list 'cond
  (mapcar #'(lambda (y) (list y 't)) forms)))
```

ここで #' 記号はそれが FUNCTION として用いられることを表わす．

それでは，マクロが FEXPR に完全に取って替わるかと言うとそうでもない．翻訳系は FEXPR に対してこのようなマクロ展開を実際に行っているので，それが翻訳されてしまえば実行効率上何も問題は生じない．しかし，解釈系にマクロ処理が委ねられると事情は一変する．その効率の低さは EVAL や APPLY を陽に呼ぶ FEXPR の比ではない．Common LISP で一部のマクロを（FSUBR のように）展開せずに直接評価することの便宜を与えようというのはこのためである．マクロがその言語の表現を豊富にすることは確かである．このことは，核になる必要最小限の“形式”の仕様を固定して置き，あとはマクロで好きなものを作ることが可能であることを示している．これにより，その言語の文法の簡潔性と安定性を図ることができる．この考えの延長線上にあるのが Common LISP である．

最後にラムダ式について述べる．LISP の成功は1

---

* ここでは T または NIL を返す AND を考える．
** Define FEXPR (LISP 1.6)

つにラムダ計算法の導入とそれを記述するラムダ式の発明にあることは明らかである．ラムダ式の評価では，ラムダ変数（仮引数）に値（実引数）を与えて，ラムダ式中の形式を評価することで1つの値（結果）を得る．この簡潔にして優美な評価法にも問題がないわけではない．それは，形式中に現われる自由変数の処隅や仮引数と実引数の個数の不一致への対応などである．前者については**3.3**で述べる．後者について言うと，EXPR の場合，仮引数の個数が実引数の個数より多いと LISP 1.5 に従えば PAIRLIS でエラーとなる．実用上これは好ましくないので，余る仮引数に対しては NIL や最後に現われる実引数などを与える方法が用いられたり，不一致を前提にして，不要の実引数に対応する仮引数を NIL と書く仕様(MacLISP)が作られた．この仕様をさらに発展させ，ラムダ式での引数渡しの機能を強化したのが Common LISP である．ここではその機能の1つを例で示すだけに留めよう．

```
(defun rcons (&key (car nil) (cdr nil))
    (cons cdr car))
(rcons) は (nil)
(rcons :car 'a 'b) は (nil. a)
(rcons :car 'a :cdr 'b)
(rcons :cdr 'b :car 'a) はともに (b. a) となる.
```

このパラメタ結合機能は Ada[36] などの手続き型言語によく見られるものである．

3.3 変数の束縛

LISP 変数の束縛戦略は動的束縛 (dynamic binding) であると言われる．それは変数の値を求めるのにこれまでの束縛の履歴を現時点から過去へと逆上る形で連想リストが走査され，その変数に束縛された最新の値が求められるためである．この戦略を実現するために考案されたのが，深い束縛と浅い束縛の2方式である．前者は連想リストがその見本であり，これをスタックに適用したのが InterLISP で用いられているスパゲッティスタックである．この方式では，束縛が生じたときにその変数の名前と値の対が連想リスト（またはスタック）に登録され，ラムダ式の終了などで束縛が不要となったとき連想リスト（またはスタック）からその対が除かれる．変数名の探索は連想リストの先頭（またはスタックの最上部）からその名前を鍵にして行われる．この探索がいわゆる線型探索となり，連想リストやスタックの深部に登録された名前の探索に時間を要することから，探索効率の向上を目的に“浅い束縛”が考案された[18]．その着想は，各変数について，環境を表わす一義名とその環境で束縛された値の対を作り，この環境名で変数の値を探索することにあった．これを LIFO スタックで実現したのが一般に知られている浅い束縛である．それは，value cell と呼ばれる各変数に対応した特定の記憶場所（たとえばアトムの値を保存する場所）にその変数に束縛された最新の値を保存し，環境の切り換えによって新たに束縛の行われた変数についてのみその（value cell にあった）旧値とその変数名とをスタックに保存し，value cell を新たに束縛された値で更新するというものである．この方式では，value cell の更新（復元を含めて）に手間を要するが，変数名の参照は，その大部分が value cell の参照で済むようになる．この方式の唯一の問題点は関数引数に現れる自由変数の扱い（いわゆる funarg 問題）である．それを次の例で示すことにしよう．

```
(DE FOO (X) (BAA (FUNCTION
    (LAMBDA ( ) X)) (ADD 1 X)))
(DE BAA (F X) (LIST X (F)))
```

の2つの関数が定義されてあるとして，

(FOO 0) の評価値は (1 1) となる．これは，FOO 中の関数 (LAMBDA ( ) X)) の自由変数 X が本来ならば FOO の仮引数 X に対応すべきところを関数引数として BAA に渡されて評価される時点で BAA の仮引数X（この値は0でなく1である）に対応してしまうからである．深い束縛ならば，BAA の形式中の X と (F)，すなわち ((LAMBDA ( ) X)) を評価する別個の環境が保持されているために後者のラムダ式中の X が BAA の仮引数Xに化けるようなことはなく，プログラマが意図したであろう値 (1 0) が結果として返される．

この funarg 問題を解決する方法として考案されたものが LISP 1.6 や MacLISP で採用されている Binding Context Pointer とか a-list pointer と呼ばれるスタックに保持された funarg 生成時の変数の束縛情報（環境）を指す特別なスタックポインタの導入であり，ZetaLISP や Franz LISP で採用された閉包 (closure) の導入である．閉包とは，変数の束縛情報を保持するための機能のことで，その変数をプログラマが直接指定できるようになっている．

浅い束縛は先に述べた理由により解釈系の実行効率を高めることになるが，翻訳系が導入されるとこの事情は一変する．翻訳系にとって都合の良い束縛戦略

は，1つの関数の中で束縛された変数（ラムダ変数など）のスコープをその関数本体だけに限ることである．このような束縛戦略を静的束縛 (static binding)，または lexical binding と呼ぶ．これは Algol 系の言語では一般に採用されている戦略であるが，自由変数をすべて大域変数と見なして，その値を束縛環境で決めようというものである．この戦略に基づいて，前述の (FOO 0) を実行すると，深い束縛によるのと同じ値 (1 0) が返される．すなわち，翻訳系が“自然”に行う静的束縛が funarg 問題を解決したことになる．しかし，これでは，解釈系の行う浅い束縛と翻訳系の静的束縛との間に不整合が生じることになる．つまり，同じ関数(群)を実行するのに解釈系と翻訳系とでは結果の異なる事態が起り得るのである．FLUID とか SPECIAL という宣言を大域変数に付けて翻訳系に指示するのはこの不整合を生じさせないためである．静的束縛を基本とする Common LISP では，この指示は翻訳系ではなく解釈系に対して行われることになる．この SPECIAL の判定は解釈系の実行効率をかなり低下させることになろうが，それにも増して翻訳系の利得が大きいことを見込しての決断であろうと思われる．

3.4 制御構造

LISP 1.5 の制御構造は，引数の個数は2個に限られるが PROG 2 による逐次実行，COND 式による条件分岐，帰納呼び出しを含む関数呼び出し，ALGOL 流のブロック構造を取り入れた PROG 形式，ERROR と ERRORSET の組合せによる大域的退出が主なものである．これらの基本的な制御構造と FEXPR（実用的には FSUBR）やマクロの組合せで，PROG 2 を n 引数に拡張した PROGN，implicit PROGN を含む COND 式，反復制御構造の DO などが作れることから，これらを含む多様な制御構造がその後の多くの「LISP」に導入されることとなった．

3.5 くず集め

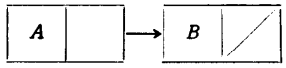
くず集め (garbage collection) の機能は言語仕様に直接含まれるものではないが，プログラマをデータ管理の煩雑さから解放したという点で，LISP の発展に果した役割は大きいものがある．くず集めについては，文献[38]に詳細な解説があるので，ここでは次節の LISP マシンとの関連で述べることにする．

印付けや複写による古典的くず集めの最大の問題点は，その処理が行われている間 LISP の実行が抑止されることであった．これを解決するために考案されたのが実時間 (real time) とか並列型 (concurrent) とか呼ばれるくず集めである．これには，印付け法を改良してくず集めとリスト処理が並行して実行できるようにした Steele や Dijkstra 提案のもの，参照計数器 (reference counter) を用いる Deutsch と Bobrow のもの，複写法を改良した Baker のものなどがある．後半の2つは，それぞれ Xerox 社の LISP マシンと MIT の LISP マシンに実装されている．

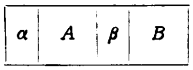
4. LISP マシン

LISP マシンの登場は，それまでの商用汎用計算機上のソフトウェア・インプリメンテーションでは効率向上が図れなかった問題のいくつかを解決することとなった．このことは主計算機の TSS 下で動作する LISP の応答速度を含めた処理能力の限界やその使い勝手に対する不満が処理の高能率化やパーソナル化を実現した LISP マシンの出現により解消されつつあることにも表われている．効率向上は時間と空間(領域)の2つの側面に対して行われている．前者についてはマシン固有の機械語命令（マクロ命令とも呼ぶ）により処理速度の向上が図られる．マクロ命令は汎用計算機の機械語に比べて機能が高く，たとえば，実記憶上のデータを参照するのと同時にポインタ・タグによりそのデータ型の判定が行えたり，マシンによっては，命令の先取り（先行制御）を行ったりする．マクロ命令の形式はマシンにより純スタック命令（0オペランド命令）形式から n オペランド形式のものまでさまざまである．

領域に関しては，S式で記述されたプログラムをマクロ命令に翻訳することで得られる使用領域量の節約と，S式自体の CDR-coding で得られる使用領域量の減少とがある．前者はS式の表現法やマクロ命令の設計に依存するし，実行速度の向上が主目的である翻訳の副次的効果であることから，その寄与率は後者より少ないと思われる．後者は圧縮リスト表現とも呼ばれる．セルの CDR 部がそれに隣接するセルへのポインタであるときその CDR 部を省いてしまう表現のことである．ただし，単に省いてしまうと，その連結状態の情報も失なわれることから，その情報はタグとして CAR 部（実質的に CDR 部のタグはない）に残される．連結状態に，CDR 部が隣り（圧縮），CAR 部が1回の間接アドレス（RPLACD の実行で生れる），通常のセルの3つがあり，これにリストの終端という状態を加えて，計4個の状態を2ビットで表わ



(a) 通常の(箱型)表現



α: CDR 部が隣接することを示すタグ値
β: リスト終端を示すタグ値
(b) COR-coding による表現

図-1 (AB) の表現法

している. CDR-coding のソフトウェア・インプリメンテーションはその処理速度の点でかなり非能率であることから, LISP マシン以外の LISP で採用されることはなかった. この点において, CONS の領域節約に果した役割が強調されるのである. すなわち, CDR-coding によれば, (AB) と (A. B) は同じ記憶容量となるが, 通常のS式表現では, LIST を用いた前者は2セルを必要とするのに対して, CONS を用いた後者は1セルで済むからである. また, CDR-coding はリスト・データの局所性を高め, ページングへの配慮がしやすいことも付記しておく.

5. 将来の展望について

5.1 標 準 化

当面, Common LISP の動向が注目されるであろう. CMU の Spice プロジェクトでは, Common LISP のフルセットが簡単に作成できるように同LISPで書かれた Spice LISP のソース・プログラム (CMU パッケージなるもの) を公開して, Common LISP の普及を積極的に進める方針である. また, 1985年度から始まる DARPA の Strategic Computing Project では公式言語として Common LISP が採用されるのではないかという観測もある. Common LISP の設計は, 当初 MacLISP 系のメンバにより開始されたが, その後, InterLISP や PSL のメンバまでも設計に加わった. 現在も仕様の部分的な改訂作業が進行中であり, これに関する意見が広く求められている. このように, Common LISP の「標準化」への布石は着実に進められているようであるが, 問題が全くないわけではない. それは, 言語仕様自体が汎用計算機向きでないという指摘や, 計算機に依存する部分の仕様が明確でなく個々のインプリメンテーションの裁量に委ねられる部分が生じ, これが Common LISP の方言を生み, ひいては Common LISP 間でのプログラムの移植性の障害になるという指摘などにも現れている.

5.2 LISP マシン

パーソナル化を指向した LISP マシンは, その優れたプログラミング環境と豊富なアプリケーションを売り物にこれまで普及してきた. ワークステーションとしてのプログラム開発やソフトウェア作成という従来の利用法に加えて, 今後, OA, CAD/CAM, CAI などの利用が増加することが見込まれる. ハードウェア技術の進歩により, マシン価格の低廉下と性能向上とが実現するならば, 今のパーソナル・コンピュータのように気軽に LISP マシンの使える時代が到来することであろう. その時点で最高台数の LISP マシン上の「LISP」が標準になるかと言うと, これはいささか尚早で, おそらく大部分のユーザは LISP の存在など意識することはないであろう.

パーソナル LISP マシン以外の LISP 専用マシンも国内で数例が報告されているが, リスト処理の高速化や並列型くず集めの実装だけを目標とする場合, そのアーキテクチャはフォン・ノイマン型からそれほどかけ離れるものではないため, 実装技術の問題だけとも言えよう.

5.3 大容量記憶管理

LISP の応用分野として知識工学が注目されている. これには知識ベースを記録するための大容量仮想記憶が必要となる. 1 GB (LISP マシンの 3600 はこの空間をサポートする) 位の仮想空間を用いる LISP では, リスト処理もくず集め[39]も従来のものと異なる配慮を必要としよう. 仮想空間でのリスト・セルの life time の解析を含めて, その効率的な管理は今後の検討課題である.

5.4 新アーキテクチャへの指向

リスト処理の高速化には, マクロ命令自体の高速化やパイプライン制御のほかに, 複数プロセッサによる並列処理が考えられる. しかし, LISP を含むリスト処理の並列化の研究は日が浅く, そのアーキテクチャも従来のものと大幅に異なることが想定される. 並列 LISP マシンの実装例は報告されているものの, さらに進んだ, データフロー・マシンやリダクション・マシンの実用化の研究は途についたばかりである.

5.5 他言語との関係

Common LISP の動向とは無関係に, 今後もほかの言語の機能の "吸収" が進むことであろう. 抽象データ型や並列機能, Flavor と呼ばれるオブジェクト指向プログラミング・パッケージの導入や PROLOG[40]

機能をもつ LISP の提案などを見ればわかるとおりである．そのとき，これらを LISP と呼ぶのか，あるいは，すでに LISP ではないのか定かではないが，いずれにせよ，LISP は今後とも研究活動が必要な多くの問題を提起することになろう．

参考文献

1) Newell, A. and Shaw, J.C.: Programming the Logic Theory Machine, Proc. of the 1957 Western Joint Computer Conference, IRE (1957).
2) Newell, A. ed.: Information Processing Language V Manual, Prentice Hall, Englewood Cliffs, NJ. (1961).
3) Gelernter, H., Hansen, J.R. and Gerberich, C.L.: A FORTRAN-Compiled List Processing Language. JACM, Vol. 7, No. 2, pp. 87-101 (1960).
4) McCarthy, J.: Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1. CACM, Vol. 3, No. 4, pp. 184-195 (1960).
5) Church, A.: The Calculi of Lambda-Conversion, Princeton University Press. Princeton, NJ. (1941) (Reprinted by Klaus Reprints, NY. 1965).
6) McCarthy, J. et al.: LISP 1.5 Programmer's Manual, MIT Press, Mass. (1962).
7) Deutsch, L.P. and Berkeley, E.: The LISP Implementation for the PDP-1 Computer (in The Programming Language LISP: Its Operation and Applications). Information International Inc. Mass. (1964).
8) McCarthy, J.: History of LISP. ACM SIGPLAN Notices. Vol. 13, No. 8, pp. 217-223 (1978) (Also in History of Programming Languages. Academic Press. NY. 1981).
9) Teitelman, W. et al.: BBN-LISP TENEX Reference Manual, Bolt Beranek and Newman Inc. Mass. (1971).
10) Teitelman, W. et al.: INTERLISP Reference Manual, Xerox PARC, CA. (1974).
11) Moon, D.: MACLISP Reference Manual, Version 0. Lab. for Computer Science, MIT, Mass. (1974).
12) Pitman, K.M.: The Revised MACLISP Manual, MIT/LCS/TR-295. Lab. for Computer Science, MIT, Mass. (1983).
13) Moon, D., Stallman, R. and Weinreb, D.: LISP Machine Manual, 5th ed. Artificial Intelligence Lab. MIT, Mass. (1983).
14) Brooks, R.A., Gabriel, R.P. and Steele, G.L. Jr.: S-1 Common LISP Implementation, Conference Record of the 1982 ACM Symposium on LISP and Functional Programming (1982).
15) Burke, G.S., Carrette, G.J. and Eliot, C.R.: NIL Reference Manual, MIT/LCS/TR-311. Lab. for Computer Science, MIT, Mass. (1984).
16) Foderaro, J.K., Skowler, K.L. and Layer, K.: The FRANZ LISP Manual, Univ. of California, Berkeley, CA. (1983).
17) MACSYMA Reference Manual. The MATHLAB Group, Lab. for Computer Science, MIT, Mass. (1974).
18) Quam, L. and Diffie, W.: Stanford LISP 1.6 Manual, Operating Note 28.7. Artificial Intelligence Lab., Stanford Univ. CA. (1972).
19) Bobrow, R.J. et al.: UCI LISP Manual, Technical Report 21, Univ. of California, Irvine. CA. (1973).
20) 記号処理シンポジウム報告集，情報処理学会・プログラミングシンポジウム委員会，蓼科 (1974).
21) Wigington, R.L.: A Machine Organization for a General Purpose List Processor, IEEE Tran. on Computer, pp. 707-714 (1963).
22) Barbacci, M., Goldbarg, H. and Knudsen, M.: C. ai—A LISP Processor for C. ai. CMU-CS-71-103. Carnegie-Mellon Univ. PA. (1971).
23) Deutsch, L.P.: A LISP Machine with very compact Programs, Proc. of 3rd IJCAI. (1973).
24) Deutsch, L.P.: Byte Lisp and its Alto Implementaions, Conference Record of the 1980 LISP Conference, pp. 231-242 (1980).
25) Greenblatt, R.: The LISP Machine, Working Paper 79, Artificial Intelligence Lab., MIT, Mass. (1974).
26) Knight, T.: CONS. Working Paper 80, Artificial Intelligence Lab., MIT, Mass. (1974).
27) Gabriel, G.: Performance and Evaluation of Lisp Systems. Stanford Univ. CA. (1984).
28) 安井 裕：LISP マシン，情報処理，Vol. 23, No. 8, pp. 757-772 (1982).
29) 情報処理学会記号処理研究会資料 17-1, 2; 18-8; 19-4; 20-5; 21-1; 22-1, 3; 23-1; 24-1, 2, 3; 27-2, 4; 30-3.
30) Marti, J. et al.: Standard LISP Report, Univ. of Utah, Utah (1978) (also in ACM SIGPLAN Notices, Vol. 14, No. 10, pp. 48-68, 1979).
31) Conference Record of the 1980 LISP Conference, Stanford, CA. (1980).
32) Ball, E. et al.: The Spice Project, Computer Science Research Review, Dept. of Computer Science, Carnegie-Mellon Univ. PA. (1982).
33) Steele, G.L. Jr.: An Overview of Common LISP, Conference Record of 1982 ACM Symposium on LISP and Functional Programming,

pp. 98-107 (1982).
34) Steele, G. L. Jr. et al.: Common Lisp, Digital Press, Mass. (1984).
35) Steele, G. L. Jr. and Sussman, G. J.: The Revised Report on SCHEME: A Dialect of LISP, Memo No. 452, Artificial Intelligence Lab., MIT, Mass. (1978).
36) Military Standard Ada Programming Language, ANSI/MIL-STD-1815 A. DoD. (1983).
37) Allen, J.: Anatomy of LISP, McGraw-Hill, NY. (1978).
38) Cohen, J.: Garbage Collection of Linked Data Structures, ACM Computing Surveys, Vol. 13, No. 3, pp. 341-367 (1981). (筆者訳：つなぎのあるデータ構造のくず集め，コンピュータサイエンス '81 (bit 別冊) 共立出版 1983).
39) Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, Texas (1984).
40) Kowalski, R. A.: Predicate Logic as Programming Language, Proc. of IFIP-74 Cong. North-Holland, The Netherlands (1974).

(昭和 60 年 4 月 30 日受付)