寄　　書

KLISP の拡張機能とその応用*

中　西　正　和**

1. まえがき

LISP[3] に基づくリスト処理言語 KLISP[4] の拡張機能とその応用の一面を紹介する．LISP は，人工知能の研究のために開発されたものであり，その構文法，処理アルゴリズム，記憶構造などが簡潔であり，プログラミング言語として，数学的な考察がなされやすいという特徴があろう．反面，形式の簡潔さは，人間の直観に対して訴えにくいということをひき起こし，プログラミングの誤まりが増える．記憶の構造の簡潔さは，多くの記憶素子を必要とし，小型の計算機での高速の処理は期待できない．

現在，LISP 言語と人間の間の接近は，補助言語の開発により，あるいは，LISP-2[8] や CPL[9] のように，ALGOL 的な言語などとの結合によってなされようとしている．一方，記憶容量の問題は，リスト構造に対するページング・アルゴリズムの研究や，ガーベージ・コレクションの方法の改良とそれにともなう経済的なリスト構造の作成[2] といったことにより解決しようとしているようである．

しかし，言語に関するこれらの改良は，関数評価方式やλ記号，汎関数などをとり入れて精神を統一しようとしている思想に反抗する方向に進み，記憶に関する研究は，記憶状態と言語との一貫した対応に水をさす結果となる．ページングの方は，一貫性に支障を来たすことにはならないかも知れないが，処理時間が非常に大きなものになるおそれがある．

KLISP[6] は，TOSBAC-3400 モデル 30 のために開発したものである．設計の前提は，磁気ドラムや磁気ディスク装置のような，リスト構造の記憶のための大容量の外部記憶装置がないこと，および会話型のシステムとして利用できるようにすることであった．さらに，つぎのことは言語と記憶構造に対しての前提であった．

（1） LISP-1.5 の言語の思想は破壊しない．

（2） リスト構造，とくに属性リストに関して何の変形も与えない．

（3） 処理速度はできるだけ速いこと．

以上のことを守った上で，24 ビット／語，16 K 語の機械に対して LISP を作ることはかなり困難であったが，結構大きなプログラムを実行させることができた．

2. KLISP システム

KLISP システムの特徴の一つは，そのユーティリティとして，GLISP[4] という，LISP 言語のトランスレータを持っていることである．これは，LISP のメタ言語をその入力とし，S 式を出力とする，全部が機械語で書かれているシステムである．

もう一つの特徴は，LISP の読み込み関数 read の拡張機能である．これは，LISP-1.5 の read が持つ機能のほかに，モニタとしての役割を演ずるための機能が加えてある．

2.1 GLISP 言語　LISP-1.5 は，プログラムはすべて S 式で書かなければならない．LISP 言語をそのまま書くことは，カッコの多さ，複雑さに加えて，λ 記号や引用符の書き方が同じ S 式であるために，一見して理解しにくいことなどが原因して，かなり面倒な仕事であろう．Henneman の A-Language[1] はトランスレータを LISP で書いた実験言語であり，非常にすぐれたものであるが，これを実用にするためには，完全なコンパイラを持ったとしても，使用者が勝手な書式を指定することができるために，ブートストラップの時間がかなりかかりそうである．そのうえ，句の優先順位を使用者が定めることは，これが数であるために面倒な仕事になる．

さらに，成長した補助言語は，リスト構造との対応

* An extended faculties of KLISP and its application, by Masakazu Nakanishi (Department of Administration Engineering, Faculty of Engineering, Keio University)
** 慶応義塾大学工学部

が複雑になり，属性リストを操作するようなプログラムを考えにくくなる．その結果，言語として1つのレベルができ，LISP が本来特徴とするプログラムが，データと同じ構造を持つために可能となるような，たとえば，LISP による LISP プログラムの生成と実行のような機能を，人間が能率よく使えなくなるおそれがある．

16 K 語程度の計算機では，有効な記憶形態をとらせるための機能を大幅に持たせることができない．そのある部分は利用者の負担にすることを考えると，最もよいのは，メタ言語を採用し，このトランスレータを LISP システムと切り離してしまうことであろう．メタ言語は，S式とほぼ1対1に対応しているために，S式の理解も早いし，若干の特殊記号を採用すれば，デバッギングは容易になるであろう．

GLISP 言語は，つぎの形の定義を，セミコロンで区切って書く．

〈関数名〉[〈引数リスト〉]=〈式〉

定義の最後には終止符を書き，つぎに評価すべき式をセミコロンで区切って書く．小文字は，すべて大文字にし，QUOTE すべきものは "▼"（引用符）で囲む．条件式の "→" は，"−〉" とし，その他の "]"，";" などはそのまま書く．

これらはすべてS式に変換され，小型の磁気ドラム，紙テープまたは磁気テープに出力される．このトランスレータは，LISP-1.5 で完全に記述することができる．

GLISP は，いくつかのプログラムをまとめて変換することができる．そのために，あとで述べる分割したプログラムを次々に変換しておくことが可能になる．

GLISP は，KLISP とは完全に切り離されているために，出力した結果を，他の LISP システムに入力することも可能である．他の LISP システムは，これをプログラムとして，あるいは mread 関数[3]の対象として利用することができるであろう．GLISP プログラムと，そのオブジェクト・プログラムを**図1**に示す．これは，リストを集合とみなしたとき，引数に与えられる集合のベキ集合を求める関数 sets を定義し，評価しようとするプログラムである．

2.2 読み込み関数

LISP の読み込み関数には，startread, advance, read など，その目的に応じて多くのものがあるが，KLISP の read は，会話を意識して作られたために少し異なった働きをする．もともと read は，1つのS式を入力装置から読み込むものであるが，KLISP の read は，この他に，読み込んだデータによって，これを単なるS式と解釈したり，プログラムとしてその評価を行なったりする機能をも持つ．また，翻訳関数の実行中に割込むことにより，任意の時点にこの関数を評価させることができる．

割り込みを除いた read のロジックは，つぎに示す LISP のプログラムのようなものである．ここで readorig* は，LISP-1.5 の read とほぼ同じであり，columnは，第1欄に "＊" があったとき真となるような関数である．

```
read[]=prog[[x; y];
        x:=readorig[];
        [and[eq[x; *LISP]; column[]]→
        go[L1]; return[x];
L1      x:=readorig[];
        [and[eq[x; *STOP]; column[]]→
        return[label[interp; λ[[x];
        [null[x]→NIL; T→
        cons[apply[car[x];
        cadr[x]; NIL]; interp[cddr[x]]]]]]
        [y]]];
        y:=nconc[y; list[x]];
        go[L1]].
```

これは，"＊LISP" や "＊STOP" を制御カードとして扱ったことになる．＊LISP がはいったとき，この read[] の値は，このあとのダブレットの値のリストとなる．これは帰納的に定義されている．それは，上の定義中の apply の引数となるダブレットの中に，read を含む関数が含まれていると，さらにこの read 関数に帰納的にはいってくる．＊LISP があったときの read[] の値を利用する関数を定義しておくことにより，プログラムを実行してできた結果を，下のレベルで利用することが可能になる．

csetq 関数を使用すれば，最も下のレベルで宣言した変数を連絡領域として使用し，複数個のプログラムを次々に読み込み，処理を続けていくことができる．

値を原子シンボルの属性リストに与えるような，たとえば cset や deflist のような関数を評価することは，その原子シンボルに半永久的な値を与えることで

* 厳密には，ここでの readorig は，S式の読み込みが終ってもレコードをかえない．実際には，*LISP を左カッコとみなし，*STOP を余分の右カッコとみなしてプログラムを1つのリストにして処理している．

```
....... KLISP-1. SYSTEM  (VERSION-3) .......

¥JOB    MF-L-011-F  SETS       M.NAKANISHI


  1     *        PROBLEM-8  POWER SET
  2     *
  3     SETS[X]=SETS2[X;X];
  4     SETS2[X;Y]=[NULL[Y]->LIST[NIL];
  5                 T->APPEND[SETS2[X;CDR[Y]];NAKANISHI[X;Y]]];
  6     NAKANISHI[X;Y]=[NULL[Y]->LIST[NIL];
  7                 T->COMB2[X;CDR[Y]]];
  8     COMB2[X;Y]=[LENGP[X;Y]->NIL;
  9                 T->APPEND[APPLIST[CAR[X];NAKANISHI[CDR[X];Y]];
 10                     COMB2[CDR[X];Y]]];
 11     APPLIST[X;Y]=[NULL[Y]->NIL;
 12                 T->CONS[CONS[X;CAR[Y]];APPLIST[X;CDR[Y]]]];
 13     LENGP[X;Y]=[OR[NULL[X];NULL[Y]]->AND[NULL[X];NULL[Y]];
 14                 T->LENGP[CDR[X];CDR[Y]]].
 15
 16     SETS[(A B C)];
 17     SETS[(A B C D E F)].


      **** GLISP PROGRAM TERMINATED! ****

       PROCESSING TIME   0 MIN     620 MSEC
```

(a)

```
....... KLISP-1. SYSTEM  (VERSION-3) .......

*       MF-L-011-F  SETS       M.NAKANISHI


*LISP

DEFINE((
  (SETS (LAMBDA (X) (SETS2 X X) ))
  (SETS2 (LAMBDA (X Y) (COND ((NULL Y) (LIST NIL)) (T (
       APPEND (SETS2 X (CDR Y)) (NAKANISHI X Y)))) ))
  (NAKANISHI (LAMBDA (X Y) (COND ((NULL Y) (LIST NIL)) (T (
       COMB2 X (CDR Y)))) ))
  (COMB2 (LAMBDA (X Y) (COND ((LENGP X Y) NIL) (T (APPEND (
       APPLIST (CAR X) (NAKANISHI (CDR X) Y)) (COMB2 (CDR X
       ) Y)))) ))
  (APPLIST (LAMBDA (X Y) (COND ((NULL Y) NIL) (T (CONS (
       CONS X (CAR Y)) (APPLIST X (CDR Y)))))) ))
  (LENGP (LAMBDA (X Y) (COND ((OR (NULL X) (NULL Y)) (AND (
       NULL X) (NULL Y))) (T (LENGP (CDR X) (CDR Y)))) ))
 ))

SETS((A B C))
SETS((A B C D E F))

*STOP


    **** READ HAS BEEN EVALUATED. ****
```

(b)

Fig. 1

ある．このような原子シンボルは，アソシェーション・リストと無関係であるから，レベルに関係なく参照したり再定義したりすることができる．

2.3 会話のための機能

人工知能の研究のためには，直接あるいは間接に会話の機能が必要となる．この会話の機能を生かしたシステムとして，KLISP-1.5[7] という KLISP の改造版が用意されているが，これは KLISP の read の特徴を生かすように設計されている．

翻訳関数 evalquote の実行中にエラーが起こったとき，およびアテンション・キーによる実行中断の指令が使用者により与えられると，式の評価のための関数 eval が自動的に read [] を評価する．会話しようとするときの入力装置は，端末タイプライタであるため，タイプライタ上に入力要求を出す．

このとき，read [] の値は無視され，割り込まれた eval 関数の値とはならないが，read [] により読み込まれたものが LISP プログラムであって，define などの属性リストの変更のための関数を含んでいると，プログラムの変更を行なうことができる．

3. オーバーレイ

リスト処理言語の1つの特徴は，どの記憶もランダムに手を結び合えるということではあるが，現実にプログラムしてみると，仕事として分割できる場合が多い．したがって，オーバーレイを行なうことが望まれる．LISP は，プログラムがデータと同じ形を持ち，翻訳の機能を使うことができるために，オーバーレイを，LISP 言語の範囲内で行なわせることができる．これを行なうのに KLISP の read は非常に適しているといえる．

ここで例示する KLISP プログラムのオーバーレイ[5] は，プログラムのファイルをランダムにアクセスするには不適であるが，逆もどりのない一直線の処理であれば，すでに読み込んであるプログラムにより，これに階層構造を持たせることもできる（**図 2**）．

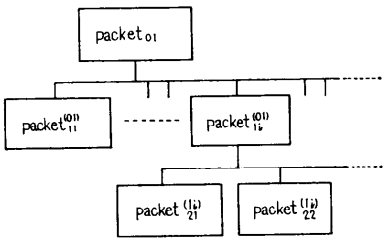

Fig. 2 structure of overlayed packets

3.1 管理プログラム

これから読み込まれるプログラムにとって，これを読み込もうとするプログラムを管理プログラムと呼ぶ．KLISP における管理プログラムの，最も簡単な例をつぎに示す．

```
executive [] ← prog [[c1 ; c2 ; … ; cn]];
                eject [];
                read [];
                …
                read []].
```

この管理プログラムは，プログラムを読み込むためだけのものである．$c_1, c_2, \cdots, c_n$ は，連絡領域であり，アソシェーション・リストにはそれぞれ NIL の値を持って登録されるが，これに値を代入するには，csetq または cest 関数を用いなければならない．それは，読み込まれたプログラムは，この c_i を持つアソシェーション・リストを持ち得ないからである（read の定義参照）．この executive に並行して，読み込まれるプログラムが共通して使用する関数を定義する．これも属性リストに定義されるために，この管理プログラムのレベルを出るまで使用することができる．executive によって，初めに読み込まれたプログラムの実行が終了すると，$c_1, \cdots c_n$ には結果が代入されている．ここでつぎの read により2番目のプログラムを読み込む．これは連絡領域を利用してつぎの仕事を行なう．このときガーベージ・コレクタが働けば，自動的に最初のプログラムが消滅する．

このような，連絡領域の方式をとると，普通の read 関数だけでも比較的簡単に書くことができる．たとえば，つぎのように書く．

```
a := read [];
apply [car [a]; cadr [a]; NIL];
…
```

しかし，これだと読み込まれるプログラムの形が，最も下のレベルのものと少し異なったものになる．また，使用者が任意の時点で，プログラムであるかまたはただのS式であるかといった指示を与えることができない．KLISP の read は，本来の read を含む合成関数であり，1つのパケットを処理する能力を持つ．

3.2 GLISP 言語の並び

オブジェクト・プログラムとして，1つのパケットとなるような GLISP プログラム，またはデータをセクションと呼ぶことにする．オーバーレイのための GLISP プログラムは，最初に最も下のレベルの管理

プログラムのセクションを書く．つづけて，この管理プログラムが読むセクションを並べる．このセクションの中に，読みこむ機能があるときは，このプログラムを管理プログラムとするようなセクションをすべて書く．

GLISP は，これらのセクションを順次変換し，外部記憶装置に出力する．すべての変換が終わると，KLISP がロードされ，これらのセクションを順に読み込み，処理を続ける．

セクションを繰り返し利用したり，遠くのセクションを読み込んだりするためには，ファイルをバック・スペースしたり，巻きもどしたり，読みとばしたりすることができなければならない．そのために rewind 関数や，＊LISP などと同じ制御カードである ＊REWIND を利用することができる．しかし，実際には，この手順がすべて使用者の負担となるため，誤まりを犯しやすく，いまのところ，このファイル構造の改良を待つほかには手がないようである．

4. むすび

LISP の属性リストは，それがプログラムによって，容易に変更されたり削除されたり作られたりすることができるようになっており，そのために他の記号処理言語に比較して非常に多くの語を必要とする．しかし，これを有効に使えば，リスト構造に密着した，かなり能率的でおもしろいプログラムをつくることができる．KLISP は，記憶を有効に使う手段として，この属性リストを縮少したり，リスト表示をまとめて記憶したりすることは行なっていない．

KLISP は，慶応義塾大学情報科学研究所TOSBAC-3400 のために昭和 42 年に生まれたものである．その後，利用者の要望を考え，機械語関数の種類を変更したり，GLISP の仕様をかえたりして今日に至っている．KLISP 本体は，入出力ルーチン，機械語関数などが約 7K を占め，残りの 9K は，利用者のためのスタックとフリーストーレジである．GLISP は約 5K である．アセンブリプログラムでの KLISP, GLISP の総ステップ数は約 12,000 である．

大学内での利用状況を見ると，GLISP を利用したのと利用しないで直接S式でプログラムしたのとでは，デバッギングの時間に関して，明らかな有意性が認められた．オーバーレイをしなければならないような仕事もぼつぼつ現われ始めており，GLISP で 300 行程のプログラムを 1 つのセクションとした数セクションのプログラムが作られている．

現在の課題としては，プログラム・ファイルの問題と，会話による利用法の開発，自己増殖などであろう．

最後に，著者の未熟な仕事に対して理解を示され，親切にご指導下された浦昭二先生，関根智明先生，土居範久先輩，原田賢一先輩に対して，深く感謝する．またプログラムなど，多くの援助を与えられた小林利臣君，竹村良孝君に厚く感謝する．

参考文献

1) Berkeley, E. C. and Bobrow D. G.: The Programming Language LISP: Its Operation and Application, MIT Press, Cambridge, Mass., (1964)
2) Hansen J. W.: Compact List Representation: Definition, Garbage Collection, and System Implementation, Comm. ACM. (Sept. 1969) 499-507.
3) Mc Carthy, J., et al.: LISP Programmer's Manual. MIT Press, Cambridge, Mass., (1962)
4) 中西：KLISP 説明書，計算機シリーズ 7，慶応義塾大学情報科学研究所 (1969)
5) 中西：KLISP プログラム集，計算機シリーズ，慶応義塾大学情報科学研究所 (1970)
6) Smith, D. C.: MLISP User's Manual. stanford Artificial Intelligenct project Memo A 1-84. stanford Uuiv. (1969)
7) 竹村良孝：タイムシェアリングシステム用記号処理言語，慶応義塾大学工学部学士論文(1970) 未発表.
8) Abrahams, P. W. et al.: The LISP 2 Programming and System, Proc. FJCC, Vol. 29 (1966) 661-76.
9) Barron, D., et al.: The Main Feature of CPL, Computer J. (July, 1963)