論　文

LISPインタプリタにおけるスタック技法とaリストの抑制法*

菱沼千明** 山下堅治*** 中西正和****

Abstract

This paper presents some usuful techniques for implementing LISP 1.5 interpreter. First, effective push down stack techniques are given in order to realize the recursive procedure peculiar to the function of LISP. Second, we show a method to curtail the excess *a*-list which is generated by universal functions when they evaluate iterative functions. The efficiency of those methods are also shown by comparing the mini-LISP, which is made to employ those methods, with other LISP 1.5 interpreters.

1. まえがき

LISP はリスト処理，記号処理に適しているだけでなく，ゲーム，人工知能，言語理論，グラフ理論などの分野に現れる組合せ理論的探索問題を解くのに極めて適しており，また解くためのアルゴリズムを開発するのに役立つ言語である．現在，LISP 言語には幾つかの異なる性格を持つものが開発され，実用に供されているが，その基本となるものは LISP 1.5[1] である．LISP 1.5 の特徴は，第一にアルゴリズムの再帰的定義を自由に許していること，第二にデータ構造が二進木リスト構造を基本とする最も簡単な形で表されていること，そして第三は処理アルゴリズムが LISP 1.5 自身で極めて簡潔に，しかし厳密に表されていることである．これらの特徴によって，LISP ではその処理アルゴリズムさえ知っていれば複雑な階層構造を有するアルゴリズムを極めて簡潔に表現することができ，またそれを具体的に解くことができる．

これらの特徴に対して，S式プログラムの見にくさ，再帰的関数定義に対する利用者の反感などは副次的な問題であるとしても，LISP の処理系に関しては次のような事柄が問題点として指摘できるであろう．

1) 手続きの再帰的呼出しを自由に許すためスタックを大幅に利用し，この結果実行速度が遅い．また，大きなスタック領域を必要とする．

2) リスト構造をデータとして持ち，ポインタ結合による二進木構造によって実現するため，単位領域当たりに格納できる情報が少ない．また，万能関数 (*eval*, *apply*) がその動作のために無駄にリストを消費する．

これらの欠点が LISP の実用化を遅らせた主な原因であろう．これに対して最近では実用のために，データ構造を複雑にしてスペース効率の悪さを改善した LISP や，言語仕様を変更して機能の向上に力を注いだ LISP などが開発されている[2)~5)]．しかし，どの LISP もすべて LISP 1.5 を基本としているにもかかわらず，LISP 1.5 自身の高速化の試みは余りなされていないようである．

小型のコンピュータ，特に主記憶容量の小さいものに対して LISP のような言語処理系を作るとき，多くの場合大幅な言語仕様の変更または制限を余儀なくされる．もしその仕様変更を少なくしようとすれば，仮想記憶装置を利用するといった方法も考えられるが，よほどうまく設計しない限り，処理効率の低下を招くだけでなく，処理系が複雑になり，仕様の変更やデバッギングを困難にするであろう．

本論文では，LISP 1.5 の外部仕様を変更することなく，処理アルゴリズムの実現方法を工夫することに

* Stack Techniques and Curtailment of *a*-list for LISP Interpreter by Chiaki HISHINUMA (Musashino Electrical Communication Laboratories, N.T.T.) Kenji YAMASHITA (Bridgestone Tire Co., Ltd.) and Masakazu NAKANISHI (Faculty of Engineering, Keio University)

** 日本電信電話公社武蔵野電気通信研究所

*** ブリヂストンタイヤ(株)

**** 慶応義塾大学工学部数理工学科

よりLISPインタプリタの高速化，スペース効率の改善を図る方法を提案する．更に，これらの方法を導入し作製した言語「ミニ LISP」がわずか4k語のミニコンピュータ上で極めて良い性能を持つことを示す．

本文の主題であるLISP 1.5インタプリタの改善法は先に述べた欠点を改善するものである．まずスタックについては，LISPの処理系の中心となる万能関数を始めとする組込み関数を実現するときにスタックができるだけ深くならない技法を示す．スタックが深くならないということは，それだけスタックを扱うための命令群の実行時間が少なくなり，結果的に高速になる．情報のスペース効率については，万能関数の動作のためだけに使われ捨てられるリストがあることに注目し，このようなリストの消費を抑えることを検討する．既にこの点に関しては，万能関数 *eval* の中での関数 *evlis* の使用を他の方法に置き換えることにより関数 *cons* による無駄なリストの消費を抑制する方法が考えられているが[6]，更に本文では定義関数 (*EXPR* function) の実行時に無駄に生成されるαリスト (association list) があることを示し，この無駄なリストの消費を抑制する方法を提案する．

2. スタック技法

手続きの本体の中で直接的に，あるいは間接的にその手続き自身を呼出して使用する方法は再帰的手続き (recursive procedure) と呼ばれ，LISPの処理系は再帰的手続きの集まりであると言ってよい．再帰的手続きの実現にはプッシュ・ダウン・スタック（以下は単にスタックという）が用いられる．従って，スタックの使用法が再帰的手続き，すなわちLISP自身の性能を大きく左右する．

再帰的手続きをスタックを用いて実現するには，まず手続きの呼出しの際に，局所的変数の値及び戻り先番地をスタックの一番上に格納し，手続きの終了まで保存しておく．そして元の手続きに戻るときは，スタック上の局所的変数とスタックの状態を元に戻し，スタックに格納してあった戻り先番地へ帰る．

以上の操作は再帰的手続きを実現するための最も一般的な例であるが，LISPのように再帰的呼出しをひんぱんに行う処理系では以上の原則に基づいていたのでは非常に大きなスタック領域を必要とするだけでなく，スタック操作のための時間を多く費すことになり能率が悪い．そこで，LISPの処理系を構成する関数群 (*SUBR*関数, *FSUBR*関数等) に特有な性質を利用し，これらの手続きの実現に以下のような技法を用いる．

1) スタックへの格納は呼出された手続き側で行う．

これは，スタックへ格納しなくとも処理が終了し，元の手続へ戻ることが多いからである．

2) スタックへは戻り番地を先に，次に引数の値の順に格納する．

これは，引数の値は参照後不必要になるのでスタックを順次押し下げることができるが，戻り先番地は一番最後に参照されるからである．

3) 呼出した手続きに制御を移し，再び制御と引数の値を復帰する必要のない手続き呼出しには，スタックにある元の引数の値を直接変更し，スタックを上昇させない．

上記3)のような形の参照を「分岐的参照」と呼ぶ(詳しくは**3.1**参照)．例えば次のような定義の *SUBR* 関数における $g[h[x]; y]$ と $f[h[x]; h[y]]$ の参照のように，f から g または f を直接呼出す場合，g と f の戻り先は既にスタックにある f の戻り先番地をそのまま利用することができ，その引数の値はスタック上の変数 x と y の値を直接書換え，制御を g, f へ移せばよい．

$$f[x; y]=[i[x]\rightarrow g[h[x]; y];\ j[x]\rightarrow f[h[x]; h[y]]]$$

ただし，関数 i, j, h の呼出しにはスタックを上昇させる．

4) あらかじめ戻り先番地を知り得るときは，スタックに戻り先番地を置かない．

5) 次に実行する手続きの終了後には二度と参照されることのない引数の値はあらかじめスタックを操作してスタック上から消しておく．

以上のスタック技法の詳しい例についてはここでは省略する．これらの技法を組み合わせて利用すると，スタックの上昇が抑えられ，スタック操作のための手数が少なくなるという効果があるだけでなく，捨ててもよいリストを積極的にスタック上から切り離すことになるので，ちり集め機能によって早い時期にリストが再生されるようになり，全体の処理能力を高められる．

3. αリストの抑制法

LISPプログラムを書く場合，同じ機能を有するプログラムであっても，プログラム機能を利用したほうが，プログラム機能を使わない再帰的関数定義 (McCarthy の条件式を用いた形の定義) によるものより

も高速で，しかも自由リスト・フルの状態になりにくいことがある．その理由は a リストの動きに違いがあるからである．

LISP 1.5 の万能関数は λ 式の処理を行う際，局所的変数が動的に宣言されるたびに a リストに新しい要素を加え，a リストを次々に伸ばしてゆくが，プログラム機能における関数 *setq* による代入作用は a リストの一部を置き換えるだけでリストを伸ばすことはしない．このためプログラム機能によるほうが a リストのために消費されるリストセルの数が少なくて済み，結果的にガーベージコレクタの動作回数が減り，その分だけ高速になるのである．

この章では，プログラム機能を使わなくとも上記のような不公平の起こらない処理方法について述べる．

3.1 分岐的参照と a リストの伸びる様子

次の2つの関数定義におけるような関数 g の参照のされ方を「分岐的な参照」と呼ぶ．

1) $f[x_1; \cdots; x_n]=g[y_1; \cdots; y_m]$

2) $f[x_1; \cdots; x_n]=[p_1\rightarrow e_1; \cdots; p_i\rightarrow g[y_1; \cdots; y_m]; \cdots; p_n\rightarrow e_n]$

ここで，$x_1, \cdots, x_n$ は変数であり，$y_1, \cdots, y_m$ はそれぞれ式 (expression) である．また，p_k, e_k $(1\leq k\leq n)$ はそれぞれ条件式の命題と式である．

分岐的参照が引き続いて起こる場合の a リストの増加の様子を次の関数を例に見てみよう．

$$reverse[x]=rev1[x; NIL]$$
$$rev1[x; u]=[null[x]\rightarrow u; T\rightarrow rev1[cdr[x]; cons[car[x]; u]]]$$

reverse にデータとして $(S_1\ S_2 \cdots S_n)$ を与える．いま，

$$x_i=(S_{i+1}\ S_{i+2} \cdots S_n)$$
$$u_i=(S_i\ S_{i-1} \cdots S_1)$$
$$x_n=NIL$$
$$u_0=NIL$$

と置き，*rev* 1 の定義式の各部分を

$$p=(LAMBDA\ (X\ U)\ q)$$
$$q=(COND\ ((NULL\ X)\ U)\ (T\ r))$$
$$r=(REV1\ (CDR\ X)\ (CONS\ (CAR\ X)\ U))$$

と置くと，$reverse[x_0]$ を評価するときの LISP 1.5 の万能関数の動きは次のようになる．

$$\begin{aligned}
&eval[(REV1\ X\ NIL); a_0]\\
&=apply[p; evlis[(X\ NIL); a_0]; a_0]\\
&=eval[q; a_1]\\
&=eval[r; a_1]\\
&=apply[p; evlis[cdr[r]; a_1]; a_1]\\
&=eval[q; a_2]\\
&=eval[r; a_2]\\
&\quad\vdots\\
&=apply[p; evlis[cdr[r]; a_n]; a_n]\\
&=eval[q; a_{n+1}]\\
&=eval[U; a_{n+1}]\\
&=u_n
\end{aligned}$$

ここで，a_0 は $reverse[x_0]$ を評価した時点での a リストを表し，また a_{i+1} は次式で与えられる．

$$a_{i+1}=nconc[((U.u_i)(X.x_i)); a_i]$$

上記の例から分るように，λ 式による *EXPR* 関数の評価を行う場合には，*eval* から *apply* を分岐的に呼出し，更に *eval* を分岐的に呼出すので，**2.** の 3) で述べたようにスタック上のパラメータを直接書換えると，スタックを上昇させずに済む．このようにすると，$i+1$ 番目の *rev* 1 の評価時，すなわち $eval[r; a_{i+1}]$ の実行時には a リストとスタックの関係は **Fig. 1** のようになり，スタック内のポインタのうち a_0 から a_{i+1} の間のリストを指すものは a_{i+1} だけである．従って，この時点で U と X の参照は a_{i+1} が指す a リストの始めから行われ，u_i と x_i がその値として得られる．ここで重要なことは，この時点で a リストの奥にある $(U.u_{i-1}), (X._{i-1}), x\cdots, (U.u_0), (X.x_0)$ などの以前の段階に作られたリストは決して参照されないということである．従って，これらのリストはすべて無駄な情報であり，生きているセルとして確保する必要はない．そこで，次にこのような無駄なリストを生じさせない方法について述べる．

3.2 a リストの抑制法

前節で示したように，関数 *rev* 1 の $i+1$ 番目の評価時の a リスト a_{i+1} は

$$a_{i+1}=((U.u_i)(X.x_i)(U.u_{i-1})(X.x_{i-1})\cdots (U.NIL)(X.x_0).a_0)$$

であるが，実質は

$$a_{i+1}=((U.u_i)(X.x_i).a_0)$$

でよい．更に，*reverse* による *rev* 1 の参照も分岐的な参照であるから，$reverse[x]$ の仮引数 x も *rev* 1

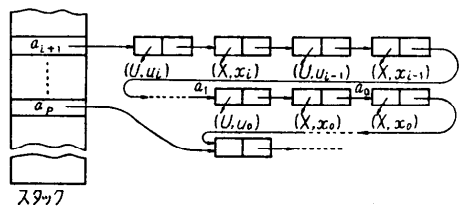

Fig. 1 A-list and the stack

$[x; u]$ のxと共通にできるので,

$$a_{i+1}=((U. u_i)(X. x_i). a_p)$$

でよい．ここで，a_p は以前に分岐的な参照によらない手続きの評価の際に作られたaリストを表す．従って，スタック内で a_{i+1} の1レベル下にあるaリストのポインタは a_p を指しているから，無駄なaリストの生成を防ぐためには，a_i から a_{i+1} を作るときに a_i から a_p の間のリストに対して *setq* に相当する操作を施せばよい．一般的にいうと，aリストの抑制法は次のようになる．

λ式による *EXPR* 関数の処理を行う際，仮引数と実引数の対リストをaリストに追加するのは，その時点のaリストからスタック内の1レベル下のaリストポインタが指す位置までに同じ仮引数名がない場合に限る．同じ引数名があるときは，その実引数を置き換える．

以上のaリストの抑制法は *apply* の定義式*のうちλ式の処理部を次のように定義したものと等しい．

```
eq[car[fn]; LAMBDA]→eval[caddr[fn];
        matchpair[cadr[fn]; args; a]];
```

ここで *matchpair* は次のように定義される．

```
matchpair[u; v; a]
  =[null[u]→[null[v]→a; T→error[F2]];
  null[v]→error[F3]; ask[car[u]; car[v]; a]
  →matchpair[cdr[u];cdr[v]; a];
  T→cons[cons[car[u]; car[v]];
        matchpair[cdr[u]; cdr[v]; a]]]
ask[x; y; a]
  =[null[a]→NIL; eq[a; $A]→NIL;
  eq[caar[a]; x]→rplacd[car[a]; y];
  T→ask[x;y;cdr[a]]]
```

ここで，$A はスタック内にある1レベル下のaリストポインタである．

なお，以上の定義はaリストの抑制法の最も基本的な表現であるが，λ式による *EXPR* 関数の処理は *eval* からいったん *apply* を介してすぐに *eval* を呼出すことになるので，aリストの抑制法は *apply* を介さずに *eval* の中に直接組入れたほうがよい．この結果，*evlis* によって消費される無駄なリストの発生も抑え得る．

さて，以上のaリストの抑制法は *EXPR* 関数が分岐的に参照される場合に限って有効であったが，更に *eval* の *SUBR* 処理部を多少変更すると次のような形をした *SUBR* 関数 s が分岐的に参照される場合にも s の最後の引数である *EXPR* 関数 g の評価時にaリストの抑制をすることができる．ただし，s はaリストを参照しない *SUBR* 関数である．

$$s[e_1; e_2; \cdots; g[l_1; l_2; \cdots; l_m]]$$

その方法は，*eval* の *SUBR* 処理部で，s の最後の引数 g を評価するために *eval* を再帰的に呼出す直前に，その時点のスタックのトップレベルにあるaリストを除去するものである．こうすると，g の評価時にはスタック内の1レベル下にあるaリストポインタは s が分岐的に呼出された時点の1レベル下のaリストを指すことになり，g が分岐的に参照された場合のスタックとaリストの関係と同じ状態になる．従って g の評価時にはaリストの抑制が行われる．

実際のLISPプログラムでは直接 *EXPR* 関数が分岐的に参照されるよりも，上の例のように分岐的に呼出された *SUBR* 関数の中に入った形で参照されることが多く，従ってaリストの抑制法は上記の変更をすることによってより効果的になるのである．

3.3 関数引数評価時の副作用と解決法

前節で述べたaリストの抑制法は関数引数を持つ *EXPR* 関数の評価を行うときにLISP 1.5と異なる結果をもたらすことがある．そのような副作用が起こるのは次の2つの場合である．

1) 自由変数を含むλ式が *EXPR* 関数の関数引数として与えられ，その関数が分岐的に参照され，しかもその *EXPR* 関数の関数定義においてλ式の自由変数名と一致する仮引数名が存在するような場合．

例えば次の関数定義で，λ式の中の自由変数 x と関数 h の仮引数 x とが一致しており，h が分岐的に参照されている．

```
g[x; y]=h[y; function[λ[[y]; cons[x; y]]]]
h[x; fn]=fn[x]
```

2) 自由変数を含むλ式が関数引数として与えられ，その本体**の中で分岐的に参照される関数があって，その関数の定義の仮引数にλ式の自由変数と同じ名前を持つものがある場合．

次の例では汎関数引数の適用の際に分岐的に参照される関数は h であり，自由変数と一致する仮引数は x である．

```
s[x; y]=g[y; function[λ[[y]; h[x; y]]]]
g[y; fn]=cons[fn[car[y]]; fn[cdr[y]]]
```

---

* 文献 1) p. 70 参照.

** λ式 $\lambda[[x_1; \cdots; x_n]; e]$ で，$x_1; \cdots; x_n$ を変数リスト部，e を本体 (body) と呼ぶ．

$h[y; x]=cons[y; x]$

まず 1) の場合の副作用を $g[A; B]$ の評価の様子を追跡して示す．この値は定義式から明らかなように $(A.B)$ になるはずである．

いま，

$$p=(LAMBDA\ (Y)(CONS\ X\ Y))$$
$$q=(FUNCTION\ p)$$

と置くと評価の様子は次のようになる．

$apply[G; (A\ B); NIL]$
$=apply[(LAMBDA\ (X\ Y)(H\ Y\ q)); (A\ B); NIL]$
$=eval[(H\ Y\ q); ((Y.B)(X.A))]$
$=apply[(LAMBDA\ (X\ FN)(FN\ X)); (B\ (FUNARG\ p\ ((Y.B)(X.A)))); ((Y.B)(X.A))]$
$=eval[(FN\ X); \alpha]$

ここで α は a リストの抑制をしないならば次のようになる．

$$\alpha=((FN.(FUNARG\ p\ ((Y.B)(X.A))))\ (X.B)(Y.B)(X.A))$$

a リストの抑制を行うと，α 内の X の対リストが置き換わるが，$FUNARG$ 関数の引数リスト内の X の対リストも同時に置き換わり，次のように変化する．

$$\alpha'=((FN.(FUNARG\ p\ ((Y.B)(X.B))))(Y.B)(X.B))$$

この結果，

$g[A; B]=(B.B)$

となり，期待される結果が得られないことになる．

この場合の副作用の原因は，上記の追跡より明らかなように，関数引数を持つ $EXPR$ 関数の分岐的参照時に a リストの抑制によって関数引数内の a リストも置き換えられてしまうことによる．従って，この副作用を回避するためには関数引数の評価方法を変更し，$FUNARG$ 関数の第 2 引数として，その時点の a リストの位置からスタック内の 1 レベル下の a リストポインタが指す位置までを複製し，更にその先のリストを継いだリストを用いればよい．すなわち，$eval$ 内の $FUNCTION$ の処理部を次のように変更する．

$eq[car[form]; FUNCTION]\rightarrow$
$\quad list[FUNARG; cadr[form]; copya[a; \$A]]$

ここで，$copya$ は次のように定義される．

$copya[x; y]=[eq[x; y]\rightarrow x;$
$\quad T\rightarrow cons[car[x]; copya[cdr[x]; y]]]$

次に，2) の場合の副作用を $s[A; (1\ 2)]$ の評価を追跡し調べる．この値は $((A.1)(A.2))$ となるはずである．評価の途中までを省略すると，

$apply[S; (A\ ((1\ 2)); NIL]$
$=cons[eval[(FN\ (CAR\ Y)); \alpha];$
$\quad eval[(FN\ (CDR\ Y)); \alpha]]$

となる．ここで α は

$$\alpha=((FN.(FUNARG\ (LAMBDA\ (Y)\ (H\ X\ Y))\ ((Y.(1\ 2))(X.A)))))\ (Y.(1\ 2))\ (X.A))$$

である．いま，

$$p=(LAMBDA\ (Y)\ (H\ X\ Y))$$

と置くと，$cons$ の 1 第引数の評価は次のように続く．

$eval[(FN\ (CAR\ Y)); \alpha]$
$=apply[(FUNARG\ p\ ((Y.(1\ 2))(X.A))); (1); \alpha]$
$=apply[p; (1); ((Y.(1\ 2))(X.A))]$
$=eval[(H\ X\ Y); ((Y.1)(X.A))]$
$=apply[(LAMBDA\ (Y\ X)(CONS\ Y\ X)); (A\ 1); ((Y.1)(X.A))]$
$=eval[(CONS\ Y\ X); ((Y.A)(X.1))]$
$=(A.1)$

このとき，最後の $apply$ から $eval$ に移る時に，自由変数である X の値が A から 1 に置き換わり，元の α は次のように変化する．

$$\alpha'=((FN.(FUNARG\ (LAMBDA\ (Y)(H\ X\ Y))\ ((Y.A)(X.1))))(Y.A)(X.1))$$

従って $cons$ の第 2 引数の評価値は

$$eval[(FN\ (CDR\ X)); \alpha']=(1.2)$$

となり，期待された結果と異なる．

この副作用の原因は，$(FUNCTION\ f_*)$ の形をした式の評価時に作られた $(FUNBRG\ f_*\ a)$ の a が f_* を評価するときに a リストの抑制作用によって変化してしまうことによる．従って，f_* の評価時に a がスタック上からポイントされるようにしておけば，この a に対して a リストの抑制は行われないから，副作用を防ぐことができる．具体的には，$FUNARG$ 関数から $FUNARG$ を取り去る部分で，$apply$ の呼び出しを分岐的に行わずに，以前の a リストをスタック上に残しておけばよい．

4. ミニ LISP

本章ではこれまでに述べたスタック技法及び a リストの抑制法を実際に計算機によって実現したシステム「ミニ LISP」について述べる*．

主記憶容量が最小の 4 k 語でも働くように，ミニ LISP では以下の手法を取入れている．

1) 性質リスト（p-list）をコンパクトにする．
2) 組込み関数の種類を必要最小限にする．
3) 処理プログラムを小さく実現する．

* このシステムの一部については既に文献 7), 10) で述べている．

Table 1 Comparison of the execution time LISP 1.5 interpreters

システム名		ミニ LISP		KLISP	MLISP	OLISP
		αリスト抑制	αリスト非抑制			
機械の名称		HITAC 10 (4k語)		TOSBAC 3400	MELCOM 7700	NEAC 2200
スタック(語)		α		744	2,500	29,000 セル
自由リスト(セル)		830-2α		3,936	25,000	
実行時間 msec (GC回数)	WANG *A*	71 (0)	66 (0)	40 (0)	330 (0)	43 (0)
	WANG *B*	402 (0)	433 (1)	260 (0)	630 (0)	123 (0)
	Bit *A**	1,175 (1)	1,170 (2)	560 (0)	データ無し	232 (0)
	Bit *B**	1,211 (1)	1,260 (2)	380 (0)	1,850 (0)	177 (0)
	Sort	58,044(135)	自由リスト不足	28,940 (11)	62,333 (3)	35,457 (9)
備考		* mapcar, mapcon は EXPR			* mapcar は EXPR	

(注) Bit *A*, Bit *B* はそれぞれ5要素，6要素のもの，Sort は 100 個のデータのもの．

4) スタックの消費を抑える．

5) *evlis* の使用を抑える．

6) αリストの生成を抑制する．

このうち性質リストは，アトム頭内に性質表示子を符号化し，それに続くリストの特定の位置に性質ポインタを置き，文字名は1文字6ビットで表してセル上に直接置いてコンパクトに表している．また，数値アトムは実装されていないメモリー空間を利用して表現している．なお，性質リストの実現方法でミニ LISP は LISP 1.5 と異なっている．

組込み関数，組込み定数などの種類は，従来の LISP の使用経験から良く使われるものだけを選び，*SUBR* 関数を 31 個，*FSUBR* 関数を 10 個，*APVAL* 定数を2個，その他のアトムを6個備えている．

処理プログラムは，基本関数やその他の手続きを閉じたサブルーチン形式で実現するなどして，可能な限り小さくしている．そのため幾らか実行時間が犠牲になっているが，スタック技法，αリストの抑制などの効果により，後で述べるように十分な性能が得られている．**Fig. 2** にミニ LISP のメモリーマップを示す．

ミニ LISP は更に次のような特徴を備えている．

1) 実装されている主記憶装置の容量を自動的に判断し，4k 語以上どのような大きさの機械でも実行できる．

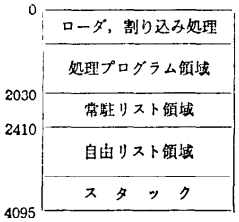

Fig. 2 Memory map of mini-LISP system

2) スタックとフリーリスト領域の境界は移動できる．これによりプログラムに応じてあらかじめスタックの大きさを設定することができる．

3) タイプライタからの指令を割込み処理する．従って実行中であっても種々の指示（実行中止，トレースの指示，解除など）を与えることができる．

4) エラー番号はエラーの発生番地によって示す．

5. ミニ LISP の性能

Table 1 に5つのテストプログラムをミニ LISP で実行した場合の実行時間とガーベージ・コレクションの回数を示す．ここで，αリストの抑制を行った場合と行わない場合の結果を比較している．更に，**Table 2** には同じテストプログラムの評価に要したスタックの最大深さを示す．テストプログラムは昭和 49 年記号処理シンポジウムにおける LISP コンテスト[8]で用いられたものであり，同時に提出された他の3つの LISP 1.5 インタプリタによる実行結果をも比較のため示した．なお，テストプログラムの詳細はここでは省略する（文献8)または9)を参照)．

Table 1 から明らかなように，ミニ LISP は4k語という小記憶容量にもかかわらず自由リストとスタックが大きい他の LISP に劣らないすぐれた性能を示している．これは本文で示したスタック技法とαリスト抑制の効果とみることができる．αリストの抑制を行った場合には，αリスト上の同一引数名の探索のため

Table 2 Required depths of stack

	ミニ LISP	KLISP	MLISP	OLISP
WANG *A*	52	63	70	43
WANG *B*	52	63	164	75
Bit *A*	73	61	データ無し	141
Bit *B*	89	61		94
Sort	514	414	1,324	817

のオーバーヘッドが生じるが，無駄なリストが消費されないため，結果的にガーベージ・コレクションの動作回数の減少による効果のほうが大きい．また，aリストの抑制をしたときには，100 個のデータのソーティングプログラムの実行が可能であることからも分かるように，記憶容量の小さな計算機ではaリストの抑制法は極めて有効である．

5. むすび

LISP 1.5 インタプリタを計算機上に実現するときの技法として，処理系の中心となる万能関数などの組込み関数の動作にスタックができるだけ深くならない方法について述べ，更に *EXPR* 関数の評価時にはaリストが無駄に生成されることがあることを指摘し，その無駄なaリストの消費を抑制する方法を提案した．ここで述べた方法はミニコンピュータのように記憶容量の限られたものに LISP インタプリタを実現するのに適している．実際にわずか 4k 語のミニコンピュータでも他の LISP インタプリタに劣らない性能を持った処理系を実現することができた．

LISP 1.5 は記号処理用の言語としては必ずしも実用的に最良の効率を持つものではないが，数学的なモデルとして採用できるような簡潔で明確な処理系の定義を持ち，他の LISP の基本をなしている．本論文で提案した内部処理効率の改善法は他の LISP にも当然導入できるものであり，このような意味から，LISP 1.5 の言語仕様を維持したまま実現方法を工夫して処理効率を上げるための研究は今後更に進められるべきであろう．

終りに本研究の機会を与えられた慶応義塾大学工学部森真作博士に深謝致します．

参考文献

1) J. McCarthy, et al.: LISP 1.5 Programmer's Manual, The MIT Press (1962).
2) 後藤: HLISP, bit, Vol. 6, No. 10 (1974).
3) 竹内，奥乃: A List Processor LIPQ, 2nd UJCC Proc. (1975).
4) L. H. Quam: Stanford LISP 1.6 Manual. SAILON 28.4, Stanford Univ. (1970).
5) W. Teitelman: INTERLISP Reference Manual, Xerox (1974).
6) 中西: LISP の処理系製作の技法について，記号処理シンポジウム報告集，p. 2 (1974).
7) 山下，大館，菱沼: ハードウェアスタックによる LISP の高速化，情報処理学会計算機アーキテクチャ研究会資料 75-10 (1975).
8) 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 (1974).
9) 中西: LISP コンテスト，bit, Vol. 7, No. 3 (1975).
10) 中西，菱沼，山下，酒井: A design of LISP Interpreter for minicomputers, IFIP TC. 2 Working Conference on Software for Minicomputers, pp. 49～55, Kesthely, Hungary (1975).

(昭和 50 年 12 月 23 日受付)
(昭和 51 年 3 月 17 日再受付)