高速補助記憶装置を使用したミニコン用 LISP 1.6 システム*

長尾　真** 中村和雄**

Abstract

The implementation of LISP 1.6 (LIST processing language 1.6) for a minicomputer (TOSBAC-40C) is described.

The implementation of a full-power LISP processor requires large program area and large free cell space, because the LISP processor must be able to run a large program efficiently. As our minicomputer has a small main memory, it is required to attach a high speed bulk core memory (512 kB) as a secondary memory. We used the secondary memory as a virtual memory by the software paging mechanism based on the LRU algorithm. Upon these address spaces, we can use maximum 64 k LISP cells.

As a processor has the ability to swap out any list expression into the secondary memory in the form of S-expression (Symbolic expression), the user can run a fairly large program that requires more cells than 64 k cells for the running. Many other ideas are employed in this processor, such as the data type of the pointer is determined by the address computation (hence the conventional data type flags in the cell are eliminated completely), the shallow binding mechanism is employed as the variable binding, the compaction and linearization of the cells are taken place at the garbage collection, and the processor works under the disk operating system.

As this processor has so many features, it is easy to use, and a large LISP program is runnable efficiently.

1. はじめに

人工知能 (AI) の分野におけるプログラム言語としては，従来からよく使用されて来た LISP[1]，SNOBOL[2]，IPL-V[3]，COMIT[4] や，さらに制御構造に柔軟性を持たせた PLANNER[5]，CONNIVER[6]，MLISP 2[7]，ECL[8] など，バックトラッキング，パターンマッチング，並列処理などの能力を備えた言語も使われている．

しかし LISP のようなインタープリタ言語は TSS のもとで実行させると，大型計算機の場合であっても能率はあまりよくない．そして領域も十分にとれるとはかぎらない (例えば 100 k セル～200 kW)．そこで我々は大規模な LISP プログググラムを実行させるためには大容量のセルをとることのできる専用計算機が必要であるとし，そのような目的にあうよう LISP システムを種々開発して来た[10]．しかしディスクメモリの仮想記憶方式のため，ワークエリアは十分であったが (最大 300 k セル以上)，その処理時間が非常に遅い欠点があった．

そこで速度をあげるために，今度高速補助記憶装置として，コアメモリ 512 k バイトを備え，このメモリ (以後バルクメモリと呼ぶ) を補助記憶とした仮想記憶方式で，仮想記憶の欠点を補うべく様々な工夫をこらした LISP 1.6 プロセッサを作成した (以後この LISP を BLISP 1.6 と呼ぶ)．

* LISP 1.6 system for a minicomputer using high speed auxiliary core memory, by Makoto NAGAO and Kazuo NAKAMURA (Faculty of Electrical Engineering, Kyoto University).

** 京都大学工学部電気工学第II学科

2. BLISP 1.6 プロセッサの構成

2.1 ハードウェア構成

BLISP 1.6 を実現した計算機は TOSBAC-40C でこの計算機に接続されている周辺機器のうちこのシステムに関係している機器の構成は **Fig. 1** の様になっている. Fig. 1 のバルクコアメモリは主として LISP を高速化するために導入したもので, これを補助記憶装置としたソフトウェアによる仮想記憶方式の LISP システムを構成した. 現時点のバルクコアメモリの性能を **Table 1** に挙げる.

なおこのバルクコアメモリは最大 1 Mバイトまで拡張可能なランダムアクセスメモリであり, LISP の様に, ポインタでリンクしたセル同志にあまり局所性がないデータ構造を処理する場合にも, 後に述べる様な種々の工夫をこらせば, 仮想記憶方式でシステムを構成しても十分実用性があると考えられる. このメモリの詳細は文献 11)に詳しい.

また, BLISP 1.6 はディスクオペレーティングシステム (DOS-40) のもとで働く様に構成されており, DOS-40 のサブシステムの一つになっている. Fig. 1 のディスクメモリ #1には DOS エグゼクティブ, サブシステム, ライブラリ, 及びユーザファイルが登録されており, #2は全領域を DOS の管理の元でファイルとして使用できる.

2.2 ソフトウェア

BLISP 1.6 の実行時のメモリマップは **Fig. 2** の構成になっており, ミニコンの主記憶装置の不足を補うため Fig. 2 の様なオーバーレイ構造で構成される. 詳細は次章以降で説明する.

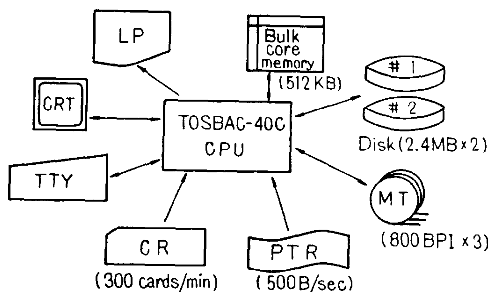


Fig. 1 Hardware Configuration

Table 1 Some characteristics of the bulk core memory

memory size (maximum)	1 MB
cycle time (minimum)	1.6 μsec
access time	0.8 μsec
read/write	40 bits parallel

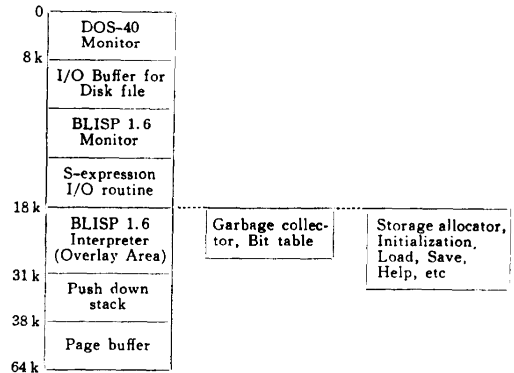


Fig. 2 Main memory map in execution of the BLISP 1.6

3. 仮想記憶の方式

BLISP 1.6 で仮想アドレスがつけられている部分は, リスト構造の領域であり, ポインタが仮想アドレスで表現されている. リスト構造の基本構成単位であるセルは 4 バイト (32 ビット) で構成されており, 従ってポインタは 16 ビットで構成されている. このポインタを使用することにより, 64k セルまでアドレッシングが出来る様になっている.

仮想記憶を実現するため BLISP 1.6 では文献 10) と同様にページング方式を採っている. ポインタとしては 1 セルの先頭をアドレス出来ればよいから, 仮想記憶上の番地は 1 セル＝4 バイト単位で番地付けされている. 1 ページの大きさは 1k＝1024 バイトに分割されている. 従って 1 ページには 256 セルを入れることができる.

論理アドレスの表現は, ページ番号 (PN) とページ内相対アドレス (RA) とから構成される. 上に述べた様に, RA の部分は 256 セルのいずれかをアドレッシングできればよい. 従って, RA として 8 ビット使用し残りの 8 ビットで PN を表現する.

ガーベッジコレクションのためのビットはガーベッジビットのための専用テーブルをもうけ, 効率化をはかっており, セルの中にガーベッジビットはない. さらに仮想記憶を実現するにはマッピングの方式と, ページフォールトが生じた時のページのリプレースメントを高速に行わなくてはならない. この様な要求を満たすために, 全ての 256 ページ分のエントリを持つページテーブル(PT)と, 完全な LRU (Least-Recently-Used) アルゴリズムを実現するための LRU-Linked-

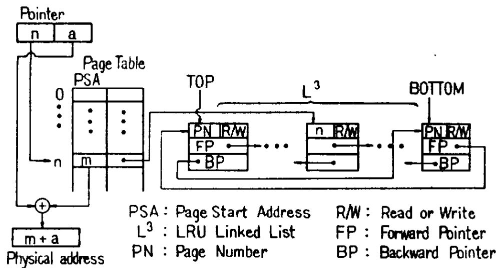


Fig. 3 Mapping from virtual address using the page table and L^3

List (L^3) を備えている．この L^3 を構成するブロックは主記憶に入っているページ数の分だけ用意すればよく，現在 26 個用意されている．また各ブロックはページ番号，read/write フィールド，前向きポインタ，後向きポインタから構成されている．PT の各エントリには対応するページが主記憶に存在する場合にはそのページの開始番地と，L^3 の対応するブロックへのポインタが入っている．

LISP の場合も，個々の関数を見れば，ある程度局所性があるものと考えられ，この意味から種々のリプレースメントアルゴリズムのうちで LRU アルゴリズムを採用した．上述の情報を使うマッピングを **Fig. 3** に示す．

Fig. 3 で TOP 及び BOTTOM の二つの特別なポインタによって，TOP から前向きにブロックを辿ることによって，最も最近に使用されたページから，最も古いページへと辿ることができる．また，PT 上の PSA 部が 0 のものは対応するページが主記憶上にない事を示しているため，この様なページをアクセスした場合はページフォールトによって，スワッピングが行われる．スワッピングの対象となるページは，BOTTOM によってポイントされているブロック内の PN によって示され，また R/W フィールドによって，スワップアウトすべきかどうかチェックされる．いかなるページにアクセスした場合でも LRU を忠実に反映するために L^3 の中の FP, BP, TOP, BOTTOM 及び適当な部分が更新される．

実際の情報のスワッピングは Fig. 2 のページバッファ・エリアとバルクコアメモリの間で行われる．なお，バルクコアメモリの I/O の平均所要時間は 1 ページ（1k バイト）当り 1.2 msec であり文献10) のディスクメモリを使用したシステムに比べ約 20〜30 倍の速度向上が得られている．

4. データ構造の内部表現

4.1 データ構造の実現法

LISP では，データ宣言が不要である代償として，実行時にデータタイプのチェックが行われ，他のプログラム言語に比して処理時間のネックになっている．この際，従来の LISP プロセッサの多くはタグやフラッグをデータ構造内に持たせることにより，データの区別を行っていた．この方法では少なくとも 1 回，セルへのアクセスが必要であり，それだけ処理時間の損失になる．BLISP 1.6 ではこの点を改善し，ポインタの表現する数値により，アドレス計算で処理が行えるため BLISP 1.6 ではデータタイプを示すタグやフラッグ類は一切使用していない．

上に述べたデータ構造を表現するために，リスト構造をストアするためのバルクコア・メモリ内は **Fig. 4** の如く割り付けられている．

Fig. 4 のグローバルネーム FRINIT, …, FWEND は，各領域の先頭を示すポインタ表現になっている．また BLKEND は実装されているバルクコアメモリの最終アドレスが入っている．このバルクコアメモリは最大 1 M バイトまで拡張可能であり，実装されている最終アドレスは BLISP 1.6 システムの Top-of-Core-Search 法により自動的にセットされる．

また，各領域の大きさは可変構造になっている．標準割り付けとしては，Fig. 4 の絶対アドレスで表された割り付けがされる．なお，FRINIT と FWEND は固定されており，FRINIT <n または FWEND≧n が成立するポインタ n に関しては，インタプリタ内で，数値そのものと解釈される．従って −1024〜+1023 までの整数値を表現するためには，余分なフリーセルや，数値セルを全く必要とせず，この範囲内の

Pointer	Area	Address
		0
	OBLIST Area	
FRINIT		4 k
	Free list Area	
STINIT		200 k
	String Area	
INTINT		202 k
	Integer Area	
FLINIT		203 k
	Float Area	
AHINIT		204 k
	Atom-header Area	
FWINIT		236 k
	Full word Area	
FWEND		252 k
	Control data Area	
		254 k
	Extension Area	
BLKEND		512 k

Fig. 4 Memory organization of the bulk core memory

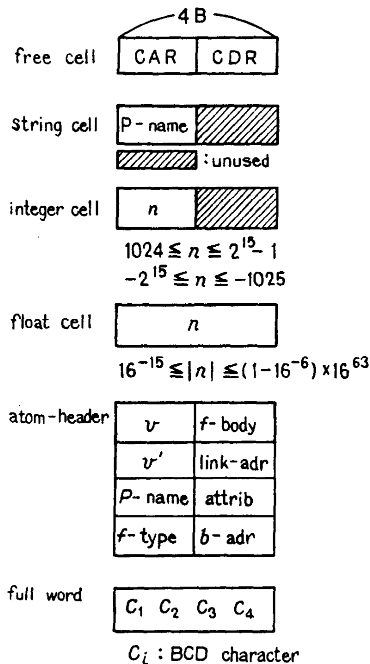


Fig. 5 The data structures in the BLISP 1.6

数値を比較するためには EQ を使えばよい．LISP においては大きな数値や浮動小数点数を使うことは比較的少なく，この意味でも余分なセルの消費を防ぐことと，数値演算の高速化に役立っている．このリスト構造が入らない空白の部分を利用して，Fig. 4 のように OBLIST 及び後述する SAVE, LOAD 等における管理情報を収めてある．残り 254 k 以上の拡張エリアは特殊な使用目的にとられており，これも後述する．なお OBLIST はハッシュテーブルになっており 2048 個のエントリを持ち，conflict は direct-chaining-method を使用している．

4.2 データ構造の内部表現

各領域の構造は **Fig. 5** の様になっている．データタイプとしてストリングを導入したのは，文字単位の処理をある程度可能にし，SNOBOL の持つ能力をいくらか LISP にも持たせるためである．

integer-cell（4 バイト）には実際の整数値が入る．但し，この部分に入る数値は上述した $-1024 \leq n \leq 1023$ の範囲外の整数値だけで，この範囲外の値はこの integer-cell を使用して表現される．floating cell には 4 byte の浮動小数点が入っている．BLISP 1.6 の数値演算関数は，数値タイプの変換を自動的に行って演算を行う．

atom-header は 16 バイトで構成されており，Fig. 5 において，v は value-cell として使用される．また Stanford 版 LISP 1.6[12] の P-list に変数値を持たせるメカニズムよりも変数へのアクセスは高速に行われ，shallow-binding が実現されている．また変数と実パラメータとの結合の際にはその時点の変数値が SPCPDL と称するグローバル変数に保存されて行き，この SPCPDL は LISP 1.5 の a-list の形式になっている．この SPCPDL を使うことにより，FUNARG 機構を実現する際に実質的な環境（変数の結合状況）が回復される．

変数値の保存をする場合，プッシュダウンスタック (PDS) を利用する方法も考えられるが，BLISP 1.6 では PDS を主記憶上にとっているため PDS エリアが少なく，スタックオーバーフロウを極力避ける必要性から，PDS を利用する方法をとらなかった．なお，現在の所，PDS エリアとして約 7 k バイト=3500 セルの分だけ確保されている．また PDS にはポインタとプログラムリンクアドレスの両者がスタックされるが，BLISP 1.6 では，PDS エリアを有効に使用する意味で両方向から使用する様にし，一方向からポインタをプッシュダウンし，他方向からプログラムリンクアドレスをプッシュダウンする様にしている．次に atom-header 内の f-body の部分にはユーザ関数の定義体への pointer が入る．

v' の部分には，BLISP 1.6 の持っている定数値が入っている．この部分は，ユーザーが誤って使用したために定数値がこわされた場合，元の定数に復元するための保存用として使用される．

atom-header 内の link-adr 部には，機械語関数へのリンクアドレスが入っている．

P-name 部には，アトムのプリントネームリストへのポインタが入っている．

attrib 部にはユーザレベルで P-list 操作が行える様にするために，属性名一属性値が対になったリストの根が入ることになる．

f-type 部の詳細は **Fig. 6**（次頁参照）の構成になっている．Fig. 6 の OVR-N は機械語関数のオーバーレイを行うための情報が入っている．現在のシステムではオーバーレイで関数が実行に移されることはないが，将来の機能拡張のため用意してある．t の 1 ビットは関数のトレースが指示された場合ビットが立てられる．type 部の 5 ビットによって関数タイプの区

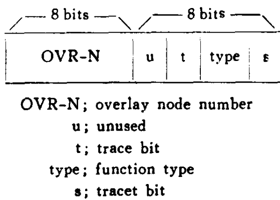


Fig. 6 The detailed figure of the f-type part in the atom-header cell

別を行う様になっている．現在関数タイプとして，ユーザが定義出来る関数4種類 (EXPR, FEXPR, MACRO, LEXPR) と機械語関数3種類 (SUBR, FSUBR, LSUBR) が用意されている．また Fig. 6 の s の部分はラムダ変数またはプロッグ変数の値をトレースする様に指示された時にビットがセットされる．

4.3 関数定義体のバルクコアメモリへの掃き出し

LISP 上で仕事をする場合，リスト構造領域が大きいに越したことはないが，現実には，様々な理由で制限せざるを得ない．特にミニコンでは制限が厳しく，これをある程度解決する一手段として BLISP 1.6 では仮想記憶方式を採用した訳であるが，これでもまだ十分とは言えない．すでに数多く開発されて来ている，AI の分野における実用的な LISP プログラムは数千～数万行の大きさを持ち，そのプログラムのみで，数十キロセル程度（有名な Winograd のシステム[13]ではプログラムのみで 50k セル程度）専有し，残りのフリーリストを使って仕事をすることになる．この種の問題点は，AI の分野ではすでに深刻な問題点になっている．将来この分野での実用的プログラムでは，250k ワード (74 ビット/ワード) 程度のメモリが必要になると予測されている[14]．

しかしどの言語で書かれたプログラムであっても，実際のワーキングセットの大きさは全体の数分の1 (1/2程度) であろうと考えられる．このことを考えれば，いつ使用されるかわからないプログラムを常に主記憶に保持しておくのはいかにも非能率的である．そこで，BLISP 1.6 では，ユーザの指示によってプログラムを外部記憶装置に掃き出しておき，実際に使用される段階になって初めて，実行可能な形に変換して実行できる機能をつけ加えた．この掃き出しに使用される情報の一つとして前述の Fig. 5 の atom-header 領域の b-adr 部が使用される．BLISP 1.6 ではこの掃き出しは S-表現 (Symbolic-Expression) イメージで行う．ここで S 式を掃き出す為に使用する外部記憶装置として，バルクコアメモリを使用し，Fig. 4 の 254k 以上の拡張エリアをこの目的に使用している．ある関数が拡張エリアに掃き出されている場合 Fig. 5 のその関数名に対する atom-header の b-adr 部には，バルクコアメモリ上のアドレスがストアされていて，そのアドレス以降に関数の定義体の S-式が文字列の形で入っている．さらにその関数の atom-header の f-body 部は切断されている．この様に関数の定義体をバルクコアメモリに掃き出すための関数として，LISP 関数 SWAP が用意されている．SWAP のシンタックスは (SWAP f-name-list flag) の形式で与えられる．f-name-list は複数個の関数名のリストである．flag は NIL 又は T を指定する．この flag は関数の定義体が，すでにバルクコアメモリに掃き出されている場合の処理を指定するもので，T を指定すれば，定義体が掃き出されているかどうかにかかわらず強制的に掃き出され，対応する atom-header の f-body の切断，b-adr のセットがされる．NIL を指定すれば，すでにバルクコアメモリ上に関数定義体の文字列が存在する場合には，対応する atom-header の f-body 部を切断するのみで，定義体を掃き出すことはしない．一方まだ掃き出されていない場合には，定義体を掃き出し，f-body 部を切断し，b-adr 部をセットする．

例えば，LISP 関数 append を **Fig. 7**(a) の様に定義して SWAP により掃き出した場合その前後では Fig. 7 (b), (c) の様になる．

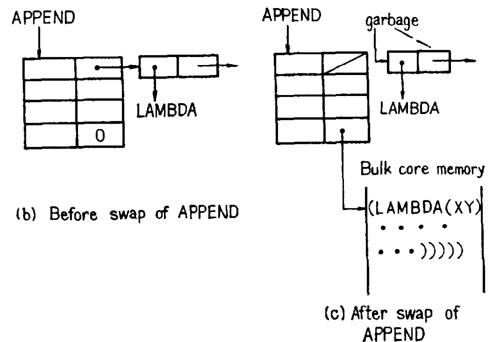


Fig. 7 An example of swapping of a function

Table 2 Some features of the BLISP 1.6 system

SUBR	123
FSUBR	31
LSUBR	11
APVAL	22
etc.	7

(a) The numbers of the built-in functions and constants in the BLISP 1.6

Free cell	49 kcells
String cell	512 cells
Integer cell	256 cells
Float cell	256 cells
Atom-header cell	2 kcells*
Full word cell	4 kcells
Stack area	3.5 kcells

*; 1 cell=16 B

(b) The standard size of the respective area

```
pair [x; y]=[null [x]→NIL;
             null [y]→NIL;
             T→cons [cons [car [x]; car [y]];
                     pair [cdr [x]; cdr [y]]]]
```

Fig. 8 The definition of pair [x; y]

バルクコアメモリに掃き出された関数をプログラムの実行時に呼んだときは，インタープリタが自動的に読み込み，f-body 部に定義体への根をセットする．

5. LISP インタープリタ

BLISP 1.6 言語仕様は **Table 2** (a), (b) の様になっている．

インタープリタ内で，関数に実引数を渡す際，全体をリストにして渡す様になっているが，このリストを構成する最高レベルのフリーセルは，すでに消滅するセルであり，この部分をスタックなどを利用することによって，防ぐことができる．例えば pair [x: y] を **Fig. 8** の様に定義し，x, y として n 個の要素のリストを与えた場合，防止しない時に消費されるセル数を C，防止した時のセルを C_s とすれば $C-C_s=10n+1$ セルの節約ができ，ガーベッジコレクションの回数も減らすことができる．

以上の点を考慮して，インタプリタが作成されている．**4.** で述べた様に，データタイプはポインタのアドレス計算で行えるので，この部分もインタープリタの高速化に寄与している．

6. ガーベッヂコレクション

LISP に限らず，動的に記憶場所を割り付けて行くプログラム言語やシステムでは，くずになった記憶場所を再生するためにガーベッジコレクションが必要になってくる．LISP においては，特にこのガーベッジコレクションをいかに高速，かつ有効に行うかがシステム全体のパーフォーマンスに大きく影響する．BLISP 1.6 では仮想記憶方式を使用している関係上，この問題は深刻である．BLISP 1.6 でガーベッジコレクションを行う際，いくつかの問題を生じたが，それらは次の点である．

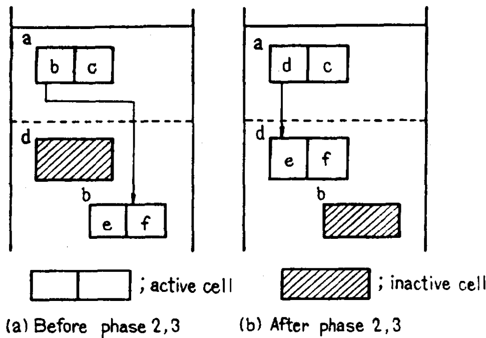
(1) 仮想記憶方式をとり入れている関係上，セルへのアクセスを出来る限り少なくしなければならない．

(2) **3.** で述べた様にポインタは 16 bit で構成されているため，セル内に余分のビット（ガーベッジ用マークビット）がない．

(3) ガーベッジコレクションのあと，アクティブセルがバラつくと，仮想記憶方式の LISP では非常に効率が悪くなる．

そこで(1)，(2)の問題点を解決するために，マークビットとしてビットテーブルを使用している．また(3)の問題点については，詰め合せ(compaction)[15]，及びフリーリスト領域の cdr 方向への物理的な直線化 (linearization) を行うことにより解決している．ガーベッジコレクション[16),17)]に限らず，仮想記憶方式においては，直接セルを参照することを出来る限り避けねばならない．詰め合せを行う場合，アクティブセル・マーキング，詰め合せ，アドレス調整の3つのフェーズを実行しなければならない．第2フェーズの詰め合せではアクティブなセルをポインタアドレスの若い方へ詰め合す．第3フェーズでは，移動したセルを参照している他のセルのポインタを移動先のポインタに変更する．第2，3フェーズの前後を図式的に示せば，**Fig. 9**(a) (b) (次頁参照) の様になる．Fig. 9 (c) は詰め合せのアルゴリズムのメタ表現である．自由変数 init, end は，領域の先頭と，最後のセルへのポインタを示す．pgreat [x; y] は与えられたポインタを単なる数と見なして $x>y$ の真偽を調べる述語関数で，search の定義中の markp [x] は x がアクティブセルの場合真になる関数である．また，incr[x]，decr [x] はそれぞれセル x の直後及び直前のセルのポインタを値とする関数である．また phase 3 の定義中の pbgreat [x] はポインタ x が属する領域の bot と比較し，$x>$bot の真偽を調べる．

また直線化を行う場合，詰め合せを実行後，リンクしているフリーリストを cdr 方向に直線的に再配置を行う．この前後を図示すれば **Fig. 10** (a), (b) (次頁参照) の様になる．1個のリストを直線化するアルゴリズムは Fig. 10 (c) で与えられる．但し直線化を開始する前にビットテーブルをクリアしておく必要があり，また実際にはアトムに付属している属性リストなども直線化の必要があるが，この部分を簡単化して示してある．Fig. 10 (c) 中の mark [x] は x に対応

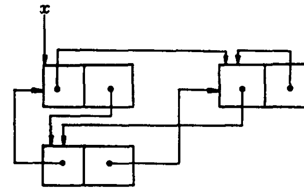


(a) Before phase 2,3 (b) After phase 2,3

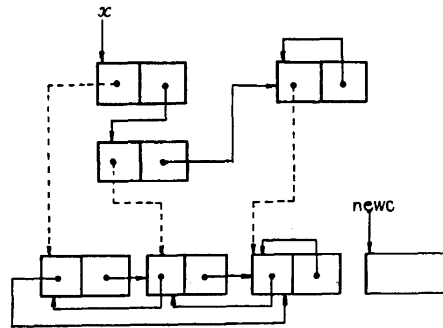
```
phase 2 [ ]=prog [[top; bot]; top: =init; bot: =end;
         LP search [top; bot];
            [pgreat [top; bot]→return [phase 3 [ ]]];
             rplaca [top; car [bot]]; rplacd [top; cdr [bot]];
             rplaca [bot; top]; go [LP]]
phase 3 [ ]=prog [[x]; x:= init;
         LP [pgreat [x; bot]→return [GC-END];
             pbgreat [car [x]]→rplaca [x; caar[x]]];
            [pbgreat [cdr [x]]→rplacd [x;cadr [x]]];
             x: =incr [x]; go [LP]]
search [x; y]=[pgreat [x; end]→error [STORAGE-EXAUSTED];
           pgreat [x; y]→prog[[ ]; top: = x; bot: = y];
           markp [x]→[markp [y]→search [incr [x]; y];
                      T→search [incr [x]; decr [y]]];
           markp [y]→prog [[ ]; top: =x; bot: =y];
             T→search [x; decr [y]]]
```

(c) The algorithm for the compacting garbage collector

Fig. 9 The compacting garbage collector



(a) Before linearization



(b) After linearization

```
linear [x; a]=[atom [x]→[a→x;
                  T→rplacd [decr [newc]; x]];
         markp [x]→[a→car [x];
                  T→rplacd [decr [newc]; car [x]]];
         T→prog [[ ]; alloc [x]; linear [cdr [x]; NIL];
             return [rplaca [car [x];
                          linear [caar [x]; T]]]]]
alloc [x]=prog [[ ]; mark [x]; rplaca [newc; car [x]];
             rplaca [x; newc];
             return [rplacd [newc; newc: =incr [newc]]]]
```

(c) The algorithm for the linearization in the cdr direction

Fig. 10 The linearizing garbage collector

するビットテーブル中のビットを立てる関数であり，グローバル変数 newc はくずになっているフリーセル領域の先頭を示す．

その他に詰め合せや直線化を行わない通常のガーベッジコレクションもあり，いずれのモードで行うかをユーザが指定出来る．なお直線化を行う場合，通常のガーベッジコレクションに比べて約2倍程の時間を要する．

7. 入出力機能その他

外部環境との interraction の道具としては Fig. 1 に示した任意の I/O 器機が使用できる．DOS-40 ソースプログラムエディタのもとで作成されたディスクファイルや磁気テープ (MT)，紙テープからソースプログラムの入力や結果の出力が自由に行える．また実行結果を保存するための指令として，LOAD, SAVE, などがあり，パーマネントファイルとして，MT やディスクファイルに貯えておくことができる．

4. で，バルクコアメモリ内の各領域は可変に出来ることを述べたが，このために初期化するコマンドが設けられており，各領域を1 kバイト単位で割り付けを行なえる．ただし，BLISP 1.6 プロセッサがジョブを開始した時点では標準サイズで初期設定が行われている．

8. 評 価

テストプログラムとして，1974 年夏に行われた記号処理シンポジウム[9]で出題された WANG algorithm，2進木発生プログラム BITA, BITB, 及びソーティングプログラムを実行した結果を **Fig. 11** (次頁参照) に示す．Fig. 11 中，( ) 内は詰め合せをしないガーベッジコレクションで行った結果である．処理時間は 50 msec ごとのクロックで測定し，同一プログラムを何回か実行した平均値である．

我々は LISP の上に言語処理プログラミングシステム PLATON を作ったが，これを用いて日本語の文章分析プログラムを書いている．この文章分析プログ

Program name		Execution time (msec)	Average times of garbage collection
WANG	A	100 (100)	0
	B	600 (600)	0
BITA	6	5,532 (5,532)	0
	7	22,432 (21,355)	0.3
	8	75,925 (73,242)	0.8
BITB	6	1,495 (1,495)	0
	7	4,345 (4,345)	0
	8	15,825 (15,358)	0.1
SORT	60	92,825 (91,275)	1.0
	80	164,200(153,750)	1.7
	100	246,850(235,750)	2.5

Fig. 11 The results of executing some programes

ラム[18]を働かせ簡単な文章を入力した場合の処理時間を測定した．このプログラムのサイズは，フリーセル 22,848 セル必要としている．またワークエリアとして使用可能なフリーセルは 20,672 セルであった．詰め合せを行うシステムと行わないシステムとで測定した結果，それぞれ 189,100 msec，174,700 msec であった．この時いずれのシステムでもガーベッジコレクションが2回行われている．なおガーベッジコレクションのみに要する時間はそれぞれ 25,600 msec，13,350 msec であった．

9. おわりに

仮想記憶方式が LISP に不向きであることは確かであるが，十分な対策を施せば必ずしも不可能でなく，実際にこのシステムを使用してみると，入出力機能の豊富さや，ファイルの使用などが可能で，インタラクティブな機能を持ち，非常に便利なシステムになっている．また LISP そのものの機能も拡張されており，関数タイプ，組み込み関数の増加，インタプリタの機能拡張などにより，プログラムも組み易くなっている．また速度の点に関しても Fig. 11 と文献9)の他の LISP のデータと比較しても決して劣っていないことがわかる．

Fig. 11 のプログラムでは1回のガーベッジコレクションに詰め合せを行う場合 10～12 秒要している．さらに 8. のデータよりわかる様に文章分析プログラムを働かせた場合 20～25 秒要している．このプログラム全体の大きさは 40k セル程度必要としているが，仕事の最初にすべての関数を掃き出しているため，実際のワーキングセットは 20k セル程度である．またこのデータではガーベッジコレクションにおいて，詰め合せをしない方が，処理時間が速いという結果が出ているが，これはプログラム自体が単純で，アクティブセルが方々に散ばるということが少ないため，詰め合せを行っても，行わなくても後の処理にそれほど影響を及ぼさないためと思われる．このガーベッジコレクションのオーバヘッドを少なくするために，6. で述べた様に，どのモードで行うかをユーザが指定出来る．

また 4. で述べた掃き出し領域が消費された場合もガーベッジコレクションを働かす必要があるが，掃き出し領域が十分あるので（258k バイト），現在はインプリメントしていない．

最後に，有益な議論を頂いた辻井助手，修士課程の吉井氏を始め研究室一同の諸氏に感謝します．

参 考 文 献

1) J. McCarthy et al.: "LISP 1.5 Programmer's Manual" MIT Press (1962).
2) D. J. Farber et al.: "The SNOBOL3 Programming Language" BSTJ (1966).
3) A. Newell (ed.): "Information Processing Language-V Manual" Prentice-Hall (1963).
4) V. H. Yngve: "COMIT Programming" MIT Press (1966).
5) C. Hewitt: "PLANNER: A Language for Manipulation Models and Proving Theorems in a Robot" MIT Project MAC, AI Memo No. 168.
6) D. V. McDermott and D. J. Sussman: "The CONNIVER Reference Manual" MIT A.I. Lab, AI-memo No. 259 (1972).
7) D. C. Smith et al.: "Backtracking in MLISP 2, An Efficient Backtracking Method for LISP" 3rd-IJCAI pp. 677～685 (1973).
8) B. Wegbreit et al.: "ECL Programmer's Manual" Harvard University, Cambridge(1972).
9) "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 (1974/7).
10) 長尾，中村: "仮想記憶の概念を使用したミニコン用リスト処理言語 LISP" 信学会研資，AL-73-66～75 (1974.01).
11) 長尾，吉井: "ミニコンへの大容量高速外部記憶装置の付加とその評価" 信学会研資，EC 74-62～69 (1975-03).
12) L. H. Quam and W. Diffie: "Stanford LISP 1.6 Manual" Stanford A. I. Project (Sept. 1970).
13) T. Winograd: "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language" MIT. Project MAC, TR-84 (Jan. 1971).
14) C. G. Bell and P. Freeman: "C.ai-A computer architecture for AI research" FJCC, pp.

779～790 (1972).
15) D.G. Bobrow (ed.): Symbolic Manipulation Language and Techniques" North Holland Publishing-Company (1968).
16) D.E. Knuth: "The Art of Computer Programming Vol. 1" Addison-Wesley Publishing Company, Inc.
17) M.L. Minsky: "A LISP garbage Collector Algorithm Using Serial Secondary Storage" MIT AI-Memo No. 58.
18) 長尾, 辻井: a) "意味および文脈情報を用いた日本語文の解析—名詞句・単文の処理"情報処理, Vol. 17, No. 1, pp. 10～18 (1976).
b) "意味および文脈情報を用いた日本語文の解析—文脈を考慮した処理" 情報処理, Vol. 17, No. 1, pp. 19～28 (1976).

(昭和50年7月21日受付)
(昭和50年9月10日再受付)