LOWELL HAWKINGSON

Segments
Areas
Structures
WORDS

1.1 DATA SEGMENT

The data segment of core storage is partitioned into three distinct areas (see Fig. 3).

ALL NON-IDENTIFIER ATOM TYPES
- Real
- Integer
- octal
- boolean
- Strings
- tables
- arrays
  - symbol
  - octal
  - boolean
- Identifiers

ATOMIC Structure

title

1. TYPE ATOM (# code)
2. SIZE of structure (# wds)
3. Self-pointer (for Reloc G.C.)

elements
- Type specific pattern
- data

FORMAL — { title / BUC to Procedure

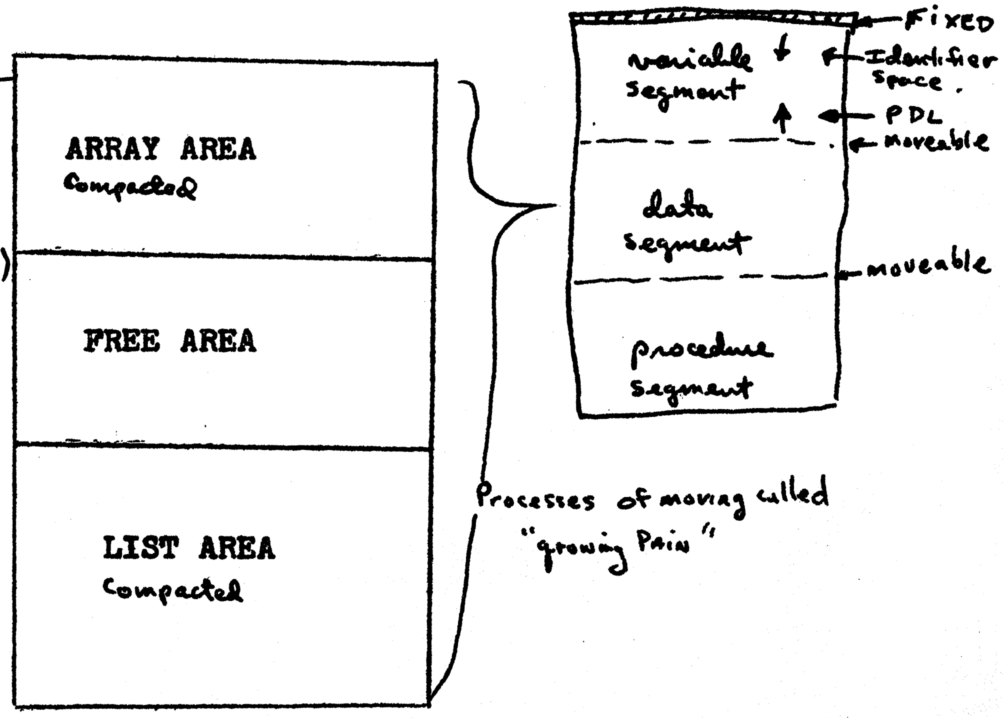

Fig. 3 Data Segment

Details of the contents of each area and of the interaction among the areas are discussed in the paragraphs that follow.

1.11. LIST AREA

The list area of the data segment contains within it structures of only one type; these structures are termed list nodes. At all times the list area must be compactly filled with nodes, i.e., there must never exist in the area a word which does not form a node.

Each list node represents an ordered pairing of two data pointers, one of which is designated as the CAR element, the other as the CDR element. Since the assumption is made here that a core storage word can accomodate at least two addresses, a list node requires exactly one word for the paired data pointers. The address of this word constitutes a reference to the node and is itself a data pointer (see fig. 4).

A data pointer occurring as the CAR or CDR element of a list node may refer to any one of the following types or classes of structures:

1) list nodes (as above)
2) atomic structures (array area)
3) identifier heads (found in the fixed area of the variable segment)
4) TRUE and NIL (special cases)

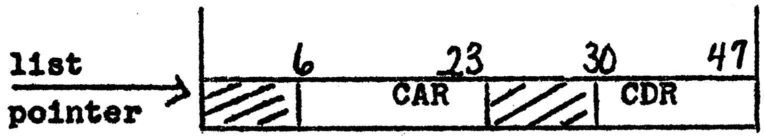

Fig. 4. List Node, Q-32 Style

CAR, CDR and the list pointer itself are data pointers.

New list nodes are created by the primitive LISP function CØNS. The procedure for selecting words in which to form them is treated in paragraph 1.13.

1.12 ARRAY AREA

The array area of the data segment contains structures of an arbitrary number of types; however, all of them. regardless of type, must conform to certain common specifications and will be referred to collectively as atomic structures. As was the case with list nodes in the list area, atomic structures must always compactly fill the array area.

Atomic structures are, essentially, ordered collections of data, both pointer and absolute, arranged in type-specific formats that conform to those specifications, outlined below, which are common to all types. Each atomic structure consists of one or more sequentially-addressed words in the array area. The first word of the structure (and half of the second, if necessary) is called the title word. It includes three ingredients, explicit or implicit:

1) type indicator, a small integer specifying the structures type
2) size, integer count of the total number of words in the structure (unneeded if this information is implied by the type indicator.)
3) self-pointer, address of the title word, conventionally located in CDR of the title word considered as a list node.

The remaining words of the structure contain the absolute and pointer elements; the number and arrangements of these elements and the significance of each one of is a function of the type and, for types allowing variable-size structures, also of the size .
Conventionally, however, pointers are permitted within a word only in the CAR-CDR positions. Atomic structures of any type, whether or not

they may be created in various sizes, are fixed in size ever after their creation. In order to expand or contract a structure, it would be necessary (but not always possible) to generate a new one of the desired dimension and to copy into it those elements of the old one which are to be preserved.

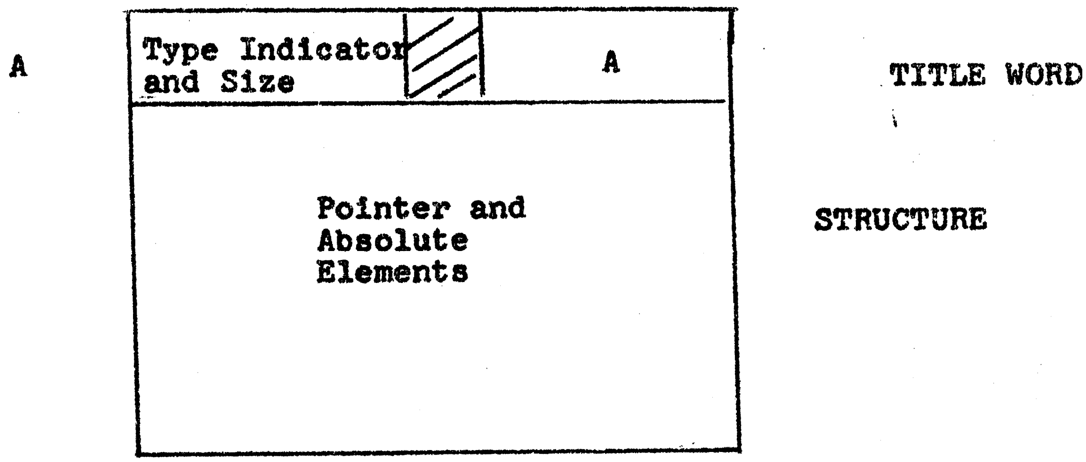

Fig. 5. Atomic Structure

Although the format and pattern of the elements of an atomic structure are relatively unrestricted, certain information about the elements must be available, for each particular type, to the storage control section of the system. Specifically, this includes the size, if constant, and not explicit in the title, and an algorithm for locating and, in some cases, determining the significance of each pointer element in the structure. This information is required by the garbage collector as discussed in paragraph 1.14. Of course, much more information about each type is needed to make possible effective programming use of it.

References to an atomic structure come in two different variaties. The data pointer variety, consisting merely of the address of the title word, represents the structure as a whole in a data context. The locative pointer variety, on the other hand, refers to a single element within the structure in a variable context and consists of both the address of the word in which the element lies and the displacement of that word from the title (see Fig. 6).

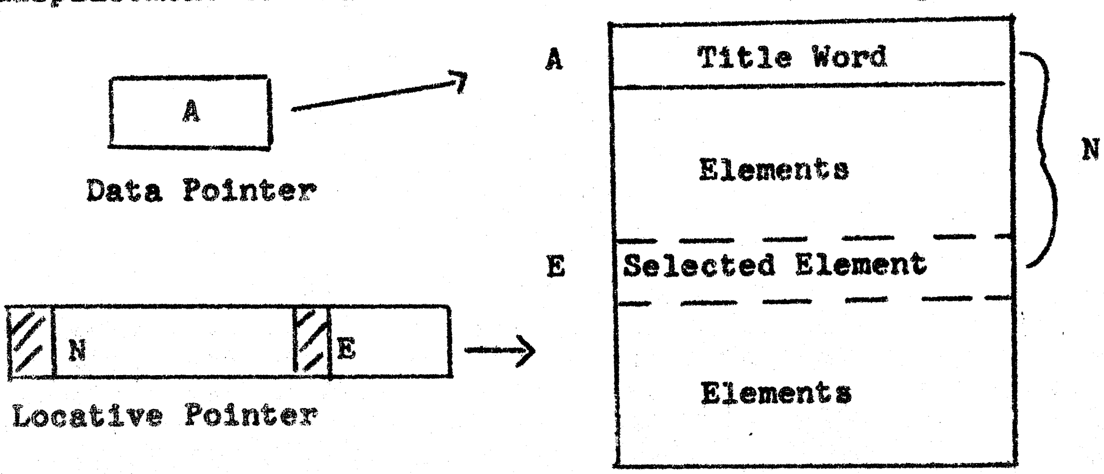

Fig. 6 Atomic Structure References

Some examples of specific types of atomic structures, as they might appear in the Q-32 LISP II system, are illustrated below (see Fig.7) Example A depicts a real number as an atomic structure; the type indicator is 5 and the implied size for this type is 2. In example B is shown a real matrix; the type indicator is 10, the total size of the structure is S, there are 3 dimensions, l_1-l_3 give the ranges for each dimension, and b_1 - b_3 are base addresses used for indexing the matrix. Finally, example C illustrates a special table structure; the type indicator is 50, the implied size is 3, and the elements include 2 data pointers and an integer, arranged as shown.

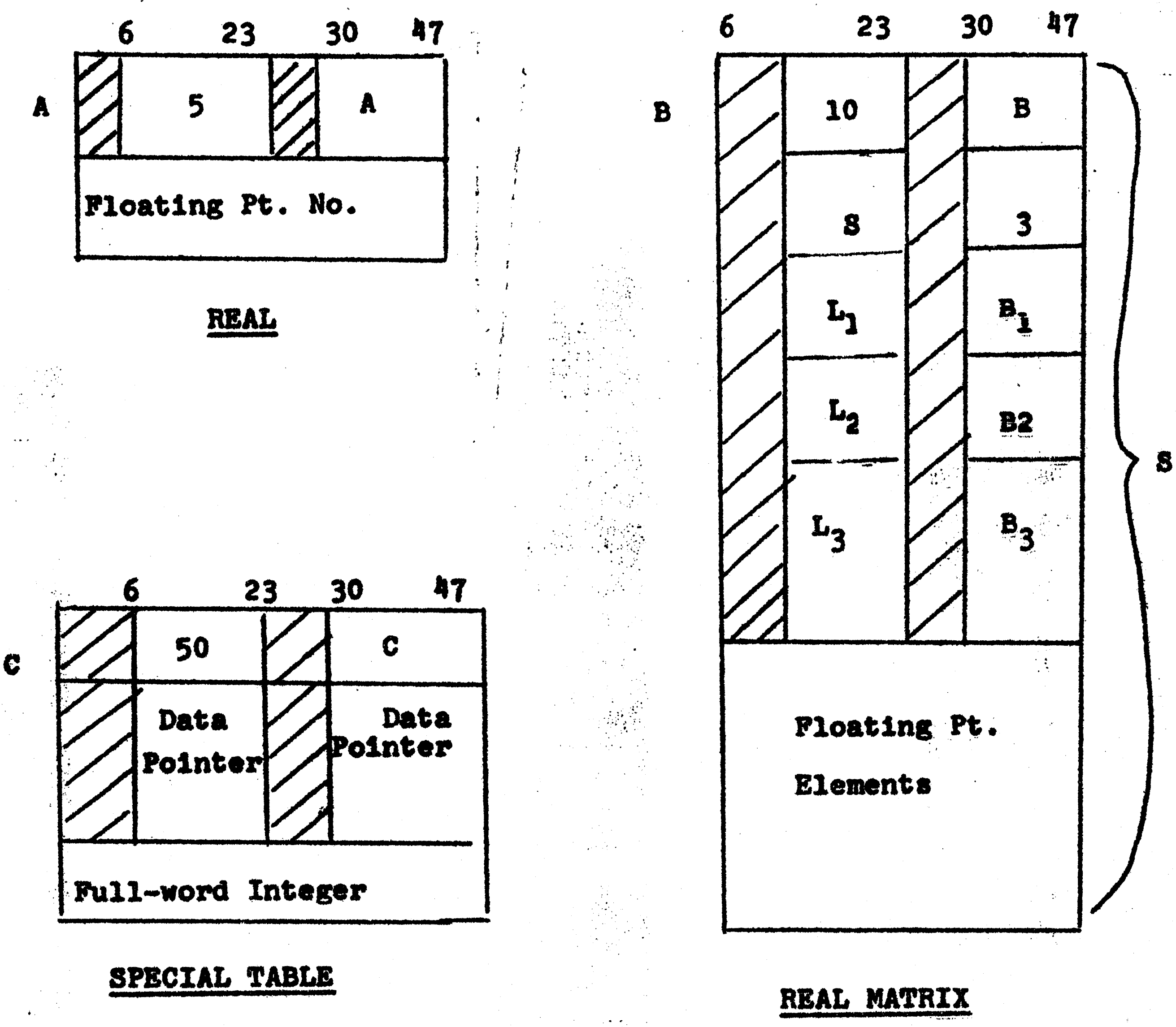

Fig. 7. Examples of Q-32 Atomic Structures

1.13 Free Area

The free storage area of the data segment contains nothing whatever of any significanee to the program; its sole purpose, in fact, is to serve as a reservoir of words out of which list nodes and atomic structures may be formed. Whenever a new list node is to be created, that word in the free area immediately adjacent to the list area is expropriated for the purpose, becoming thereafter a word in the list area. A similar procedure is employed for the generation of atomic structures, except that many words may be taken at a time instead of just one and that the other end of the free area is involved.

The efficacy of the procedure descrived above is, of coru upon the availability in free storage of the requisite number of words one or many as the case may be. Since this area is continually being depleted by the creation of structures, the moment is bound to arrive sooner or later, when it is unable to satisfy the demand for words made upon it. At this point, a process known as garbage collection must be initiated and carried through to a successful completion. if the program is to be continued. Hopefully, the garbage collector will be able to increase substantially the dimension fo the free storage area, at least by an amount sufficient to meet the demand at hand. The gargage collection Algorithm is presented in paragraph 1.14 below.

1.14 GARBAGE COLLECTION

The algorithm for garbage collection of the data segment will be presented below in LISP II source language. It will be far from complete, for the most part neglecting the related garbage collection. problems for the other segments of core storage, which will be discussed later in more informal terms. Many subfunctions will be described, but not explicitly defined. Despite the limitations of such an approach, however, the primary purpose, that of exhibiting the algorithm for compacting the list and array areas by structure relocation, will have been served.

That portion of the total garbage collection scheme which affects the data segment consists of seven or eight distinct steps (they do not all really deserve to be called passes) The first step involves the marking of structures which must be preserved, i.e., are not garbage; the remaining steps represent a progression toward the ultimate compacting of the saved structures, requiring their re-assignment and relocation and the updating of all references to them. A brief description of each of the steps is offered below as a prelude to the actual LISP precedure definitions themselves.

The partitioning of the data segment upon the initiation of garbage collection is defined by four boundaries B1-B4 (see Fig.8) The assumption is made that the extreme boundaries of the segment , B1 and B4, are to be displaced by amounts DB1 and DB4 respectively, allowing the segment as a whole to be expanded, contracted, or shifted (an essential part of the so-called "growing pain").

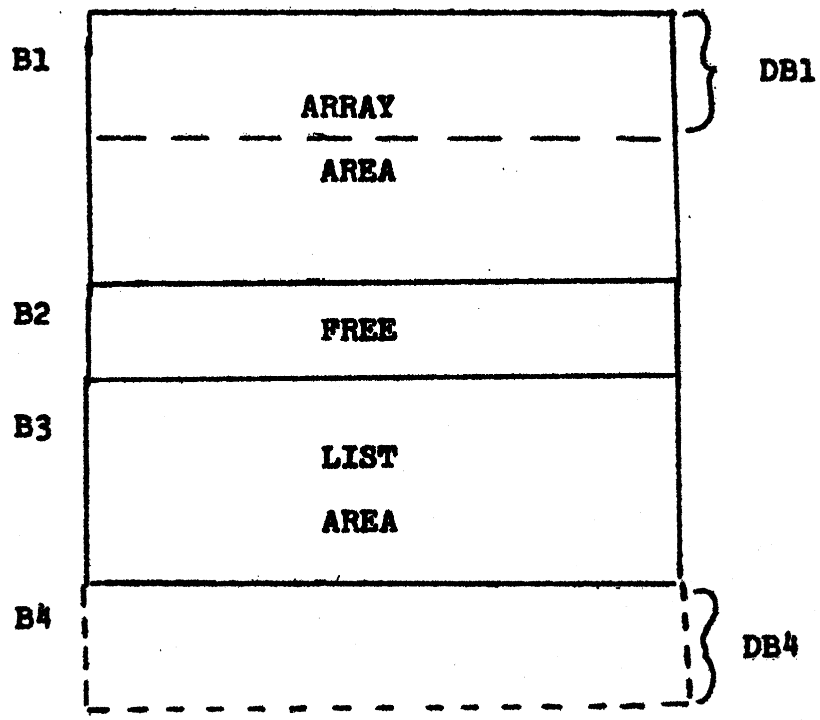

Fig. 8 Boundaries of Data Segment

The principal procedure of the garbage collector GC has seven arguments: pointers B1 -B4, integers DB1 and DB4, and an integer N which specifies the minimum number of words that must be restored. Procedure GC advances one by one through the eight steps, each of which is outlined below, and before returning updates the pointers B1-B4 and checks that the regenerated freee area does indeed contain at least N words.

1. MARKVARS (not defined) marks all list and atomic structures that are to be saved by applying procedure GCMARK to each data pointer (or locative transformed into a title pointer) in the variable segment. GCMARK determines what its argument is pointing to and acts accordingly: an identifier head is marked by MARKI; any other atomic structures, if unmarked (NOT MARKEDA(X)), is marked by MARKA and then given to MARKELS which is identical to MARKVARS except that it works on a single atomic structure; a list node, if unmarked (NOT MARKEDL(X)), is marked by MARKL, after which GCMARK is applied recursively to its CAR and CDR elements. Of all procedures used here, only GCMARK is defined; the rest are assumed. Predicates IDP and BOOLP test for identifiers and value TRUE or NIL, respectively.

2. GC2 assigns new locations for all marked atomic structures, such that B1 + DB1 is the first location so assigned and such that the structures, when moved during step GC6 to their new locations, will be compacted and will retain their original ordering. The new address for the title word of each atomic structure replaces the CDR of its old title word (previously a self-pointer) Sub-function SIZEA determines the size of an atomic structure.

3. GC3 compacts the list area by repeating the following two steps until it is no longer possible to do so: move the topmost marked node down into the bottommost unmarked word; store a pointer to the latter in CDR of the former. After this iteration has been completed, the CDR of all atomic structures and of those nodes which have been moved will contain their respective relocation addresses.

4. UPDATEVARS (not defined) replaces each data pointer (or locative properly transformed) occurring in the variable segment by the value of UPDATEP applied to it. UPDATEP returns the relocation address, CDR of the structure, for all atomic structures and moved list nodes; the pointer displaced by DB4 for unmoved nodes; the pointer argument itself otherwise.

5. GC4 updates all data pointers which occur as elements of marked atomic structures. UPDATELS (not defined) is identical to UPDATEVAR except that it substitutes for the data pointers found within a single atomic structure. Note that the procedures MARKELS and UPDATELS must have available to them the information required to locate pointer elements, as mentioned in paragraph 1.12. Similarly SIZEA must be able to determine the size of any given atomic structure.

6. GC5 replaces each CAR and CDR element of the nodes in the now compacted part of the list area by UPDATEP thereof.

7. GC6 moves each marked atomic structure to the new location assigned for it during step GC2 (or to a temporary location if necessary, in order to avoid conflicts when boundary B1 is being displaced by a positive DB1.

8. GC7 displaces the compacted array when DB1 is positive and list area when DB4 is non-zero, such that the new boundaries of the data segment B1 and B4 will become what was previously B1 + DB1 and B4 and DB4 respectively. As in step GC6, it is assumed that any room required from variable or procedure segment for the displacement is already available.

Several considerations which are relevant to but have been neglected in the garbage collection algorithm presented here are: the unmarking of structures (somewhat machine dependent) ; the fact that DB1, DB4, and N would not normally be parameters of GC, but would rather be computed heuristically after the first (marking) step; the remainder of the "growing pain" problem, since displacement of B1 and B4 implies the moving of variable and procedure segment boundaries; the gathering of statistics on garbage collection.

```
PROCEDURE GC (B1,B2, B3, B4, DB1, DB4, N)
     FLUID POINTER B1,B2,B3,B4;
     INTEGER DB1,DB4, N;
     BEGIN FLUID POINTER NB2, NB3;
          MARKVARS ():
          NB2 ← GC2( ); NB3 ← GC3( );
          UPDATEVARS ( ); GC4( ); GC5 ( );
          GC6( ); GC7( );
          B1 ← B1+DB1; B2 ← NB2;
          B3 ← NB3+DB4; B4 ← B4+DB4;
          IF B3-B2 N THEN ERROR ('(GC ERROR))
          RETURN
END



PROCEDURE GCMARK (X); BEGIN
     IF IDP (X) THEN MARKI (X) ELSE
     IF ATOM (X) AND NOT MARKEDA (X) THEN
          BEGIN MARKA (X); MARKELS (X) END ELSE
     IF NOT BOOLP (X) AND NOT MARKEDL (X) THEN
          BEGIN MARKL (X); GCMARK (CAR (X));
               GCMARK (CDR(X)) END;
RETURN
END


PROCEDURE GC2 ( );
     BEGIN POINTER P1,P2;
          P1 ← B1; P2 ← B1+DB1;
     L1:  IF P1=B2 THEN RETURN (P2);
          IF NOT MARKEDA (P1) THEN GO TO L2;
          CDR (P1) ← P2;
          P2 ← P2+SIZEA (P1);
     L2:  P1 ← P1+SIZEA (P1);
          GO TO L1
END


PROCEDURE GC3 ( );
     BEGIN POINTER P1, P2;
          P1 ← B3; P2 ← B4;
     L1:  P2 ← P2-1;
          IF MARKEDL (P2) THEN GO TO L1;
     L2:  IF MARKEDL (P1) THEN GO TO L3;
          P1 ← P1+1; GO TO L2;
     L3;  IF P1 > P2  THEN RETURN (P1);
          WORD (P2) ← WORD (P1);
          CDR (P1) ← P2+DB4;
          P1 ← P1+1; GOTO L1
END
```

```
PROCEDURE UPDATEP (X); RETURN
     IF B1 < X AND X < B4 THEN
          IF X < NB3 THEN CDR (X) ELSE X +DB4
          ELSE X


PROCEDURE GC4 ( );
     BEGIN POINTER P1;
          P1 ← B1
     L1:  IF P1 = B2 THEN RETURN;
          IF MARKEDA (P1) THEN UPDATELS (P1);
          P1 ← P1+SIZEA (P1);
          GO TO L1
     END


PROCEDURE GC5 ( );
     BEGIN POINTER P1;
          P1 ← NB3
     L1:  IF P1=B4 THEN RETURN;
          CAR (P1) ← UPDATEP (CAR(P1));
          CDR (P1) ← UPDATEP (CDR(P1));
          P1 ← P1+1; GO TO L1
     END


PROCEDURE GC6 ( );
     BEGIN POINTER P1, P2; INTEGER J, K;
          K ← IF DB1 < 0 THEN 0 ELSE DB1;
          P1 ← B2;
     L1:  IF P1=B2 THEN RETURN:
          IF MARKEDA (P1) THEN BEGIN
             P2 ← CDR (P1)-K;
             FOR J ← 0 STEP 1 UNTIL (SIZEA (P1)-1)
                 DO WORD (P2+J) ← WORD (P1+J)
          END;
     P1 ← P1+ SIZEA (P1); GO TO L1;
END
```

```
PROCEDURE GC7  (   );
     BEGIN POINTER P1, P2; INTEGER J;
          IF DB4 < 0 THEN BEGIN
             P1←NB3; P2← NB3+DB4;
             FOR J← 0 STEP 1 UNTIL (B4-NB3-1)
               DO WORD (P2+J)      WORD (P1+J)
             END;
     IF DB4 > 0 THEN BEGIN
             P1 ← B4; P2← B4+DB4;
             FOR J← 1 STEP 1 UNTIL (B4-NB3)
                 DO WORD (P2-J) ← WORD (P1-J)
          END;
     IF DB1 > 0 THEN BEGIN
             P1← NB2 - DB1; P2← NB2;
             FOR J←1 STEP 1 UNTIL (P1-B1)
                DO WORD (P2-J) ←WORD (P1-J)
          END;
     RETURN
END
```