MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Project MAC
Cambridge, Massachusetts

Artificial Intelligence Project
Memo 58 (Revised)

Memorandum MAC-M-129
December 27, 1963

A LISP Garbage Collector Algorithm Using Serial Secondary Storage*

by M. L. Minsky

Paper to be presented at the First International LISP Conference, Mexico City, Mexico, December 30 - January 3, 1964.

This paper presents an algorithm for reclaiming unused free storage memory cells in LISP. It depends on availability of a fast secondary storage device, or a large block of available temporary storage. For this price, we get

1. Packing of free-storage into a solidly packed block.
2. Smooth packing of arbitrary linear blocks and arrays.
3. The collector will handle arbitrarily complex re-entrant list structure with no introduction of spurious copies.
4. The algorithm is quite efficient; the marking pass visits words at most twice and usually once, and the loading pass is linear.
5. The system is easily modified to allow for increase in size of already fixed consecutive blocks, provided one can afford to initiate a collection pass or use a modified array while waiting for such a pass to occur.

collect[z;n] is the function that finds all list-structure depending from z, and puts out on drum the

*Work reported herein was supported by MIT Project MAC and sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

To handle blocks, insert after the marking operator a transfer to a sub-program like the one below, which uses two new program variables:

```
        write[n;a;d]
        q := h
        p := d
START   q := q+1
        p := p-1
        equal[p;0] → go [BEGIN]
        n := n+1
        writefullword[n;q]
        mark[q]
        go[START]].
```

When reloading, simply put (y . z) into register x.

mark[x] plants a mark in the word in register x, so that collect can tell that this word was encountered before. The value of mark[x] is x.

mkd[x] is a predicate that is T if x is marked. We assume also that

$$atom[x] \supset m[x]$$

for expository purposes.

v[x] = [atom[x] → x;T → cdr[x]]

This function gives access to the re-location address of the packed list-structure; these addresses are stored in the decrement fields of marked words. Only study of the algorithm will reveal why the old decrement values can be so replaced.

```
collect[z;n] = prog[[a;d;m;h]
BEGIN    [null[z] → return[DONE]
         h := car[z]
         a := car[h]
         d := cdr[h]
         m := mkd[h]
         [~ m → rplacd[mark[h];n]]
         [m ∨ mkd[a] ∨ mkd[d] →
              write[[m → d;T → n];
                    [mkd[a] → v[a];m → n;T → n+1];
                    [m → d+1;mkd[d] → v[d];T → n+1];
         z := cdr[z]
         [~ m ∧ ~ mkd[d] ∧ ~ mkd[a] → z := cons[h;z]]
         [m ∨ mkd[d] ∧ ~ mkd[a] → z := cons[a;z]]
         [~ m ∧ ~ mkd[d] → z := cons[d;z]]
         [~ m → n := n+1]
         go[BEGIN]]
```

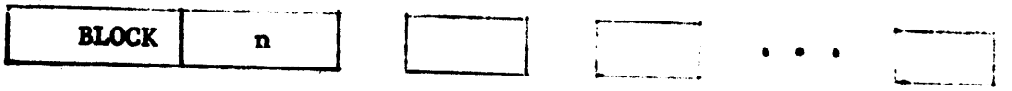
data for assembling the packed version of same. The packed version will, when loaded, begin at register n. To use collect, one must construct a z which leads to all the structure that must be saved.

Collect has the additional feature that all cdr sequences end up linearly packed! There are probably some important applications of this.

Collect uses a push-down list of words whose reading-out has had to be postponed because neither of the two pointers could be evaluated in terms of the re-location address. This list (here concatenated with z) can be as long as the longest chain of cdr's whose car's are not atoms. This can be a nuisance; the list would (in general) be shorter if we rewrite collect to pursue the car directions in preference to the cdr chains. In any case, the ordering on z is immaterial, hence it can be buffered and got out on secondary storage.

Note: because of the replacd, collect as written will destroy itself if it can be reached through z. Obviously, the real collect will be a special compiled program.

We present only the basic algorithmic idea. It is easy to see how to patch it to introduce a special atom "BLOCK" which, when it appears in the form



signals the existence of a consecutive block of n full words following that word in core memory.

The algorithm as presented stops copying when it encounters an ATOM. This is done for clarity, but for real application one would have to make things more complicated.

Definitions:

write[x;y;z] writes the triple [x;y;z] on to the drum.