The views, conclusions, or recommendations expressed in this document do not necessarily reflect the official views or policies of agencies of the United States Government.
The research reported in this paper was sponsored by the Advanced Research Projects Agency Information Processing Techniques Office and was monitored by the Electronic Systems Division, Air Force Systems Command under contract F1962867C0004, Information Processing Techniques, with the System Development Corporation.

TECH MEMO

a working paper

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406
Information International Inc. / 11161 Pico Boulevard / Los Angeles, California 90064

TM- 3417/500/00

AUTHOR Jeff Barnett for P.S.
P. Stygar

TECHNICAL Jeff Barnett
J. Barnett

RELEASE Carl Weissman
C. Weissman, S.D.C.
Ken Anschultz
D. Anschultz, I.I.I.

for J. I. Schwartz

DATE 4/26/67 PAGE 1 OF 16 PAGES

(Page 2 Blank)

LISP 2 Garbage Collector Specifications

ABSTRACT

This document is intended to be a basis for further detailing of the work completed on storage management for LISP 2 on the IBM 360. It includes brief descriptions of housekeeping information, the four phases of garbage collection, the decentralization of primitives, the organization of the pushdown stack, and the role of various data zones. The relevance of these to the overall system is outlined, together with some of the remaining design and programming problems.

TABLE OF CONTENTS

1. INTRODUCTION

In this document, LISP 2 storage management is described metaphorically, using the terminology of city planning. The technical terms used in this description include zone, zoning laws, reapportionment, structural unit, site, block, relocation, forwarding address, map, surface and substrate. This is an extension of the usual systems terminology, which includes such terms as location, field, area, space, and border. Hopefully, the correspondence between the everyday and the technical will be clear and useful. The garbage collector performs the role of gathering together the inactive sites. The reapportionment of blocks among zones is better known as the growing pain.

Storage management provides for 20 to 30 data zones in the basic implementation. There are user facilities for the definition of further data zones, including such specialized data zones as association spaces, SLIP spaces, queues, and stacks. A "garbage collector" is specified for this version of LISP 2; it uses a fresh technique called "plex processing." The word "plex" is an abbreviation of "plexus," and means an interwoven combination of elements or parts in a cohering structure. Plex processing is the interconnecting of n-tuples. Note that the etymology of the word "n-tuple," as in quintuple, may be traced to the Latin word "plex," which means fold or weave, as in fivefold. An n-tuple is a data element with N components, in which the components are accessed by name. For example, the traditional list node is a 2-tuple with component names CAR and CDR. The decision to use plex processing was made to motivate a detailed study of the problems associated with the new feature. Several simplifications in the design of storage management and the primitives occurred as a result. Similar simplifications will occur in other parts of the system.

2. TERMINOLOGY

A field is a consecutive sequence of bits in a memory word.

A structural unit is an aggregate of fields that represents some elementary data object.

A locator is a unit that is an encoded representation of the information necessary to access a field of a structural unit.

A site is the storage space allotted to hold a structural unit.

The surface of a structural unit is the information associated with the unit by virtue of its classification as a data unit. For example, the surface of a list node includes the fact that the data unit is a list node, that it has two components CAR and CDR, that the data type of the components is SYMBOL, that CAR is the left half-field and CDR is the right half-field of a word. The surface includes an n-tuple called the structural unit descriptor which contains information and functionals necessary to process the data unit. The

surface is determined from the site on which the unit is situated and from the data zone containing the site.

The substrate of a structural unit is the aggregate of values contained in the fields of the structural unit. The substrate belongs to the particular data unit, and may be thought of as an instance of the surface of the data unit. For the example of a list node, the fact that the node has components CAR and CDR is the surface of the node, whereas the contents of the CAR and CDR fields make up the substrate of the node. The substrate includes pointers which must be followed during the marking phase of the garbage collector, and which must be updated when the referenced data units move.

A storage block is a set of contiguous memory locations. The computer memory is partitioned into blocks at the time the system is generated. The size of the blocks is uniform, predetermined, and a power of 2, typically 512 or 1024.

A zone is an integral number of consecutive storage blocks. The partitioning of the blocks into zones is varied via a reapportionment mechanism. For example, a zone that contains no active structural units will have no blocks, and is said to be dormant. A zone can grow or shrink in quantum steps as determined by the storage block size. There are zoning laws to restrict the kinds of structural units that may appear within the zone. The zone control is a tuple containing procedures and functionals to allot sites within the zone, to specify the surface information about the structural units on the sites in the zone, and to sweep over the zone during the various phases of garbage collection. Various statistics are accumulated, such as the occupancy of each zone and the count of the number of structural units of each general type. In addition to the zone control, there is a tuple associated with the zone called the zone descriptor which contains parameters such as the primary border, or low address of the zone, and the conjugate border, or high address of the zone. The control is a parameter of the descriptor. The separation of the control and the descriptor allows a multiplicity of zones with the same control. Thus it will be possible to have several list spaces.

The storage map is an array that specifies for each storage block the descriptor of the zone to which the block belongs. The map is used to access the LISP primitive for the data units in the data zone, as these are stored in the zone control.

3. STRUCTURAL UNIT DESCRIPTOR (SUD)

A structural unit is one of the following data objects: a list node, a hash node, a number cell, an identifier, a genid, an array, a record, a table, a function descriptor, a quote cell, a free variable, an own variable, a locator, a string, an absolute value cell or a symbolic value cell. From the point of view of storage management, a structural unit is a grouping of fields in which the fields are associated together by virtue of their relative placements. The fields might be associated together because they are in contiguous memory

locations, or because they are in symmetric positions relative to the boundaries of a data zone. The fields are not associated together via links, symbol pointers or displacements. The field grouping is treated as a unit by the storage management facilities: the unit is marked once, has one forwarding address, and is relocated en masse. Some one of the fields in the unit is understood to be the base, and all locators of fields in the unit must address relative to the base of the unit. The placement of the fields relative to the origin is called the layout of the structural unit.

With each structural unit is associated a structural unit descriptor (SUD). This is an n-tuple containing parameters, lists and functions as required by the various aspects of the system in order to describe and process the unit. All units of the same general layout and classification will have the same SUD. The SUD controls a data unit in the sense that it contains the functionals required to process the unit. In the case of records, the SUD is introduced as a side effect of the user's definition of the record. In other cases, the SUD is introduced along with the core image generation of the system.

The information contained in the SUD is used by the following:

1. the I/O, in outputting data objects, and in inputting n-tuples;
2. the dynamic typer, which yields the structure name for an n-tuple;
3. the equality primitive, where if two data objects have the same data descriptor, then a functional contained in a field of the data descriptor is fired to carry on the comparison of the data objects;
4. the allocation of storage for new data objects;
5. the garbage collector, where the mark/follow and the fix phases require the coordinate of each pointer field, and the plan and move phases require the size of the data object.

The structural unit descriptor is obtained via the function SUD (X).

A problem which has not been explored in detail is the question of whether there should be a special descriptor for arrays that makes use of a SUD for the entries of the array. The SUD approach seems to work for everything else: n-tuples, number cells, list nodes, identifiers, variable elements, etc.

4. GARBAGE COLLECTOR OPERATION

The garbage collector operates in four passes, called the mark phase, the plan phase, the fix phase and the move phase. During each phase the appropriate control functional for each zone is fired on that zone, meaning that the functional is evaluated with the zone descriptor as an argument. When the functional is fired, a brief flurry of activity occurs, during which the storage management

processes for that zone during that phase are performed. The order in which the zones are processed is irrelevant: the zones may be processed in a different order for each garbage collection, or even each phase. All that matters is that each zone be processed once during each phase, by the control functional appropriate during that phase. In some instances a control process is a no-op. For programming convenience the processing order of the zones is determined by the storage map. The dormant zones are ignored.

4.1 MARK PHASE

During the mark phase, the zone control component SCANS is fired on each zone descriptor. The purpose of this phase is to determine which data units are active. At the start of the phase, the active units include the variables, the quote structures and the nontrivial identifiers. At the end of the phase, every data unit referenced from the substrate of an active unit has been marked as active. A zone control component, MARK, when fired on a data unit will flag the site as active. A zone control component, ACTIVE, specifies whether or not a site has already been flagged as active. This latter functional is also used during the plan and fix phases. The SUD component FOLLOW is fired on a data unit to guarantee that the site of each unit referenced from the substrate is active. The substrate is followed once for each active data unit, regardless of how many pointers reference the unit. If a referenced site is not active, then it is marked and its substrate is followed. The time spent marking is proportional to the number of active data units. The time spent following is proportional to the number of pointers in active data units. The time spent scanning is proportional to the number of units, active and inactive, in the scanned zones.

4.2 PLAN PHASE

The plan phase prepares each of the data zones for the operations to be performed during the fix and move phases to follow. During the plan phase, it is determined exactly where each data unit will be after the move phase. Many of the data units will be relocated, either during the plan phase or during the move phase, and a forwarding address must be associated with the current site of unit to indicate the new site.

Data units are relocated so that all the active data units in a zone will be on contiguous sites, and all the vacant sites will be contiguous. As a result of compacting a data zone, the allocation of sites for new data units in the zone is simplified. Also, the number of storage blocks assigned to a zone can be kept to a minimum. The storage blocks not assigned to any zone are put into a pool of vacant storage blocks, from which they will be allotted to other data zones depending on the dynamic requirements of the system.

The reassigning of sites to data units is called reallocation. The actual movement of data units onto the reassigned site is called relocation. The reassigning of storage blocks to zones is called reapportionment.

4.2.1 Local Planning

Local planning is that aspect of the plan phase which governs the reallocation of sites to data units within a data zone. We will distinguish between pure zones, uniform zones and mixed zones. A pure zone is structurally homogeneous or equi-typical: all the data units in the zone have the same structual unit descriptor, so that the SUD is determined from the zone, and the only housekeeping information associated with each site is a bit for marking purposes. Sometimes the marking bit is contiguous with the unit, and sometimes there is a bit map for the zone starting at the conjugate border. This arrangement of the unit on the site is called a pure format. A uniform zone is homogeneous with respect to size; the data units in the zone have a uniform size, though they are of mixed SUDs. The housekeeping information associated with each site includes a bit for marking and a pointer to the SUD. This arrangement of the unit on the site is called a uniform format. A mixed zone is heterogeneous: the data units are of mixed sizes and mixed structural description. The housekeeping information associated with each site includes a pointer to the SUD and a pointer to the site. This arrangement of the unit on the site is called a mixed format. The self-pointer is set to NIL during the mark phase to indicate an active data unit. The self-pointer is set to the forwarding address of the data unit during the plan phase. After the move phase, the self-pointer points at the site, until the next garbage collection.

4.2.2 Folding

In pure zones and in uniform zones, such as list node zones and numerical zones, both the reallocation of sites and the local relocation of data units is performed during the plan phase. The following operation is repeated until all the active data units are contiguous in the primary area of the zone: the highest active data unit is moved to the lowest vacant site, and a forwarding address is placed on the vacated site indicating the new site. The vacated site is now inactive, and the new site is active. The algorithm uses two pointers: one to the low end, which moves up looking for vacant sites, and one to the high end, which moves down looking for active units.

The time spent scanning is proportional to the number of sites. The time spent folding is proportional to the number of units relocated and the size of the sites.

4.2.3 Vacant Sites List

In pure zones and in uniform zones there is an alternative reclamation technique to folding. The technique involves a sweep over a zone linking the vacant sites together. This technique is useful when global conditions are such that there is little global advantage to compacting; under such conditions the fix and move phases may be bypassed, with a resulting decrease in overhead. The design of the local planning for each zone must allow folding as an option, to be used at the discretion of the global planning mechanism.

The vacant sites list is the only reclamation technique useful in bolted zones, in which the data units may not be relocated relative to the primary border of the zone. Examples of bolted zones include the function descriptor zone, the variable element zone, and the zone of quote-structure bases. These zones are bolted because the data units within are referenced from compiled code. The time spent scanning is proportional to the number of sites in the zone.

4.2.4 Collapsing

In mixed data zones, the relocation of the data units is postponed to the move phase, in which the space occupied by inactive sites will be squeezed out in the process of collapsing the zone, that is, moving all the active sites together. The order of appearance of the active sites is preserved by the collapsing operation. During the plan phase the effect is simulated by using two pointers: the put pointer and the scan pointer. The put pointer points to the simulated top of the primary area of the zone, and indicates the address of the site at which the next data unit is to be allocated. The scan pointer points to the next site to be examined under the reallocation. The size of the scanned site is determined. If the scanned site is active, the put pointer is assigned to the self pointer of the site and the put pointer is incremented by the site size. The scan pointer points to the next site to be examined under the reallocation.

The scan is then continued by incrementing the scan pointer by the site size. The time spent scanning is proportional to the number of sites in the zone.

4.2.5 Pruning

A data unit is said to be unique if the information that makes up the data unit is constant throughout the lifetime of the data unit. A unique data unit has a read-only substrate. If a data unit is unique, then it is advantageous to conserve storage space by representing the data unit uniquely. A data unit is said to have a unique representation if all copies of the data unit are on the same site. Examples of unique data units include identifiers, variable elements, numbers and hashed list nodes. An identifier is a unique string. When a site is to be allocated for a unique data unit, then a check must be made to determine whether or not the data unit is already represented. Hashing techniques are used to increase the efficiency of this search, in which a hash number is computed on the basis of the constant information in the data unit and all the data units in a given zone with the same hash number are linked together. The hash link is considered to be a property of the site, and is not followed during the marking process.

Pruning is the name given to the process of following the hash links and unlinking the inactive data units. Pruning occurs before reassigning sites to unique data units, and is considered to be a part of the housekeeping performed by the garbage collector during the plan phase. The time spent pruning is proportional to the number of uniquely represented data units. The alternative to

pruning is a technique called rehashing, which is performed during the fix phase and accomplishes both the pruning and the fixing of the hash links. When pruning is used to preserve the hash structures, the fixing of the hash links must occur during the fix phase.

4.2.6 Global Planning

The global planning governs the growth of zones and the relationships between zones. We define a mutual to be a collection of zones in which the sites are inter-allocatable. A mutual is the smallest programmer-selectable workspace. The most common instance of a mutual will be a single zone. The next most common instance will be a set of zones with the same zone control. The most general mutual would bring together zones having separate zone controls.

When the allocation processes exhaust the available sites in a zone, a vacant storage block must be found and added to the mutual of which the zone is a member. In some cases, such as list node zones, the operation involves no more overhead than constructing a new zone descriptor. In the more common case, the operation must be performed by appending a contiguous vacant block, which might involve a ripple of zone relocation.

The global planning makes use of the statistics accumulated during the mark phase: the occupancy of each data zone, the number of instances of each class of data unit, and the total site allocation in each zone since the last garbage collection. The global planning must be very flexible in design and in operation. Heuristics are used at the option points to smooth the overall operation; it must have flexible communications with the local planning mechanisms.

4.2.7 Reapportionment

The reassigning of blocks to zones is accomplished by moving the primary border, the conjugate border, or both. A border movement entails a relocation of the portions of data units contiguous with the border. The moving of the primary border is a somewhat involved operation, because the data units in the zone are usually allocated starting at the primary border. In the case of the list nodes, the moving of the primary border can be achieved at the relatively low cost of relocating one block of nodes. In other cases, the effect must be achieved by first planning to do it, then performing the actual movement of the primary area of the zone at the end of the move phase. This latter operation is called a growing pain. The operation is performed to move a vacant block from one side of a zone to the other. Storage blocks can be appended only to the conjugate border of a zone. The reassigning of the conjugate border, whether to enlarge a zone or to shrink it, is a simple operation. The conjugate border is adjusted automatically after the data units in a zone have been compacted to vacate the blocks contiguous at that end.

4.2.8 Inter-Allocation

When several zones with the same control are organized as a mutual, the re-allocation of sites may result in the movement of some data units to some other zone in the mutual, in order to keep the total vacant space within the mutual to less than one storage block.

4.2.9 Reformatting

When there are enough instances of an n-tuple of a given class, a pure zone is created, filled with data units from a mixed n-tuple zone. The conversion from the mixed format to the pure format accomplishes a net release of storage space due to the decreased housekeeping information associated with the pure format. The new pure zone becomes part of a mutual containing the mixed zone.

When there are too few instances of an n-tuple of a given class, it is preferable to store all the instances in the mixed format. The conversion from the pure format to the mixed format is performed only if the pure space is a member of a mutual containing a mixed space. The conversion accomplishes a net release of storage, because n-tuples in the pure format require a minimum of one storage block, whereas in the mixed format less total space may be consumed in spite of the increased housekeeping information.

4.3 FIX PHASE

During the fix phase every pointer in every active data unit is replaced by a forwarding address associated with the site of the referenced data unit. In the case of a pointer that references a non-relocatable data unit, the fix operation is an identity operation. The time spent scanning is proportional to the number of active data units, except in bolted zones, where the time spent scanning is proportional to the number of sites in the primary area. The time spent fixing is proportional to the number of pointers in active data units.

4.4 MOVE PHASE

The move phase carries out the specifications of the plan phase. Every data unit is moved to the location planned. The time spent is proportional to the total size of the relocated data units. There is no overhead for the data units not moved.

5. GARBAGE COLLECTION OVERHEAD

Garbage collection is often viewed as a convenience: the programmer is spared the detailed bookkeeping associated with allocating storage, erasing data structure and reallocating storage. He does not have to program the destruction of the data units that make up the complex data structure he wishes to manipulate. He is thus free to concentrate on the problem-oriented aspects of the algorithm and the data structure. However, there are a number of general principles affecting the overhead of garbage collection. These principles, when followed, will result in an increased effectiveness and efficient use of garbage collection as a tool.

The following is a simplified derivation showing the basic relations between the main factors affecting overhead. The subsections following the derivation discuss the simplifications made in the derivation; the general principles which may be deduced from the result of the derivation; and the implications of the general relations for system design.

Let A = space occupied by active data structures

B = space taken up in core by compiled programs

C = amount of core storage managed by the LISP 2 system

D = amount of space for temporary data structures allocated during the run of a program

Then $C - B - A$ = amount of space available for allocation of new data structure

$\frac{D}{C-B-A}$ = approximate number of garbage collections to occur during the run

kA = time for one garbage collection

Thus $\frac{kAD}{C-B-A}$ = total overhead generated by the run

5.1 ACTIVE DATA STRUCTURES

The garbage collection overhead is spent largely on the active data structures. The appellation "garbage collection" is misleading in the sense that the time is not spent gathering together the inactive sites but in transforming the data organization. The cost of garbage collection is inversely proportional to the derived benefits: the more space reclaimed, the less the overhead; the less space reclaimed, the greater the overhead.

In the derivation it was assumed that the overhead per garbage collection is proportional to the amount of space occupied by active data structures. Actually, the overhead also varies linearly with the number of active data units, the number of pointers in active data units, and the size of the data zones.

The "A" factor appears in both the numerator and the denominator, so that a reduction in the "A" factor has a second-order effect on reducing overhead. There are a number of system and programming techniques available for reducing the "A" factor.

5.2 COMMON SUBSTRUCTURES

In many applications, such as algebraic manipulation, the data structure contains many common sub-expressions. By arranging the computation so that the common sub-expressions have a common representation, a decrease in the "A"

factor results. This technique is available to a programmer, and requires careful algorithm construction. In many cases the overlapping of common data substructures spells the difference between success and failure of a run.

In this version of LISP 2, a number of techniques are available to guarantee unique representation of unique data structures. Thus numbers are uniquely represented when in the cell format. Constant strings, called identifiers, are uniquely represented. The unique representation of constant list structures is facilitated through the programmer option of hashing list nodes.

5.3 ARRAYS AND N-TUPLES

When appropriate, the use of arrays and n-tuples in building complex data structures has several processing and overhead advantages. The advantage in processing is that the time to access a component is uniform. The advantage in overhead is that arrays and n-tuples involve less space and fewer pointers to represent a given amount of information. It pays to "shrink out the connections". However, using n-tuples will result in a space-saving only if an n-tuple is used frequently enough to balance the space taken up by the n-tuple definition.

5.4 TEMPORARY DATA STRUCTURES: ALLOCATION AND ERASURE

The derivation of the overhead equation assumes that for each garbage collection, the amount of active data structure remains constant. In practice, some of the data units active at the last collection will have become inactive, and some of the data units constructed since the last collection are still active. The important principles governing the "D" factor are: (1) the programmer should prefer algorithms that consume less storage for the representation of temporary data structures, and (2) the space taken up by temporary data structures should be released as soon as possible. The first point focuses on the allocation aspect of algorithms. An algorithm that uses few CONS'es is better than one using many. An algorithm that consumes less total space to achieve the same effect is to be preferred. However, there is a tradeoff between the compactness of data structures and the complexity of the algorithms necessary to process the data: complex algorithms are more difficult to program, sometimes run slower, and usually increase the "B" factor. The programmer should look for data representations that are both compact and easy to process.

The second point focuses on the erasure aspect of algorithms. There are a few obvious and straightforward techniques that may be applied to an algorithm after it has been debugged: a variable referencing a temporary data structure keeps the structure active. If the variable is set to NIL, or if the block in which the variable is bound is exited, then there are fewer references to the data structure. A data structure is erased only when there are no references to it. During input operations, the variables that are to receive the incoming data structure may be NIL'd before starting the input operation. A number of erasure facilities can probably be provided through the use of compiler optimization techniques.

5.5 INPUT/OUTPUT

In reducing the amount of resident active data structure, attention should be paid to the data structures that are independent of the current task and are not in use. These structures should be placed on secondary storage, where they will not participate in the overhead. To accomplish this effectively, it must be possible to externally represent data structures in a self-contained, structure-preserving format.

A print/read capability should be available which preserves the graph interconnections of the data units that make up the data structure. This capability would preserve the relations of the common sub-expressions, and would allow the external representation of circular data structures.

5.6 CODING OF THE GARBAGE COLLECTOR

Obviously the "k" factor would be reduced if the garbage collector were coded in machine language. However, the advantages of programming in a higher-level language such as LISP 2 substantially outweigh the possible gains in efficiency: the language itself improves through changes made during the implementation; the implementors learn one language; the coding of the garbage collector improves as improvements occur in compilation techniques; the availability of the system programs to the user is increased.

Note that because the zones are processed independently during each phase, the advent of multiprocessing and parallel processing techniques will result in a substantial improvement in overhead figures.

5.7 CORE STORAGE

Obviously, an increase in the "C" factor would result in less overhead. The cost of more core needs to be balanced against the increased efficiency. LISP systems tend to be large.

5.8 SOFTWARE PAGING

A reduction in the "B" factor through the use of software paging techniques will affect overhead favorably. Software paging is the use of secondary storage to hold binary programs not relevant to the current task. The "B" factor is separated from the "A" factor because the former is simpler to control using automatic techniques. The storage management of binary programs is independent of the operation of the garbage collector. Several software paging packages have been proposed for this version of LISP 2, and one has been coded for the Q-32 LISP 2.

5.9 STORAGE BLOCK SIZE

There is a tradeoff between the size and the number of storage blocks. If there are more storage blocks, then the storage map will be larger. If the storage blocks are large, then the amount of committed vacant space becomes

considerable. Once a storage block is assigned to a zone, it is committed to holding the data units characteristic of that zone. On the average there will be half a block of vacant committed space per zone.