Not-quite-linear Random Access Memories

Jim des Rivieres
Wilf LaLonde
Mike Dixon

SCS-TR-20
August 1982
Revised March 1, 1983

School of Computer Science
Carleton University
Ottawa, K1S 5B6
Canada

This research was supported by the Natural Sciences Engineering Research Council of Canada.

Not-quite-linear Random Access Memories

Jim des Rivieres, Wilf LaLonde, and Mike Dixon
School of Computer Science
Carleton University

Abstract

Not-quite-linear random access memories (NQL-RAMs) are an alternative to conventional linear RAMs, that are better suited to list processing applications, such as LISP systems. Some of their advantageous characteristics are:

General-purpose.
Like conventional memories, NQL-RAMs are quite simple and flexible. They do not impose severe constraints on the shapes of objects that may be represented. All words within a multi-word object have addresses.

Compact representations.
NQL-RAMs are well-suited to applications involving a large number of small objects. Tagged pointers, tagged words, "immediate" pointers, "invisible" words, and compact list representations ("CDR-coding") are all supported with memory system overhead as little as three bits per word.

Memory level garbage collection.
Sufficient information is encoded in an NQL-RAM to support application independent garbage collection and compaction, with linearization of sequence-like multi-word structures.

Implementable.
All primitive operations of NQL-RAMs can be efficiently implemented either in microcode or with special-purpose hardware. Conventional memory technologies can be used at the implementation level.

The general structure of NQL-RAMs, their use in list processing, and their implementation are all discussed in this paper.

Keywords and Phrases: data structure representations, linear random access memories, list processing, garbage collection, LISP, compaction, linearization, memory management, memory architecture.

1. Introduction

Linear random access memories (RAMs) are one-dimensional arrays of fixed-sized storage units (words), each of which has a unique address. Upon presenting any address to the memory unit, the contents of that word can be retrieved or changed, all for a small fixed cost that is (ideally) independent of the word addressed, its contents, and the past history of the memory unit. Word addresses are represented in the memory and registers as consecutive integers; this makes it very efficient to calculate the address of an arbitrarily distant word given a base address and an integer offset (displacement). This property is especially important for applications that deal extensively with large rectangular arrays, since any element of the array can be accessed equally easily (and cheaply). Their highly regular structure allows linear RAMs to be economically realized with hardware. Low cost of production, together with their simple general-purpose structure, account for their ubiquity --- it is hard to find a computer that does not use linear RAMs as the basis of their primary store.

At the outset, we must make it clear that we will be more concerned with memory structures in the abstract than with actual realizations. Of central interest will be memory structures that can support the needs of list processing systems, such as LISP.

1.1 Linear RAMs are not always ideal

A linear RAM affords the lowest levels of system software with what can be best described as "a big block of words". (In practice, machines may make several linear RAMs available at the machine language level, but this complication need not concern us here.) As mentioned before, this very simple topology is convenient for efficiently representing rectangular arrays. But for a growing number of sophisticated applications, the capabilities of such simplistic memory structures are less than ideal and fall far short of providing for the basic needs common to all these systems. This shortcoming is reflected by the substantial amount of work invested in the development and implementation of memory managers.

Three characteristics of data structures found in list processing applications that do not mesh well with bare linear RAMs are:

- Heterogeneous structures
- Time-varying storage requirements
- Indeterminate lifespans

A wide variety of techniques have been developed to allow the efficient implementation of list processing systems on top of conventional linear RAMs. The basic technique is to abandon the notion of storing objects in entirely consecutive memory locations in favour of "linked" data structures [ref Knuth, Vol.1 Ch. 2] whereby complex objects are stored in a number of small blocks of consecutive locations and linked together explicitly with stored pointers (links, addresses, or references). LISP systems are excellent examples of systems based almost entirely on linked data structures [ref Allen].

When linked data structures are used extensively and there are a wide variety of "pieces" floating around inside a single region of memory, it usually becomes necessary to tag them so that their type can be determined. There are several ways that this can be done. First, tags can be stored with each piece. Such tags are most useful for recording information about the shape of the object carrying the tag. Alternatively, tags can be associated with all pointers [ref Bishop]. This technique is most appropriate for recording time-invariant information about the referent of the pointer; it can also be used to encode access rights. Situations can arise where both pointer tags and word tags are desirable. When a system contains numerous small objects, the storage requirements for the tags may be excessive. Schemes exist in which all objects with the same tag are stored in pages of memory reserved for that variety of object, allowing tag information to be kept at the page level instead of with each word or each pointer [ref Steele BIBOP].

Considerable space savings can be realized by not squandering memory on very small unchanging objects, such as small integer values; instead, the values can be encoded in special "immediate" pointers [ref Moore]. Another benefit of this scheme is uniformity --- everything can be viewed as pointers.

If memory is a scarce resource and object lifespans are unknown, it becomes necessary to employ an automatic scheme that can identify and recycle the storage occupied by objects that have expired. This is typically done with either "reference counting" or "garbage collection" techniques [ref Cohen]. When consecutively stored objects of an assortment of sizes are stored in a single region of memory, fragmentation may make it difficult to reuse all unused words. Even schemes that maintain several regions, one for each size of object, can suffer from fragmentation on a macroscopic level. Techniques for repacking memory are often an integral part of garbage collectors.

Still further space savings can be achieved by using compact representation for frequently occuring structures. This has been done quite successfully in the case of LISP lists [ref Bobrow&Clark, Weinreb&Moon]. When this is done, it is usually also the case that the garbage compactor will shuffle memory so that the more compact representations can be utilized.

All of the techniques touched on above, and many more, have been used, or proposed for use, in some implementation of some LISP dialect at some time in the past two decades. Throughout this paper, we will make reference to the problems encountered by LISP implementors not because they are the only systems ill-served by linear RAMs --- the pain is felt much more widely --- but because they are the problems most familiar to us. (Hopefully, thith dithtinct biath will not prove to be an inconvenienth to our readerth.)

1.2 NQL-RAMs are an alternative to linear RAMs

We will not propose to impose a structure on top of RAMs; instead, we will suggest an alternative memory structure that we call a not-quite-linear random access memory (NQL-RAM). From the point of view of an application, an NQL-RAM looks similar to a conventional RAM --- it still has blocks of words. But there are several differences:

- o NQL-RAMs are oriented to "next" and "previous" locations, instead of arbitrary offsetting
- o consecutive words can be detached
- o a location can be replaced by another
- o all words contain a pointer, with small atomic data values encoded as "immediate" pointers
- o both words and pointers have tags

An NQL-RAM can be used to represent both sequence-like and vector-like data structures. The word and pointer tags are not used by the memory system itself; they are solely for the use of the application system. Objects may overlap; all words in a complex object have addresses. Structures may be moved without difficulty. Moreover, the information that must be present to implement the primitive operations on NQL-RAMs is sufficient to allow garbage collection, compaction, and linearization to be performed as part of the NQL-RAM implementation; i.e., where it is quite invisible to the application system.

Moreover, the primitive operations on NQL-RAMs are quite simple; efficient implementations are feasible with either microcode or special-purpose hardware (software implementations, while feasible, might be too expensive to be useful due to the low-level nature of memory systems). When compared with linear RAMs, the memory system overhead can be as low as three bits per word --- the rest of the word holds application-dependent information.

In this report, we begin by presenting a model of linear RAMs. A number of varieties of not-quite-linear RAMs are then obtained by relaxing some of the constraints of the original model. We then present an implementation of NQL-RAMs, showing how NQL-RAMs can be mapped to linear RAMs, how the primitive operations can be efficiently coded, and how linearizing garbage collection can be performed. In the final section, we discuss the implications of NQL-RAMs and topics for future research, including an NQL-RAM-based machine.

2. Linear RAMs and NQL-RAMs

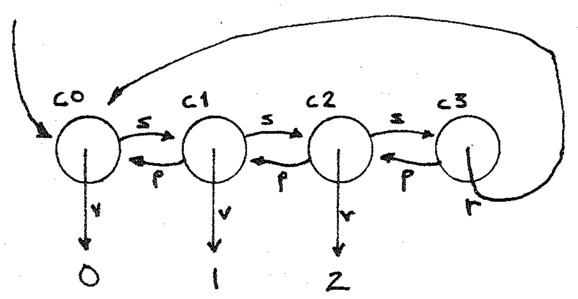
For the purposes of this paper, we will call the smallest addressable unit of storage in a memory a "word", and will assume that a word is large enough to hold the encoding of the address of any other word in the memory, with a little room to spare. We will assume that the words can be numbered 0, 1, 2, ..., Memory-size - 1, where Memory-size is a positive power of 2, and that addresses are the straightforward binary encoding of these word numbers. We will also assume that all operations on addresses are done modulo Memory-size.

2.1 Cell Graphs are used to model memory structures

In the spirit of Early´s V-graphs [ref Early], we can describe one view of a (tagged) linear memory in terms of an abstract structure that we will call a "cell graph". Each word in the memory will be modelled by a "cell". Cells will have unique "names"; this corresponds to the notion of addresses. (Since there is a one-to-one correspondence between cells and cell names, no harm is done by blurring the distinction between them. We´ll just call them cells.)

Cell graphs will have cells and atomic data as nodes with four kinds of arcs to interconnect them. A "successor" arc relates a cell to the cell representing the next higher-numbered word in memory; all cells but the one representing the highest numbered word have exactly one successor. A "predecessor" arc relates a cell to the cell representing the next lower-numbered word in memory; all cells but the one for word 0 have a single predecessor. A "contents" arc relates a cell to either a primitive atomic datum, such as an integer or a character, or another cell; we will refer to these as the cell´s "value" and "referent", respectively. In addition to a cell´s contents, we will also assume that each cell has a "cell tag" arc terminating at one of a small set of cell tag values (these values are application-dependent). The cell graph for a four word memory is shown in <Figure 1: 4-word>. The cells are named "C0", "C1", "C2", and "C3"; "s", "p", "r", and "v" are used to indicate successor, predecessor, referent, and value arcs, respectively; cell tag values are written inside the circles just to reduce clutter. Here, the first three cells contain integer values, whereas the last cell refers to cell C0; all four cells have the cell tag value "?". (In the future, we will not depict contents or cell tags when they are irrelevant.)

Some readers may have noted that the cell graph described above is not the usual model of a linear RAM in which every word is directly connected to every other word simply by offsetting. Indeed, our cell graph would model a linear SEQUENTIAL access memory if it weren´t for the referent arcs that allow jumping to an arbitrarily distant cell. However, we hope to show that this is a convenient way of viewing a linear random access memory and the ways that they are used to represent



<Figure 1: 4-word>

more complex data structures.

The operations that would allow us to manipulate such a cell graph are quite simple:

Contents(c)
: Designates either the atomic datum or cell at the terminus of the contents arc emanating from cell c.

Set_Contents(c,x)
: Establishes a contents arc from cell c to the atomic datum or cell denoted by x. The contents arc emanating from c prior to this operation is deleted.

Has_Value(c)
: True iff cell c´s contents is an atomic datum (as opposed to a cell).

Cell_Tag(c)
: Designates the value indicated by the cell tag arc from cell c.

Set_Cell_Tag(c,t)
: Changes the cell tag of cell c to t. The previous cell tag arc is discarded.

First(c)
: True iff cell c has no predecessor (i.e., true only of the cell that represents the word numbered 0).

Last(c)
: True iff cell c has no successor.

Successor(c)
: Designates the cell that is the successor of cell c. It is an error if cell c has no successor (i.e., if Last(c) holds).

Predecessor(c)
: Designates the cell that is the predecessor of cell c. It is an error if c has no predecessor (i.e if First(c) holds).

So far, we have provided no operations that would alter the successor and predecessor relationships between cells, nor any way of creating new cell graphs. We can assume that the applications level manipulating the cell graph is given the name of one of the cells, say, the first. The application would then be able to use any of the primitive operations to manipulate the cell graph in any way that it chooses, subject, of course, to the rules enforced by the primitives.

We shall now consider some less conventional operations --- operations to modify the successor and predecessor relationships between cells. Although arbitrary changes could be allowed, implementation considerations and usefuleness of the operations will limit the full range.

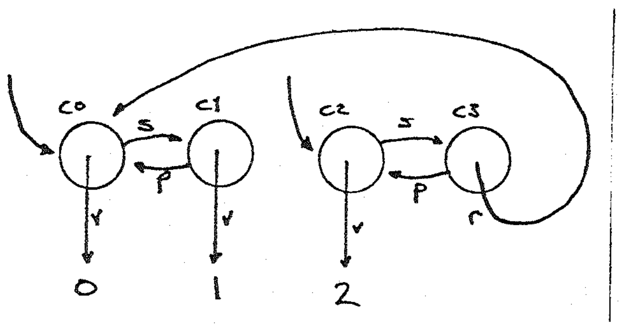
So as not to appear completely arbitrary in the choosing of a such a set of operations, we will consider progressive additions to the basic set and an application for the new operations. We don´t claim that the resulting set is either minimal or complete; however, we do believe that the resulting memory devices may retain the generality and cheap hardware realization of linear RAMs while at the same time providing a richer set of capabilities. By challenging some of the fundamental assumptions upon which linear RAMs are based, we hope to see an improvement in the memories with a concommitant simplification in the lower levels of application programs. Having presented cell graphs, our straw dog model of linear RAMs, we will hack away at it.

2.2 Successor and predecessor relationships can be dismantled

The first victim will be the permanence of successor and predecessor relationships. Why do two cells that are initially connected by successor and predecessor arcs in a cell graph have to remain connected? When storage is being allocated dynamically, it is often the case that two unrelated data structures will happen to occupy adjacent blocks of memory locations. The ability to disconnect adjacent cells would allow one to explicitly indicate that these cells need not be kept together since the program makes no use of their adjacency.

Breaking inter-cell connections also makes sense at a macroscopic level. If a program was given a single large cell graph, it would be able to subdivide it into as many pieces as it needed. Calling these pieces "memoroids" {Note Sydney J.}, we note that each of the memoroids would have the same general topology as the original cell graph, except that each would have a smaller number of cells. It would still be possible to create contents arcs from one memoroid to its own cells or to the cells of any other memoroid. <Figure 2: split-4-word> shows the cell graph of <Figure 1: 4-word> after it has been split into a pair of two-celled memoroids.

Let´s postulate the existence of a new primitive operation, Forget_Successor(c), that will delete the successor and predecessor arcs between cell c and Successor(c) if it exists. Assuming that the global variable Root initially locates the first cell of the entire cell graph, we can write an allocation procedure that returns a memoroid of a



<Figure 2: split-4-word>

requested size. (Algorithms will be expressed in a PASCAL-like language)

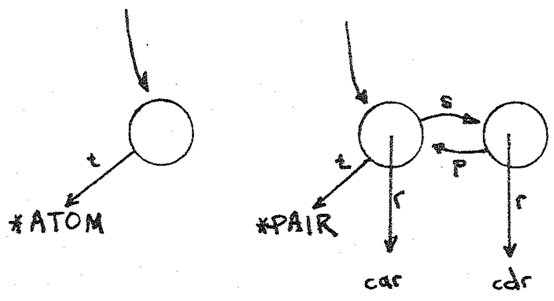
```
FUNCTION New (N)
   LOCAL VARIABLES First_Cell_In_Memoroid, Last_Cell_In_Memoroid
   BEGIN
      First_Cell_In_Memoroid := Root
      FOR I := 1 TO N DO
         IF Last(Root) THEN Error END
         Last_Cell_In_Memoroid := Root
         Root := Successor(Root)
      END
      {Disconnect the memoroid from the Root memoroid.}
      Forget_Successor (Last_Cell_In_Memoroid)
      RETURN First_Cell_In_Memoroid
   END
```

We will assume this definition of New throughout the following examples. Ultimately, New will be promoted to the status of primitive operation.

2.3 Example: LISP data structures

As an example of the use of memoroids, we will show how LISP's (dotted) pairs and atoms could be represented and manipulated. A quick review of LISP is in order. The eight LISP primitives that we will discuss are as follows. Atoms are zero-component structures that are manufactured by GENSYM(); every call to GENSYM creates a different atom. Pairs are two-component structures constructed by CONS(x,y) where both x and y are LISP objects. Each call to CONS creates a distinct pair. A LISP object is either an atom or a pair. EQ(x,y) is true iff x and y are (is?) the same LISP object. ATOM(x) is true iff x is an atom. The first component of a pair "p" can be extracted with CAR(p); i.e., EQ(CAR(CONS(x,y)),x) always holds. Similarly, the second component of such a pair can be retrieved with CDR(p). The relationships between a pair and the LISP objects it has as its components are mutable. RPLACA(p,x) and RPLACD(p,y) will change the the first and second fields of pair p to x and y, respectively.

Because pairs can have two other pairs as component, and because the relationships between a pair and its components can change, it will



<Figure 3: LISP-memoroids>

be most convenient to use a linked data structure. Atoms can be represented by a one-celled memoroid with cell tag "*ATOM". Pairs can be mapped to two-celled memoroids; the first cell, which will be used as the root of a pair, will carry the tag "*PAIR" and have a referent to (the root of) the memoroid representing the CAR object; the second cell will have a referent to the CDR object´s memoroid (no need for a cell tag on this cell). <Figure 3: LISP-memoroids> summarizes these static properties.

The eight operations can be implemented as follows:

```
FUNCTION GENSYM ()
   LOCAL VARIABLES Result
   BEGIN
      Result := New(1)
      Set_Cell_Tag (Result, "*ATOM")
      RETURN Result
   END

FUNCTION CONS (X, Y)
   LOCAL VARIABLES Result
   BEGIN
      Result := New (2)
      Set_Cell_Tag (Result, "*PAIR")
      Set_Contents (Result, X)
      Set_Contents (Successor(Result), Y)
      RETURN Result
   END

FUNCTION EQ (X, Y)
   RETURN X = Y       {i.e., the same cell.}

FUNCTION ATOM (X)
   RETURN Cell_Tag (X) = "*ATOM"

FUNCTION CAR (P)
   IF Cell_Tag(P) = "*PAIR" THEN RETURN Contents(P) ELSE Error END

FUNCTION CDR (P)
   IF Cell_Tag(P) = "*PAIR" THEN RETURN Contents(Successor(P)) ELSE Error END
```

```
PROCEDURE RPLACA (P, X)
   IF Cell_Tag(P) = "*PAIR" THEN Set_Contents (P, X) ELSE Error END

PROCEDURE RPLACD (P, X)
   IF Cell_Tag(P) = "*PAIR" THEN Set_Contents (Successor(P), X) ELSE Error END
```

2.4 Garbage collection

One of the things that goes hand in hand with the implementation of LISP´s data structures is a method of identifying and reusing the storage for LISP objects that are no longer needed. This task, called "garbage collection", is usually done by first identifying all of the active LISP objects, and then recycling the rest. The only active LISP objects are ones to which the program holds a direct pointer, and ones that are a component of some other active LISP object. This process of marking all active LISP objects is captured by the procedure called Mark.

```
PROCEDURE Mark (X)
   BEGIN
      IF Cell_Is_Marked(X) THEN RETURN END
      Mark_Cell (X)
      IF NOT First(X) THEN Mark (Predecessor(X)) END
      IF NOT Last(X) THEN Mark (Successor(X)) END
      IF NOT Atomic(Contents(X)) THEN Mark (Contents(X)) END
   END
```

(Here we are assuming that Cell_Is_Marked(X) will be true iff cell X has been passed to Mark_Cell at some earlier time. How this is accomplished is not germane to the discussion.)

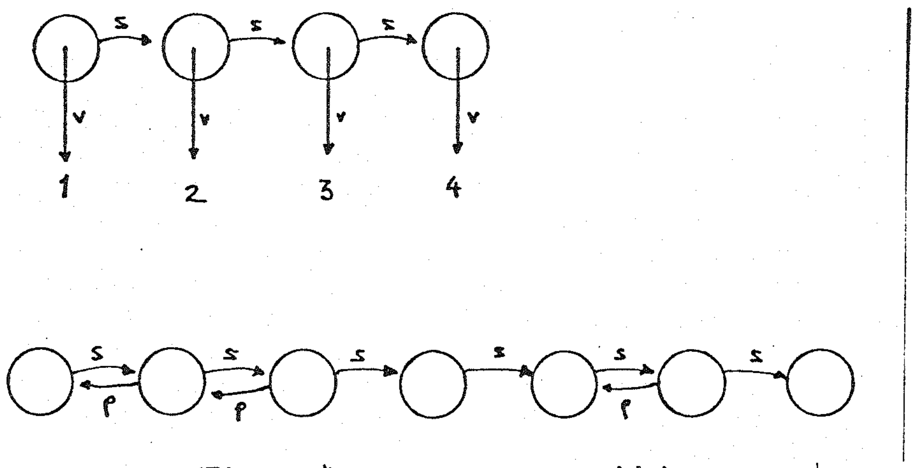
The important thing to note is that Mark is ignorant of the structure of LISP objects. It uses only the cell graph primitives to visit each of the cells in a memoroid and their referents. If Mark had been given Root prior to issuing any Forget_Successor commands, all of the cells would have ended up being marked. However, once Forget_Successor has been used there is a distinct possibility that some cells will never get marked. The moral: there is no useful notion of accessibility at the level of conventional linear RAMs; hence, garbage collection must be accomplished at a higher level. But if there is a way to indicate that consecutive memory locations are not related as far as the application level is concerned, a useful notion of accessibility emerges.

The other part of garbage collection, the recycling of inaccessible cells, cannot be expressed at the application level in terms of the primitive operations because there is no way of locating any of the inaccessible cells. However, it would be possible to do this at the level at which the primitives are implemented. All of the information that is required to correctly implement the primitives also serves the needs of a compacting garbage collector.

2.5 Successor and predecessor need not be symmetric

Until now, Successor(c1) = c2 iff Predecessor(c2) = c1. The next straw we remove will destroy this symmetry. We will postulate the existence of an additional primitive, Forget_Predecessor(c), that will remove any predecessor arc emanating from cell c without erasing any successor arc terminating at c. The result will be that Successor(c1) = c2 need not imply Predecessor(c2) = c1 --- or even that c2 has a predecessor. Note, however, that Predecessor(c2) = c1 still implies Successor(c1) = c2.

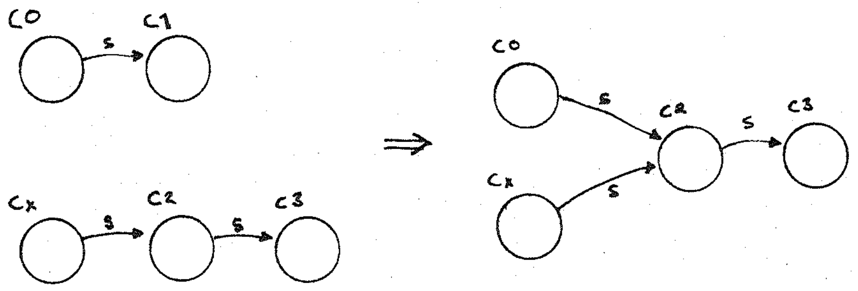
It is now possible to build memoroids that are traversible in the forward direction only (i.e., via successor arcs), in contrast to the old ones whose cells could also be visited in backwards order. This property is reminiscent of sequential files and other sequence types, so we will call them "sequence memoroids" and use the term "vector memoroids" for the original variety. We will take the class of sequence memoroids to include all of the hybrids that have a mixture of cells with and without predecessors because they can be viewed as sequences of vector memoroids. Some sequence memoroids are depicted in <Figure 4: sequence-memoroids>.



<Figure 4: sequence-memoroids>

One of the main properties of sequence data types is the ability to extend them (think of a sequential file). Another characteristic is that they are often implemented in such a way as to allow different sequences to share storage whenever possible (think of LISP's lists). There should be no doubts that both of these properties could be achieved by representing sequence data types with a bunch of two-celled vector memoroids, each holding one element of the sequence and a referent to the remainder of the sequence (this is exactly how most LISPs represent lists.) Unfortunately, the provided set of operations (so far) on sequence memoroids are not flexible enough to encode the fully general "next" relationship required for sequences --- it is not yet possible to construct two cells that share the same successor, and all successor relationships are created at "the beginning of time" when the original cell graph was constructed.

What we propose to do is add yet another primitive operation --- one that will alter the successor topology of a cell graph. Replace(c1,c2) will replace cell c1 with cell c2. So that we don't "warp" the memory structure too severely; we will insist that cell c1 be isolated; that is, First(c1) and Last(c1) both hold. The result of Replace(c1,c2) will be that the referent, cell-tag, and contents arcs emanating from c1 will be vaporized along with the cell c1, and all successor and referent arcs previously terminating at c1 will be redirected to end at cell c2 (see <Figure 5: replace>). This operation, albeit a strange one, will have the desired effect of making the sequence-like properties of sequence memoroids rich enough to support sequence data types, without making it too different from linear RAMs so as to preclude a reasonable implementation.



<Figure 5: replace>

2.6 Example: Linear LISP lists

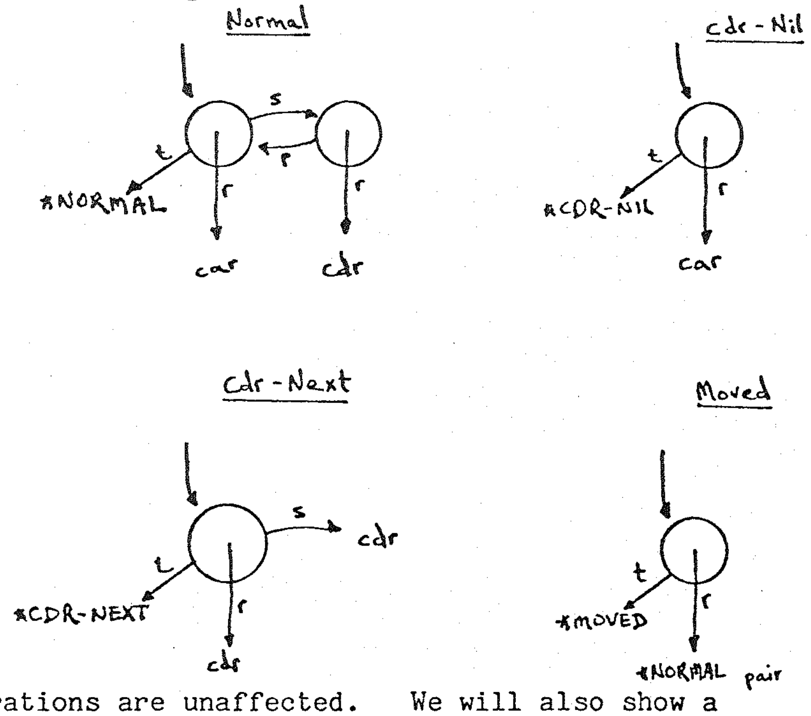
LISP uses CDR-linked pairs to represent lists; that is, the first element of a list is remembered in the CAR component of a pair and the CDR component is used to point to the list containing the rest of the elements in the list. The distinguished atom NIL is used to terminate lists, making it synonymous with the empty list. Such a large preponderence of a LISP data base is made up of lists that considerable space can be saved by adding an alternative representation of pairs in which the CDR relation is encoded by proximity instead of pointers. This trick is known "CDR-coding" [ref Allen] because each word representing a pair must carry a two bit tag indicating one of four possibile pair representations:

NORMAL	This word and the one following it constitute a normal pair. This word contains a pointer to the CAR component; the next word points to the CDR component.
CDR-NIL	This word encodes a pair representing a one element list. As usual, this word contains a pointer to the CAR component, but the CDR component is the atom NIL (however, no pointer to NIL is stored). This one word structure allows a substantial space saving because this sort of pair occurs quite frequently.
CDR-NEXT	This word encodes a pair representing one segment of a list; this word contains a pointer to the CAR component, and the next word IS the

CDR component of this pair (instead of a pointer to it).

MOVED	This word encodes a pair that has had to be moved elsewhere; the contents is a pointer to that location.

These four representations can be described with sequence memoroids quite readily (see <Figure 6: CDR-coding>).



<Figure 6: CDR-coding>

Most of the other operations are unaffected. We will also show a new constructor, LIST, that will make good use of the economical representation for pairs.

```
FUNCTION CONS (X, Y)
   LOCAL VARIABLES Result
   IF EQ(Y,Nil_Atom) THEN
      Result := New(1)
      Set_Cell_Tag (Result, "*CDR-NIL")
      Set_Contents (Result, X)
      RETURN Result
   ELSE
      Result := New(2)
      Set_Cell_Tag (Result, "*NORMAL")
      Set_Contents (Result, X)
      Set_Contents (Successor(Result), Y)
      RETURN Result
   END
```

Note that "real" implementations of CONS will look for the case when the next word to be allocated immediately precedes the CDR object and create a CDR-NEXT cell instead of a two-celled NORMAL pair. This trick, combined with allocating memory from high addresses to low, can cut down on the amount of storage required. The implementation of CONS shown above does not do this. However, by writing a version of New that allocated cells from the other end of the cell graph and did not break the successor arc from the next cell to be allocated to the first cell in the memoroid just allocated, it would be possible to achieve the equivalent effect.

```
FUNCTION LIST (X[1..N])
   LOCAL VARIABLES Result, Finger, I
   IF N = 0 THEN
      RETURN Nil_Atom
   ELSE
      Result := New(N)
      Finger := Result
      FOR I := 1 TO N - 1 DO
         Set_Cell_Tag (Finger, "*CDR-NEXT")
         Set_Contents (Finger, X [I])
         Finger := Successor(Finger)
         Forget_Predecessor (Finger)
      END
      Set_Cell_Tag (Finger, "*CDR-NIL")
      Set_Contents (Finger, X[N])
      Forget_Predecessor (Finger)
      RETURN Result
   END

FUNCTION CAR (P)
   CASE Cell_Tag(P) OF
      "*CDR-NIL", "*CDR-NEXT", "*NORMAL":  RETURN Contents(P)
      "*MOVED":   RETURN CAR(Contents(P))
      OTHERWISE:   Error
   END

FUNCTION CDR (P)
   CASE Cell_Tag(P) OF
      "*NORMAL":   RETURN Contents(Successor(P))
      "*CDR-NIL":  RETURN Nil_Atom
      "*CDR-NEXT": RETURN Successor(P)
      "*MOVED":    RETURN CDR(Contents(P))
      OTHERWISE:    Error
   END

PROCEDURE RPLACA (P, X)
   CASE Cell_Tag(P) OF
      "*CDR-NIL", "*CDR-NEXT", "*NORMAL": Set_Contents (P, X)
      "*MOVED":  RPLACD (Contents(P), X)
      OTHERWISE:  Error
   END
```

The only primitive that is complicated by CDR-coding is RPLACD. The problem is simple: the CDR relationship on pairs is fully mutable, yet two of the representations, CDR-NEXT and CDR-NIL, have already used their only fully mutable cell relationship, contents, for the CAR relationship. That´s where the MOVED variation becomes necessary. A new NORMAL pair is constructed elsewhere with the CAR from the old pair and the new CDR. Then, the old cell´s tag is set to MOVED and its contents changed to point to the new pair.

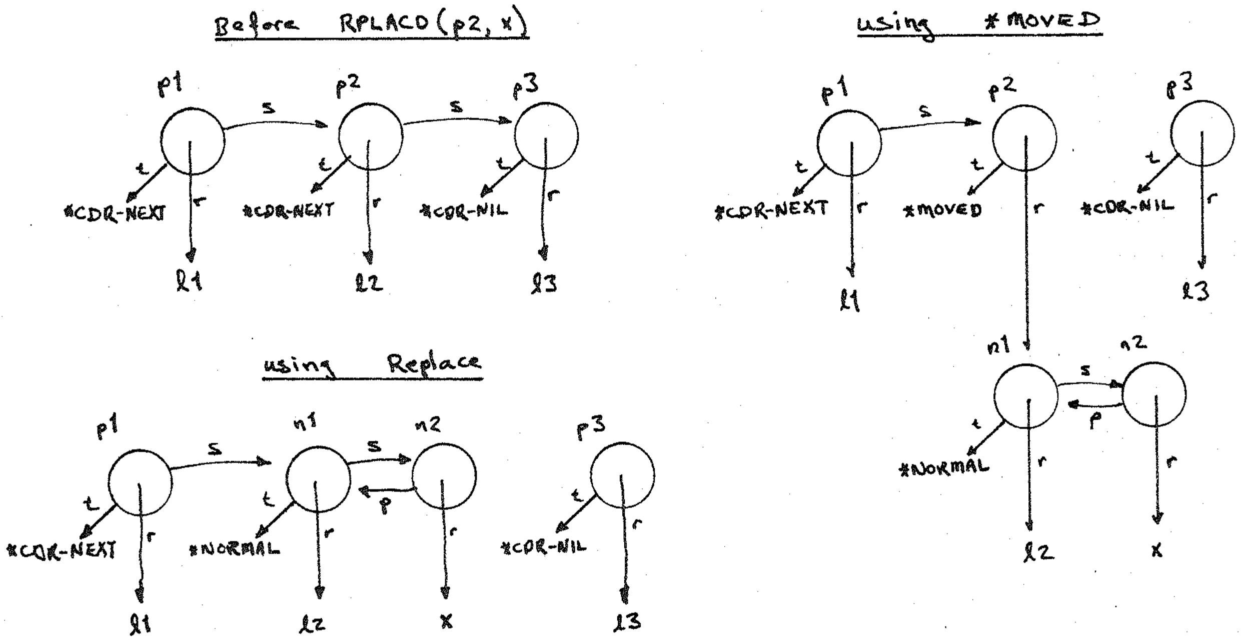
```
PROCEDURE RPLACD (P, X)
   LOCAL VARIABLES Extension
   CASE Cell_Tag(P) OF
      "*NORMAL":   Set_Contents (Successor(P), X)
      "*MOVED":    RPLACD (Contents(P), X)
      "*CDR-NEXT", "*CDR-NIL":
         IF EQ (X, Nil_Atom) THEN
            Set_Cell_Tag (P, "*CDR-NIL")
         ELSE
            Extension := CONS(Contents(P),X)
            Set_Cell_Tag (P, "*MOVED")
            Set_Contents (P, Extension)
         END
      OTHERWISE:    Error
   END
```

These are not the most clever implementations of the primitives possible. A standard trick is to ensure that no primitive ever returns a MOVED variety of pair. This reduces the buildup of long chains of MOVED pairs and simplifies the coding of the primitives.

There is one other small point that could be addressed. When cells are changed from the CDR-NEXT to CDR-NIL or MOVED variations, as happens in RPLACD, the successor arc from that cell could be discarded using Forget_Successor. Otherwise, some cells might appear to remain accessible from that cell even though they are not.

The preceding rendition of CDR-coded lists made no important use of the capabilities of the primitives that support sequence memoroids. However, the situations that cause RPLACD to change CDR-NEXT and CDR-NIL cells into MOVED ones are precisely those where Replace can be used; i.e. instead of handling MOVED cells via a special tag, we can use the more direct topology changing operations. Thus, instead of changing the cell so that its referent is the new pair, we replace that old cell by the (root cell of the) new pair. This will allow us to get rid of the MOVED form of pair. The modified versions of CAR, CDR, and RPLACA are obtained by deleting the MOVED case. The effects of both versions of RPLACD are shown in <Figure 7: rplacd>.

A revised RPLACD follows.



<Figure 7: rplacd>

```
PROCEDURE RPLACD (P, X)
   LOCAL VARIABLES Extension
   CASE Cell_Tag(P) OF
      "*NORMAL":    Set_Contents (Successor(P), X)
      "*CDR-NEXT", "*CDR-NIL":
         IF EQ(X,Nil_Atom) THEN
            Set_Cell_Tag (P, "*CDR-NIL")
            Forget_Successor (P)
         ELSE
            Extension := CONS(Contents(P),X)
            Forget_Successor (P)
            Replace (P, Extension)
         END
      OTHERWISE:     Error
   END
```

We can go even further, and eliminate NORMAL tags entirely by always encoding the CDR relationship with successor arcs. This will require only minor changes to CAR, CDR, RPLACA, and RPLACD. Only CONS needs to be rewritten:

```
FUNCTION CONS (X, Y)
   LOCAL VARIABLES Result, Dummy
   IF EQ(Y,Nil_Atom) THEN
      Result := New(1)
      Set_Cell_Tag (Result, "*CDR-NIL")
      Set_Contents (Result, X)
      RETURN Result
   ELSE
      Result := New(2)
      Set_Cell_Tag (Result, "*CDR-NEXT")
      Set_Contents (Result, X)
      Dummy := Successor(Result)
      Forget_Predecessor(Dummy)
      Replace (Dummy, Y)
      RETURN Result
   END
```

One last comment about "real" implementations of CDR-coded lists. Since there is a choice of representations for pairs, some taking more storage than others, it is usually the case that the garbage collector will attempt to "linearize" list structures so as to minimize the amount of storage. As we will show in the next section, all of the information required to linearize arbitrary sequence memoroids is already present.

2.7 Summary

The adjectival phrase "not quite linear" is somewhat applicable to both of the non-standard memory structures proposed in this section. Forget_Successor, by itself, permits the subdivision of linear RAMs into smaller linear RAMs. This gives rise to a useful notion of accessibility upon which to base garbage collection. A quite workable memory system could be based on the standard primitives together with Forget_Successor and New.

The addition of Forget_Predecessor allows cells to be isolated without destroying successor arcs that terminate at a cell. This asymmetry means that the topology of the memory will be coherent even when several cells have a common successor --- a situation that can only be created by Replace. The memory structure that results has promise in list processing applications.

A different class of memories could be obtained by eliminating the predecessor relationship entirely. Removing Predecessor, Forget_Predecessor, and First from the set of primitives gives a variety of sequence-like memory without bi-directional accessibility.

3. Implementation of NQL-RAMs

In this section we present an implementation of an NQL-RAM with both vector and sequence memoroids. Since a conventional linear memory will be used to represent the NQL-RAM, we will use the cell graph terms (e.g., "cell" and "cell name") when referring to entities in the memory system being implemented, and use conventional terminology ("word" and "address") for the memory system being used at the implementation level. (For reasons that will become apparent, the implementation scheme that we will present is called a "Snakes and glue" realization of an NQL-RAM.)

3.1 Memory system interface

The memory system interface that we will use is quite crude. All communication from the application level to the memory system will be done with procedure calls. While we assume that the application level has the capability of dealing with small integers and booleans, we will, for the most part, keep the operands to and the results of memory system primitives in a set of memory system registers. Such a low level protocol was chosen because it allows us to discuss the numerous implementation issues at their own level.

Altering our terminology somewhat from preceding sections, we will say that each register can contain an "object", of which there are two varieties: "value" objects and "cell" objects. Value objects represent atomic data; cell objects represent cell names. Also, each object carries an object tag (not to be confused with a cell tag); object tags are provided so that an application can keep a small amount of information with each object. Here are the operations on objects that do not involve cells:

Make_Value_Object(T,V,R) Creates a new value object with value V and object tag T and places that object in register R.

Value(R) Returns true iff register R contains a value object.

Get_Value(R) Returns the value of the value object in register R. Error if register R does not contain a value object.

Get_Object_Tag(R) Returns the object tag of the object contained in register R.

Set_Object_Tag(R,T) Changes the object tag of the object in register R to T.

Copy_Object(R1,R2) Copies the object in register R1 into register R2. R1 is unaffected.

Same_Object(R1,R2) Returns true iff either both R1 and R2 contain value objects and their values are the same, or both R1 and R2 contain cell objects for the same cell. The object tags are not considered.

Same_Tagged_Object(R1,R2) Same as Same_Object(R1,R2), except the object tags must be identical.

Notice that none of these operations will allow a value object to be converted into a cell object, or vice versa. The application level is forced to deal with all cell objects via the interface. Because of this property, the application level will always be completely independent of the way in which objects and cells are implemented.

A note on errors. Throughout the implementation we will run across situations in which errors will be detected. Although no mechanism is shown by which the application could be informed of these problems, we assume that this will be done (somehow).

The operations on the NQL-RAM proper are those of the preceding section --- only their names and calling conventions have been changed. Initially, all cells in the NQL-RAM are inaccessible. New_Vector and New_Cell_With_Successor are the only operations that can increase the number of cells available to the application. The cell graph manipulation primitives are as follows:

First_Cell(R) Returns true iff the cell object in register R designates a cell having no predecessor. Error if R does not contain a cell object. (All of the primitives on cell objects are strict, so we need not repeat this caveat).

Last_Cell(R) Returns true iff the cell object in register R has no successor.

Successor_Cell(R1,R2) Makes register R2 contain a cell object that designates the successor of the cell designated by the cell object in register R1. Error if Last_Cell(R1). The default object tag[1] will be associated with the cell object in R2.

Prececessor_Cell(R1,R2) Sets register R2 to the (cell object for) the predecessor of (the cell designated by the cell object of) R1. Error if First_Cell(R1). Again, the default object tag is used on the result object.

Forget_Successor(R) Breaks any successor and predecessor links between cell R and its successor (if any).

Forget_Predecessor(R) Breaks the predecessor connection between cell R and its predecessor, if it has one.

Fetch_Cell_Contents(R1,R2) The object that is the contents of cell R1 is placed in register R2.

1. R_2 designates (i.e. points to) a cell whose contents (see Fetch-Cell-Contents) has a user defined tag. Thus R_2 has no associated tag. Rather than leave it undefined, it is set to a default.

Store_Cell_Contents(R1,R2) The contents of cell R1 is changed to the object contained in register R2.

Isolate_And_Replace(R1,R2) Isolates cell R1 via Forget_Successor(R1) and Forget_Predecessor(R1). Then, provided that cells R1 and R2 are different, the cell R1 is vaporized and cell R2 inherits all successor and referent arcs terminating at the deceased cell.

Get_Cell_Tag(R) Returns the cell tag (not the object tag) associated with the cell designated by the cell object contained in register R.

Set_Cell_Tag(R,T) Changes the cell tag of the cell designated by the cell object contained in register R to T.

New_Vector(N,R) Sets register R to be a cell object for the first cell of a previously inaccessible group of N (>0) cells. All cells except the first will have predecessors; all cells except the last have successors; object tag, cell tag, and cell contents are all set to their default values.

New_Cell_With_Successor(R1,R2) Sets register R2 to be a never-before-seen cell whose successor is cell R1. Object tag, cell tag, and cell contents are all defaulted.

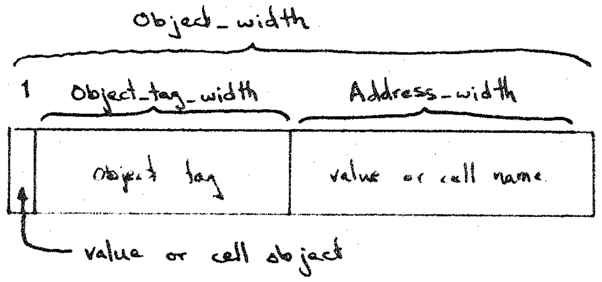
Offset(R1,N,R2) Sets register R2 to a cell object for the cell that is N cells distant from the base cell, R1. If N = 0, the base cell will be used (but the object tag will be the default); if N > 0, the target cell is reached by N successor arcs; if N < 0, predecessor arcs are followed.

3.2 Mapping cells to words

The key idea used in mapping cells in an NQL-RAM to words of a linear RAM is that cells related by successor and predecessor arcs will be stored in adjacent locations whenever possible The cell´s tag and contents will be stored in the corresponding word. A few extra bits in the word will be needed to record whether or not the cell has a predecessor and/or successor. Cell names can be represented by the address of the word that corresponds to the cell.

Without loss of generality, assume that the underlying linear RAM is dedicated to the implementation of the cells of the NQL-RAM, has a total of Memory_Size words where Memory_Size = 2**Address_Width, and that the word addresses are just integers in the range 0 to Memory_Size - 1. Assuming that object tags are Object_Tag_Width bits wide, and that Memory_Size atomic values will be sufficient, an object will require Object_Width = 1+ Object_Tag_Width + Address_Width bits --- the extra bit is required to distinguish cell objects from value objects (see <Figure 8: object-format>).

When c1 = Predecessor(c2), we say that cell c1 is "strongly glued



<Figure 8: object-format>

right" to c2. When Successor(c1) = c2 but either c2 has no predecessor or its predecessor is some cell other than c1, we say that c1 is "weakly glued right" to c2. The topology of NQL-RAMs is such that there may be more than one cell weakly glued right to a given cell, but at most one cell strongly glued right to it. It is also possible for a cell to be weakly glued right to itself.

Define M to be a total function that maps cells in the NQL-RAM to words in the implementation level linear RAM. This cell-to-word map must satisfy the following three desiderata:

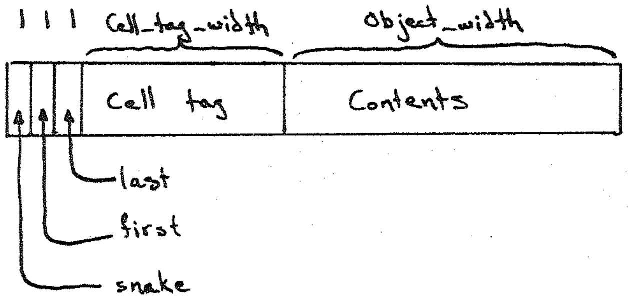
o Every cell must be mapped to a different word.

o If cell c1 is strongly glued right to cell c2, then these cells will be mapped to consecutive words (i.e. M(c1)+1 = M(c2)).

o If cell c1 is weakly glued right to cell c2, then either c1 and c2 will be mapped to consecutive words, or the word following that of c1 (i.e., M(c1)+1) will not be used to represent a cell.

That is, successor and predecessor relationships between cells of a vector memoroid will always be encoded implicitly by adjacency in the underlying memory. However, since several cells in a sequence memoroid may share the same successor cell, there will be situations in which weakly glued cells cannot always be stored side by side. In these cases, we will use the word following the first cell to encode a pointer to where the cell was mapped. All references to these words are automatically treated as references to their contents (e.g., the way "*MOVED" cells were treated in section 2.6). Known by various names, including "invisible words", "indirect words", "forwarding addresses", and "broken hearts", we choose to call them "snakes", from the childrens board game called Snakes and Ladders in which landing on a snake penalizes that player by causing his/her piece to slide down the snake to its tail. Is should be clear that an arbitrarily convoluted NQL-RAM having N cells can always be mapped to any linear RAM with at least 2N words, the worst case being when every cell is weakly glued to itself.

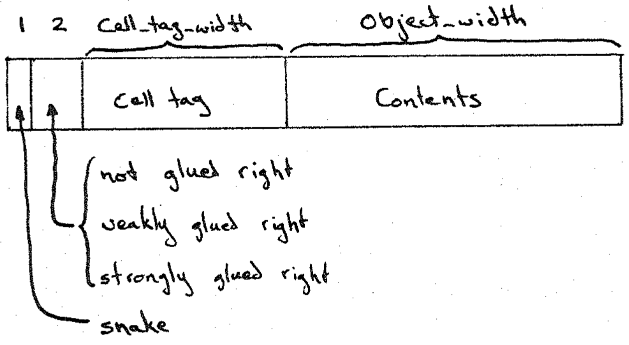
With the general properties of the cell-to-word mapping decided, the control information that must be put in a word amounts to three bits: one bit to indicate whether the word represents a snake or a cell, one to say that the cell encoded by this word has no predecessor, and



<Figure 9: loose-cell-words>

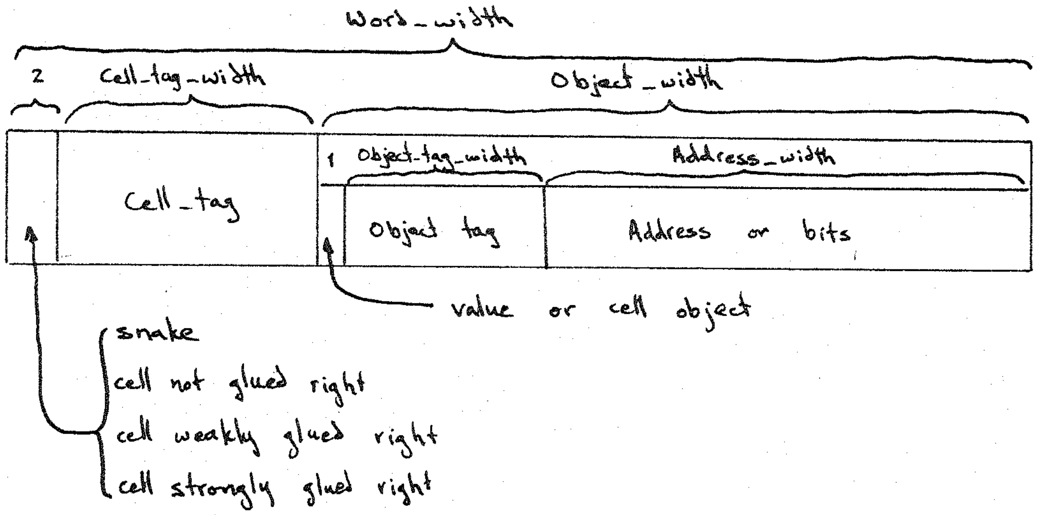
one to say that it has no successor. Assuming that cell tags will be given Cell_Tag_Width bits, a 3 + Cell_Tag_Width + Object_Width bit word format is shown in <Figure 9: loose-cell-words>.

While this cell representation does have the advantage of being self-contained, some of the information is redundant. In particular, the asymmetry of the predecessor and successor relationships ensures that if cell c1 has no successor, then any cell mapped to the word following c1´s will have no predecessor. Instead, we can store in a cell´s word a field that reflects how the cell is related to its successor: "strongly glued", "weakly glued", or no glue (has no successor) (see <Figure 10: non-redundant>). A snake word, which is never related to the following word, would always have glue setting "none". The principle advantage of this encoding is that Forget_Successor need only change the glue on the current cell´s word, while Forget_Predecessor need only alter the glue on the word preceding the current cell´s.



<Figure 10: non-redundant>

Once the switch to encoding glue information has been made, a further optimization is possible. Since the glue field on a snake word is always "none", the glue and snake fields can be combined into the two-bit control field that is shown in <Figure 11: tight-cell-words>. This word format, which we shall adopt, requires that the width of a word be Word_Width = 2 + Cell_Tag_Width + Object_Width. With the exception of the two bits used for snakes and glue and the value/cell bit required in the contents field object, all remaining bits are solely for the application´s use. Snake words must also be accounted as memory system overhead; how significant this will be is a function of how



<Figure 11: tight-cell-words>

Replace is used to warp the initially flat memory topology, and how the garbage compactor works.

To put this in perspective, an NQL-RAM with 16M cells could be mapped to 16M 32-bit linear memory with 5 bits left to be split between object and cell tags, assuming 24-bit word addresses are stored.

3.3 Registers and normalization

The presence of snake words creates a bit of a problem with the representation of cell objects. By rights, cell objects should contain the address of the word to which the cell is mapped. However, cell words will occasionally be changed to snake words by Replace, so we must admit the possibility that cell objects are not represented uniquely --- a cell object will encode either the address of its cell´s word or the address of a snake word whose contents field contains a cell object for that same cell. We will say that a cell object is "canonical" if the former case applies; value objects are always canonical. By "normalization" we mean the process of deriving a canonical object from a possibly non-canonical one. Normalization of a cell object involves accessing the word to which it points. If it is not a snake word, the cell object was canonical. If it is, the cell object stored in the snake word must be extracted and normalized instead.

Normalization is a relatively expensive operation since it required at least one memory access for a cell object, even canonical ones. With very few exceptions, primitive operations must deal with canonical cell objects --- after all, only canonical one provide direct access to the words that represent cells. It is an important implementation issue to decide when to perform the normalization of objects held in registers. The technique of keeping all objects in registers canonical at all times is undesirable since it would require that the Fetch_Cell_Contents operation, upon encountering a cell with a cell object as its contents, would have to make an additional memory access to see if that cell object was canonical. (This would probably be disasterous if the underlying memory was "virtual".) Instead, we will opt for a policy of normalizing only when necessary.

Nevertheless, normalization of registers will be a frequent enough operation to cause concern about unnecessary re-normalizations. So, for each register there will be a single flag that indicates whether the cell object in that register is known to be canonical. To ensure that inconsistencies never arise, Replace will simply reset all of the canonical flags.

The principle disadvantage of normalization only when necessary is that in some circumastances lengthy chains of snakes can build up. For example, this will happen when a cell c1 is replaced by cell c2 which, in turn, is replaced by c3, and so on. We propose three approaches for combatting this problem. First, we will provide a primitive, called Normalize, that the application may call whenever it suspects that a register contains a object that may be getting too un-canonical. Normalize will have no visible affect on the state of the world as viewed from the application's level, but it will arrange that the object held in the register be canonical. Second, all registers and all words will be guaranteed to contain canonical objects after garbage collection. The third approach involves associating with each memory system register the address of the word from which the object currently in the register was extracted (if any). Whenever a register normalization turned out to be non-trivial, the contents field of the source word could be set to the result of the normalization, of course, provided that the word's contents hadn't be altered since the register was loaded. We have not incorporated this in the implementation given in Appendix I; it would probably be worthwhile in environments where Replace sees heavy use, but would likely be unnecessary when the topology of the memory remains essentially flat. Still, it only marginally complicates the implementation and need not slow down normalization except in those cases where it does additional useful work.

3.4 Implementation of the primitive operations

Code for all of the primitive operations is given in Appendix I. Most of it is straightforward, so only a few salient points need be addressed here. The first has to do with New_Vector, the main cell allocation primitive. The pool of unallocated cells is located by the (internal) memory system register Creation_Free. In contrast to the version of New presented in the preceding section, Creation_Free locates the last unallocated cell in the block, not the first. Thus, New_Vector allocates cells in successively lower memory locations and returns an cell object for the lowest numbered (i.e. first) word in the block. It also assumes that the unallocated pool is a vector memoroid. Moreover, when a block of cells is given out, the cell located by Creation_Free will retain its weak connection to the first cell of the newly allocated block. This will permit New_Cell_With_Successor to capitalize on situations in which the successor of the to-be-allocated cell is the most recently allocated cell (or the first cell of the most recently allocated vector memoroid) without having to violate accessibility.

The second point is that Successor_Cell and Fetch_Cell_Contents may return a non-canonical result, whereas the results of New_Vector, New_Cell_With_Successor, and Predecessor_Cell are always canonical cell objects.

The final point has to do with Isolate_And_Replace. All snakes are created by Isolate_And_Replace (or other parts of the system that call Isolate_And_Replace). Thus the burden is on this operation to ensure that all cell objects have a canonical version; that is, it must be careful not to create circular snake chains. This can be accomplished simply by recognizing that requests to replace a cell by itself can be satisfied without creating a snake. Providing that the test for sameness is done on canonical cell objects, no infinite snakes can ever be created. We offer the following informal proof of this fact. If there are no infinite snakes in the system, then the normalizations done by Isolate_And_Replace will terminate, yielding two canonical cell objects. These two cells, call them x and y, will be mapped to words M(x) and M(y), respectively. If M(x) <> M(y) then turning word M(x) into a snake with referent y will simply mean that all extant cell objects formerly normalizing to the canonical object for cell x will now normalize to the canonical object for cell y. All other cell objects will normalize as before. Thus, as long as there are no infinite snakes initially there never will be any. One final observation: snake words, once created, will never be changed (except by the garbage collector).

3.5 Costs of the primitive operations

We will use the number of memory accesses that are required to perform the various primitives as a measure of the cost of these operations. Let R be the cost of fetching an entire word from the underlying RAM, and W be the incremental cost of writing a word that was just read. Let N be the cost of normalizing a cell object, c, that is not known to be canonical. If c is canonical this can be determined simply by reading the cell´s word at cost R. If c´s word is a snake, at least one more read will be required. So, N = nR for some n >= 1. Let N´ be the cost of normalizing a cell object that may be known to be canonical. If the register´s canonical flag is set, N´ will be zero. Otherwise, N´ = N.

The cost of the various primitive operations with respect to memory accesses is given in Table I. There are four general prices categories: the cost of reading the current cell´s word (e.g. Last_Cell, cost N) reading the word preceding the current cell´s word (e.g. First_Cell, cost N´+R), updating the current cell´s word (e.g. Forget_Successor, cost N+W), and updating the word preceding the current cell´s word (e.g. Forget_Predecessor, cost N´+R+W).

Table I - Costs of primitive operations (in memory access)

New_Cell_With_Successor	N´+2(R+W) best, N´+3(R+W) worst
New_Vector	(n+1)(R+W) (for n cells)
First_Cell	N
Last_Cell	N´+R
Successor_Cell	N (could be reduced to N´ if check omitted)
Predecessor_Cell	N´+R (could be reduced to N´ if check omitted)
Forget_Predecessor	N´+R+W (could eliminate W sometimes)
Forget_Successor	N´+W (could eliminate W sometimes)
Fetch_Cell_Contents	N
Store_Cell_Contents	N+W
Get_Cell_Tag	N
Set_Cell_Tag	N+W
Isolate_and_Replace	N+W+R+W+N´ (could eliminate one W sometimes)
Same_Object	N´+N´

R = cost of one read
W = cost of one write
N = cost of normalizing a cell object not known to be canonical. N = nR, n>=1
N´= cost of normalizing a register. If object is known to be canonical cost is 0, else cost is N

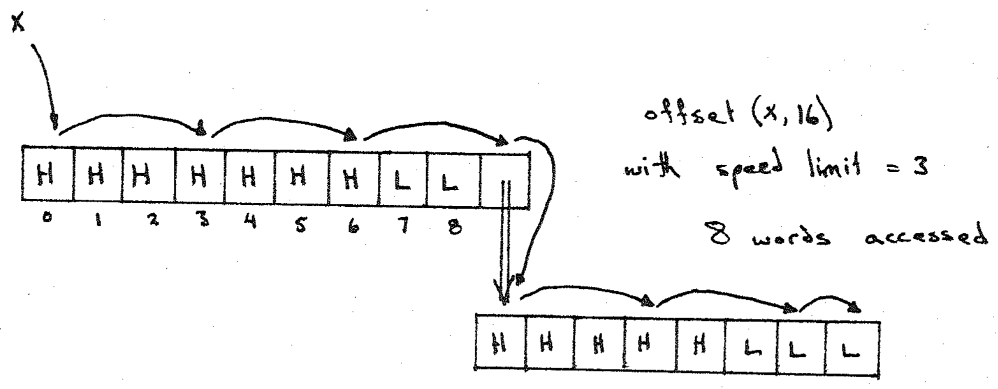
3.6 Offsetting in an NQL-RAM

Consider the following definition of the operation that locates the cell that is a given distance away from a base cell.

```
PROCEDURE Offset (R1, N, R2)
   BEGIN
      IF Value(R1) THEN Error END
      Copy_Object (R1, R2)
      IF N = 0 THEN
         Set_Object_Tag (R2, Default_Object_Tag)
      ELSE IF N > 0 THEN
         FOR I := 1 TO N DO
            Successor_Cell (R2, R2)
         END
      ELSE
         FOR I := 1 TO -N DO
            Precesessor_Cell (R2, R2)
         END
      END
   END
```

Done in this manner, the cost would be a function of the offset value. This property implies that NQL-RAMs will not be very good for applications that deal with large arrays. This is undesirable since there are few systems, including LISPs, that do no support some sort of array facility. How, then, can we get around this bias without sacrificing safety? Although we know of no way to make offsetting as efficient an operation as it is with conventional memories, we will show how faster implementations of Offset can be done at the expense of slowing down some of the memory warping operations and an extra bit in every word. This bit, when set, would indicate that the Kth successor (K fixed at 8, say) of the current cell does exist and can be found in the Kth word following the current word. Offset, when moving in the forward (i.e. successor) direction would use these bits to tell it whether it can jump ahead by as many as K words instead of cautiously advancing one cell at a time (<Figure 12: gear-shifting>). In other words, this is a "gear-shifting" technique. When the bit in the word is set, Offset may travel in high gear as fast as the speed limit (K). A word with the bit reset signals that low gear should be used because the road ahead either ends or snakes.

The implications for the rest of the implementation are really quite minor. When memory is initialized, the "high-speed" bit would be set in all but the last K words of memory. Forget_Successor (and the primitives that use it: New_Vector, Isolate_And_Replace, New_Cell_With_Successor) would have to update the bits whenever it actually has to break the bond between a cell and its successor. This will involve jumping back K-1 words from the current word and resetting the high-speed bit on all intervening words (it can stop as soon as it encounters a word whose bit is already reset). The other changes would be to the garbage collector --- when a structure is moved, the high-gear bits would have to be set appropriately.



<Figure 12: gear-shifting>

In summary, this technique will improve the performance of forward offsetting into both vector and sequence memoroids without sacrificing safety, while only marginally slowing down some of the primitives that we expect would be used relatively infrequently. Morover, the speed limit can be chosen to suit the application. We have not incorporated this proposal into the implementation given in Appendix I, but revised versions of Forget_Successor and Offset can be found in Appendix II.

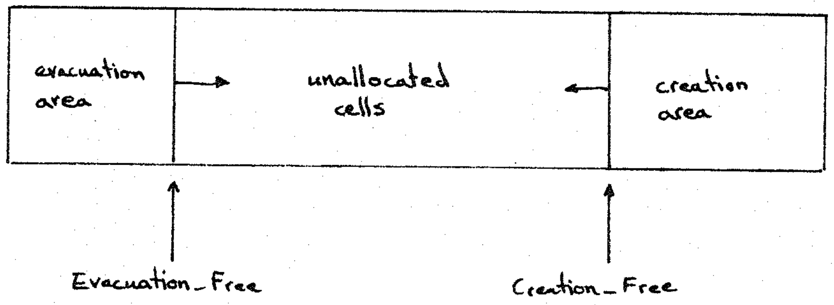
3.7 Garbage collection

Garbage collection is the process of identifying and recycling storage that is no longer being used. At any instant in time, the application level program has direct access to a limited number of cells in the NQL-RAM. By implementation assumptions discussed in an earlier section, all of these cells are identified by objects kept in the registers. Let this set of directly accessible cells be called ACC. Some primitive operations can change the registers, and therefore the elements of ACC, but only in a very controlled way. Since all access to the NQL-RAM is mediated by the primitive operations, the contents of the registers together with the rules enforced by the primitives make it possible to determine exactly what cells the application program could possibly access at some point in the future (ignoring cells that are yet to be created). The only primitives that will give the application direct access to a cell not in ACC are Successor_Cell, Predecessor_Cell, and Fetch_Cell_Contents. Therefore, the set of "accessible" cells, ACC*, can be defined to be the smallest set containing the set ACC and closed under the successor, predecessor, and referent relations.

At the cell level, any cell that is not in ACC* is "inaccessible". At the word level, we will say that any word w is "in use" if either
(i) w = M(c) for some cell c in ACC*, or
(ii) there are cells c1 and c2 in ACC* such that Successor(c1) = c2, M(c1) = w-1, and M(c2) <> w.
Any word that is not in use is said to be "garbage".

Sufficient information exists at the implementation level to determine which words are in use and which are garbage.

The second task of a garbage collecting scheme, the recyling of


evacuation area
unallocated cells
creation area
Evacuation_Free
Creation_Free

<Figure 13: free-storage>

garbage words, is usually done in one of two ways. The first way is to keep a list of free words and try to to put them back into circulation through the storage-consuming primitives. This kind of garbage collection never involves changing the cell-to-word mapping, M, for any accessible cells. The other general strategy involves altering M for accessible cells --- these are the "compacting" garbage collectors.

The garbage collector that can be found in Appendix III is of the two-space, stop-and-copy variety [ref Baker]. The underlying memory is partitioned into two equal-sized "spaces". At any time only one of the spaces is used (except during garbage collection); the other space is completely devoid of accessible cells. The free block, a vector memoroid, is always within the current space. When a storage allocation request cannot be satisfied because not enough available cells remain, the garbage collector is invoked. This causes the current space to be "condemned" --- all accessible objects must be evacuated by copying them into the formerly inactive space. The exodus begins by requesting that all of the cell objects accessible from the (user-visible) registers be moved across. Then the scavenger hunts through those objects in the new space looking for words that reference cells that are still in the condemned space and gets them moved across. When the scavenger is done, all cell objects in the new space, and all registers, will contain pointers into the new space, never the old one. The old space, now devoid of accessible cells, is then re-initialized. The application is allowed to proceed; all cell allocation requests being satisfied from the new space. During scavenging, words in the new space are allocated from low addresses to high. That is, the evacuation area lies below the word pointed to by the internal register Evacuate_Free and grows by eating into the front of the unallocated block of cells, whereas the creation area lies above the Creation_Free and grows by eating into the tail-end of the same block (see <Figure 13: free-storage>).

The heart of the garbage collector is Move. When given a cell in the old space, Move will copy that cell´s contents and cell tag over into the new space, and then Replace the old cell with its new version. But there are some complications. First, if a cell is strongly glued to either its successor or predecessor, these cells must be moved at the same time so as to preserve their contiguity. A second complication is introduced by the desire to minimize the number of snakes created in the new space. Move therefore tries to move weakly connected sequences of cells at the same time. This process will terminate when a Last cell has been moved, or when an immovable cell is encountered. A cell can be considered immovable for one of two reasons. One reason is that the cell has already been moved. The other reason is that the cell has a predecessor that has not been moved. When an immovable cell is encountered, a snake must be created in the new space to point at the immovable cell. Just so long as the process of Moving does not alter the topology of the NQL-RAM for any accessible cells, and Move ensures that the cell passed to it as a parameter is evacuated from the old space, the garbage collector will eventually terminate. Interestingly enough, Move does not violate accessibility and can be written using the normal primitives.

The garbage collector does have the following properties:

- Completely transparent --- the cell graph topology is not altered.

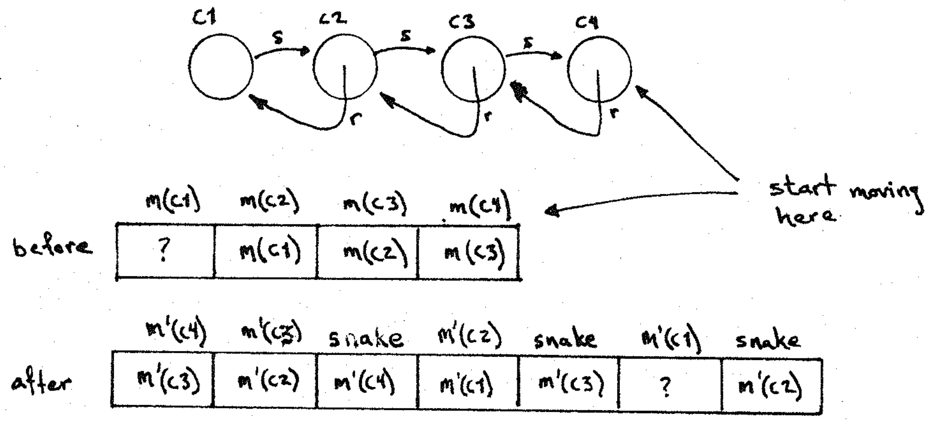
- Chains of snakes are collapsed --- after garbage collection, all registers and all in-use words in the underlying memory will contain canonical objects.

It is much more difficult to make claims about the effectiveness of the linearization of sequences of weakly glued cells. What is certain is that there is no guarantee that the accessible cells copied into the new space will occupy fewer words than before --- but at least we can assert that the absolutely worst case would less than double the number of words required (see <Figure 14: back-links>). However, we do make the following observations:

- CDR-coded lists done with Replace (see preceding section) will be linearized as well as with [ref Baker]. However, the garbage collector is completely general, and will endeavour to linearize any sequence memoroid.

- If a cell is only accessible as the successor of a single cell, then these cell will be stored contiguously after garbage colection. In other words, the system will tend not to break up sequence memoroids for no reason.

- The linearization strategy will tend to be more effective when pointers go backwards in time (i.e., new objects tend to point to older ones).



<Figure 14: back-links>

The two major shortcomings of this variety of garbage collector can removed without undue difficulty. The "embarassing pause" noticed when the garbage collector is running can be alleviated by employing an incremental version of our algorithm (see [ref Baker] for details). The pointless moving of semi-permanent cell structures can be countered by adopting a technique such as that of [ref Lieberman&Hewitt 1981] (the authors have implemented a stop-and-copy variation of an earlier version of their scheme [ref Liebweman&Hewitt 1980] and encountered no significant problems.)

4. Future Research and Conclusions

At present, the only use NQL-RAMs have seen is in the (software) implementation of a reflective LISP processor [ref Smith]. The complete separation of memory management decisions from other language implementation issue has greatly simplified this task. The next step, to be taken within the year, is to implement an NQL-RAM in microcode on a machine that has a large virtual address space.

A much more radical step is also in the planning stages: a general purpose single-user machine that will be totally NQL-RAM-based. The instruction set of the main processor will be designed to take advantage of the flexible structure of the NQL-RAM. For example, with instruction streams represented with sequence memoroids, code can move in the underlying memory without problems. (Some amusing side effects exist: code patched with Replace could be linearized by the garbage collector; unconditional branch instructions to a known instruction could be done with snakes; unreachable code may be garbage collected unless glued down.) The entire NQL-RAM memory system will be implemented on a completely separate processor.

One other intriguing topic of future research is the design of a cache to go with an NQL-RAM. Because of the immediate availability of the glue information in each word, the cache might be better able to predict what not to bother caching. Perhaps it would even be able to use the same information to try and guess which words will be required next.

It would also be interesting to see what kinds of non-list processing applications could make effective use of an NQL-RAM's non-standard topology. For example, could PROLOG [ref Warren], Smalltalk [ref BYTE], and APL [ref Iverson] implementations use an NQL-RAM?

In conclusion, we have presented a new form of memory system and shown how it can be implemented efficiently. These memories are general purpose and can support many of the basic needs common to list processing systems, such as garbage collection, in an application independent way. We boldly suggest that NQL-RAMs might be well suited to serve as the basic memory structures for 5th generation computers.

Acknowledgments

The authors would like to thank all of the members of the Ottawa chapter of the Knights of the Lambda Calculus, and especially Nick Briggs, Greg Nuyens, Mike Wilson, and Dave Thomas. They enthusiastically contributed numerous ideas, comments, and suggestions, many of which have been incorporated in this paper. Thanks also to Brian Smith of Xerox PARC for discovering such an elegant concept as procedural reflection and incorporating it in a "semantically rationalized" version of LISP --- our quest for a equally elegant and efficient implementation of 3-LISP led us to distill existing LISP implementation techniques into the form described herein. Sydney J. Hurtubise (whereabouts unknown) also gets an honourable mention for contributing the term "memoroids", which evokes both an entirely appropriate image and a wholly inappropriate one --- simultaneously.

<u>Appendix I</u>

```
TYPES
   Word_Type = RECORD
                  Control: Snake_And_Glue_Type
                  Tag: Cell_Tag_Type
                  Contents: Object
               END

   Object    = RECORD
                  Value: Boolean
                  Tag: Object_Tag_Type
                  Data: Address_Type
               END

   Cell_Tag_Type       = 0 .. (2**Cell_Tag_Width) - 1
   Object_Tag_Type     = 0 .. (2**Object_Tag_Width) - 1
   Snake_And_Glue_Type = (Snake, None, Weak, Strong)
   Address_Type        = 0 .. (2**Address_Width) - 1
   Register_Type       = 0 .. 25

VARIABLES
   Memory:    ARRAY [Address_Type] OF Word_Type
   Register:  ARRAY [Register_Type] OF Object
   Canonical: ARRAY [Register_Type] OF Boolean

CONSTANTS
   First_Register     = 0
   Last_User_Register = 15
   Creation_Free      = 16
   Evacuation_Free    = 17
   Scan               = 18
   Next_Cell          = 19
   This_Cell          = 20
   Scratch1           = 21
   Scratch2           = 22
   Scratch3           = 23
   Scratch4           = 24
   Scratch5           = 25
   Last_Register      = 25

PROCEDURE Normalize (R: Register_Type)
   IF NOT Register[R].Value AND NOT Canonical[R] THEN
      WHILE Memory[Register[R].Data].Control = Snake DO
         {Must take care to preserve object tags.}
         Register[R].Data := Memory[Register[R].Data].Contents.Data
      END
      Canonical[R] := True
   END
```

```
PROCEDURE Set_Object_Tag (R: Register_Type; T: Object_Tag_Type)
   Register[R].Tag := T

PROCEDURE Get_Object_Tag (R: Register_Type): Object_Tag_Type
   RETURN Register[R].Tag

FUNCTION Value (R: Register_Type): Boolean
   RETURN Register[R].Value

FUNCTION Get_Value (R: Register_Type): Address_Type
   BEGIN
      IF NOT Register[R].Value THEN Error END
      RETURN Register[R].Data
   END

PROCEDURE Make_Value_Object (T: Object_Tag_Type; V: Address_Type;
                             R: Register_Type)
   BEGIN
      Register[R].Value := True
      Register[R].Tag := T
      Register[R].Data := V
   END

PROCEDURE Same_Object (R1, R2: Register_Type): Boolean
   BEGIN
      IF Register[R1].Value <> Register[R2].Value THEN
         RETURN False
      END
      IF NOT Register[R1].Value THEN
         Normalize (R1)
         Normalize (R2)
      END
      RETURN Register[R1].Data = Register[R2].Data
   END

PROCEDURE Same_Tagged_Object (R1, R2: Register_Type): Boolean
   BEGIN
      IF Register[R1].Value = Register[R2].Value
      AND Register[R1].Tag = Register[R2].Tag THEN
         RETURN Same_Object(R1,R2)
      ELSE
         RETURN False
      END
   END

PROCEDURE Copy_Object (R1, R2: Register_Type)
   {Copy register R1 into R2.}
   BEGIN
      Register[R2] := Register[R1]
      Canonical[R2] := Canonical[R1]
   END
```

```
FUNCTION First_Cell (R:Register_Type): Boolean;
   BEGIN
      IF Register[R].Value THEN Error END
      Normalize (R)
      RETURN Memory[Register[R].Data-1].Control <> Strong
   END

FUNCTION Last_Cell (R: Register_Type): Boolean;
   BEGIN
      IF Register[R].Value THEN Error END
      Normalize (R)
      RETURN Memory[Register[R].Data].Control = None
   END

PROCEDURE Successor_Cell (R1, R2: Register_Type)
   {R2 is the destination.}
   BEGIN
      IF Register[R1].Value THEN Error END
      Normalize (R1)
      IF Memory[Register[R1].Data].Control = None THEN Error END
      Register[R2].Value := False
      Register[R2].Tag := Default_Object_Tag
      Register[R2].Data := Register[R1].Data+1
      Canonical[R2] := False
   END

PROCEDURE Predecessor_Cell (R1, R2: Register_Type)
   BEGIN
      IF Register[R1].Value THEN Error END
      Normalize (R1)
      IF Memory[Register[R1].Data-1].Control <> Strong THEN Error END
      Register[R2].Value := False
      Register[R2].Tag := Default_Object_Tag
      Register[R2].Data := Register[R1].Data-1
      Canonical[R2] := True
   END

PROCEDURE Forget_Predecessor_Cell (R: Register_Type)
   BEGIN
      IF Register[R].Value THEN Error END
      Normalize (R)
      IF Memory[Register[R].Data-1].Control = Strong THEN
         Memory[Register[R].Data-1].Control := Weak
      END
   END

PROCEDURE Forget_Successor_Cell (R: Register_Type)
   BEGIN
      IF Register[R].Value THEN Error END
      Normalize (R)
      Memory[Register[R].Data].Control := None
   END
```

```
PROCEDURE Set_Cell_Contents (R1, R2: Register_Type)
   {The cell located by R1 has it contents changed to what´s in R2.}
   BEGIN
      IF Register[R1].Value THEN Error END
      Normalize (R1)
      Memory[Register[R1].Data].Contents := Register[R2]
   END

PROCEDURE Fetch_Cell_Contents (R1, R2: Register_Type)
   {The contents of the cell located by R1 is put in R2.}
   BEGIN
      IF Register[R1].Value THEN Error END
      Normalize (R1)
      Register[R2] := Memory[Register[R1].Data].Contents
      Canonical[R2] := False
   END

PROCEDURE Set_Cell_Tag (R: Register_Type; T: Cell_Tag_Type)
   {The cell tag of the cell located by R will be changed to T.}
   BEGIN
      IF Register[R].Value THEN Error END
      Normalize (R)
      Memory[Register[R].Data].Tag := T
   END

FUNCTION Get_Cell_Tag (R: Register_Type): Cell_Tag_Type
   {The cell tag of the cell located by R will be returned.}
   BEGIN
      IF Register[R].Value THEN Error END
      Normalize (R)
      RETURN Memory[Register[R].Data].Tag
   END

PROCEDURE Isolate_And_Replace_Cell (R1, R2: Register_Type)
   {The cell located by R1 will by the cell located by R2.
    All cells that had the first cell as a successor beforehand
    will be connected to the second cell afterwards.}
   BEGIN
      IF Register[R1].Value OR Register[R2].Value THEN Error END
      Normalize (R1)
      Normalize (R2)
      Forget_Predecessor_Cell (R1)
      Forget_Predecessor_Cell (R1)
      IF Register[R1].Data <> Register[R2].Data THEN
         Memory[Register[R1].Data].Contents := Register[R2]
         Memory[Register[R1].Data].Control := Snake
         Register[R1].Data := Register[R2].Data
         FOR R := First_Register TO Last_Register DO
            Canonical[R] := False
         END
         Canonical[R1] := True
         Canonical[R2] := True
      END
   END
```

```
PROCEDURE Replace_Cell (R1, R2: Register_Type)
BEGIN
      IF NOT First_Cell(R1) OR NOT Last_Cell(R1) THEN Error END
      Isolate_And_Replace (R1, R2)
   END

PROCEDURE New_Vector (N: Integer; R: Register_Type)
   VAR I: Integer
   BEGIN
      IF N <= 0 THEN Error END
      IF Space_Remaining() < N THEN
         Garbage_Collect ()
         IF Space_Remaining() < N THEN Error "Out of memory." END
      END
      Forget_Successor_Cell (Creation_Free)
      FOR I := 1 TO N DO
         Set_Cell_Tag (Creation_Free, Default_Cell_Tag)
         Set_Cell_Contents (Creation_Free, Default_Contents)  {a value}
         Predecessor_Cell (Creation_Free, Creation_Free)
      END
      Successor_Cell (Creation_Free, R)
      Forget_Predecessor_Cell (R)
      Set_Object_Tag (R, Default_Object_Tag)
   END

PROCEDURE New_Cell_With_Successor (R1, R2: Register_Type)
   BEGIN
      IF Space_Remaining() < 2 THEN
         Garbage_Collect ()
         IF Space_Remaining() < 2 THEN Error "Out of memory." END
      END
      IF NOT Last_Cell(Creation_Free) THEN
         Successor_Cell (Creation_Free, Scratch1)
         IF Same_Object(Scratch1, R1) THEN
            Copy_Object (Creation_Free, R2)
            Predecessor_Cell (Creation_Free, Creation_Free)
            Forget_Predecessor_Cell (R2)
            Set_Cell_Tag (R2, Default_Cell_Tag)
            Set_Cell_Contents (R2, Default_Contents)
            Set_Object_Tag (R2, Default_Object_Tag)
            RETURN
         END
      END
      New_Vector (2, R2)
      Successor_Cell (R2, Scratch1)
      Isolate_And_Replace_Cell (Scratch1, R1)
   END
```

```
PROCEDURE Offset (R1: Register_Type; N: Integer; R2: Register_Type)
   BEGIN
      IF Register[R1].Value THEN Error END
      Copy_Object (R1, R2)
      IF N = 0 THEN
         Set_Object_Tag (R2, Default_Object_Tag)
         RETURN
      END
      IF N > 0 THEN
         FOR I := 1 TO N DO
            Successor_Cell (R2, R2)
         END
         RETURN
      END
      FOR I := 1 TO -N DO
         Predecessor_Cell (R2, R2)
      END
   END
```

Appendix II

```
PROCEDURE Offset (R1: Register_Type; N: Integer; R2: Register_Type)
   {Gear-shifting version.}
   BEGIN
      IF Value(R1) THEN Error END
      Copy_Object (R1, R2)
      IF N = 0 THEN
         Set_Object_Tag (R2, Default_Object_Tag)
         RETURN
      END
      IF N < 0 THEN
         FOR I := 1 TO -N DO
            Predecessor_Cell (R2, R2)
         END
         RETURN
      END
      WHILE N > 0 DO
         Normalize (R2)
         IF Memory[Register[R2].Data].High_Speed THEN
            IF N >= Speed_Limit THEN
               Register[R2].Data := Register[R2].Data + Speed_Limit
               Canonical[R2] := False
               N := N - Speed_Limit
            ELSE
               Register[R2].Data := Register[R2].Data + N
               Canonical[R2] := True
               N := 0
            END
         ELSE
            Successor_Cell (R2, R2)
            N := N - 1
         END {IF}
      END {WHILE}
   END
```

```
PROCEDURE Forget_Successor_Cell (R: Register_Type)
   {Gear-shifting version.}
   BEGIN
      IF Register[R].Value THEN Error END
      Normalize (R)
      IF Memory[Register[R].Data].Control <> None THEN
         Memory[Register[R].Data].Control := None
         Memory[Register[R].Data].High_Speed := False
         Copy_Object (R, Scratch4)
         Register[Scratch4].Data := Register[Scratch4].Data - (Speed_Limit-1)
         WHILE Memory[Register[Scratch4].Data].High_Speed DO
            Memory[Register[Scratch4].Data].High_Speed := False
            Register[Scratch4].Data := Register[Scratch4].Data + 1
         END
      END
   END
```

<u>Appendix III</u>

```
VARIABLES
   Current_space: 0..1

INTERNAL FUNCTION Make_Object (V: Boolean;
                              T: Object_Tag_Type;
                              A: Address_Type): Object
   VAR Result: Object
   BEGIN
      Result.Value := V
      Result.Tag := T
      Result.Data := A
      RETURN Result
   END

INTERNAL PROCEDURE Garbage_Collect ()
   VAR R: Register_Type
   BEGIN
      Current_Space := 1 - Current_Space      {flip}
      Initialize_Space ()
      FOR R := First_Register TO Last_User_Register DO
         Move (R)
      END
      Scavenge_All ()
   END
```

```
INTERNAL PROCEDURE Initialize_Space ()
   BEGIN
      IF Current_Space = 0 THEN
         Register[First_Word_In_Space] :=
            Make_Object(False,Default_Object_Tag,0)
         Register[Last_Word_In_Space] :=
            Make_Object(False,Default_Object_Tag,Memory_Size/2-1)
      ELSE
         Register[First_Word_In_Space] :=
            Make_Object(False,Default_Object_Tag,Memory_Size/2)
         Register[Last_Word_In_Space] :=
            Make_Object(False,Default_Object_Tag,Memory_Size-1)
      END
      FOR I := Register[First_Word_In_Space].Data TO
               Register[Last_Word_In_Space].Data DO
         Memory[I].Control := Strong
         Memory[I].Contents.Value := True
      END
      Memory[Register[Last_Word_In_Space].Data].Control := None
      Canonical[First_Word_In_Space] := True
      Canonical[Last_Word_In_Space] := True
      Copy_Object (First_Word_In_Space, Evacuation_Free)
      Copy_Object (Last_Word_In_Space, Creation_Free)
   END
```

The scavenger runs through all of memory below Evacuation_Free moving any cells in the old space into new space. The scavenger violates the normal rules of accessibility and must be careful to avoid snakes in the new space. The contents field of each word, whether cell or snake, must be made to contain a normalized pointer into the new space. Since the process of "cleansing" a word may cause further cells be be moved to the new spaces, the scavenger´s job is not complete until all words below Evacuation_Free have been visited.

```
INTERNAL PROCEDURE Scavenge_All ()
   BEGIN
      Register[Scan] := Register[First_Word_In_Space]
      WHILE Register[Scan].Data < Register[Evacuation_Free].Data DO
         Register[Scratch1] := Memory[Register[Scan].Data].Contents
         Move (Scratch1)
         Memory[Register[Scan].Data].Contents := Register[Scratch1]
         Register[Scan].Data := Register[Scan].Data + 1
      END
   END

INTERNAL PROCEDURE Move (R: Register_Type)
   LOCAL VARIABLES Done, This_Cell_Has_No_Predecesor,
      Next_Cell_Has_No_Predecessor, This_Cell_Is_Immovable,
      Next_Cell_Is_Immovable: Boolean
   BEGIN
      IF Register[R].Value THEN RETURN END
      IF NOT In_Old_Space(R) THEN RETURN END

      {Find the first cell of the strongly connected block
```

```
 containing R.}
Copy_Object (R, Next_Cell)
WHILE NOT First_Cell(Next_Cell) DO
   Predecessor (Next_Cell, Next_Cell)
END

{Start the process of moving successive cells to the new space.}
Done := False
Next_Cell_Has_No_Predecessor := True
Next_Cell_Is_Immovable := False
REPEAT
   Copy_Object (Next_Cell, This_Cell)
   This_Cell_Has_No_Predecessor := Next_Cell_Has_No_Predecessor
   This_Cell_Is_Immovable := Next_Cell_Is_Immovable

   {Acquire a new cell from the Evacuation_Free block.}
   Copy_Object (Evacuation_Free, New_Cell)
   IF Last(Evacuation_Free) THEN Error "Out of space." END
   Successor_Cell (Evacuation_Free, Evacuation_Free)

   IF This_Cell_Is_Immovable THEN   {Create a snake.}
      Isolate_And_Replace_Cell (New_Cell, This_Cell)
      Done := True
   ELSE
      {Transcribe contents and cell tag from old to new.}
      Fetch_Cell_Contents (This_Cell, Scratch2)
      Set_Cell_Contents (New_Cell, Scratch2)
      Set_Cell_Tag (New_Cell, Get_Cell_Tag(This_Cell))

      {Weaken glue to predecessor if necessary.}
      IF This_Cell_Has_No_Predecessor THEN
         Forget_Predecessor_Cell (New_Cell)
      END

      IF Last_Cell(This_Cell) THEN    {End of the road.}
         Done := True
         Forget_Successor_Cell (New_Cell)
      ELSE
         Successor_Cell (This_Cell, Next_Cell)
         Next_Cell_Has_No_Predecessor := First_Cell(Next_Cell)
         Next_Cell_Is_Immovable := False
         IF NOT In_Old_Space(Next_Cell) THEN
            Next_Cell_Is_Immovable := True
         END
         IF NOT Next_Cell_Has_No_Predecessor THEN
            Predecessor_Cell (Next_Cell, Scratch3)
            IF NOT Same_Object(This_Cell,Scratch3) THEN
               Next_Cell_Is_Immovable := True
            END
         END
      END

      {Make the switch.}
      Isolate_And_Replace_Cell (This_Cell, New_Cell)
```

```
            END
         UNTIL Done
         Normalize (R)
      END

   INTERNAL PROCEDURE Initialize_NQL_RAM ()
      VAR R: Register_Type
      BEGIN
         Default_Contents := Make_Object(True,Default_Object_Tag,X'DEAD')
         FOR R := First_Register TO Last_Register DO
            Register[R] := Default_Contents
            Canonical[R] := False
         END
         Current_Space := 0
         Initialize_Space ()
      END

   INTERNAL FUNCTION In_Old_Space (R: Register_Type): Boolean
      BEGIN
         Normalize (R)
         RETURN (Register[R].Data < Register[First_Word_In_Space].Data) OR
              (Register[R].Data > Register[Last_Word_In_Space].Data)
      END
```

References

1. Allen,J. Anatomy of LISP. McGraw-Hill, New York, 1978.
2. Bobrow,D.G. and Clark,D.W. "Compact encodings of list structure" ACM TOPLAS 1, 2 (Oct. 1979), 266-286.
3. Knuth,D.E. The Art of Computer Programming, Vol. 1: Fundamental Algorithms (2nd edn.). Addison-Wesley, Reading, Mass., 1973.
4. Baker,H.G.Jr. "List processing in real time on a serial computer" Comm. ACM 21, 4 (Apr. 1978), 280-294.
5. Lieberman,H. and Hewitt,C. A Real Time Garbage Collector Based on the Lifetimes of Objects, MIT A.I. Memo 569A, Cambridge, Mass., Oct. 1981.
6. Steele,G.L.Jr. Fast Arithmetic in MACLISP, MIT AI Memo 421, Cambridge, Mass., Sep. 1977.
7. Sansonnet,J.P., Castan,M., et Percebois,C. "M3L: A list-directed architecture", Proc. 7th Annual Symposium on Computer Architecture, May 1980 (appears in SIGARCH Newsletter 8, 3, 105-112).
8. White,J.L. "Address/memory management for a gigantic LISP environment" Proc. 1980 LISP Conference, Stanford, Calif., Aug. 1980, 119-127.
9. Deutsch,L.P "ByteList and its Alto implementation" Proc. 1980 LISP Conference, Stanford, Calif., Aug. 1980, 231-242.
10. Burton,R.R., Masinter,L.M., et al. "Overview and Status of DoradoLisp" Proc. 1980 LISP Conference, Stanford, Calif., Aug. 1980, 243-247.
11. Smith,B.C. Reflection and Semantics in a Procedural Language. MIT LCS Tech. Report 272, Cambridge, Mass., Jan. 1982.
12. Bishop,P.B. Computer Systems with a Very Large Address Space and Garbage Collection. MIT LCS Tech. Report 178, Cambridge, Mass., May 1977.
13. Steele,G.L.Jr. Data Representation in MACLISP, MIT AI Memo 420, Cambridge, Mass., 1977.
14. Prini,G. and Rudalics,M. "The Lambdino storage management system", BYTE 4, 8 (Aug. 1979), 26-32.
15. Hartley,A.K. INTERLISP-11. Bolt Beranek and Newman, Cambridge, Mass., Mar. 1979.
16. Moore,J.S. The INTERLISP Virtual Machine Specification. Xerox PARC CSL Report 76-5, Palo Alto, Calif., Sep. 1976.
17. Teitelman,W. INTERLISP Reference Manual. Xerox PARC, Palo Alto, Calif., Oct. 1978.
18. Weinreb,D. and Moon,D. Lisp Machine Manual. MIT AI Lab, Cambridge, Mass., January 1979.
19. Stallman,R.M. Phantom Stacks. MIT AI Memo 556, Cambridge, Mass., July 1980.
20. Cohen,J. "Garbage collection of linked data structures" ACM Computing Surveys 13, 3 (Sep. 1981), 341-367.
21. Sussman,G.J., Holloway,J. et al. "Scheme-79 - Lisp on a Chip" Computer, July 1981, 10-21.
22. Lieberman,H. and Hewitt,C. A Real Time Garbage Collector That Can Recover Temporary Storage Quickly. MIT AI Memo 569, Cambridge,

Mass., Apr. 1980.
23. Earley,J. "Towards an Understanding of Data Structures" Comm. ACM 14, 10 (Oct. 1971), 617-627. 24. Warren,D., Pereira,L.M., and Pereira,F. PROLOG -- The language and its implementation compared with LISP. Proc. Symp. on A.I. and Prog. Lan.; SIGPLAN Notices, Aug. 1977, 109-115.
25. BYTE Magazine, Sept. 1981.
26. Iverson,K. A Programming Language. Wiley, New York 1962.