Computing Cyclic List Structures

F. Lockwood Morris
Syracuse University, lockwood@ecs.syr.edu

Jennifer Schwarz
Syracuse University

2-80

Computing Cyclic List Structures

June 1980

F. Lockwood Morris
J.S. Schwartz

SCHOOL OF COMPUTER
AND INFORMATION SCIENCE
SYRACUSE UNIVERSITY

Computing Cyclic List Structures

F. Lockwood Morris*

Syracuse University

Jerald S. Schwarz

Bell Laboratories

Abstract

It is argued that list structures containing cycles are useful and unobjectionable Lisp entities. If this is so, it is desirable to have a means of computing them less foreign to the equational-definition style characteristic of Lisp than are the list-structure-altering primitives *rplaca* and *rplacd*. A notion is developed of a reasonable system of mutually recursive equations, guaranteed to have a unique solution in list structures. The notion is given in terms of the computations invoked by the equations, without reference to the forms of expressions appearing in them. A variety of programming examples are presented, including a curious implementation of the Knuth-Morris-Pratt string matching algorithm. Two methods of implementing the recursive definition facility are discussed.

1. Introduction

Lisp has two outstanding virtues which make for ease of use and clarity. The first is that it invites programming by recursive definitions, which is to say a specialized form of declarative, or "logic" programming. One writes down a selected body of true equations about a function of interest, and these provide a definition by which without further ado it may be computed. To see that a function is correctly programmed one need only assent to the truths, and assure oneself that computation will terminate - perhaps meeting some criterion of efficiency - for all cases which will arise; the latter task, impossible in general, is ordinarily neither difficult nor highly subject to error.

The second virtue underpins the first: arbitrarily complex structures may be considered as entities, and may serve as the values of identifiers, as the arguments and results of functions. One thinks in whole list structures, not in pointers and *cons* cells. What justifies this high-level thinking is that entities ordinarily do not shift under one's feet; indeed if one abstains from assignment to *car* and *cdr* fields - from *rplaca* and *rplacd* - they never do so. The fortunate piece of design which has given rise to this virtue seems to have been the provision of *cons* as official primitive, and the stigmatization of assignment to fields within list structures as a practice to be engaged in only with extreme caution.

We do not regard Lisp's orientation towards trees (i.e., in favor of abstracting away from the occurrence of shared substructures), which is provided by the predicate *equal* and by the visible form of S-expressions, as of great significance. We are happy to think of list structures as they really are (with sharing), and to exploit *eq* as it really is in all Lisps known to us - the identity predicate, defined between any two objects. We have commonly found it expedient to make use of list structures in which so many *car*-*cdr* paths led to the same substructures that their S-expression representations would have been of astronomical size.

* This authors' research was supported by NSF Grant MCS75-22002.

Moreover, we have from time to time had good reason to use list structures which contained *car-cdr* cycles, and which would make infinite S-expressions. In Lisp as it is, such structures - which we regard as innocuous - cannot be created without recourse to *rplaca* and *rplacd*. This brings us to our topic: to advocate the provision, in modern versions and offshoots of Lisp, of a facility for the creation of list structures which may contain cycles as the solutions to recursive equations.

2. The proposed facility

The possibility of list structures which contain themselves as subparts, and which therefore satisfy recursive structural equations, has been from time to time alluded to in the literature, for example by Landin [1] and Burge [2], and recently Henderson [3]. So far as we are aware, however, there have been no systematic experiments in constructing such objects and exploiting them in programming. It seems most useful, therefore, rather than attempting to specify in detail a language feature for which there is as yet no widely felt need, instead to characterize informally a class of intuitively reasonable computations of list structures by recursive definition, introduce a working notation for such definitions, and then display (Sections 5 and 6) a collection of programming examples. In Section 7 we shall give a tentative discussion of what appear to be feasible implementation techniques.

(With some trepidation, we pass over as not germane to the present development such notions as Landin's "streams" and the "lazy evaluation" of Henderson and Morris [4], and Friedman and Wise [5]. These, we take it, are methods of simulating computation with actually infinite lists and trees - e.g., the list of all the prime numbers - by computing breadth first, and only elaborating the infinite quantities' defining expressions so far as is necessary. In the present context, we prefer to deal with list structures which exist in a more everyday sense, and to be able to think of a computation with list structures independently of any programming language, as the combination of *car*'s, *cdr*'s, *cons*'s, *atom*'s and *eq*'s which the structures undergo. This view tends to deny to programming languages a central place in computer science, and to regard them as more or less convenient notations for prescribing computations; it is a view which has been highly fruitful in the design and analysis of algorithms.)

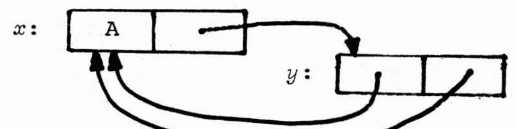
To proceed with the characterization of "reasonable computation": A computation of a list structure from some others may be such that the latter are subjected to no primitive operations other than *cons*, and thus can play no useful role other than as parts of the result. Such a computation one may call <u>purely</u> <u>constructive</u> (with respect to the structures regarded as being "computed from"). A system of equations

$$\begin{aligned} x_1 &= e_1 \\ &\vdots \\ x_n &= e_n \end{aligned}$$

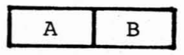
in which the e's specify computations purely constructive with respect to the values of such of the x's as appear in them is readily seen to have a unique solution, provided only that it contains no completely circular subsystems of variables equated to each other, on the pattern of $x_1{=}x_3$, $x_3{=}x_1$. For example, the system

$$\begin{aligned} x &= cons[\mathrm{A},y] \\ y &= cons[x,x] \end{aligned}$$

has the solution which may be pictured



"Unique solution" here must be understood in the Heraclitian sense in which *cons*[A,B] may be said to yield



uniquely; each time we solve our equations we should get a structurally identical but distinct solution.

A clear grasp of the existence of solutions should make programs which employ such systems of equations as definitions of their left-hand-side variables at least somewhat understandable. It may be as well, however, to supply at this point a deliberately vague account of how one might expect solutions to be computed: namely, the right-hand-side expressions may be evaluated in any order and in perfectly everyday fashion, some sort of dummy values being made available for those x's the production of whose actual values still lies in the future; as each actual value is produced, it should somehow be made to coalesce with or replace all occurrences of its correponding dummy. It is important to realize that the right-hand-side computations may be arbitrary, provided they have the necessary purely constructive character, and that therefore the expressions need not be restricted to any particlar syntactic form.

The observation just made, that the right hand sides can be worked out in any order, one being completely evaluated if we like before the next is begun, shows that the reasonableness condition on a set of equations may be liberalized as follows: if the equations can be so ordered (before being numbered 1 through n) that each expression e_i specifies a computation purely constructive with respect to the values of x_i, $x_{i+1}, \ldots, x_n$, having regard to occurrences of these values not only under their own names but also as parts of the values of $x_1, \ldots, x_{i-1}$, and if completely circular definitions are avoided, then a unique solution exists. Thus, for example,

$$x = cons[y, \mathrm{A}]$$
$$y = cdr[x]$$

is reasonable (and rather dull), but

$$x = cons[y, \mathrm{A}]$$
$$y = car[x]$$

is circular, since the second equation in effect gives y as the definiens of y.

It is natural to require that equations be written in an order which manifests their reasonableness. To destroy the symmetry of the idea "set of simultaneous equations" is regrettable, but the examples will show that considerable power is gained by use of the liberalized condition.

3. Notation

The notation to be used here is in essence the Lisp M-language. Application is shown with square brackets and commas: $f[x,y]$, except that *car*, *cdr*, *null*, and *eq* are made into operators: $\underline{a}$, $\underline{d}$, $\underline{n}$, $\equiv$, and that applications of *cons* and *list* are shown with the corresponding S-expression punctuation: $(x.f[y])$ for $cons[x,f[y]]$ and $(x\ y\ z)$ for $list[x,y,z]$.

The principal extension to conventional Lisp is to suppose a form of mutually recursive declaration of identifiers:

$$\underline{\text{lectrec}}\ x_1 = e_1$$
$$\underline{\text{and}}\ \vdots$$
$$\underline{\text{and}}\ x_n = e_n\ \underline{\text{in}}\ \ \text{body}\quad .$$

The intention is to express a system of equations as described above, and to yield the value of "body" in an environment in which $x_1, \ldots, x_n$ have been bound to the solutions. A similar construction, introduced by plain let, makes a more readable equivalent to application of a lambda abstraction to arguments.

An inessential but convenient extension is to permit expressions with a tuple of values and functions with a tuple of results, and correspondingly to permit declarative equations to have multiple left hand sides. E.g. (division with remainder):

```
quotrem[x,y] = if x<y then 0,y
               else let q,r = quotrem[x-y,y]
                        in q+1, r    .
```

Such transitory tuples could of course be assembled and decomposed by *cons*, *car*, and *cdr*, but to do so would be to obscure the more momentous *cons*'s.

4. Atlases

A further digression needs to be taken before considering examples of programming with letrec. Any function that makes a tour of an arbitrary list structure which may contain cycles needs to keep some record of where it has been, so that the structure may not after all seem infinite to it. In operational terms, one thinks of marking the cells of the structure, or entering them as keys with some associated information in some dictionary, which in keeping with the topographical metaphor may better be called an "atlas."

We shall here hypothesize a purely functional atlas facility, consisting of the constructors *empty* and *extend*, the predicate *defined*, and the selector *value*, interrelated by the identities:

```
defined[x, empty[]] = false
defined[x, extend[x' , y,a]]  =
     if x ≡ x' then true else defined[x,a]
value[x, extend[x',y,a]]  =
     if x ≡ x' then y else value [x,a]   .
```

See the appendix for some remarks on the realization of atlases.

5. Small examples

A first programming example is provided by the problem of making an isomorphic copy of an arbitrary list structure, to be built out of the same atoms but with all new dotted pairs. Here the necessary atlas tabulates a function mapping pairs in the old structure to image pairs in the new one, and letrec allows us to conjure up a value of this function out of thin air, before we know the subobjects of which it will ultimately be the *cons*. Thus:

```
copy[x] = let x',h' = cop[x,empty[]] in x'
cop[x,h] = if atom[x] then x,h
   else if defined[x,h] then value[x,h],h
   else letrec y',h' =
              cop[a x, extend[x,x',h]]
        and z',h" = cop[d x,h']
        and x' = (y'.z')
        in x',h"    .
```

It is perhaps of interest to redefine the same function in a more imperative style, to suggest how an atlas facility with a destructive extension operation could be accommodated. We ask the reader's indulgence for the eclectic notation:

```
copy[x] = prog h
   letrec cop[x] =
      if atom[x] then x
      else if defined[x,h] then value[x,h]
      else letrec x' = [h:=extend[x,x',h];
                         (cop[a x] . cop[d x])]
            in x'
  in [h:=empty[]; cop[x]]    .
```

An isomorphism predicate is readily programmed on the model of *copy*. The natural notion of *equal*, however, according to which two structures are unequal only if some path of *car*'s and *cdr*'s leads to an atom in one and to something not that atom in the other, seems to require the techniques of fast unification - see [6] for an exposition - for its efficient realization.

As a second programming exercise let us take the problem of translating an acyclic representation of an arbitrary labeled directed graph to a representation in which arcs are represented directly as connections within the structure. As acyclic representation, it is natural to take a graph on m nodes, labeled with the atoms $L_1,\ldots,L_m$, to be the list of its nodes, and to take for its i'th node the list $(L_i\ L_{i,1}\ \ldots\ L_{i,d(i)})$, where $L_{i,1},\ldots,L_{i,d(i)}$ are the labels of the nodes directly accessible from the node labeled L_i. The goal is to build a corresponding structure in which the nodes themselves have replaced uses of their labels; that is, to represent the graph by a list $(n_1\ldots n_m)$, where n_i is the list $(L_i\ n_{i,1}\ \ldots\ n_{i,d(i)})$. The following function definitions are plausible:

```
digraph[g] = dig[reverse[g], nil]

dig[l,m] =
   if n l then m
   else letrec m' =
            ((aal . neighbors) . m)
        and result = dig[dl,m']
        and neighbors =
          map[λx. findnode[x,result],dal]
        in result   .
```

Here *result* is always to be in effect the final representation of the whole graph; *neighbors* is to be the list of nodes adjacent to a single node, obtained by looking up their labels in *result*, and m and m' are successive subgraphs, being built up (with labels in reverse order to their occurrence in the argument l) as the recursion of *dig* digs deeper. To carry out the construction of nodes we need the unremarkable auxiliary functions

```
map[f,l] =
   if nl then nil else (f[al].map[f,dl])
```

- this is the historical "mapcar" - and

```
findnode[lab,res] =
      if aa res ≡ lab then a res
      else findnode[lab, d res]   .
```

It is clear that by the use of a function similar to *digraph* one can construct any list structure whatever, by translating from an acyclic representation of it computed by conventional means; however, this is unlikely to be the most natural way of proceeding. As a somewhat whimsical next example, let us take the construction of a graph whose nodes correspond to the vertices of a cube and whose arcs correspond, in reciprocal pairs, to the cube's edges. For simplicity we may do without labels, so that each node will be nothing but a list of its adjacent nodes; and as the representing structure will be strongly connected, we may accept any one of its nodes as standing for the whole cube (and similarly with other graphs which arise along the way).

To motivate the following solution, observe that any graph may be regarded, loosely speaking, as a polygon, and consider the operation of stretching a polygon out into a prism - i.e., making two copies with corresponding nodes joined up. Given a general function for this operation, we can then compute the cube as

```
      prism[prism[prism[nil]]]   .
```

Prism is not hard to write - though it is perhaps confusing to keep in mind that a graph, a node, and a list of nodes are all the same thing; a suitable atlas maps each node in the argument graph to the corresponding edge (represented by the pair of its two ends) in the result:

```
prism[g] = a value[g, pris[g, empty[]]]

pris[g,h] =
 if defined[g,h] then h
 else letrec h' = extend[g, (g1.g2),h]
      and h" = prislist [g,h']
      and g1 = (g2.map[λx.a value[x,h],g])
      and $g2$ = ($g1.map[\lambda x.$d $value[x,h],g]$)
      in $h$"
$prislist[g,h]$ =
      if n$g$ then $h$
      else $prislist[$d$g, pris[$a$g,h]]$    .
```

6. The Knuth-Morris-Pratt string matching algorithm [7]

This is a hard example, intended to display the techniques under discussion as useful in the programming of a somewhat serious algorithm. The problem is to count the occurrences of a given list of characters $p = (p_1 \ldots p_m)$ - the "pattern" - as a contiguous substring of a typically much longer list of characters, the "text". The obvious algorithm can - albeit for rather pathological strings - require time proportional to $m \times n$, n being the length of the text. However, the job can be done in time proportional to $m + n$; the insight necessary to see that this must be so is to observe that if the text is cut at any point, the total count is determined by the number of complete occurrences of p to the left of the cut, the remainder of the text to the right, and the longest prefix of p which ends at the cut: all other possible occurrences of p across the cut are determined by that one. Therefore, there must be an automaton of $m + 1$ states - one for each prefix of p - which rolls along the text inspecting each character once and always assuming the state corresponding to the longest prefix of p just passed over.

For an abstract version of the algorithm, we may suppose that the states of the automaton are the natural numbers 0 through m, and that its law of operation is given by a function

$$transit: [0 \ldots m] \times \text{character} \rightarrow [0 \ldots m] .$$

Then we may write our program as a simple interpreter for the automaton:

```
$counter[k,s]$ =
      if $k=m$ then 1 + $counter[2ndbest[m],s]$
      else if n$s$ then 0
      else $counter[transit[k,$a$s],$d$s]$    .
```

The number of occurrences of p in a text t will be given by $counter[0,t]$. Here rather than count a pattern occurrence and inspect the character following it both at once, we have allowed the automaton to change state without reading a character from m to $2ndbest[m]$, which is defined to be the length of the longest prefix of p which is also a proper suffix of p; thus $2ndbest[m]$ is the state from which to begin considering the next possible occurrence of p to the right of the one just counted. We must, therefore, interpret a combination of state k and remaining text s as "k is the length of the longest as yet uncounted prefix of p which ended just before the start of s".

The automaton's transition function, depending on the structure of p, is still to be defined. The helpful idea is to define $2ndbest[k]$ for all states k, as the length j of the longest prefix $p_1 \ldots p_j$ of p which is also a proper suffix of $p_1 \ldots p_k$. Intuitively, given that $<k,s>$ corresponds to some position where the pattern may occur in the text, $<2ndbest[k], s>$ corresponds to the next possible occurrence to the right.

Now, considering the intended effect of $transit$ - to always land us in the highest numbered state consistent with what we know - we can write the following truth about it:

```
$transit[k,c]$ =
   if $p_{k+1} = c$ then $k+1$
         {a prefix of $p$ is extended}
   else if $k=0$ then 0
         {empty prefix could not be extend-
          ed; have to try it again at the
          next character}
   else $transit[2ndbest[k],c]$
```

{a non-empty prefix could not be extended; outcome of *transit* should be the same as if we had confronted the second-best prefix with this character instead} .

The special treatment of state 0 here is annoying, and stems from the empty prefix's having no second best. It is convenient to define $2ndbest[0]=-1$, a new state signifying "none of the pattern has been seen, and moreover $\underline{a}s \neq p_1$". The equation about *transit* may now be rewritten to involve state -1, but to treat states $0,\ldots, m-1$ uniformly:

```
$transit[k,c]$ = $\underline{\text{if}}$ $k=-1$ $\underline{\text{then}}$ 0
                 $\underline{\text{else}}$ $\underline{\text{if}}$ $p_{k+1}=c$ $\underline{\text{then}}$ $k+1$
                 $\underline{\text{else}}$ $transit[2ndbest[k],c]$   .
```

This equation can serve to computer *transit* recursively; it provides a second level of the abstract interpreter, which now requires only a tabulation of *2ndbest* to become executable.

Now to introduce some list processing: in a Lisp context it is natural, rather than storing values of *2ndbest* in an array, to directly model the automaton's state diagram in list structure. The following modelling scheme will make for easy interpretation: call the "concrete" states $q_{-1}, q_0, \ldots, q_m$ to avoid confusion, and define

$$q_{-1} = (\text{SKIP } q_0)$$

$$q_i = (\text{TEST } p_{i+1}\ q_{i+1}\ q_{2ndbest[i]}) \qquad i=0,\ldots,m-1$$

$$q_m = (\text{ACCEPT } q_{2ndbest[m]}) \quad .$$

Each state may be regarded as an instruction, listing the kind of action required of the interpreter and the necessary parameters.

The "concrete" interpreter for this model of the automaton is easy to write on the model of the abstract one. Corresponding to *counter* we have:

```
$count[q,s]$ =
    $\underline{\text{if}}$ $\underline{\text{a}}\ q \equiv$ ACCEPT $\underline{\text{then}}$ $1+count[\underline{\text{ad}}\ q,s]$
    $\underline{\text{else}}$ $\underline{\text{if}}$ $\underline{\text{n}}\ s$ $\underline{\text{then}}$ 0
    $\underline{\text{else}}$ $count[trans[q,\ \underline{\text{a}}\ s],\underline{\text{d}}\ s]$
```

and corresponding to *transit*:

```
$trans[q,c]$ = $\underline{\text{if}}$ $\underline{\text{a}}\ q \equiv$ SKIP $\underline{\text{then}}$ $\underline{\text{ad}}\ q$
               $\underline{\text{else}}$ $\underline{\text{if}}$ $\underline{\text{ad}}\ q = c$ $\underline{\text{then}}$ $\underline{\text{add}}\ q$
               $\underline{\text{else}}$ $trans[\underline{\text{addd}}\ q,c]$   .
```

Note: *count* and *trans* are fixed functions, good for all patterns, whereas *counter* and *transit* had knowledge about p built in.

There is still the problem of constructing the automaton from the pattern. Let us first consider how to compute *2ndbest*, beyond the conventional value $2ndbest[0] = -1$. Observe that $2ndbest[1]$ is zero (the only possible value) and that for $k>1$, $2ndbest[k]$ is at most one more than $2ndbest[k-1]$ - this when $p_k = p_{2ndbest[k-1]+1}$ - and otherwise is at most one more than $2ndbest[2ndbest[k-1]]$, and so on. Thus, for $k>0$, $2ndbest[k]$ depends only on $2ndbest[k-1]$ and p_k, say $2ndbest[k] = Ext[2ndbest[k-1],p_k]$, where we can describe *Ext* as follows:

```
$Ext[j,c']$ =
   $\underline{\text{if}}$ $j = -1$ $\underline{\text{then}}$ 0
         {a roundabout way of making
          $2ndbest[1]$ come out to 0}
   $\underline{\text{else}}$ $\underline{\text{if}}$ $p_{j+1} = c'$ $\underline{\text{then}}$ $j+1$
         {case where $2ndbest[k]$ is found to
          one more than some other $2ndbest$
          value}
   $\underline{\text{else}}$ $Ext[2ndbest[j],\ c']$
         {case where we must appeal to some
          next best}
```

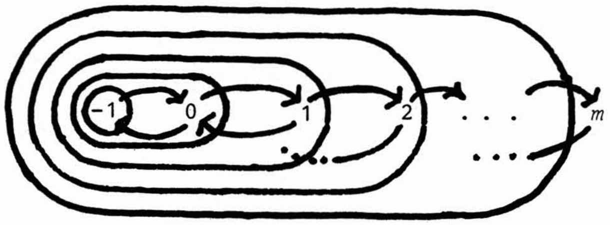
Ext has turned out to be the same function as *transit*; so to sum up, a complete description of *2ndbest* is given by

```
$2ndbest[k]$ = $\underline{\text{if}}$ $k=0$ $\underline{\text{then}}$ -1
               $\underline{\text{else}}$ $transit[2ndbest[k-1],\ p_k]$ .
```

Now we are ready to embody our knowledge about *2ndbest* in a function, *compile*, which will construct the automaton-as-list-structure from the pattern, and so will satisfy

$$counter[0,t] = count[compile[p],\ t] \quad .$$

Observe that the general shape of the automaton is



Each part enclosed by a contour has exactly two connections with the first state outside the contour. This makes it plain that our recursion should be over successively shorter prefixes of the pattern; i.e., that we should reverse p to begin with. Each recursive level of the compiler should construct states $q_{-1}, \ldots, q_k$ from $p_1 \ldots p_{k+1}$; it should come out with both q_0 (the final answer) and $q_{2ndbest[k+1]}$ (for building in to q_{k+1}); and as an additional argument it will need q_{k+1} itself, to form the "success continuation" of q_k. Thus for the top level of the compiler we may write

```
compile[p] =
    letrec i,j = compil[reverse[p],final]
    and final = (ACCEPT j)
    in i   .
```

We need a *compil* which satisfies

$$compil[(p_{k+1} \cdots p_1), q_{k+1}] = q_0, q_{2ndbest[k+1]} \quad .$$

This is achieved by the following definition:

```
compil[rp, succ] =
     if n rp then succ, (SKIP succ)
     else letrec i,j = compil[drp, this]
              and this = (TEST a rp  succ  j)
              in i, trans[j, arp]   .
```

Note that atlases were not needed here, and that subject to a little reasoning about how much work *trans* can do during compilation, a complete invocation of $count[compile[p],\ t]$ should indeed run in time linear in the sum of the lengths of text and pattern. What makes the program bizarre is that the automaton is put into operation in furtherance of its own construction. We regret the involved nature of this example, but it may serve to make the point that the construction of cyclic structures cannot in general be confined to intervals when "ordinary computing" is not going on; rather the two are likely to be thoroughly intertwined.

This approach to implementing the Knuth-Morris-Pratt algorithm was invented by one of the authors (Schwarz) who constructed the automaton by a different method, which exploited lazy evaluation. The other author was delighted to discover that his less exotic approach to computing with cyclic structures would handle the example, although perhaps less perspicuously than can be done with lazy evaluation.

7. Implementations

We shall now attempt to suggest means by which the plan sketched in Section 2 for computing solutions to recursive list structure equations can be carried out, with as little disturbance as possible to ordinary notions of Lisp implementation. In outline, the plan is clear: evaluate the right hand sides in the order given, and as each value is obtained, ascribe it to its left-hand-side variable. The only difficulty arises when evaluation of some one of the variables must precede ascription of its computed value to it.

The notion of "replacing all occurrences" of a variable's dummy value by its computed value, in the sense of overwriting all the arbitrarily scattered instances of a pointer, appears computationally infeasible. The next most straightforward approach appears to be to implement "coalescing" of dummy with computed values, by the device of so-called "invisible pointers" which have been proposed for a variety of uses. An invisible pointer is a specialized kind of cell which contains the address of another cell and which, it is arranged, is transparent to all pointer-following operations. That is, an operation such as *car*, whenever it is apparently about to deliver an invisible pointer cell as result, must look through it and deliver instead the cell it points to (unless this is in turn transparent ...). To serve as a dummy value, we need an invisible pointer cell initialized to a quiescent (and opaque) state in which it appears to be an ordinary object, although it is amenable to none of the five primitive operations other than *cons*. To make a dummy value disappear when its time is up it is necessary only to make it point to the corresponding computed value.

In practice, it seems that values of variables which are needed before they have been computed turn out most often to be newly-minted pairs. It is unnatural, but tolerable, to restrict what is an allowed system of equations by demanding that this should always be the case. If this is done, invisible pointers may be dispensed with; instead *cons* cells with some conventional contents can serve as dummy values. Under this regime, when a right-hand-side value has been computed, and if its corresponding dummy or, one may say, "predicted" value has ever been accessed, the mechanism administering solution of equations must verify that the value as computed is a pair, and copy its *car* and *cdr* fields into the value as predicted. If the rules have been followed, the *cons* cell of the value as computed is now garbage and disappears; no one will ever know that it was unable to coalesce with the predicted, and now only, value of the variable.

Of the preceding examples, the only one which falls afoul of the restriction just laid down is *digraph*: when a node of the graph has outdegree zero, the varible *neighbors* will take the value nil. *Digraph* can be fixed up, tediously, by making a special case of such nodes; this appears to be the typical situation.

The second suggestion given for implementation can serve as a guide to hand-translating programs with recursive list structure definitions into present-day Lisp; in this light the present work may be of some immediate value to practitioners as a proposal for the extremely disciplined use of *rplaca* and *rplacd*.

Appendix: More about atlases

At least three respectably efficient schemes of representing atlases suggest themselves: actual marking, which in its most flexible form probably amounts to equipping *cons* cells as well as atoms with property lists; hash tables, provided each cell comes with a permanent identifying number - this could cheaply be its address, so long as relocating garbage collection is not practiced; and some form of ordered trees keyed by cell addresses - these, though slower than hashing, would tolerate order-preserving garbage compaction.

It appears that the side-effect-free specification given above for atlases would not be met by property lists or hash tables. On the other hand 2-3 trees, for example, could be made to satsify it, as could of course the association lists of classical Lisp. As may be seen, however, by study of the examples, "pure" atlases, for which extension is nondestructive, are probably hardly ever essestial; hence an imperatively oriented atlas facility based on marking or hashing may prove to be worth providing instead of, or as well as, a functional one (see Schwarz [8]). Atlases are widely applicable and, we suggest, are more natural to Lisp than arrays.

References

1. Landin, P.J., "The mechanical evaluation of expressions", *The Computer Journal 6*, 4 (January 1964), pp. 308-319.

2. Burge, W.H., *Recursive Programming Techniques*, Addison-Wesley, Reading Mass. (1975).

3. Henderson, P., *Functional Programming: Application and Implementation*, Prentice Hall International (1980).

4. Henderson, P. and Morris, J.H., "A lazy evaluator", *Third ACM symposium on Principals of Programming Languages*, Atlanta (1976), pp. 95-103.

5. Friedman, D.P. and Wise, D.S., "Cons should not evaluate its arguments", *Third International Colloquium on Automata, Languages, and Programming*, Michaelson, S. and Milner, R., eds., Edinburgh University Press, (1976).

6. Morris, F.L., "On list structures and their use in the programming of unification", Syracuse University, C.I.S. report 4-78.

7. Knuth, D.E., Morris, J.H., and Pratt, V.R., "Fast pattern matching in strings," *SIAM Journal of Computing 6* 2 (June 1977), pp. 323-350.

8. Schwarz, J.S., "Verifying the safe use of destructive operations in applicative programs", *Proc. 3rd International Symposium on Programming*, B. Robinet ed., Dunod, Paris (1978).