A List Scanning Facility for Icon

Rodney A. Norden

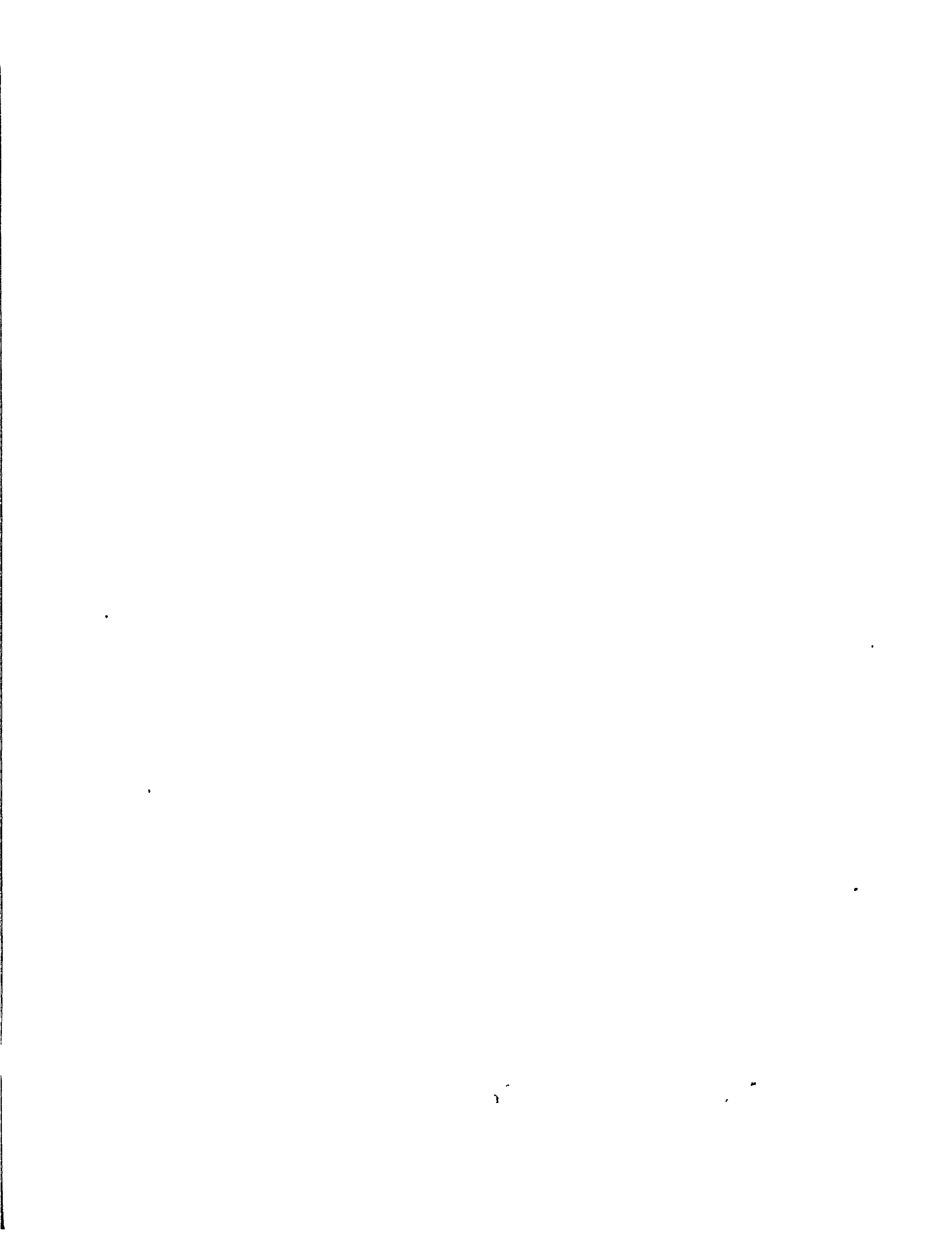
TR 78-8

Department of Computer Science

The University of Arizona

May 16, 1978

*This work was supported in part by the National Science Foundation under Grant MCS75-01307.



1. Introduction

This paper describes an experimental list scanning extension to the Icon programming language. The facility is modeled after the novel string scanning features provided by Icon. The reader should be familiar with Icon [1], and a working knowledge of SNOBOL4 [2] and SL5 [3] also will be helpful, since Icon shares a philosophical base with these languages.

One of the most important aspects of Icon is its use of generators and goal-directed evaluation to replace the concept of string pattern matching, which is central to SNOBOL4. These new features provide the advantages of string pattern matching without the numerous disadvantages associated with patterns [4].

Facilities for list structure scanning described here are also based on generators and goal-directed evaluation. A deliberate attempt was made to develop facilities analogous to those used for strings rather than modeling a system after existing list processing systems such as LISP.

The new list scanning system has several features which have no counterpart in string processing. This is a consequence of the complexity of list structures, contrasted with the simple linearity and homogeneity of strings.

2. List Processing Facilities

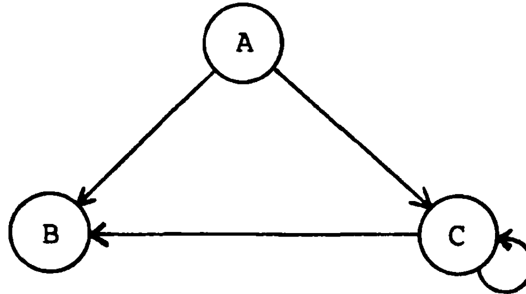
2.1 Definition of Lists

The term list as used here refers to heterogeneous linear arrays with an origin of 1. A list of n elements is created by <u>array</u> n. A list of n specific elements can be specified explicitly

$$[x_1,\ldots,x_n]$$

The sections that follow describe various other ways of constructing lists.

One use of lists is to represent general directed graphs. In this case, each list is capable of representing a node containing an arbitrary number of value fields and an arbitrary number of pointer fields to other nodes of the graph. Consider, for example,



The following program segment constructs a list representation of this graph in which the value of each node is the first element of the corresponding list.

```
l1 := ["A",,]
l2 := ["B"]
l3 := ["C",l2,]
l1[2] := l2
l1[3] := l3
l3[3] := l3
graph := l1
```

Because of the complexity of relationships that may occur among lists such as those that represent graph structures, it is convenient to have a diagnostic facility for printing lists. The function ldump(l) is provided for this purpose. A call to ldump with the graph above as an argument, i.e. ldump(graph), produces

```
list1 --> ["A",list2,list3]

list2 --> ["B"]

list3 --> ["C",list2,list3]
```

The ldump function assigns the names list1, list2, and so on to lists in the order in which they are encountered in the dumping process. Note how cycles and pointers (references to lists) are represented by ldump.

Indexing into lists is defined in a manner analogous to that for strings. A position in a list is assumed to refer to a point in front of the corresponding element. Position 1 is in front of the first element from the left; position 2 is between elements 1 and 2 (i.e. in front of element 2), and so on. Negative indexes refer to positions in front of the n-th element from the right end of the list. An example follows.

```
["A","B","C",["D","E"]]
|   |   |   |         |
1   2   3   4         5
-4  -3  -2  -1        0
```

2.2 Basic List Operations

The three fundamental operations on lists are list concatenation, sublist generation, and list comparison.

Concatenation of lists is a natural generalization of concatenation of strings. The operation l1 ||| l2 performs the concatenation, producing a new list with the elements of l1 followed by those of l2. For example, given two lists a and b

```
a := ["A","B"]

b := ["C","D"]
```

the expression a ||| b produces

```
["A","B","C","D"]
```

The null list, i.e. the list consisting of zero elements, is the identity with respect to list concatenation. It is analogous to the null string and, in fact, the null string may be used interchangeably with the null list in list concatenation.

The function lsection(l,i,j) returns a sublist composed of the elements of l between positions i and j, inclusive. The values of i and j may be negative in accordance with the list indexing conventions given earlier. For example, if

```
c := ["A","B","C","D","E"]
```

then either lsection(c,2,5) or lsection(c,2,-1) returns the sublist

```
["B","C","D"]
```

Note that this function is analogous to section(s,i,j) for strings.

List comparison is performed by the operation l1 === l2 as follows: if l1 and l2 are lists of the form

$l1 := [x_1,x_2,...,x_n]$

$l2 := [y_1,y_2,...,y_m]$

then l1 === l2 succeeds if and only if

(a) n = m

(b) for 1 <= i <= n

(1) type(x_i) == type(y_i) and

(2) compare(x_i,y_i) succeeds, or

(3) type(x_i) == "list" and x_i === y_i succeeds

2.3 List Scanning

List scanning is analogous to string scanning and provides its advantages: an implicit subject to which operations apply and implicit cursor movement. List scanning thus provides automatic bookkeeping and a concise notation for list processing.

In list scanning, the value of the keyword &lsubject is the implicit subject of scanning. The value is established by an assignment of the form

```
&lsubject := ["A","B","C",["D"]]
```

This expression sets the subject to the given list and also automatically sets the keyword &lcursor to 1, positioning it at the beginning of the list. All subsequent list scanning operations refer implicitly to these two <u>global</u> keywords until they are reset.

The operation l ?? e operates in the manner of s ? e for string scanning. This operation sets the value of &lsubject to l (automatically setting &lcursor to 1) and then evaluates e. The previous values of &lsubject and &lcursor are saved before the evaluation of e and restored afterwards. This feature allows recursive and nested list scanning. Examples of its use are given later.

2.4 List Sequencing

Sequencing through a list structure is an important operation. There are two keywords that are generators operating implicitly on &lsubject which return successive elements starting at the current value of &lcursor and continuing to the right end of &lsubject:

(1) &element, which generates successive elements of a &lsubject.

(2) &list, which generates successive elements of &lsubject that are lists.

These generators are most commonly used with the *every* construct, which provides a means for operating on each element of a list or on each list contained in ("pointed to" from) a list. For example

```
&lsubject := ["A","B","C","D","E"]
every x := &element do write(x)
```

generates

```
A
B
C
D
E
```

In another example,

```
&lsubject := ["A","B","C",["D"],["E","F"]]
every x := &list do write(&lcursor)
```

generates the following

```
4
5
```

2.5 List Scanning Functions

The list scanning functions are similar to the string scanning functions. The differences are a consequence of the special features of lists. The list scanning functions apply exclusively to the value of &lsubject.

A typical list scanning function is lmove(n), which adds n to the value of &lcursor and returns as value the sublist of &lsubject between the old and new values of &lcursor. If the value of &lcursor would be out of the range of &lsubject, the operation fails and &lcursor is not changed. Assuming the value of &lsubject to be the list

```
[3,"A","B",["D","E"]]
```

then the expression

```
ldump(lmove(&element))
```

prints the following list

```
list1 --> [3,"A","B"]
```

Another scanning function is ltab(n), which sets &lcursor directly to n and returns as value the sublist between the old and new values of &lcursor. If the value of &lcursor would be

out of the range of &lsubject, the operation fails and &lcursor is not changed. This function is similar to lsection(l,i,j), but lsection requires starting and ending positions to be explicitly specified and ltab assumes the current value of &lcursor as a starting position. As an example, consider

```
&lsubject := [3,"A","B",["D","E"]]
lmove(&element)
rest := ltab(0)
```

The value of rest after execution of this code is the list

```
[["D","E"]]
```

The scanning function lupto(x) is a generator. The value returned by lupto(x) is the position of the first occurrence of x as an element in &lsubject, starting at the current value of &lcursor. Unlike the corresponding string scanning function, upto(c), lupto(x) does not specify a set of objects, but only a single value as a distinct element of &lsubject. Since there may be more than one occurrence of x in &lsubject, there is more than one possible value of the expression lupto(x). These values are generated (in increasing sequence) as needed. For example,

```
&lsubject := ["A","B","B",["D","E"]]
every write(lupto("B"))
```

prints

```
2
3
```

the positions of each "B" in &lsubject.

The prefix operation !l allows the user to specify a sublist for scanning purposes. A comparison is made for the sublist starting at the current value of &lcursor. If the comparison is succcessful, &lcursor is advanced through the sublist and the scanning operation succeeds; otherwise, it fails. Consider the following code segment

```
l := ["A",["B"],"C","D",["E"]]
l ?? {
     lmove(2)
     if ![“C","D"] then ldump(&element)
    }
```

The scanning operation succeeds and prints

```
listl --> [["E"]]
```

2.6 The Marking Facility

Since list structures such as directed graphs are usually non-linear and contain cycles, it is frequently necessary to have a means of detecting which nodes have been processed or which parts of the structure have been traversed. For this purpose, a list marking facility is provided. Its features include two keywords, &mark and &unmark, which are used for marking a list and removing a mark from a list, respectively. An auxiliary table provides storage for pointers to marked lists and the values with which the lists are marked. Since the marking facility maintains a separate table of marked nodes, there is no physical alteration of the structure being marked.

The auxiliary marking table is created by an assignment to &marktable of the form

```
&marktable := table
```

&marktable is independent of &lsubject and it survives any scanning operation. A new table may be created at any time by another assignment to &marktable. Once created, any valid operation on tables may be applied to &marktable, such as conversion to an array for sequential processing.

The keywords &mark and &unmark automatically reference the current value of &element. If &element is not a list, both &mark and &unmark references fail. &element can be marked with an arbitrary value by assignment to &mark. For example

```
d ?? {&list; &mark := 1}
```

marks the first list element in d with a value of 1. The value assigned in marking can be changed by another assignment to &mark whenever the list is the value of &element. The keyword &unmark removes a marked list and its associated value from &marktable, but no assignment need be made. The &unmark expression fails if &element is not marked. For example

```
every &list do
   if &unmark fails
      then write("unmarked node encountered")
      else ldump(&element)
```

This expression prints a diagnostic message if &element is not marked. If &element is marked, it is unmarked and printed.

While &mark and &unmark operate on tables, they provide an accessing method not available with the direct use of tables. Assignment of a value to &mark adds &element and the value to the table, but referencing &mark and &unmark both fail when &element is not present and no table insertion is made.

3. Examples

This section contains a number of examples of the list scanning facilities and typical programming techniques illustrating its use.

3.1 Graph Construction and Representation

There are many ways of representing directed graphs for computer processing. In one method of representation each node is represented as a string of the form

```
label:value:node1,node2,...,nodeN
```

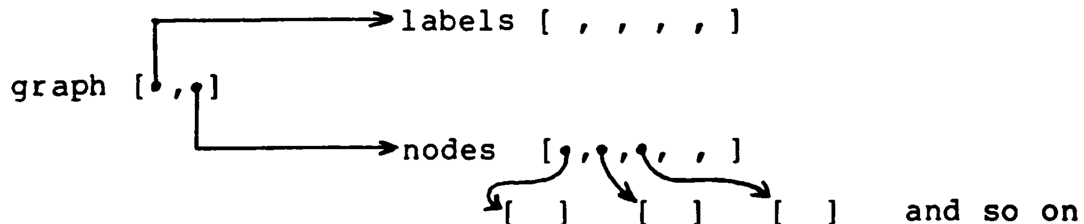
where the label field contains a label for the node, the value field contains the node's value, and the "node" field contains zero or more references (arcs) to neighbors of the node [5]. This concise notation is easy to process with Icon's string processing capabilities.

The string representation of the graph given in Section 2.1 provides an example:

```
l1:"A":l2,l3
l2:"B":
l3:"C":l2,l3
```

The labels l1, l2, and l3 are auxiliary to the graph itself and simply provide names for the nodes.

One possible list representation for directed graphs consists of a list for each node and two other lists: one for the node labels and one for references to the nodes (arcs). Each node's list contains the value as its first element and references to neighbors in the remaining elements. The general form of the list representation follows:



As an example of this representation, the following output results from ldump applied to this list representation of the graph of Section 2.1:

```
list1 --> [list2,list3]
```

```
list2 --> ["11","12","13"]

list3 --> [list4,list5,list6]

list4 --> ["A",list5,list6]

list5 --> ["B"]

list6 --> ["C",list5,list6]
```

Here list2 is the list of labels and list3 is the list of nodes. The nodes themselves are list4, list5, and list6. The entire graph is represented by list1, which points to the label list and the node list. This representation of graphs will be assumed throughout the remaining examples in Section 3.

The following procedure converts the the string representation of a graph contained on a file to the list representation. The procedure graph(f) uses two passes for the conversion. The procedure pass1(f) reads the string data for each node and inserts the various fields into their proper positions in the lists. The procedure pass2(g) uses the list processing facility to convert labels of a node's neighbors into actual references ("pointers") to the nodes themselves.

```
# The procedure graph(f) constructs a graph representation using lists
# It reads data from file f of the form
#
#       label:value:node1,node2,...,nodeN
#
procedure graph(f) local graph

   if graph := pass1(f) fails then fail # error in data
   graph := pass2(graph)
   return graph
end
```

```
procedure pass1(f) local labels, lists, temp

   while &subject := read(f) do {
        # isolate label field
        if labels := labels ||| [tab(upto(":"))] then move(1)
                                                     else fail

        # isolate value field
        if temp := [tab(upto(":"))] then move(1) else fail

        # add all node fields
        while temp := temp ||| [tab(upto(","))] do move(1)

        # handle last node reference
        temp := temp ||| [tab(0)]
        lists := lists ||| [temp]
       }

   return [labels,lists]
end
```

Note the use of list concatenation to construct a list an element at a time.

```
# The procedure pass2(g) uses list scanning to automatically
# set &lsubject to the list of nodes, g[2], and to process
# every arc of each node.  Note that any &lsubject in effect
# outside the procedure is automatically saved at entrance.
# Note also the use of lupto(label) to get the position
# of the current value of label in the label list.
#
procedure pass2(g) local label

   g[2] ?? {                       #isolate list of nodes
      every &element ?? {          # process every node
         ltab(2)                   # skip value of node

         # process every arc of each node, and replace label
         # by corresponding node

         every label := &element do
            &element := g[2][g[1] ?? lupto(label)]
         }
      }
   return
end
```

Note lupto(label) searches the list of labels, g[1], and returns the index of the corresponding node in the list of nodes, g[2].

A useful property of a graph is the number of nodes and edges it contains. The number of nodes in a graph g is simply length(g[1]). Computing the number of edges is slightly more

complex. A count is made of all pointers to lists. These represent directed edges in the graph representation.

```
procedure edges(graph) local edges

   edges := 0
   graph[2] ?? {

   # count arcs for all nodes in graph

      every &element ?? {
         every &list do edges+
         }
   }

   return edges
end
```

3.2 Access Path Determination

Determination of access paths is a common problem in management of heap storage where variable-sized blocks are allocated.

In this example, the marking facility is used to mark all nodes accessible from a given list. The algorithm is similar to the one used by SNOBOL4 [6]. The argument to procedure mark is the list of pointers to nodes, referred to here as the basic block. The procedure mark(block) recursively marks all lists in the structure, starting rrov the basic block. It assumes there is no pointer to the basic block so that it does not need to be marked.

```
# The procedure mark(block) uses the procedure mark2(block) to
# mark all lists accessible from its argument, the basic block.
# The procedure mark(block) is needed to set up &marktable.
#
procedure mark(block)
   &marktable := table
   block ?? {
      every &list do {           # process every list in block
         &mark := 1                      # mark it
         mark2(&element)                 # follow pointers
         }
      }
   return &marktable
end
```

```
procedure mark2(block)
   block ?? {
      every &list do                    # process all lists
         if &mark fails then {    # if not marked
            &mark := 1                  # mark it
            mark2(&element)             # follow pointer recursively
            }
      }
   return
end
```

3.3 Topological Sort of a Directed Graph

Following Knuth [7], a partial ordering of the elements of a set G is a relation between the elements of G satisfying three properties for arbitrary and not necessarily distinct elements x, y, and z in the set. The symbol <= means "precedes or equals".

Property 1: if x <= y and y <= z, then x <= z.

Property 2: if x <= y and y <= x, then x = y.

Property 3: x <= x.

Topological sorting can be viewed as the process of finding a linear ordering of objects in which a given partial order can be embedded. Topological sorting is useful in the analysis of activity networks where a large, complex project is represented as a directed graph in which the nodes correspond to the goals in the project and the edges correspond to activities. Some goals must be completed before others are begun. The topological sort gives an order in which these goals can be achieved.

In the following examples, a depth-first search is used to topologically sort the nodes of a graph. The process involves isolating nodes with no outgoing arcs to unprocessed nodes.

To contrast conventional list processing with the facilities described here, two solutions follow. The first solution uses arrays and tables with standard processing techniques. The second solution uses list scanning.

Solution 1 - Topological Sort without List Scanning

```
global topsort, position, labeled
        .
        .
        .
# The procedure topsort handles the referencing environment for the
#       recursive procedure tsort
#
procedure topsort(graph) local nodes, index, x
   nodes := length(graph[1])
   labeled := table nodes         # for labeled ancestors
   topsort := array nodes         # for sorted nodes
   position := nodes + 1
   x := graph[2]                  # retrieve pointer fields of all nodes

   # step through all nodes looking for nodes with no labeled ancestor

   every index := 1 to nodes do
        if labeled[x[index]] == "" then tsort(x[index])

   return topsort
end

# The procedure tsort uses a depth first recursive search to find a
#       node with no outgoing edges to unprocessed nodes
#
procedure tsort(v) local w, index
   labeled[v] := 1                # mark current node

   every index := 1 to length(v) do {
        w := v[index]
        if labeled[w] == "" then tsort(w)
        }

   # insert v into topsort at a position before any descendant

   topsort[position-] }:= v

   return
end
```

Solution 2 - Topological Sort with List Scanning

```
# The procedure topsort handles the referencing environment for the
#       recursive procedure tsort
#
procedure topsort(graph)

   graph[2] ?? {                # set &lsubject to list of nodes

   # process all nodes in graph

      every &list do
         if &mark fails then {   # any marked ancestor?
                &mark := 1
                &element ?? {topsort := tsort(topsort)}
                }
      }

   return topsort
end


# The procedure tsort uses a depth first recursive search to find a
# node with no outgoing edges to unprocessed nodes.
# Note that topsort is null at the first call; list concatenation
# extends it automatically.
#
procedure tsort(l)

   # process all unmarked descendants of current &lsubject

   every &list do
      if &mark fails then {
                &mark := l
                &element ?? {l := tsort(l)}
                }

   # insert &lsubject in place before any descendant

   l := [&lsubject] ||| l

   return l
end
```

The topological sort with list scanning has the following advantages. The bookkeeping needed in Solution 1 is not present; it is handled automatically by the system. Solution 2 is also more readable and concise. The list processing system allows a level of abstraction that is not present in Solution 1. The user may think directly in terms of processing every list for some property, and very nearly code that idea directly.

4. Conclusions

4.1 Current Status

A working version of the list scanning facility has been implemented in a prototype of Icon based on SL5 [3]. As implemented, there are only minor syntax differences from the full Icon version presented in this paper. All the facilities are consistent with the design of Icon, so there are no technical problems in implementing them in the production version of Icon.

4.2 Evaluation

The list scanning facility provides a natural linguistic mechanism that stresses the essential details of a list manipulation problem while suppressing most of the unnecessary details. It allows the user to think and program directly in such terms as: for every list perform an operation, process the current element, and so on. It provides a readable and concise notation for problems involving list manipulation.

There are several problems with the list scanning facility in its present form. Many of these problems are shared with Icon's string scanning facility [4]. Typical of this class of problems are the scope of &lsubject, the hazards of global variables for communication, and the choice of a complete yet small set of primitives. A better choice of primitives should result as experience is gained with list scanning.

The list scanning facility is potentially inefficient, since all operations that return lists must allocate space. The semantics of the operations require copying of the lists to avoid erroneous side effects.

4.2 Suggested Extensions to the Facility

One natural direction for future development is a unified scanning facility that is polymorphous, allowing strings and lists in all operations. For example, x1 || x2 could represent both list and string concatenation. Similarly, &subject could be either a string or a list and tab(n) could move the cursor position regardless of the type of &subject.

There are both advantages and disadvantages involved in this approach. An advantage is the smaller vocabulary that would result from combination of list and string scanning facilities. Only the generator &list and the keywords &mark, &unmark, and &marktable associated with the marking facility are exclusively list oriented. &element, applied to strings, could reference the current character in &subject. All other list operations can be unified with their string counterparts.

A unified vocabulary would provide many useful analogies that would make the facility easier to learn and to use. The same conceptual operation would be represented by the same operator, regardless of the types of its operands, with the type-specific operation selected internally.

Unification of string and list scanning also has some disadvantages. There would be a greater possibility for error, since the exact nature of the generic operations would not be obvious from context, although type declarations would alleviate this problem to some extent. Similarly, in the absence of type declarations, necessary run-time type checking would also introduce overhead in the implementation of the generic operations.

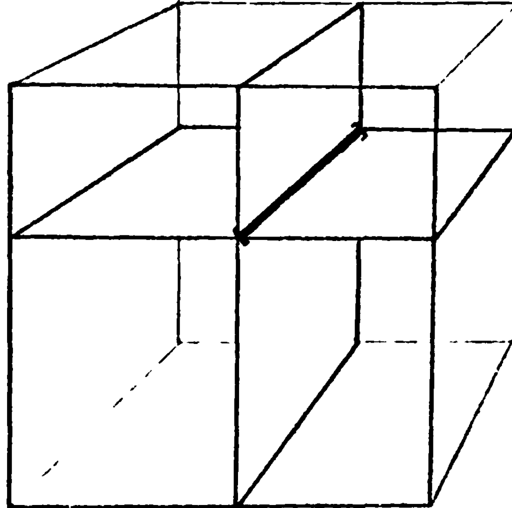
A further problem is that there are operations that do not have natural meanings for both types. For example, while tab(n) for strings is an important operation, the equivalent operation on lists, although included in the present list scanning facility, seems unnatural in many applications. Similarly, some string scanning operations, such as upto(c), do not have complete generalizations for lists (see Section 2.5).

There are many other possibilities for extending the facility. For example, it would be convenient in some cases to have multiple subjects. These could be used in simultaneous scanning of two or more list structures. A serious disadvantage exists with the use of multiple subjects, however, since any such facility would reintroduce the complexity which the scanning facility is designed to suppress.

Other facilities are needed to support more general list scanning. For example, if &marktable is converted to an array, the current list scanning facility cannot operate directly on it, since the resulting array has two dimensions and is not a list as defined in Section 2.1. An array sectioning operator to convert a two-dimensional array to a set of linear arrays or lists would be useful.

Another possible extension is the generalization of list scanning to arrays of arbitrary dimensionality. This poses a difficult problem. Much of the advantage of list scanning is derived from the simplicity inherent in implicit cursor movement. In a linear array, that is, a list, the cursor is simply an integer and cursor movement is effected by incrementing and decrementing its value. In an array of n dimensions, a position is uniquely specified by n coordinates (x1,x2,...,xn). The list scanning functions can be generalized accordingly (for example, tab(a1,...,an)), but the result is complex and defeats the inherent advantages of scanning: conciseness and simplicity.

An alternative approach is to use list scanning <u>within</u> arrays of arbitrary dimensionality. A linear array may be specified in an n-dimensional array by the intersection of n - 1 hyperplanes orthogonal to the axes. For example, in the three-dimensional array shown below, the two planes intersect to select a row that is a linear array.



Similarly, the combination of a position and direction specifies a list that can be translated into &lsubject and &lcursor. List scanning can then be carried on along this list without regard for the other coordinates which may be considered implicit and fixed. The specification of a change in direction orthogonal to &lsubject produces new values of &lsubject and &lcursor along the orthogonal direction. Assignment of selected coordinates to a keyword, such as &direction, could accomplish this change of direction, selecting a list that is orthogonal to the current one.

This facility is not difficult to implement, but its utility remains to be proved. For example, are there kinds of data that can be organized as n-dimensional arrays which can be usefully processed by this type of list scanning with its necessarily "tunnel-vision" characteristics?

If there is any problem with the directions of future work, it lies in the selection from a number of interesting, but basically incompatible alternative extensions to the list scanning facility.

5. Acknowledgements

I am indebted to Ralph E. Griswold for many of the ideas that appear in this report and for his constant encouragement. His many critical readings of drafts of this paper and his helpful comments are also appreciated.

References

1. Griswold, Ralph E., David R. Hanson, and John T. Korb. The Icon Programming Language; a Preliminary Report. Technical Report TR 78-3, Department of Computer Science, The University of Arizona, Tucson, Arizona. April 10, 1978.

2. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. The SNOBOL4 Programming Language, 2nd ed. Prentice-Hall, Englewood Cliffs, New Jersey. 1971.

3. Griswold, Ralph E. and David R. Hanson. "An Overview of SL5", SIGPLAN Notices, Vol. 12, No. 4 (April 1977), 40-50.

4. Griswold, Ralph E. An Alternative to the Concept of "Pattern" in String Processing. Technical Report TR 78-4, Department of Computer Science, The University of Arizona, Tucson, Arizona. April 10, 1978.

5. Reingold, E. M., Nievergelt, J., and Deo, N. Combinatorial Algorithms. Prentice-Hall, Englewood Cliffs, New Jersey. 1977.

6. Griswold, Ralph E. The Macro Implementation of SNOBOL4, A Case Study in Machine-Independent Software Development. W. H. Freeman, San Francisco. 1972.

7. Knuth, Donald E. The Art of Computer Programming, Vol. 1, Fundamental Algorithms. Addison-Wesley, Reading, Mass. 1968.