Unifying List and String Processing in Icon*

Allan J. Anderson and Ralph E. Griswold

TR 83-4

February 26, 1983

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS81-01916.

Unifying List and String Processing in Icon

Icon, like its ancestor SNOBOL4, has extensive facilities for string processing but only limited facilities for list processing. Furthermore, except for a few elementary operations, Icon's list and string processing facilities are dissimilar. There are, however, many problems in which data is best represented with a combination of strings and lists [1].

Both strings and lists are sequences of values, providing the basis for a set of operations that apply to either type. One approach, of course, is to consider strings to be lists of characters. This simplifies the problem at the expense of relinquishing the concept of a string as a data object in its own right — a concept that is the basis for much of the usefulness of languages like SNOBOL4 and Icon.

This report describes an alternative approach in which strings and lists are retained as separate data types but with a set of operations that is defined over values of both types. The resulting language is designated as $\text{Icon}_{\mathcal{E}}$ in this report to distinguish it from standard Icon.

The reader should be generally familiar with Version 5 of Icon [1].

1. Basic Design Decisions

The most basic decision to be made is whether to have separate but similar operations for strings and lists. The goal of unification strongly suggests that the operations on strings and lists be the same, but polymorphous. This also avoids an excessively large vocabulary of operations.

Icon already has several polymorphous operations that apply to strings and lists, as well as to other types. For example, the element generation operator

```
!x
```

generates the 'elements' of either a string or a list. The elements of a string are its characters, or more precisely, its one-character substrings. For example,

```
!"Hello"
```

generates the values "H", "e", "l", "l", "o". The elements of a list consist of its values. Thus

```
!["Hello", "world"]
```

generates the values "Hello" and "world".

In Icon, concatenation is not polymorphous and there are separate operations for string and list concatenation:

```
s1 || s2
```

and

```
a1 ||| a2
```

In $\text{Icon}_{\mathcal{E}}$, these two operations are fused into a single polymorphous operation:

```
x1 || x2
```

The letter s is used in this report to indicate an expression whose value is a string, while a is used to indicate an expression whose value is a list. The letter x is used to indicate an expression whose value may be either a string or a list.

One way to handle lists in polymorphous operations is to consider lists to be of a 'higher type' to which strings can be converted in the fashion that integers are converted to real numbers. The natural conversion of a string s to a list would be [s]. In this interpretation

```
a || s
```

would be equivalent to

```
a || [s]
```

However, if strings were converted to lists automatically, then in general

(a || s1) || s2 ≠ a || (s1 || s2)

Consequently, concatenation would not be associative. In order to preserve useful properties of several operations on strings and lists, there is no type conversion between strings and lists in $\text{Icon}_{\mathcal{E}}$. Thus

```
a || s
```

is erroneous in $\text{Icon}_{\mathcal{E}}$.

There are several problems with existing polymorphous operations on strings and lists in Icon. Consider

```
s[i]
```

which references the ith character of s, and

```
a[i]
```

which references the ith value of a. Although both reference the ith element, there is a difference when an assignment is made to such a reference. In the case of strings,

```
s1[i] := s2
```

is essentially an abbreviation for

```
s1 := s1[1:i] || s2 || s1[i + 1:0]
```

In other words, one element can be replaced by any number of elements. (Only positive position specifications are considered in this report. Computations with nonpositive specifications apply in all cases, but it simplifies the presentation to consider only positive specifications.) On the other hand,

```
a[i] := x
```

changes the *value* of the ith element in a. This is quite different from the string case. For example, if

```
a := ["three", "bad", "apples"]
```

then

```
a[2] := "good"
```

changes the value of a to

```
["three", "good", "apples"]
```

more significantly

```
a[2] := ["very", "good"]
```

changes the value of a to

```
["three", ["very", "good"], "apples"]
```

not to

```
["three", "very", "good", "apples"]
```

which would be the case for list concatenation. The different interpretations of s[i] and a[i] result from the fact that strings are homogenous sequences of characters (even though there is no character data type in Icon), while list elements are arbitrary. String operations are applicative in this sense; a string operation may produce another string but it cannot change the value of the string on which it operates. Some operations on lists are not applicative, however.

Similarly,

```
s1[i:j] := s2
```

is an abbreviation for

```
s1 := s1[1:i] || s2 || s1[j + 1:0]
```

(Indices in range specifications are considered to be in non-decreasing order here to simplify the presentation.) On the other hand

```
a1[i:j] := a2
```

is erroneous in Icon, although it has a perfectly consistent interpretation as an abbreviation for list concatenation. The decision to interpret this as an error in Icon was arbitrary and probably a mistake. Consequently,

```
s[i]
```

is equivalent to

```
s[i:i + 1]
```

but

```
a[i]
```

is not, in general, equivalent to

```
a[i:i + 1]
```

To remove these discrepancies, a new subscripting operation is included in Icon$_{\mathcal{E}}$

```
x!i
```

which references the ith element of x. Thus

```
a!i
```

in Icon$_{\mathcal{E}}$ is equivalent to

```
a[i]
```

in Icon. To assure consonant interpretations for lists and strings, in an expression such as

```
s1!i := s2
```

s2 must be a one-character string. That is, an assignment to a substring expression changes the value of that element but does not change the size of the string or list. On the other hand,

```
a[i]
```

in Icon$_{\mathcal{E}}$ is equivalent to

```
a[i:i + 1]
```

in Icon. For compatibility

```
a1[i:j] := a2
```

in Icon$_{\mathcal{E}}$ is an abbreviation for

```
a1 := a1[1:i] || a2 || a1[j + 1:0]
```

Thus *subsection* references are distinct from *subscript* references in Icon$_{\mathcal{E}}$ and both have consonant interpretations for strings and lists.

2. Extending String Operations to Lists

Since Icon has many more operations on strings than it has on lists, a major problem in the design of Icon$_{\mathcal{E}}$ is finding an interpretation of these operations for lists that is consistent with their use on strings. Operations that are strictly based on the order of elements in strings, such as `reverse(s)`, can be trivially extended to lists. The string analysis functions, however, require more careful consideration.

A function like

```
find(s1, s2)
```

can be characterized as finding a consecutive sequence of the *elements* of `s1` in `s2`. There is an equivalent interpretation for lists, so that

```
find(a1, a2)
```

produces the position of sequence of the elements of `a1` in `a2`. For example,

```
find(["green", "apples"], ["three", "green", "apples"])
```

produces the value 2.

The interpretations of

```
find(s, a)
```

and

```
find(a, s)
```

are based on element-by-element comparisons. For example,

```
find("apples", ["green", "apples"])
```

produces 2, while

```
find(["green", "apples"], "is 'greenapples' one word?")
```

produces 5.

The analysis functions in Icon$_{\mathcal{E}}$ are, of course, generators and may produce sequences of values.

Several of Icon's string analysis functions depend on the characters that occur in a string without regard to their order. An example is

```
upto(c, s)
```

which produces the position in `s` at which any character in `c` occurs. The concept of a cset in Icon was introduced because of contexts such as this in which membership in a set of characters is important.

In generalizing string analysis functions to include lists, it is natural also to generalize csets to sets of arbitrary values. The notation

```
{x1, x2, ..., xn}
```

denotes an Icon$_{\mathcal{E}}$ operation that constructs a set consisting of the values `x1`, `x2`, ..., `xn` and is analogous to the list construction operation:

```
[x1, x2, ..., xn]
```

The use of braces to denote set construction conflicts with the use of braces to enclose a sequence of expressions in the case where there is only one expression in the sequence. This difficulty is ignored here, but is discussed in Section 5.1.

In order to integrate sets with the rest of Icon, two extensions are incorporated in Icon$_{\mathcal{E}}$. The function

```
set(a)
```

produces a set consisting of the elements of the list `a`, and

```
sort(c)
```

produces a sorted list of the elements of the set c. (The letter c is used here for sets — 'collections'.)

```
sort({"bad", "apples", 3})
```

produces the list

```
[3, "apples", "bad"]
```

To avoid the proliferation of types and operations, sets subsume csets in Icon$_{\mathcal{E}}$. Thus

```
set(s)
```

replaces

```
cset(s)
```

and produces a set consisting of the elements of the string s. For example,

```
set("apple")
```

is equivalent to

```
{"a", "p", "l", "e"}
```

and produces the same set as the literal 'apple'. The other cset operations of Icon can be incorporated in an upward-compatible fashion in Icon$_{\mathcal{E}}$, with the cset complement ~c being interpreted as &cset -- c.

The interpretation of the elements in a set depends on the context in which the set is used. Consider, for example,

```
upto(c, x)
```

If x is a string, the Icon interpretation of c applies. This raises the issue of how to handle values in c that are not one-character strings. There are several possibilities, two of which fit into the framework of Icon: to ignore such values or to treat them as errors. Icon$_{\mathcal{E}}$ takes the former approach. Thus, if

```
vset := {"x", "y", "z", "delta"}
```

the value delta in vset is ignored in

```
upto(c, s)
```

This is consistent with the interpretation that the element delta does not occur in any string. Therefore

```
upto(vset, "3+delta")
```

fails. On the other hand

```
upto(vset, [3, "+", "delta"])
```

produces the value 3.

3. Scanning

The disparity between the string and list processing facilities of Icon is most apparent in scanning. Nonetheless

s ? *expr*

may produce a value of any type — whatever *expr* produces. For example,

```
s ? {
   a := []
   while put(a, move(1))
   a
   }
```

produces a list of the one-character substrings of s. (The result produced by the compound expression is the result of the last expression, a.)

By making the scanning operation polymorphous, lists can be scanned.

a ? *expr*

The interpretation of &subject and &pos for lists is obvious.

If the subject is a list, the matching functions move(i) and tab(i) return a list of the elements between the previous and new values of &pos. For example,

```
a ? tab(3)
```

produces a list of the first three values in a, but fails if the size of a is less than 3. The string analysis functions have the same interpretation for scanning as they do in Icon: omitted trailing arguments default to &subject and &pos.

Consider the following scanning expression that produces a string consisting of the odd-numbered characters (elements) of a string s:

```
s ? {
   s1 := ""
   while s1 ||:= move(1) do
      move(1)
   s1
   }
```

A similar scanning operation can be formulated to produce a list of the odd-numbered elements of a list a:

```
a ? {
   a1 := []
   while a1 |||:= move(1) do
      move(1)
   a1
   }
```

Although string and list scanning have the same form in this example, the fact that list elements can be of any type introduces complexities and the need for additional features. For example, the following scanning operation might be formulated to write all the even-numbered elements of a list:

```
a ? while move(1) do write(move(1))
```

This formulation is erroneous, however. The expression move(1) does not produce the next element of a, but rather a one-element list that contains the next element. Of course, the value can be obtained by subscripting, as in

```
a ? while move(1) do write(move(1)!1)
```

but this is cumbersome. Since the need to access individual elements of a list occurs frequently in practice, the keyword &element is included in Icon$_{\mathcal{E}}$ as a synonym for

```
&subject!(&pos)
```

The scanning operation above then can be reformulated as

```
a ? while move(2)
   do write(&element)
```

Note that this formulation requires only one use of move for each evaluation of the loop.

The keyword &element applies equally well to string subjects, although its use in this case is less frequent, since strings usually are not processed on a character-by-character basis.

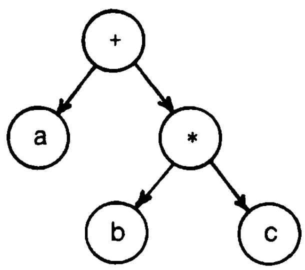
In Icon a list is a pointer (reference) to the structure that contains the elements of the list. Thus

```
a := [3, "green", "apples"]
```

assigns a pointer to a. Consequently, a list of lists can be used to represent a tree. For example,

```
tree := ["+", ["a", ["*", ["b"], ["c"]]]]
```

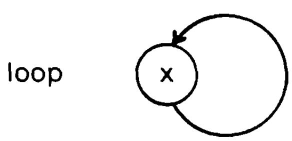
represents a tree in which the strings are node values and subtrees are represented by lists. This tree can be visualized as



Assignment in Icon copies pointers, not structures. Since any kind of value may occur in a list, structural loops can be constructed. A simple example is

```
loop := ["x"]
put(loop, loop)
```

which produces a loop that can be visualized as



Thus arbitrarily complicated structures can be built using lists.

To simplify the systematic processing of such structures, the keyword &visit is included in Icon$_{\mathcal{E}}$. If the subject is a list, &visit generates all its elements in preorder, but each one only once. If the subject is a string, however, &visit simply produces the subject. For example, if tree is a list representing a tree as given in the earlier example, then

```
tree ? every x := &visit do
   if type(x) == "string" then write(x)
```

writes all the node values in tree:

```
+
a
*
b
c
```

On the other hand, if loop is a list with a loop as given above,

```
loop ? every write(image(&visit))
```

writes

```
list(2)
"x"
```

4. Examples

4.1 String and List Representations of Trees

One situation in which both strings and lists are useful is in processing trees. On input, a tree is represented by a string. For processing, this string is converted to a list structure such as the one shown in Section 3. For output, this list structure is converted back into a string. Consequently, string-to-list and list-to-string conversion procedures are needed.

In string-to-list conversion, the subject of scanning is a string and the value produced is a list. A procedure is:

```
procedure ltree()
   local a
   if a := [tab(upto('('))] then {
      move(1)
      while put(a, tab(bal(', )')) ? ltree()) do
         move(1)
      }
   else a := [tab(0)]
   return a
end
```

This procedure is used in the form

```
a := (s ? ltree())
```

If s contains a left parenthesis, the substring up to that character is the node value and becomes the first value in the evolving list. The subtrees are then added by applying ltree recursively. If s does not contain a left parenthesis, it is a leaf node and is returned as a one-element list.

List-to-string conversion is similar, but slightly complicated by the fact that the string representation of a tree is not as homogeneous as the list representation. A procedure is:

```
procedure stree()
   local s
   s := &element
   move(1)
   if pos(0) then return s else s ||:= "("
   repeat {
      s ||:= (&element ? stree()) || ","
      move(1) | break
      }
   return s[1:-1] || ")"
end
```

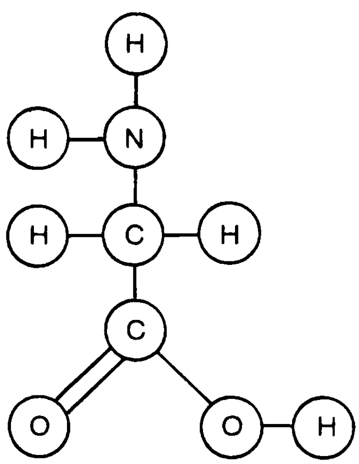
For example, the following program segment writes out all the subtrees of a:

```
a ? every x := &visit do
   if type(x) == "list" then write(x ? stree())
```

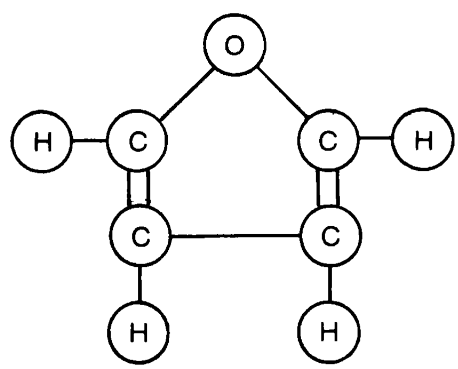
4.2 Graphs

String representations of trees and other structures without loops usually can be processed with relative ease, so that list scanning does not offer substantial advantages over string scanning. On the other hand, string representations of structures with loops often are awkward to process and list scanning can be used to advantage. An example follows.

Molecules can be represented by graphs in which the atoms are node and the bonds are edges. Two examples are



glycine



furan

In order to process such molecules, different atoms of the same element are distinguished by integer prefixes. String representations of these molecules, in which the edges are given in parentheses are:

```
glycine := "N1(H1,H2,C1)H1()H2()C1(C2,H4,H3)H3()H4()C2(O2)O1(C2,C2)O2(H5)H5()"
furan   := "O1(C1,C4)C1(C2,C2,H1)H1()C2(C3,H2)H2()C3(C4,C4,H3)H3()C4(H4)H4()"
```

The representation of molecules in the program consists of a list for each atom, in which the name of the atom is the first value and subsequent values represent the bonds by pointing to adjacent atoms. In addition, there is a table in which there is an entry value for each atom whose corresponding assigned value is the list for the atom. The following procedure, adapted from a general graph-construction procedure in [1], builds the list representation of a molecule:

```
procedure graph(s)
   local nodes, arcs, a, name
   nodes := table()
                                          # start with the name of the node
   s ? while nodes!(name := tab(upto('('))) := a := [name] do {
      move(1)
      arcs := tab(bal(')') + 1)
      arcs ? while put(a, "" ~== tab(upto(', )'))) do
         move(1)
      }
   every a := !nodes do           # change the names to their lists
      every i := 2 to *a do
         a!i := nodes!(a!i)
   return nodes
end
```

The following procedures scan for patterns in molecules, which are limited to alcohols, ethers, amines, nitriles, and molecules containing acyl groups and double and triple bonds between carbon atoms.

```
procedure analyze(mol)
   local node, A, neighbor
   every node := !mol do {
      node ? (
         A := atom(),
         neighbor := move(1),
         case A of {
            "C": {                          # check for double & triple bonds
               if (found(neighbor, 3)) then
                  write("molecule contains a triple bond")
               else if found(neighbor, 2) then
                  write("molecule contains a double bond")
               }
            "O": {                          # check for alcohols, ethers, & acyl
               if found(neighbor, 2) then
                  write("molecule contains an acyl group")
               else if ((neighbor!1) ?:= atom()) == "H" then
                  write("molecule contains an alcohol group")
               else write("molecule is an ether")
               }
            "N": {                          # check for amines & nitriles
               if (found(neighbor, 3)) then
                  write("molecule contains a nitrile group")
               else write("molecule contains an amine group")
               }
            }
         )
      }
end

procedure atom()
   local s
   s := ((move(1)!1) ? tab(upto('0123456789')))
   return s
end
```

```
procedure found(L, n)              # look for n occurrences of L
   local i
   i := 1
   every find(L) do i +:= 1
   if i = n then return
   else fail
end
```

For glycine, the output is

```
molecule contains an amine group
molecule contains an acyl group
```

For furan, the output is

```
molecule is an ether
molecule contains a double bond
molecule contains a double bond
```

This program can be extended to cover more functional groups and to include positional considerations (for example, alpha, para, ortho, and also rings). While analyze simply writes the configurations that are found, it could be extended to set a flag or to note the position in the molecule.

5. Implementation of a Portion of Icon$_{\mathcal{E}}$

A portion of Icon$_{\mathcal{E}}$ has been implemented. The majority of this implementation consists of a library of Icon procedures that overload the built-in functions and operations of Icon that have been changed in Icon$_{\mathcal{E}}$. The library procedures are described in Section 5.2. A preprocessor and the programmer-defined control operation extension to Icon [2] are used as well.

5.1 Preprocessing

The preprocessor translates Icon$_{\mathcal{E}}$ syntax into standard Icon syntax wherever possible. The preprocessor is based on the Icon translator [3], but emits translated program text instead of the usual intermediate code that is the input to the Icon linker. Since this preprocessor uses the same grammar as the Icon translator, except for specific changes made for Icon$_{\mathcal{E}}$, it is faithful to the rest of the Icon syntax. Where no direct translation is possible, the preprocessor produces calls of library procedures.

To provide polymorphous concatenation, the preprocessor performs the following translations:

```
s1 || s2     →    Cat(s1,s2)
a1 ||| a2    →    Cat(a1,a2)
s1 ||:= s2   →    s1 := Cat(s1,s2)
a1 |||:= a2  →    a1 := Cat(a1,a2)
s1 @ s2      →    s1 || s2
a1 % a2      →    a1 ||| a2
s1 @:= s2    →    s1 ||:= s2
a1 %:= a2    →    a1 |||:= a2
```

Thus both list and string concatenations are translated into calls of the support procedure Cat. The types of the arguments are, of course, presumed. The translation of the infix operators @ and % make the built-in concatenation operations available to Cat. The preempted transmission and remaindering operations are unavailable in this implementation of Icon$_{\mathcal{E}}$.

To provide the Icon$_{\mathcal{E}}$ distinction between subscripts and subsections, the preprocessor performs the following translations:

```
x!i          →    x[i]
x[i]         →    x[i:Poseq(i,x) + 1]
```

where Poseq is a library procedure that produces the positive equivalent of position i in x. The translation of x[i] to a range specification with upper and lower bounds prevents assignments to list subsection references in $\mathrm{Icon}_{\mathcal{E}}$. Such expressions cannot be used as abbreviations for list concatenation in this implementation. This problem cannot be handled by the preprocessor, since it is not possible to detect all subsection references syntactically. For example, a procedure call may return a subsection reference. The one-character restriction on assignment to s!i is not supported.

To provide for the set construction operation, the preprocessor performs the following translation:

```
{x1,...,x2}     →     set([x1,...,x2])
```

where set is a library procedure that constructs a set from a list. The ambiguity involving sets and compound expressions is resolved by the translation

```
{x}             →     {x}
```

Consequently, one-element sets cannot be constructed directly. However,

```
{x,x}
```

can be used to circumvent this problem.

To replace the built-in scanning control structure by a library procedure, the preprocessor provides the following translations:

```
x1 ? x2         →     Scan{x1,x2}
x1 ?:= x2       →     x1 := Scan{x1,x2}
```

where Scan is a library procedure. Since scanning is a control operation in which the arguments are not evaluated according to the rules for evaluation of procedure arguments, the braces are used in the call to indicate a programmer-defined control operation.

Finally, the preprocessor provides the following translations for keywords:

```
&pos            →     Pos
&subject        →     Subject
&element        →     Subject[Pos]
&visit          →     Visit()
```

Subject and Pos are global identifiers used by library procedures in place of the scanning keywords. Visit is a library procedure.

5.2 Library Procedures

The library procedures that are essential to $\mathrm{Icon}_{\mathcal{E}}$ are described in the following sections, along with examples of the full repertoire of the polymorphous string and list processing functions in $\mathrm{Icon}_{\mathcal{E}}$. The library procedures are written in $\mathrm{Icon}_{\mathcal{E}}$.

Polymorphous concatenation is performed by:

```
procedure Cat(x1, x2)
   return (string(x1) @ string(x2)) |
      if type(x1) == type(x2) == "list" then (x1 % x2) |
      stop("string or list expected")
end
```

Cat illustrates the need for type checking and selection of the operation to be performed accordingly. The expression

```
string(x1) @ string(x2)
```

uses explicit conversion to combine checking with the actual operation of string concatenation. The operators @ and % are translated by the preprocessor into || and |||, respectively.

Sets are implemented in $\mathrm{Icon}_{\mathcal{E}}$ by tables. Since a table is a set of pairs, each containing an entry value and an assigned value, a convenient method of representing sets is to have the entry and assigned values be the same in each pair. That is, each entry in the table is assigned its own value. A record type with one field is used to allow sets to be identified:

```
record Set(t)

procedure set(x)
   local t, y
   if type(x) == "Set" then return x
   t := table()
   every y := !x do
      t!y := y
   return Set(t)
end
```

If x is already a set, it is returned unchanged. Otherwise a table is constructed, each element of x is entered, and a record of type Set is returned.

The equivalence among values in Icon$_{\mathcal{E}}$ is essential to list processing. While two strings in Icon are equivalent if they have the same elements in the same order, this is not true of lists. For example, the comparison operation

```
["green", "apples"] === ["green", "apples"]
```

fails, since the two list values do not point to the same structure, even though the two structures have the same elements. In order for Icon$_{\mathcal{E}}$ to work properly, *equivalence comparison* is needed. In this operation, two values are equivalent if they are the same or if their elements are equivalent. A procedure for determining the equivalence of objects is:

```
procedure Eq(x, y, done)
   local i
   /done := table()                          # set up table on "outside" call
   if x === y then return y                  # equivalence test
   if string(x) == string(y) then return y   # equivalent strings
   if type(x | y) ~== "list" then fail
   if *x ~= *y then fail                     # check size first
   /done!x := table()                        # table for x if new
   if (done!x)!y === y then return y         # if already compared, return
   else (done!x)!y := y                      # add new value compared to x
   every i := 1 to *x do                     # now compare elements
      if not Eq(x!i, y!i, done) then fail
   return y
end
```

General value comparison is made first. If x1 and x2 are lists, a recursive comparison is made of the first elements and then of the rest of the lists. The table done, which is nonnull only when Eq calls itself recursively, serves to mark lists already processed. Each entry value in done is a table of values for which x has been compared. Thus, structures that contain loops are handled properly. A similar technique is used to extend the function image to produce a string representation of lists. In this representation, each list is labeled, and loops are treated by using only the labels.

Given the operation for determining the equivalence of two values, a test for set membership can be formulated quite compactly:

```
procedure Member(x1, x2)
   return Eq(!x2.t, x1)
end
```

The expression x2.t produces the table for the set and

```
!x2.t
```

generates the values in the set as necessary until one is found that is equivalent to x1 or until there are no more, in which case the membership test fails.

As mentioned in Section 5.1, scanning is a control operation and cannot be implemented directly by a procedure. Instead, the programmer-defined control operation extension to Icon is used. In this extension, procedures that are called with braces instead of parentheses are control procedures that get the arguments in the call as a list of co-expressions. The arguments therefore are not evaluated before the control procedure is invoked, but rather are evaluated by the control procedure as required, by activating the co-expressions. See [2] for the details of this mechanism.

In the case of scanning, the previous value of the subject and position must be saved before the arguments of scanning are evaluated. This allows the proper maintenance of these values for nested scanning. The global declaration and scanning procedure are:

```
global Subject, Pos

procedure Scan(a)                      # Scan{x1, x2}
   local tsubject, tcursor
   suspend 6(                          # 6th expression is x2
      tsubject := Subject,             # save Subject
      Subject <- |@a!1,                # get new Subject from x1
      tcursor := Pos,                  # save Pos
      Pos <- 1,                        # initial new Pos
      a!2 := ^a!2,                     # refresh x2 for each evaluation
      |@a!2,                           # activate x2
      Subject <- tsubject,             # restore Subject
      Pos <- tcursor                   # restore Pos
      )
end
```

Subject and Pos take the place of &subject and &pos in Icon. Subject is necessary, since a list value cannot be assigned to &subject. This procedure is somewhat arcane, since it is necessary to save and restore Subject and Pos whether or not a scanning operation succeeds or fails and also to maintain their proper values if the scanning expression is resumed for additional results. A full explanation of the interaction of the scanning operation with other expressions is given in [4].

The matching functions move(i) and tab(i) are implemented as described in [4]:

```
procedure move(i)
   suspend .Subject[.Pos:Pos <- Pos + i]
end

procedure tab(i)
   if i <= 0  then i := *Subject + i + 1
   suspend .Subject[.Pos:Pos <- i]
end
```

The analysis functions operate for strings and lists on the basis of the equivalence of elements as described in Section 5.2. A representative procedure is:

```
procedure upto(x1, x2, i, j)
   if /x2 := Subject then /i := Pos else /i := 1
   /j := 0
   j := Poseq(j, x2)
   x1 := set(x1)
   if (x2 := string(x2)) | (type(x2) == "list") then {
      suspend Member(x2!(k := i to j), x1) & j > k
      }
   else stop("string or list expected")
end
```

The first part of this procedure establishes the proper default values in case the trailing arguments are omitted. The first argument, **x1**, is then converted to a set. If **x2** is either a string or a list, it is processed element by element until an element of **x1** is found. Note that the procedure suspends with each position at which an element is found.

The generator **&visit**, which is translated into **Visit()** by the translator, provides a way of visiting all the elements of a list only once, even if there are structural loops. The procedure is:

```
procedure Visit(x, done)
   /x := Subject
   if string(x) then return x
   /done := table()
   if \done!x then fail
   done!x := x
   suspend x | Visit(!x, done)
end
```

The arguments that are passed to **Visit** are nonnull only when **Visit** calls itself recursively, as is the case with **done** in **Eq**.

6. Conclusions

The unification of list and string processing that is proposed here primarily is based on the development of polymorphous operations in a way that allows the two types to be treated similarly. In particular, the distinction between subsection and subscript references resolves a problem that is troublesome in the present version of Icon.

The extension of string analysis functions to lists, which is based on the concept of elements, leads to the replacement of csets by sets. The generalization of scanning leads to new primitives that are needed to process the more complicated structure of lists.

As illustrated in Section 5, most of Icon$_{\mathcal{E}}$ can be implemented by a preprocessor and a library of procedures that replace or extend the built-in ones of Icon. This implementation is usable and easy to modify. Although a complete implementation of Icon$_{\mathcal{E}}$ as a modification of the standard implementation of Icon is not a major project, more experience with the proposed facilities is needed before undertaking such a modification. In particular, other uses of list scanning may suggest the need for additional facilities or for approaches to unifying and simplifying the present ones.

Acknowledgements

Steve Wampler and Dave Hanson provided a number of helpful comments on the content of this report, as well as on its presentation.

References

1. Griswold, Ralph E. and Madge T. Griswold. *The Icon Programming Language.* Prentice-Hall, Inc. 1983.
2. Griswold, Ralph E. and Michael Novak. "Programmer-Defined Control Operations in Icon", *The Computer Journal.* In press.
3. Coutant, Cary A. and Stephen B. Wampler. *A Tour Through the C Implementation of Icon; Version 5.* Technical Report TR 81-11a, Department of Computer Science, The University of Arizona, Tucson, Arizona. 1981.
4. Griswold, Ralph E. *Models of String Pattern Matching.* Technical Report TR 81-6, Department of Computer Science, The University of Arizona. 1981.