Tables in Icon*

Ralph E. Griswold

TR 84-16

August 29, 1984

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants MCS81-01916 and DCR-8401831.

Tables in Icon

1. Introduction

Symbol tables are important components of many language processors, such as assemblers and compilers. In these applications, a symbol table serves to associate information such as location or type with a symbol such as an identifier. Various kinds of associative lookup facilities are found in AI languages and set-manipulation languages [1-6].

A general associative lookup facility has many potential applications in general-purpose high-level programming languages as well. Most high-level languages do not support such a facility, however. Instead, a programmer needing associative lookup must supply it and implement it using other data structuring facilities of the language. Although the techniques for implementing associative lookup are generally well known, they are relatively complicated and involve a significant programming effort. Furthermore, sophisticated approaches often are needed, since efficiency is a serious concern when the number of distinct "symbols" is large.

SNOBOL4 was the first general-purpose, high-level programming language to provide a table data type for associative lookup [7, 8]. Although SNOBOL4 is best known for its pattern-matching facilities, its table facility is widely used and is responsible for the use of SNOBOL4 for many applications.

Tables in the style of SNOBOL4 have been included in two subsequent languages, SL5 [9] and Icon [10]. Other languages that include similar types of tables include Awk [11], B [12], Bs [13], and *EZ* [14]. The full potential of tables is generally overlooked, however.

This report describes some of the uses of tables, including uses that are not obvious, discusses design considerations, and describes implementation techniques briefly. Tables as they exist in Icon are used for most of the material that follows. Tables in SNOBOL4 are similar to those in Icon and most of the programming techniques that are described are applicable to SNOBOL4 as well. This report assumes that the reader has a general knowledge of Icon, although tables are described in detail.

2. Tables in Icon

A table in Icon is a data aggregate containing elements that are pairs of values, called *elements*. Each element consists of an *entry value* and a corresponding *assigned value*. Subscripting a table with an entry value produces the corresponding assigned value. There is no restriction on the types of entry and assigned values of a table. In this sense, tables may be heterogeneous.

Tables are themselves data objects. A table is created by the function

table(*d*)

The value *d* is the *default assigned value* for elements in this table. For example,

```
t := table(0)
```

creates a table with the default assigned value 0 and assigns this table to t. A newly created table is empty — it contains no elements initially.

The subscripting expression

t[*expr*]

looks up the value of *expr* in t. The result of a subscripting expression is a variable. If a value is assigned to this variable, this value becomes the assigned value for the entry value *expr*. For example,

```
t["the"] := 3
```

creates a new table element with the entry value "the" and the assigned value 3. As noted above, any value can be used for an entry value or assigned value of a table element. For example,

```
t[1] := ["one", 1]
```

creates an element with an entry value that is an integer and an assigned value that is a list. No type conversion is performed for either entry values or assigned values:

```
t[1]
```

and

```
t["1"]
```

refer to different elements.

If the value of a table subscripting expression is used in a computation, the variable is dereferenced, producing the assigned value. For example,

```
write(t["the"])
```

writes 3.

Tables grow in size as values are assigned to new entry values. The size of a table, which is the number of distinct entry values to which values have been assigned, is denoted by *t. There is no way to remove an element from a table (see Sec. 4). Note that entry values are distinct but that assigned values need not be.

If a table is subscripted by an entry value that is not in the table, no new element is added unless an assignment is made to the subscripting expression. However, the value of the subscripting expression is the default assigned value. Thus if there is no element for the entry value "these" in the table t above,

```
write(t["these"])
```

writes 0.

The expression !t generates the elements of t. Each element is generated once, but in no predictable order. The result of !t is a variable — equivalent to a subscripting expression. For example,

```
every write(!t)
```

writes the assigned values of all the elements in t, while

```
every !t := 0
```

changes the assigned values of all the elements in t to 0.

The random element selection operator applies to tables also and produces a variable. For example,

```
?t := 0
```

changes the assigned value of a randomly selected element of t to 0.

The evaluation of a table subscripting expression involves a lookup process that is inherently time consuming (see Sec. 5). While there is little extra overhead in the two references to x in expressions such as

```
x := x + 1
```

there may be substantial overhead in

```
t[x] := t[x] + 1
```

Icon's augmented assignment operations avoid this extra overhead; the expression

```
t[x] +:= 1
```

should be used in preference to the one above.

There is no direct way of determining the entry values of the elements in a table. Yet in typical applications, the entry values are computed during program execution and are not known apart from their existence in the table. In order to determine the entry values of elements in a table, it is necessary to create an auxiliary structure. The function

```
sort(t, i)
```

creates a list corresponding to the table t (the table t is not modified in the process). This list contains an value for each element in t. Each value in the list is itself a list that contains two values, the entry and assigned values of the corresponding table element. If i is 1 (the default), the two-element lists occur in the sorted order of the entry values, while if i is 2, they are in the sorted order of the assigned values.

To write out the elements of a table in the sorted order of the entry values, a program segment such as the following can be used:

```
a := sort(t)
every pair := !a do
   write(pair[1], "\t", pair[2])
```

Assignment in Icon does not copy structures. In fact, a table value is a pointer to the actual structure, so that

```
t1 := t
```

assigns the same table to t1 that t points to. Sometimes it is useful to copy a table, which is done by copy(t). The result is a new table that is physically distinct from t, but that contains the same elements as t at the time of the copy.

3. Table Programming Techniques

While tables can be used in the same way as they are in compilers and assemblers, most applications of tables in high-level languages are more general and are more aptly characterized by a number of programming paradigms that are described in the following sections. The examples are given in terms of Icon; most of these examples apply, with minor modifications, to SNOBOL4.

3.1 Tabulation

One class of programming tasks is characterized by the analysis of an input file, during which significant strings are located and counted. Examples are tabulating the words in a file and tabulating operation codes in an assembly-language program. In the case of tabulating operation codes, the set of all possible codes in usually known, although it may be large and hence awkward to incorporate directly in a tabulation program. In the case of word tabulation, the set of all possible words that may occur in an input file usually is not known, and may not even be finite if the term "word" is interpreted liberally. Tables, however, can be used to accumulate the significant strings without having to enumerate them in the program. A typical program for counting words* is

```
procedure main()
   letter := &lcase ++ &ucase
   count := table(0)
   while text := read() do
      text ? while tab(upto(letter)) do       # find the beginning of a word
         count[tab(many(letter))] +:= 1       # put the word in the table
   result := sort(count)
   every pair := !result do
      write(left(pair[1], 15), right(pair[2], 4))
end
```

Note that the default assigned value for count is 0, corresponding to the initial count for all words. This

*The definition of a "word" in this program is a naive one. It can be made more sophisticated without changing the use of tables.

allows the use of augmented assignment to set the count of each newly entered word to 1. Augmented assignment not only avoids a second lookup in the table, but it also allows a transient result of string scanning to be used without assigning it to an auxiliary identifier.

3.2 Set Operations

Since table elements are pairs, a table can be used to represent a set by using a nondefault assigned value for every entry value that corresponds to a member in the set. Because Icon has operators for determining if an expression has a null value, it often is convenient to use the null value for the default assigned value when using a table to represent a set, as in

```
s := table()
```

If the null value is the default assigned value, any nonnull assigned value can be used for members in the set. For example,

```
s["a"] := 1
s["b"] := 1
```

can be used to add the members "a" and "b" to the set s. Similarly, if only the set of words in an input file is of interest, as opposed to how many times each occurs, the augmented assignment expression in the previous program can be replaced by

```
count[tab(many(letter))] := 1
```

and the output loop can be replaced by

```
every write((!result)[1])
```

If the default assigned value is null, testing for set membership is easy, as is illustrated by

if \s["the"] then $expr_1$ else $expr_2$

which evaluates $expr_1$ if "the" is in s but evaluates $expr_2$ otherwise.

For the general manipulation of sets, it is useful to have procedures that correspond to the usual set operations. For set membership, the following procedure can be used:

```
procedure in(s, x)
   return \s[x]
end
```

Note that if the test succeeds, the value returned is the assigned value. Otherwise, the call fails. If the entry value is desired, the procedure can be recast as

```
procedure in(s, x)
   return \s[x] & x
end
```

A member can be deleted from a set by assigning the default assigned value to it, as in

```
procedure delete(s, x)
   s[x] := &null
   return
end
```

While this procedure removes the member from the set, it does not remove the corresponding element from the table that is representing the set. Consequently, the size of a set is not, in general, the size of the table that represents it. A procedure to compute the size of a set is

```
procedure size(s)
   local i
   i := 0
   every \!s do i +:= 1
   return i
end
```

Set operations like union and intersection are even more awkward, since there is no direct way to determine the members of a set (the entry values in the corresponding table). One approach to set union is

```
procedure union(s1, s2)
   local s3, a, pair
   s3 := copy(s1)
   a := sort(s2)
   every pair := !a do
      if \pair[2] then s3[pair[1]] := 1
   return s3
end
```

A new set is created by copying s1 and then the members of s2 are added to it. Note that the members of s2 cannot just be added to s1 — that would change s1 instead of creating a new set. Observe that it is necessary to "filter out" null-valued table elements.

A more general approach is to provide a procedure that generates the set members:

```
procedure member(s)
   local a, pair
   a := sort(s)
   every pair := !a do
      if \pair[2] then suspend pair[1]
end
```

This procedure now can be used in other set operations. For example, set union becomes

```
procedure union(s1, s2)
   local s3
   s3 := copy(s1)
   every s3[member(s2)] := 1
   return s3
end
```

The awkwardness involved in producing the members of a set can be reduced by always making the assigned value of a table element that corresponds to a set member the same as its entry value, as in

```
s["the"] := "the"
```

Then the values generated by !s are the set elements and the procedure member is not needed. Thus set union becomes

```
procedure union(s1, s2)
   local s3, x
   s3 := copy(s1)
   every x := \!s2 do
      s3[x] := x
   return s3
end
```

If the assigned value is the same as the entry value for members of a set, this precludes the default assigned value from being a member of the set. It is easy to pick a default assigned value that does not create a problem in practice. Structures in Icon are created during program execution and are represented by pointers to the actual data aggregates. Each such pointer is distinct. For example,

```
a := []
```

creates an empty list and assigns it to a. This list is distinct from every other value in the program. Thus

```
setdefault := []
s := table(setdefault)
```

can be used to produce a table s to represent a set. Only the value of setdefault is precluded from set membership — hardly a restriction in practice, since this value is created solely for the purpose of providing a default assigned value.

If this approach is taken, the set operations must compare assigned values of elements to setdefault. For example,

```
s[x] ~=== setdefault
```

succeeds if x is a member of s.

In a program in which there are many sets, the default assigned value can be computed initially and used globally.

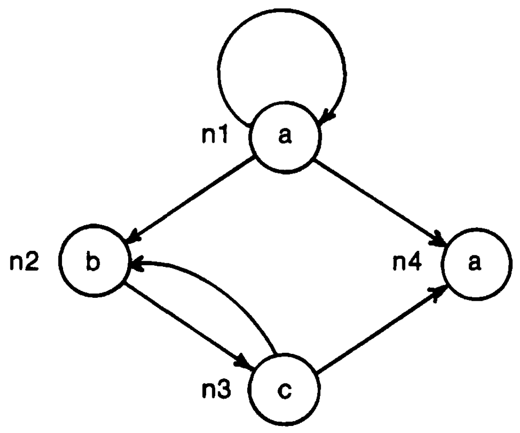
It is interesting to note that it is possible to determine the default assigned value of a table from the table itself, without any other information. By definition, the default assigned value is produced for an entry value that is not in the table. Since any newly created structure cannot be in the table previously, an expression such as

```
tdefault := t[[]]
```

assigns the default assigned value of t to tdefault.

3.3 Traversing Structures

Icon uses pointer semantics for structures; a list, table, or record value is a pointer to an aggregate of data for the structure. This allows complex data structures, such as graphs, to be represented easily. Consider, for example, a directed graph in which each node contains a value, as in



This graph can be represented by lists, using one list to represent each node. In each list, the first element is the value and the remaining elements represent arcs to the other nodes. For the graph above, the nodes might be

```
n1 := ["a",,,]
n2 := ["b",]
n3 := ["c",,]
n4 := ["a"]
```

The arcs then can be filled in by*

```
n1[2] := n1
n1[3] := n2
n1[4] := n4
n2[2] := n3
n3[2] := n2
n3[3] := n4
```

Consider the problem of traversing such a graph, printing the values of all nodes that are reachable from a given node. This process is inherently recursive, but as illustrated in the example above, there may be loops.

The usual method of handling loops in structures involves marking the nodes that have been processed and not processing a marked node. Garbage collection schemes often employ this approach, for example.

The problem with marking is that there must be some space in each node for a mark. Furthermore, in order to be able to traverse the structure again, the marks must be removed after the traversal. An alternative approach is to keep a list of all processed nodes and to check this list when a new node is encountered. With this approach it is not necessary to remove the marks when the transversal is done — the list simply can be discarded. Maintenance of such a list requires a lookup procedure; tables are ideal for this application, since they provide a built-in lookup mechanism.

Consider the problem of printing the values in all nodes reachable from node n1 (including the value of n1). The form might be

```
done := table()
nprint(n1, done)
```

with the procedure

```
procedure nprint(n,t)
   if \t[n] then return
   t[n] := 1
   write(n[1])                        # print the value of the current node
   every nprint(n[2 to *n], t)        # process the successor nodes
end
```

The table t contains (pointers to) the nodes that have been processed. When nprint(n, t) is called, there first is a test to see if n has already been processed (that is, if it is an entry value in t). If so, nprint returns immediately. If it is not, it is entered in t by assigning a nondefault value (1). Then the procedure nprint is called recursively on the nodes that n points to. Note that t is passed on in these recursive calls.

The need for an externally specified table (done, in the the example above) can be avoided by omitting the second argument in the initial call of nprint and initializing t on entry to nprint, as in

```
procedure nprint(n, t)
   /t := table()
   if \t[n] then return
        :
        :
end
```

In a call of the form nprint(n), t has the null value, /t succeeds, and t is assigned a new table. Note that all calls of nprint by itself have the form nprint(n, t), in which case t is passed along. This "dynamic" initialization is

*Arcs that go to previously created nodes can be filled in as new nodes are created. The more straightforward approach is used here for clarity.

used in favor of

```
        initial t := table()
```

so that nprint can be used repeatedly on different graphs.

This technique can be applied to a wide variety of similar problems. The key to its applicability is that list and table values are pointers. Thus, the table t can be passed as an argument and the lists can be used as entry values in it.

3.4 Graphs

If the structure of a graph is known *a priori* and is fixed, it can be represented by lists as shown in the preceding section. In many cases, however, the structure of a graph is not known in advance and may change during program execution.

In such cases, tables offer a natural alternative to lists. In this representation, each node is a table whose entry values are the tables to which the node has arcs. Thus, the nodes are like sets, and any non-default entry value can be used to indicate the presence of an arc. The value associated with a node can be represented by selecting a specific entry value, such as "value".

For the graph given in the preceding section, the table representation might be

```
        n1 := table()                          # first construct the nodes
        n2 := table()
        n3 := table()
        n4 := table()

        n1["value"] := "a"                     # enter the values
        n2["value"] := "b"
        n3["value"] := "c"
        n4["value"] := "a"

        n1[n1] := 1                            # then construct the arcs
        n1[n2] := 1
        n1[n4] := 1
        n2[n3] := 1
        n3[n4] := 1
        n3[n2] := 1
```

Note that weight or other values could be associated with the arcs by chosing appropriate assigned values.

In the case of a graph, it is also useful to have a table of all the nodes, as a kind of "root" to provide access to all nodes. (It is not possible, in general, to get from any one node to all the others via arcs.) Such a root node is given by

```
        graph := table()
        graph[n1] := 1
        graph[n2] := 1
        graph[n3] := 1
        graph[n4] := 1
```

Note that graph is a table of tables. Structures of this kind are useful in many similar situations.

3.5 Tuples

There are a number of situations in which it would be useful to have "multi-dimensional" tables with several entry values, rather than just one. Consider, for example, the representation of the seating capacities for classrooms on a college campus.

The buildings might have names such as Math, Engr, and so forth. To represent seating capacities, it would be useful to subscript a table by both the building name and the room number, as in

```
capacity["Math", 200] := 30
              :
capacity["Engr", 200] := 65
```

and in general to refer to seating capacities by expressions of the form

```
capacity[bldg, room]
```

This problem is so common that most programmers are familiar with the common idiom of concatenation of the two "entry values" to construct a single composite entry value, as in

```
capacity[bldg || room]
```

This solution is, nonetheless, somewhat contrived. Furthermore, there are situations in which the construction of a single, unambiguous entry value by the concatenation of multiple entry values is not practical.

A similar problem often arises in case expressions. For example, in symbolic algebra, the operation to be performed may depend on the types of the operands. Consider the addition of two values, which may be either integers or strings that represent symbolic quantities. It is tempting to try to use lists, as in

```
case [type(op1), type(op2)] of {
   ["integer", "integer"]  : op1 + op2
   ["string", "integer"]   : op1 || "+" || op2
   ["integer", "string"]   : op2 || "+" || op1
   ["string", "string"]    : op1 || "+" || op2
   }
```

This method does not work, however, since every newly constructed list in Icon is a distinct structure and the comparison operation used in case expressions (and in determining equivalent entry values in tables) succeeds only if two objects are the *same*, not just if they have equivalent values. Thus

```
["integer", "integer"] === ["integer", "integer"]
```

always fails, since the two lists are distinct.

In the absence of a comparison operation that determines if two lists have equivalent values, it would be useful to have a completely general method of constructing a unique object from a list of values. Such an object might be called a *tuple*, with the hypothetical syntax

$<expr_1, expr_2, \ldots, expr_n>$

For example, the comparison

```
<"integer", "integer"> === <"integer", "integer">
```

would succeed.

Tuples can be implemented by using a tree of tables, in which the interior nodes are tables and the leaves contain the values in the tuples. The root node table contains a table for each tuple size that has been entered, and thus divides tuples into classes by their length. When a tuple is entered, a path through the tree is traversed, based on the values in the tuple. If the traversal encounters a non-existent value, a new path is created. Eventually, the traversal arrives at a leaf node that either already corresponds to the tuple or has been constructed in the process.

In the absence of the hypothetical syntax for tuples given above, a tuple is created by calling a procedure whose argument is a list of the tuple values:

```
procedure tuple(tlist)
   local tb, i, e
   static tuptab
   initial tuptab := table()           # create the root node
   /tuptab[*tlist] := table()          # if there is no table for this size, make one
   tb := tuptab[*tlist]                # go to table for size of tuple
   i := 0                              # assign default value to i
   every i := 1 to *tlist-1 do {       # iterate through all but last value
      e := tlist[i]                    # ith value in tuple
      /tb[e] := table()                # if it is not in table, make a new one
      tb := tb[e]                      # go to table for that value
      }
   e := tlist[i + 1]                   # last value in tuple
   /tb[e] := copy(tlist)               # if it is new, enter copy of the list
   return tb[e]                        # return that copy as the value; it is unique
end
```

Thus a tuple is created by

tuple([$expr_1$, $expr_2$, ... $expr_n$])

so that

```
tuple(["integer", "integer"]) === tuple(["integer", "integer"])
```

succeeds.

Tuples can be used as unique table entry values, so that in the seating capacity example above,

```
capacity[tuple([bldg, room])]
```

can be used. Similarly, the case expression for symbolic addition can be written as

```
case tuple([type(op1), type(op2)]) of {
   tuple(["integer", "integer"])  : op1 + op2
   tuple(["string", "integer"])   : op1 || "+" || op2
   tuple(["integer", "string"])   : op2 || "+" || op1
   tuple(["string", "string"])    : op1 || "+" || op2
   }
```

4. A Design Retrospective

The design of tables in Icon evolved through the development of SNOBOL4 and SL5. The changes that were made during this evolution provide some insight into the role that tables have played in programming and the problems that have been encountered.

The default assigned value is always the empty string in SNOBOL4. Thus the programmer cannot select a default assigned value that is useful in distinguishing the properties of a table.

The conversion of a table to a structure that can be accessed by position also is somewhat different in SNOBOL4. In SNOBOL4 an array is constructed from a table by the expression*

```
a = SORT(t)
```

The result is an *n*-by-2 array with a row for each of the *n* elements of t that does not have an empty string as assigned value. Elements of t that do have the empty string as their assigned value are not included in the

*The function SORT(t) is not available in all dialects of SNOBOL4 The general method is

```
a = CONVERT(t, "ARRAY")
```

which produces an unsorted *n*-by-2 array

array.

The first column in the array contains the entry values and the second column contains the assigned values. The rows are sorted by the entry values, although some dialects of SNOBOL4 provide for other sorting orders.

Note that this is similar to the result of sorting a table in Icon, except that in SNOBOL4, table elements with empty-string assigned values always are "filtered out". This is a stronger interpretation of the default assigned value than Icon has.

Neither SNOBOL4 nor Icon has a satisfactory way to determine if a specific value is in a table. Referencing a table with an entry value for an element that is not in a table produces the default assigned value, but the default assigned value also can be assigned to an element that is in a table; hence this value is not sufficient in itself to determine membership.

Note that there are two concepts of "membership" of an entry value in a table: "logical" and "physical". As a matter of programming practice, an entry value usually is logically in a table only if its corresponding assigned value is not the default assigned value. However, explicitly assigning the default assigned value to an element does not physically remove the element. Specifically, such an assignment does not change the size of the table. Membership can determined by constructing an auxiliary structure (which imposes a different concept of membership in SNOBOL4 and Icon, as mentioned above), but that method is cumbersome and inefficient.

In Icon, table membership can be determined in a "destructive" way by first testing to see if the assigned value for an entry value is the default assigned value, and if it is, assigning the default value to it. If the size of the table increases as a result, the entry value was not in the table previously. However, it is entered in the table in the process, and there is no way to remove it (see the discussion below). Furthermore, this is a devious and complicated way to test membership.

In Version 2 of Icon [15], the table membership problem was handled somewhat differently and was motivated by the observation that tables often are used in a two-phase fashion. In the first, "insertion", phase, elements are added as the table is subscripted with new entry values and given corresponding non-default assigned values. In the second, "lookup", phase, the table is subscripted with arbitrary values, and the action taken depends whether or not the value is already in the table.

To accommodate this kind of usage, tables in Version 2 of Icon can be opened and closed. If a table is open (the initial state of a newly created table), new elements can be added and a reference to an entry value that is not in the table produces the empty string (as in SNOBOL4, there is no choice of default assigned value in Version 2 of Icon). Thus an open table in Version 2 is much like a table in Version 5. However, if a table is closed, using the function close(t), an attempt to subscript a table with an entry value that is not in the table fails. Thus membership can be tested easily. Furthermore, subscripting a closed table with an entry value that is in the table produces the corresponding assigned value, but not a variable: it is not possible to change an assigned value in a closed table.

This mechanism works well, provided that insertion and lookup naturally occur in distinct phases of program execution, but it is awkward and cumbersome if insertion and lookup are interleaved. Furthermore, this mechanism adds to the vocabulary of the language. After a period of use, this mechanism was abandoned in subsequent versions of Icon in favor of allowing the programmer to specify the default assigned value.

There is, of course, no conceptual problem with providing a function that tests for membership in a table. However, the need for such a function has not been sufficient to dictate its addition. Nevertheless, the absence of a reasonable way to test for membership in a table remains an aesthetic irritant.

A more serious problem is the lack of a method for removing an element from a table. If the default assigned value is used to distinguish logical membership in a table, an element can be logically removed by assigning the default assigned value to it. This does not remove the element physically, however, and the size of the table remains the same, as discussed below. If there are many such elements, the amount of memory they occupy may be a problem and the speed of lookup may be degraded.

Again, there is no conceptual problem with providing a function to remove an element from a table. Such a function certainly would make some programming tasks easier and more straightforward.

The difficultly of determining the set of all entry values in a table causes more trouble than determining if a specific entry value in in a table. The problem is that there may be no way to know any of the entry values without creating an auxiliary structure, as is illustrated by the word tabulation program given in Sec. 3.1.

The SNOBAT dialect of SNOBOL4 [16] provides a method of determining entry values by adding two functions: first(t), which produces a pointer to the first element of the table t, and next(p), which produces a pointer to the next element after p.

The natural approach to this problem in Icon would be to provide a generator that produces successive entry values. In fact, this would be more general than the current table element generator, !t, since the assigned values can be obtained from the entry values. If !t were redefined to generate the entry values in the table t, then the output portion of the word tabulation program in Sec. 3.1 could be rewritten as

```
every word := !count do
   write(left(word, 15),right(count[word], 4))
```

Furthermore, set membership could be determined, albeit inefficiently, by

```
x === !t
```

The question of generating the elements of a table, whether by subscripting expressions or entry values, raises the question of the order in which elements are produced. Because of the way that tables usually are implemented (see the next section), this order normally is unpredictable, although sorted orders can be produced from auxiliary copies. It is worth mentioning that the original implementation of SNOBOL4 provided a well-defined ordering — chronologically by the time of first use of an entry value as a subscript. Chronology is not a common feature of programming languages, but it has its uses. For example, in SNOBOL4 it is easy to produce a list of the words in a file in the order of their first occurrence.

The tuples example in Sec. 3.5 raises two interesting design questions. The obvious one is the issue of multi-dimensional tables. In terms of design, there is little problem with extending tables to allow multiple entry values. One question that would have to be decided is whether a given table would have a fixed number of entries, or whether it could be subscripted by different numbers of entry values at different times. The choice is primarily between the ease of implementation and generality.

In fact, the main impediment to the inclusion of multi-dimensional tables lies in implementation considerations. Furthermore, tuples, as a built-in language feature, offer more generality, since they can be used in a variety of contexts, such as case expressions. Note that tuples, in themselves, constitute a form of table.

The less obvious issue illustrated in the procedure that implements tuples at the source level is the problem of table creation. The paradigm

```
/t1[e] := table()
t2 := t1[e]
```

appears twice in this procedure and it occurs in most contexts where tables of tables are constructed.

EZ [14] solves this problem by automatically creating a table if an attempt is made to subscript a value that is not a table. Thus

```
t2 := t1[e]
```

automatically creates a table and assigns it to t1 if t1 is not table-valued. This rather unusual feature is consonant with the general design philosophy of *EZ*, in which almost any operation takes place automatically if it is possible. Awk also creates its associative arrays implicitly.

Icon, on the other hand, cannot create tables automatically, since the same syntax is used for subscripting strings, lists, and records, as well as tables. For example,

```
s[i]
```

references the ith character in the string s; assignment of a new table to s in such an expression would be disastrous.

Nonetheless, programming involving lists of tables and tables of tables would be substantially easier if tables could be created automatically in context. All that is needed is a unique syntax for table subscripting.

5. The Implementation of Tables

The implementation of tables presents a number of interesting problems. The most obvious concern is the efficiency of the lookup process, since the number of entry values in a table may be very large. In Icon (and SNOBOL4) the fact that any type of value can be used as an entry value (as opposed to restricting entry values to being strings) complicates the implementation. A less obvious, but nonetheless important problem is avoiding the creation of table elements for entry values that are referenced, but for which no corresponding assignment is made.

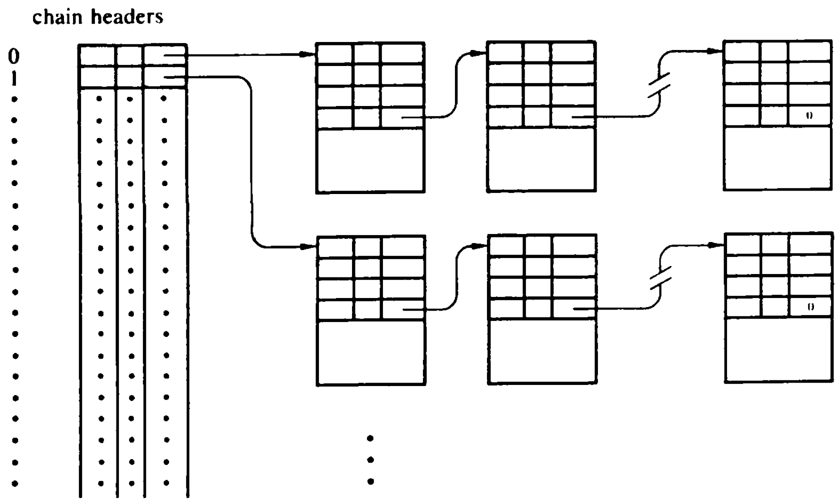
5.1 Data Structures for Representing Tables

In the original implementation of tables in SNOBOL4 [17], tables are represented using fixed-sized blocks in which elements are simply adjacent pairs of values. Table subscripting is done by examining the alternate values in the block, starting at the beginning. If the entry value is not found, a new pair is added in unused space at the end of the block. When space in the initial block is exhausted, a new block is linked onto the end, and so on. The table creation function in SNOBOL4 has two optional arguments that allow the user to specify the sizes of the initial block and any additional blocks, respectively. Considering how large tables get in some programs, it is surprising that the linear lookup in this implementation has not proved to be more of a problem than it has been.

An interesting byproduct of this implementation is that table elements are ordered chronologically. In fact, this order is specified as a feature of the language [7], although it is generally not supported in other implementations of SNOBOL4.

The need for sophisticated data structures and accessing techniques for associative lookup facilities was recognized early in the development of programming languages that supported such features [2]. The obvious approach is to hash the entry values. Simple hashing techniques are not adequate, however, because of the dynamic nature of tables and their unpredictable size. The current technique for implementing tables in Icon originated in the implementation of the main symbol table in the original implementation of SNOBOL4 [17].

In SNOBOL4, every non-empty string, including those constructed during program execution, is a potential identifier, so that identifiers can be created during program execution. Since identifiers can be created during program execution, SNOBOL4's symbol table is much like a table created by a programmer. Techniques for efficiently managing this main symbol table were a serious concern in the original implementation of SNOBOL4. The approach taken was to combine hashing with chaining. There are a fixed number of "hash bins", each of which is the header of a chain of elements. Each element contains, among other things, the string, a link to the next element in the bin, and an *order number*. When a new string is created, hash computations are performed to produce two numbers: a bin number and an order number, mentioned above. The bin number determines which bin the string goes in. Within a bin, elements are linked in the order of increasing order numbers. The symbol table in SNOBOL4 therefore has the following structure:



When a string is looked up in the table, a search is performed on the elements in the selected bin. If a point in the chain is reached at which the order number of the new string exceeds the order number of the current element, the string is not in the table and a new element for it is created and linked into the chain at that point. If an element is found with the same order number as the new string, there are two possibilities: (1) the string in the element is the same as the new string, or (2) there has been a "collision" in which two different strings have the same order number. The cases must be distinguished by comparing the two strings. Note that string comparison is necessary only if order numbers are the same.

Having several hash bins serves to divide strings into classes by a simple computation and to reduce the average length of the chains. The specification of the number of bins involves a trade-off between the space required for the bin headers and the average chain length. The symbol table for SNOBOL4 has 256 bins. Shorter chains improve the efficiency of the lookup process, of course, but increasing the number of bins is effective in this regard only to the extent that the hashing function that computes the bin number has the desired effect.

The SPITBOL 360 [18] and SITBOL [19] implementations of SNOBOL4 use a similar technique, combining hashing and chaining, although they do not use the order-number concept. In SITBOL, the default number of bins is 37, but a different number can be specified as an argument of the function that creates a table. Version 5.9 of Icon uses hash bins, chaining, and order numbers. The implementation of tables in Version 5.9 of Icon is described in detail in [20].

5.2 Different Types of Entry Values

Many programming languages restrict associative lookup to strings. As illustrated by the programming examples given earlier in this report, however, the utility of tables is considerably enhanced by allowing any type of entry value.

In languages like SNOBOL4 and Icon, it is easy to use any kind of value as an entry value, since the implementations of these languages represent all values by *descriptors* [17, 21]. Descriptors are the same size, regardless of the type of value that they represent. If the value is too large to be stored in a descriptor, a pointer to the value is stored in the descriptor instead. This is the case for strings, lists, tables, and records. Thus all values can be manipulated uniformly. In fact, in the original implementation of SNOBOL4, the bit pattern in a descriptor is a unique representation for the value, so that the lookup of an entry value in a table is particularly simple. In other implementations, some kinds of values (such as strings) require comparison of the values that are pointed to.

Hash computations are complicated by allowing non-string entry values. Techniques for hashing string values are well known [22]. Types like integers are relatively straightforward to hash — the residue modulo the number of bins can be used for the bin number, and the integer itself can be used for the order number. Other data types are more challenging. Icon, like SNOBOL4, not only has many data types, but some of them are comparatively esoteric.

For the purposes of designing hash functions, data types can be divided into two categories: those for which the data value itself contains information that can be used in hashing (for example, strings and integers), and those for which the data value provides little or no information that can be used in hashing (for example, records, and lists).

Records can be segregated by type, but for records of the same type there is nothing to use for hashing, since all the components of a record are subject to change. Lists present the same problem.

It should be noted that care is needed in devising hashing functions for structures. It is easy to mistakenly use an attribute of a structure for hashing that appears to be invariant, but in fact is not. For example, the size of a list in Icon may change and hence cannot be used for hashing. Similarly the address of a structure cannot be used in SNOBOL4 or Icon, since it may be changed as a result of relocation during garbage collection.

There are two alternatives here: (1) add some unique identification to such objects that can be used for hashing, or (2) ignore the problem and hash all such objects into the same bin with the same order number.

Unique identification might take the form of a date stamp or a serial number. Any added information of this kind has the drawback that it requires additional space in each structure. Ignoring the problem, on the other hand, results in a linear lookup for such objects.

Which alternative is the best depends somewhat on context. For example, in implementations where memory is at a premium, the addition of additional space to each structure might degrade performance or even be a limiting factor for some programs. In practice, subscripting tables with structures is not common. See Sec. 3.3, however. On the average, it may be more efficient to tolerate linear lookup for such cases than to burden *all* programs, most of which never subscript tables with structures, with the overhead of larger structures.

5.3 Element Creation

The original implementation of SNOBOL4 creates an element when a table is subscripted with a new entry value, whether or not a value is assigned to the subscripting expression. The effect of this is that tables grow in size, simply by being subscripted. This problem is particularly serious when a table is used to store data related to a small number of entry values, but in which many different entry values may be used as subscripts, just to check for these few relevant values [23].

Consider, for example, a hyphenation program in which there is a table that contains a small number of exceptions to the hyphenation algorithm. In this case, every word to be hyphenated is looked up in the table, although the number of relevant entry values is small. In the original implementation of SNOBOL4, every new entry value results in an added, and useless, element.

Methods of avoiding this problem were employed in subsequent implementations of SNOBOL4. The mechanism used in SITBOL is particularly elegant and has many other applications [24]. When a table is subscripted with a new entry value in SITBOL, instead of constructing a new table element and adding it to the table, a *trapped variable* is created and returned as the result of the subscripting expression. This trapped variable is a descriptor that points to a small block of data that contains a pointer to the table that was subscripted and the entry value. Trapped variables are distinguished from other variables by a single bit in their descriptors. When assignment is made to a trapped variable, the information in the data block that it points to is used to construct a new table element, which is added to the table. If the trapped variable is only dereferenced, however, the default assigned value is produced (the empty string in SNOBOL4), but no element is added to the table. Thus an element is added to a table only if an assignment is made to it. The mechanism is efficient, since it is only necessary for the assignment and dereferencing routines to check a single bit to determine whether or not the special processing is required.

6. Conclusions

Tables in SNOBOL4 originally were conceived of as a way to associate values with strings — to provide an associative lookup facility for strings. The "descriptor semantics" of SNOBOL4 led naturally to the ability to subscript a table with any kind of value. This capability continued in SL5 and Icon. The generality of subscript values separates languages like Bs [13] that only allow string entry values from languages like Awk[11] and EZ [14] that have the greater generality.

The examples given in this report illustrate the importance of general entry values. The feasibility of providing this generality depends, of course, on other language semantics and on the implementation. Having every language value uniformly represented by a fixed-sized descriptor makes generalized entry values trivial. It may be very hard to implement this feature in a language in which different kinds of values have different sizes. B provides a reasonable compromise between generality and implementation difficulties by allowing tables to have any kind of entry value, but requiring all the entry values for a specific table to be of the same type.

Tables in Icon represent extensive experience with the use of tables and many variations and experiments with the design of specific features of tables. Despite this, there are many unresolved problems — table membership, removing elements from tables, access to table elements when the entry values are not known, multi-dimensional tables, and automatic table creation.

Tables are appearing in new programming languages with greater frequency — B, Bs, and EZ are recent examples. This seems, therefore, to be an appropriate time to give attention to the development of a coherent model of tables that could be used in a variety of language frameworks.

Acknowledgements

The idea of tuples is due to Bill Mitchell, who developed the technique that is described in Sec. 3.5.

The original implementation of tables in Version 5 of Icon was done by Cary Coutant and Steve Wampler. Rob McConeghy subsequently added order numbers and improved the hashing algorithm.

Dave Hanson, Bill Mitchell, Janalee O'Bagy, and Steve Wampler provided a number of helpful suggestions on the presentation of the material in this report.

References

1. Newell, Allen. *Information Processing Language-V Reference Manual.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1961.
2. Feldman, Jerome A. and Pail D. Rovner. "An Algol-Based Associative Language", *Communications of the ACM*, Vol. 12, No. 8 (August 1969), pp. 439-449.
3. Reiser, John F. *Sail.* Stanford Artificial Intelligence Laboratory Memo AIM-289, Stanford, California. 1976.
4. McCarthy, et al. *Lisp 1.5 Programmer's Manual.* MIT Press, Cambridge, Massachusetts. 1962.
5. Cohen, Shimon. "The A-Table Data-Type for LISP Systems", *SIGPLAN Notices*, Vol. 14, No. 10 (October 1979), pp. 36-47.
6. Dewar, Robert B. K. *The SETL Programming Language.* Technical report, Courant Institute of Mathematical Sciences, New York University. 1979.
7. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language*, second edition. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1971.
8. Griswold, Ralph E. "SNOBOL", in *History of Programming Languages*, Richard L. Wexelblat, ed. 1981. p. 622.
9. Griswold, Ralph E. and John T. Korb. *A Catalog of Built-In SL5 Operators and Functions.* Technical Report S5LD3g, Department of Computer Science, The University of Arizona. 1977.
10. Griswold, Ralph E. and Madge T. Griswold. *The Icon Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1983.
11. Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger. *Awk — A Pattern Scanning and Processing Language (Second Edition).* Technical report, Bell Telephone Laboratories, Murray Hill, New Jersey. 1978.
12. Meertens, Lambert and Steven Pemberton. *Description of B.* Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. 1984.
13. *UNIX System User's Manual; Release 5.0.* "BS(1)", Bell Laboratories, Murray Hill, New Jersey. June 1982.
14. Fraser, Christopher and David R. Hanson. *The EZ Reference Manual.* Technical Report TR 84-1, Department of Computer Science, The University of Arizona. 1984.
15. Griswold, Ralph E. and David R. Hanson. *Reference Manual for Version 2 of Icon.* Technical Report TR 79-1, Department of Computer Science, The University of Arizona. 1979.
16. Silverston, Stephan M. "Extensions to SNOBOL4 in the SNOBAT Implementation", *SIGPLAN Notices*, Vol. 12, No. 9 (September 1977) pp. 77-84.
17. Griswold, Ralph E. *The Macro Implementation of SNOBOL4; A Case Study in Machine-Independent Software Development.* W. H. Freeman and Company. San Francisco, California. 1972.
18. Dewar, Robert B. K. *SPITBOL Version 1.0.* Technical report, Illinois Institute of Technology. 1971.
19. Gimpel, James F. *SITBOL.* Technical report, Bell Telephone Laboratories. 1972.
20. Griswold, Ralph E. *The Implementation of Data Structures in Icon.* Technical report, Department of Computer Science, The University of Arizona. In preparation.

21. Griswold, Ralph E., Robert K. McConeghy, and William H. Mitchell. *A Tour Through the C Implementation of Icon; Version 5.9.* Technical Report TR 84-11, Department of Computer Science, The University of Arizona. 1984.

22. Knuth, Donald E. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley, Reading, Massachusetts. 1973. pp. 506-549.

23. Santos, Paul J. Letter to the Editor, *SIGPLAN Notices*, Vol. 13, No. 9 (September 1978). pp. 23-24.

24. Hanson, David R. "Event Associations in SNOBOL4 for Program Debugging", *Software — Practice & Experience*, Vol. 8 (1978). pp. 115-129.