Data Structures in the Icon Programming Language

Ralph E. Griswold The University of Arizona

ABSTRACT: The Icon programming language provides a rich variety of data structures with sophisticated facilities: sets of arbitrary values, tables with associative lookup, lists with positional and deque access mechanisms, and records that extend the type repertoire of the language. Instances of these structures are created at run-time and grow and shrink as values are added to or removed from them. Storage management is automatic.

This paper describes these structures and their use in combination with Icon's goal-directed evaluation mechanism. Examples illustrate the use of pointer semantics and heterogeneity and how the natural geometrical interpretation of structures like trees and graphs in the problem domain is imaged in the programming domain.

The work described in this paper was supported by National Science Foundation Grants MCS-8101916, DCR-8401831, DCR-8502015, and CCR-8713690.

1. Introduction

The data structures that a programming language provides have an important influence on the kinds of problems for which the language is appropriate. They also determine how easy it is to program solutions to these problems in the language. These considerations are of central importance in rapid prototyping, artificial intelligence applications, and, in general, in nonnumerical computation.

Most "traditional" programming languages provide only arrays and records. Persons who program in these languages have to fabricate, at the source level, more complicated data structures that are needed for specific problems. For example, a C programmer who needs to look up symbols in a table must provide the look-up mechanism and handle storage allocation.

Some programming languages go well beyond these primitive and static data structures. Most notably, LISP [McCarthy et al. 1962; Steele 1984] in various forms provides a variety of data structures motivated largely by problems in artificial intelligence. APL generalizes the notion of arrays and provides many powerful and concise operations on them [Polivka & Pakin 1975]. SETL [Schwartz et al. 1986] brings the powerful mathematical notions of tuples and sets into the programming domain. The SNOBOL languages [Farber et al. 1964; Farber et al. 1966; Griswold et al. 1971] started with a focus on string processing, but then added sophisticated data structures, including tables with associative look-up [Griswold 1975; Gimpel 1976]. More recent languages with associative look-up mechanisms include Awk [Kernighan & Pike 1984], B [B], Rexx [Cowlishaw 1987], and ProLex [Fabisinski 1989].

The development of powerful data structuring capabilities in the SNOBOL languages continued in SL5 [Griswold 1976] and culminated in Icon [Griswold & Griswold 1983], which is a general-purpose programming language with powerful facilities for processing both strings [Griswold to appear] and structures. The data structures in SNOBOL4, SL5, and Icon were originally motivated by providing support for organizing and accessing string data in sophisticated ways. However, the data structuring capabilities of these languages have proved to be centrally important in their own right. In fact, one of the major applications of Icon is for the rapid prototyping of software systems [Fonorow 1988].

Icon has sets that may contain values of arbitrary types, tables with associative look-up, lists with deque access mechanisms, and records. This paper describes the essential aspects of these data structures. The material related to Icon that is essential to understanding its data structure facilities is covered briefly in the following sections. A more thorough understanding of Icon [Griswold & Griswold 1983] may be helpful in understanding details in some of the examples.

2. *Variables, Values, and Types*

Icon, unlike most programming languages, has no type declarations. Any variable may take on a value of any type at any time during program execution, as in

```
x := 1
  :
x := "Hello, world!"
```

On the other hand, while Icon has no compile-time type system in the ordinary sense, it has a strong run-time type system. The types of all operations are checked and operands are coerced, if necessary, to expected types. Thus,

```
sum := sum + read()
```

increments `sum` by the numeric value of a string read from a file, provided the string can be converted to a number. (Program execution terminates with an error message if the string is not

numeric in form.) Note that the conversion of a string to a numeric value is automatic; the programmer does not have to specify this.

In addition, it is possible to determine the type of any value. For example,

```
write(type(x))
```

writes the type of the value of `x`. One view of this is that variables are not typed in Icon, but values are.

While the original motivation for not providing type declarations in the SNOBOL languages that preceded Icon was to make it easier to write programs (at the expense of compile-time type checking), the ultimate importance of this approach is that structures – aggregates of values – need not be homogeneous with respect to type. The value of this language feature is discussed in Section 5.

Of course, untyped variables have a profound effect on the implementation of the language [Griswold & Griswold 1986; Walker 1988]. Since any variable can be assigned a value of any type, in some sense all values must be the same size. Like typical implementations of LISP, Icon uses a descriptor representation of values. Icon's descriptor contains type information, flags that are useful for identifying classes of values, and a representation of the value itself. If the value is small enough (as it is for integers), it is contained in the descriptor. If the value is too large to fit into the descriptor (as in the case of strings and structures), the descriptor contains a pointer to the value.

While these are implementation matters, they lead in a natural way to pointer semantics for structures. That is, the value for a structure such as a set or list is a pointer to the corresponding aggregate of values. In this sense, a structure value *is* a pointer. This allows arbitrarily large and complex aggregates to be treated as first-class values, and these values are small, independent of the size of the aggregates to which they point. For example, assignment just copies a pointer, not the aggregate of values to which it points.

The role of descriptors is seen in Icon's value-comparison operation,

```
x1 === x2
```

which compares two values of any type. The comparison is successful for structures if the *descriptors* for `x1` and `x2` are identical. Thus, two structures are the same for this operation if they point to the same aggregate of values. Similarly, the operation

```
x1 ~=== x2
```

succeeds if `x1` and `x2` are not identical.

There are many consequences of pointer semantics, both in how structures are viewed conceptually and in how they are manipulated. With pointer semantics, a programmer can model a problem-domain object like a directed graph in the programming domain in a natural way. These considerations are treated in more detail in Section 5.

3. *Expression Evaluation*

The way that expressions are evaluated in a programming language has a strong influence on how structures are used. In conventional programming languages like Pascal and C, the evaluation of an expression always produces exactly one result. Access to structures is straightforward, but imperative in nature and often tedious. IPL-V [Newell 1961], CLU [Liskov 1981], and SETL provide iterators that simplify processing of specific structures. In Prolog [Sterling & Shapiro 1986], on the other hand, expression evaluation is much more sophisticated (as well as largely hidden from the programmer) and data is searched for desired combinations of characteristics automatically. In this case, programming paradigms are more declarative and problem-oriented.

While Icon is superficially an imperative programming language, it has a sophisticated expression-evaluation mechanism: expressions that can generate sequences of results, goal-directed evaluation, and novel control structures. These aspects of expression evaluation in Icon make many operations on structures more concise and natural than they are in conventional imperative languages, while providing control that is often difficult to achieve in declarative programming languages. Furthermore, unlike

iterators in other programming languages, generators and goal-directed evaluation are general features of expression evaluation in Icon and apply to all kinds of computation.

Unlike most programming languages, the evaluation of an expression in Icon may succeed and produce a result, or it may fail and not produce a result. Success and failure, not Boolean values, drive control structures in Icon. Consider, for example, the traditional way of writing all the values in a list `L`:

```
i := 1
while i <= *L do {
   write(L[i])
   i := i + 1
   }
```

Here `*L` is the number of elements in the list `L` and `L[i]` references the `i`th element of `L`. In Icon, the expression `i <= *L` does not produce a Boolean value. Instead, it succeeds if the comparison succeeds but fails if the comparison fails. The `while` loop is controlled by this success or failure, not a Boolean value, although the appearance is the same. The utility of the success/failure concept comes from its broader applicability. An expression that succeeds and produces a useful computational value in one circumstance may fail in another circumstance. For example, `L[i]` fails if `i` is out of range. Thus, the loop above can be recast in a more compact form:

```
i := 1
while write(L[i]) do
   i := i + 1
```

Note that it is not necessary to know how big a list is in order to traverse it.

It is important to understand that failure is a normal aspect of expression evaluation in Icon, not the indication of an error. Failure occurs when a computation cannot be performed but the situation is not erroneous. This distinction between failure and error, which is not found in most programming languages, is centrally important to expression evaluation in Icon and how programs in Icon are written. The concept of failure allows conditional computations to be cast in a natural way and provides the context in which alternative computations can be specified.

Alternatives are provided by *generators*, which are expressions that are capable of producing more than one result. This is a natural concept in processing structures, which are aggregates of values. For example, `!L` generates the values in the list `L` in order from beginning to end.

A generator produces one value at a time, suspending evaluation every time it produces a value so that it can be resumed if another value is needed. If only one value is needed from a generator, only one is produced.

```
write(!L)
```

just writes the first value in `L`, even though `L` may contain many values.

The need for multiple values from generators arises from *goal-directed evaluation*, in which suspended generators are resumed if they are contained in an expression that otherwise would fail. This is illustrated by

```
if !L === 0 then write("zero found")
```

The first value produced by `!L` is compared to zero. If this comparison fails, `!L` is resumed to produce another value. This process continues until the comparison is successful, in which case the message is written, or until there are no more values for `!L`, in which case no message is written. The power of goal-directed evaluation is illustrated by

```
if !L1 === !L2 then write("common value")
```

which writes a message if `L1` and `L2` contain a value in common.

The *iteration* control structure,

every $expr_1$ do $expr_2$

resumes $expr_1$ repeatedly and evaluates $expr_2$ for each value $expr_1$ produces. For example,

```
every x := !L do
   write(x)
```

writes all the values in `L`. The `do` clause is optional, and the example above can be rewritten more concisely as

```
every write(!L)
```

Compare the conciseness of this form with the method of performing this computation using a `while` loop. The use of iteration and goal-directed evaluation in combination is shown by

```
every write(!L1 === !L2)
```

which writes all of the values common to `L1` and `L2` (a comparison operation returns its right operand if it succeeds). It should be noted that the expression-evaluation mechanism of Icon does not improve algorithmic efficiency for such operations (here there are `*L1 * *L2` comparisons). Instead, it provides a natural formulation that is concise and demonstrably correct.

Icon has several generators. For example,

```
i to j
```

generates the integers from `i` to `j`. One generator of general usefulness is the *alternation* control structure

$expr_1$ | $expr_2$

which generates the values of $expr_1$ followed by the values of $expr_2$. Thus,

```
!L === (0 | -1)
```

succeeds if `L` contains a value that is 0 or -1.

Programmer-defined procedures also can be generators, suspending (rather than returning) so that they can be resumed to produce additional values. For example, the following procedure generates the values in the list `L` that are not equal to `x`:

```
procedure cull(L,x)
   every y := !L do
      if x ~=== y then suspend y
   fail
end
```

The `fail` control structure at the end of the procedure returns, but indicates that no result is produced. The `fail` is optional here, since flowing off the end of a procedure body has the same effect.

The control structure `suspend`, like `every`, iterates over all the values generated by its argument. Consequently, the procedure above can be written more compactly as

```
procedure cull(L,x)
   suspend x ~=== !L
end
```

4. *The Basics of Icon Data Structures*

Different kinds of structures are provided to allow aggregates of values to be organized and accessed in different ways. Sets allow values to be grouped according to some common property. Lists allow values to be organized sequentially and to be accessed by position. Lists also can be accessed as stacks and queues. Tables provide associative look-up in which keys select corresponding values. Finally, records provide fixed organizations in which values are referenced by name.

The following sections describe the basic properties of these data structures as they are cast in Icon.

4.1 Sets

A set is an unordered collection of distinct values with no structure imposed on the values beyond their membership in the set. An empty set with no values is created by `set()`. For example,

```
words := set()
```

assigns a new, empty set to `words`.

A value is added to a set by `insert(S,x)`. For example,

```
insert(words,"the")
```

adds the string `"the"` to `words`. Since a set is a collection of distinct values, adding a value to a set that already contains the value has no effect. A value is deleted from a set by `delete(S,x)`. If the value is not in the set, this operation has no effect (it does not fail). Membership in a set is tested by `member(S,x)` which succeeds if `x` is in `S` but fails otherwise. The size of a set, which

is the number of values in it, is given by `*S`. The size of an empty set is 0. The size of a set increases and decreases as values are added to it and deleted from it.

In addition to these functions related to set membership, there are three operations on sets as a whole:

`S1 ++ S2`	union
`S1 ** S2`	intersection
`S1 -- S2`	difference

In each case, the result is a *new* set; `S1` and `S2` are not affected. For example,

```
commonwords := words1 ** words2
```

assigns a new set to `commonwords` that consists of the values that are in both `words1` and `words2`.

The operation `!S` generates the values in `S`. Since there is no inherent order for the values in a set, the order in which they are generated is unpredictable. The operation `?S` produces a randomly-selected value in `S`. For example,

```
delete(S,?S)
```

deletes a randomly selected value from `S`.

The values in a set need not all be of the same type. For example, the result of

```
one := set()
insert(one,1)
insert(one,1.0)
insert(one,"one")
```

is a set with three members, all of different types.

4.2 Lists

Many problems require access to an ordered collection of values. Icon provides lists for this purpose. Icon lists are one dimensional arrays (vectors) with an origin of 1.

Lists can be constructed in several ways. The values in the list can be given explicitly, as in

```
one := [1,1.0,"one"]
```

which assigns a list of three values to `one`. If the values in a list are not known when the list is created or if the list is large, the function `list(n,x)` can be used. It produces a list of `n` values, all of which are `x`. Both `[]` and `list(0)` produce empty lists.

Unlike a set, the values in a list are ordered and can be referenced by position. For example, the value of `one[2]` is 1.0. As mentioned earlier, an out-of-bounds list reference, such as `one[4]`, fails, and `!L` generates the values in `L` in order. As for a set, `?L` produces a randomly-chosen value in `L` and `*L` produces the size of `L`.

Since the values in a list are ordered, specific list values can be changed, as in

```
one[2] := 2.0
```

In other words, `L[i]` is a variable, as are `!L` and `?L`. For example,

```
every !L := 0
```

changes every value in `L` to 0. Note that it is not necessary to know the size of `L` to perform this operation.

Two other operations on lists are concatenation,

```
L1 ||| L2
```

and sectioning,

```
L[i:j]
```

Both of these operations produce new lists.

Since a list is a sequence of values, it is natural to access it by position. There are two other commonly-used structures that consist of sequences of values: stacks and queues. Rather than provide two additional structure types (or require programmers to model them with fixed-sized lists), Icon provides stack and queue (deque) access mechanisms for lists.

The following functions add or remove elements from the ends of lists:

`push(L,x)`	prepend `x` to the left end of `L`
`put(L,x)`	append `x` to the right end of `L`
`pop(L)`	remove and return the leftmost value in `L`

`get(L)`	remove and return the leftmost value in `L`
`pull(L)`	remove and return the rightmost value in `L`

With `push` and `put`, a list grows automatically. There is no limit to the size of a list except the amount of available memory. With `pop`, `get`, and `pull`, a list shrinks. These functions fail is the list is empty. Note that `pop` and `get` are synonymous.

While deque access functions are provided for manipulating lists as stacks and queues, they also are useful for other purposes, such as building lists whose sizes are not known in advance. Suppose, for example, that a list of all lines of an input file is needed. In most programming languages, a fixed-size array would be allocated. Depending on the number of lines in the input file, part of the array would be wasted, or it might overflow. In Icon, all that is needed is

```
lines := []
while put(lines,read())
```

It might seem that positional and deque access mechanisms are in fundamental conflict and likely to lead to programming errors. In practice, lists are used in either one mode or the other, or the mode changes but remains fixed during a different phases of program execution. For example, during the initial phase of program execution, a list such as `lines` above may be constructed using the queue access function. Once built, this list may be accessed in another phase of program execution strictly in a positional fashion. Thus, the fusion of positional and deque access mechanisms provides flexibility is manipulating sequences of values from viewpoints that may change during the course of program execution.

4.3 Tables

There are many situations in which access to an aggregate of values needs to by key rather than by position. An example is tabulating the words in a file, where each word has an associated count. Icon provides tables for such purposes. Tables are created like sets, as in

```
words := table()
```

which assigns an empty table to `words`. Tables are subscripted like lists, except the subscripts (keys) can be of any type. For example,

```
words["the"] := 1
```

associates the value 1 with the key `"the"` in `words`. If there is not already a value for the key `"the"` in the table, the size of `words` increases by one, reflecting a new key and value.

A table has a default value, which is given as the argument when it is created. For example,

```
words := table(0)
```

provides the default value 0 for `words`. The default value is provided for keys to which no other value has been assigned. For example,

```
write(words["automaton"])
```

writes 0 if no other value has been assigned for the key `"automaton"`.

Conceptually a table is a set of key/value pairs that constitutes a many-to-one mapping, that is, a function. The domain and range of a table include all possible Icon values. The default value serves to make the function complete, mapping all keys that have not been assigned values to the default value.

The operations on other structures apply to tables also. For example, `*T` is the size of `T` – the number of keys for which values have been assigned.

4.4 Records

Records in Icon are similar to those in many other programming languages with the notable exception that records extend Icon's type repertoire.

Records types are declared, as in

```
record complex(r,i)
```

which defines `complex` to be a record type with fields `r` and `i`. Records of type `complex` are created by a function of the same name. For example,

```
origin := complex(10.0,0.0)
```

creates a new `complex` record and assigns it to `origin`.

Fields are referenced using the infix dot operator. For example,

```
write(origin.r)
```

writes 10.0, the value of the `r` field of `origin`, while

```
origin.i := 20.0
```

changes the `i` field of `origin` to 20.0.

As mentioned above, a record declaration adds a new type to Icon's repertoire. Although such types are really subtypes of a general record type, they are on a par with other types, such as `list` and `table`, as far as type determination is concerned. For example,

```
write(type(origin))
```

writes `complex`. Thus, for example, complex arithmetic can be added to Icon in the form of procedures that perform different arithmetic operations, depending on the types of the operands.

Several records may have the same field name, as in

```
record verb(value,count)
record noun(value,count)
       ⋮
```

Such field names need not be in the same order for all records to which they apply.

In addition, record type names can be the same as built-in type names. An example is

```
record list(car,cdr)
```

Different types with the same name are differentiated internally, but they are indistinguishable via the `type` function.

5. *Pointer Semantics*

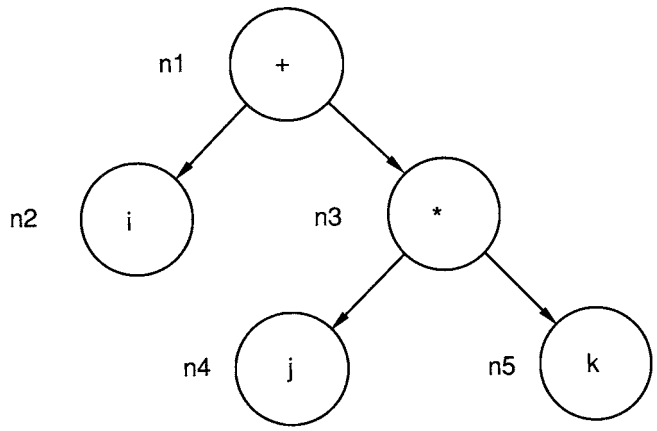
As mentioned earlier, pointer semantics provide much of the power of data structures in Icon. Using pointers, physical structures in the problem domain can be modeled directly in the programming domain. A few examples follow.

5.1 Trees

Consider binary trees composed of records, such as

```
record node(value,left,right)
```

This declaration defines a `node` data type with three fields. The `value` field provides a place for a value associated with a node, while `left` and `right` hold pointers to left and right subtrees (nodes), respectively. For example, the expression `i + j * k` might be represented as



This tree could be constructed node by node, as in

```
n4 := node("j")
n5 := node("k")
n3 := node("*",n4,n5)
n2 := node("i")
n1 := node("+",n2,n3)
```

In general, of course, such a tree would be constructed by parsing a string and building corresponding nodes in the process.

Where no field value is specified, the default value is null (similar to *nil* in other programming languages). This corresponds to the absence of a pointer to a node. The operation

```
\x
```

succeeds if `x` is non-null (for example, a pointer).

Generators provide a natural way of traversing such a structure:

```
procedure traverse(T)
   suspend T | traverse(\T.left | \T.right)
end
```

The use of `traverse` is illustrated by

```
every write(traverse(T).value)
```

which writes the values of all nodes in the binary tree `T`.

Lists can be used for general trees that are not limited to an out-degree of two. For example, nodes can be represented by lists in which, by convention, the first element holds a value and the remaining elements are arcs (pointers) to subtrees. Note that heterogeneity is essential to this method of representing trees. A more structured approach is to use a record such as

```
record node(value,arcs)
```

where the first field holds a value as before and the second field is a list of arcs (pointers) to subtrees. Again, heterogeneity is important.

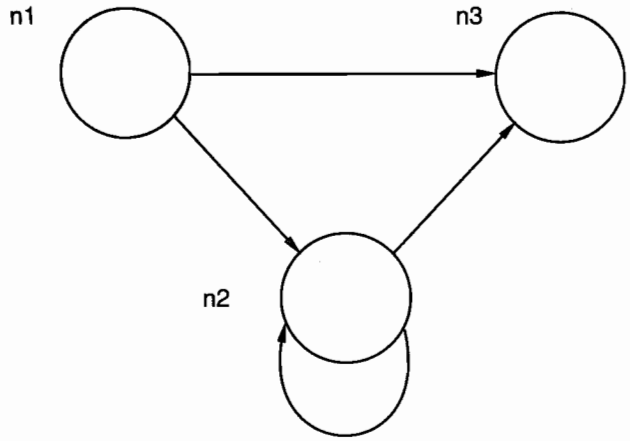
Other kinds of structures can be handled in a similar way. For example, the representations above can be used for dags as well as trees; it is simply a matter of arranging pointers to represent the desired structure.

Note that in all cases, the values associated with nodes need not be strings. They could, for example, be structures.

5.2 Graphs

While directed cyclic graphs can be represented by lists or records, there is a more general and interesting way of representing graphs. This method is based on the observation that with pointer semantics, a set is a pointer to a collection of objects that are its

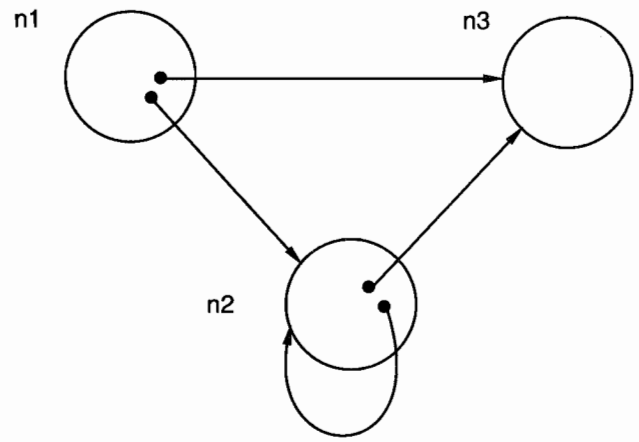
members. But a node in a graph has pointers (arcs) to other nodes. Thus, a node in a graph can be represented by a set that contains the nodes (sets) that the node points to. For example, the graph



can be represented by

```
n1 := set()
n2 := set()
n3 := set()
insert(n1,n2)
insert(n1,n3)
insert(n2,n3)
insert(n2,n2)
insert(n1,n3)
```

The corresponding Icon data structure is



where

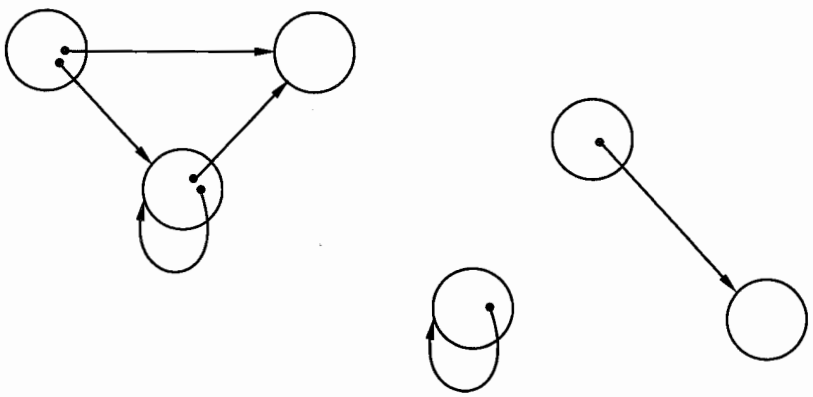
indicates the member of a set that is a (pointer) to a set. Thus, there is a simple geometrical transformation that illuminates the relationship between a graph and its representation as a sets whose members are sets.

Many graph operations are easy to perform on this representation. For example, the following procedures form the transitive closure of a graph starting at node `n`.

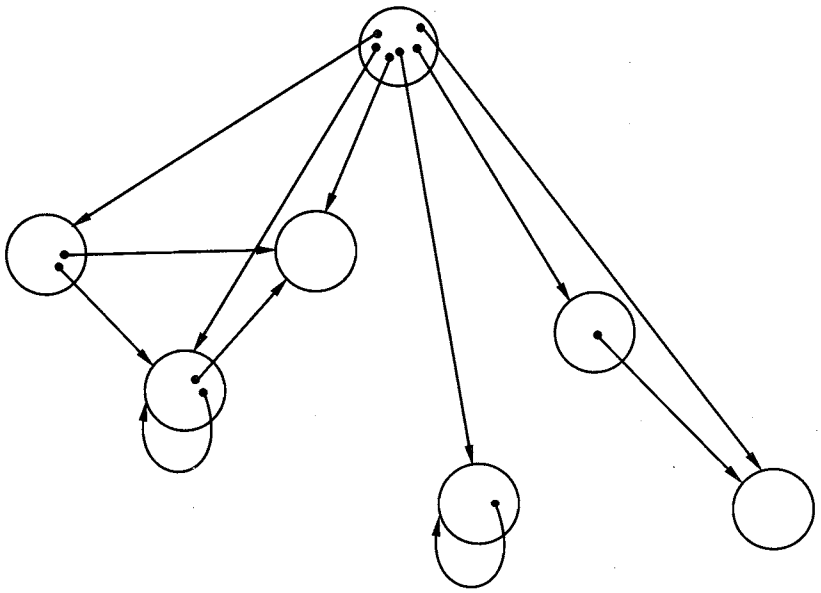
```
procedure closure(n)
   S := set()
   insert(S,n)              # start with node itself
   return accumulate(S,n)
end

procedure accumulate(S,n)
   every n1 := !n do  # process nodes reachable from n
      if member(S,n1) then
         next               # skip those already found
      else {
         insert(S,n1)       # add new node
         accumulate(S,n1)   # recurse with new node
         }
   return S
end
```

Note that the set containing the closure also can be interpreted as a graph node with arcs to all the nodes in the closure.

In handling graphs in general, where not all nodes are reachable from any one node and there may be disconnected subgraphs, an additional structure is necessary. Consider, for example:



As for forests, a list of nodes may be useful. Geometrically, this is a tree of (graph) nodes: If a set is used in place of a tree, the geometrical interpretation is a set of nodes. This, of course, is just another graph:



5.3 Labelings

With structures in general (and graphs in particular), labeling often is needed to keep track of nodes within a program and to relate them to external data. For example, labels in previous diagrams are used informally to identify components of structures, and correspond to variables in the code used to construct them, as in

```
n1 := set()
```

String labels may be needed outside the program (for example, to write out results of processing structures). Tables provide a natural way to associate string labels with structures. For the example above, this might take the form

```
Node := table()
Node["n1"] := n1
   :
```

Thus, the node labeled `n1` is obtained by

```
Node["n1"]
```

and so on.

The converse mapping is needed more frequently. This can be accomplished using a table whose keys are nodes and whose labels are the corresponding values:

```
Label := table()
Label[n1] := "n1"
    :
```

Note that the keys in `Node` are strings, while the keys in `Label` are sets. Since the keys in a table need not be homogeneous, both kinds of keys can be used in a single table:

```
Graph := table()
Graph["n1"] := n1
Graph[n1] := "n1"
    :
```

Such a "two-way" table, which contains both label-to-structure and structure-to-label relationships, combines the information in a single structure. In this structure, looking up a node produces its label, while looking up a label produces the corresponding structure. The advantage of using this technique is that the top-level characterization of a graph is contained in a single structure. Again, heterogeneity allows a useful programming technique.

6. *Combinations of Data Structures*

In most nonnumerical problems, there are several kinds of data that need to be accessed in different ways. The data structuring repertoire of a programming language determines how this data is represented, how it is accessed, and what operations have to be provided in addition to the built-in ones. Consider, as an example, a program that reads a context-free grammar and produces randomly selected sentences from the corresponding language.

Data structures are centrally important in the design of such a program. Assuming that the size and details of the grammar are not known in advance, the structures for representing it must be

constructed during program execution and must be flexible enough to handle a wide variety of possible program input.

There are two main considerations in designing the necessary structures: how sentences are generated from them and how they can be constructed from the program input.

Suppose the grammar is in a BNF form with each nonterminal symbol defined by alternative definitions composed of sequences of terminal and nonterminal symbols. An example is:

```
<element>::=<variable>|(<expression>)
<expression>::=<term>|<term><addop><expression>
<term>::=<element>|<element><mpyop><term>
<addop>::=+|-
      :
```

A sentence for a specified nonterminal symbol can be produced by selecting one of its alternative definitions at random and processing each symbol in this definition (in order from left to right is convenient for programming purposes). For example, if the specified nonterminal is `<element>` and the second alternative is picked, the symbols to be processed are `(`, `<expression>`, and `)`.

A terminal symbol, such as a parenthesis, contributes to the sentence being generated, while a nonterminal symbol, such as `<expression>`, is replaced by a randomly chosen definition for it as above.

The process of producing a sentence is stack-based. The top symbol is popped. If it is a terminal symbol, it is appended to the evolving sentence. If it is a nonterminal symbol, the symbols from a randomly chosen definition for it are pushed.

A natural way to distinguish between terminal and nonterminal symbols is by type. In Icon, terminal symbols are naturally treated as strings, while nonterminal symbols can be represented by a defined type:

```
record nonterminal(name)
```

so that `<expression>` is represented by

```
nonterminal("expression")
```

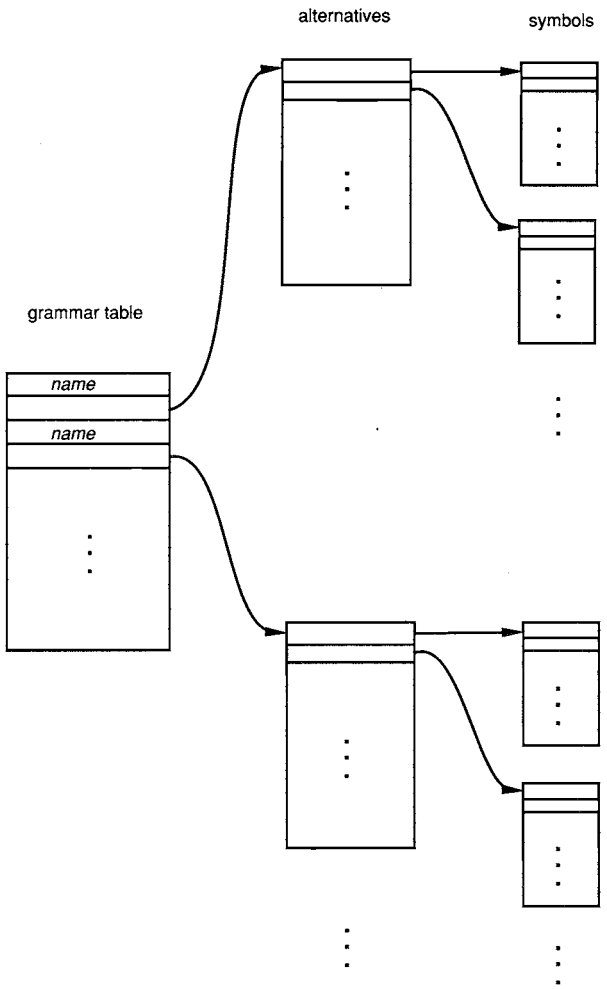
The sequence of symbols in an alternative is represented by a list. For example, the list for the second alternative for `<element>` is

```
[ "(", nonterminal("expression"), ")" ]
```

Note that such lists are heterogeneous, consisting of values of type `string` and `nonterminal`.

In a similar fashion, alternatives are lists – lists of symbol-sequence lists. Thus, the grammar is represented by a two-tier list of lists. Note that selecting an alternative at random is trivial.

Finally, it is necessary to get from the name of a nonterminal to its corresponding structure. This can be done with a table whose keys are the nonterminal names and whose values are the corresponding lists of lists. The overall structure for a grammar therefore has the form



Suppose the structure for a grammar has been built and is in the table `grammar`. A procedure to generate sentences for the nonterminal `goal` is:

```
procedure sentences(goal)
   local pending, sent, symbol
   repeat {
                        # initial condition
      pending := [nonterminal(goal)]
      sent := ""
      while symbol := pop(pending) do
                        # pop the top symbol
         if type(symbol) == "string" then
            sent := sent || symbol
                        # concatenate terminal symbol
         else           # push symbols for nonterminal
            pending := ?grammar[symbol.name] ||| pending
      suspend sent      # produce the completed sentence
      }                 # go around again
end
```

Here `""` is the zero-length, empty string and `||` is string concatenation. Note that list concatenation is used to prepend all the symbols for the randomly selected alternative to the list of pending symbols. This is equivalent to pushing them one by one, but it is simpler and also faster.

This leaves the problem on constructing the structures for the grammar. Since the sizes of the lists of alternatives and symbols are not known in advance, it is convenient to build them as queues, even though they are accessed by position later. The choice of queues rather than stacks is irrelevant for the lists of alternatives, since they are only accessed by random position, but is necessary for the lists of symbols so that they are in the correct order for the list concatenation in the sentence-generation procedure.

The procedures for constructing the grammar are naturally phrased in terms of generators:

```
global grammar

procedure main()
   grammar := table()
   every grammar[name()] := definitions()
      :
end
```

```
procedure definitions()
   deflist := []
   every alternative() do
      put(deflist,symbols())
   return deflist
end

procedure symbols()
   symlist := []
   every put(symlist,symbol())
   return symlist
end
```

The procedures `name`, `alternatives`, and `symbol` read the input, analyze the definitions, and generate the nonterminal names, definitions, and symbols, respectively. These procedures use Icon's high-level string scanning facility, which is beyond the scope of this paper. They are comparable in size to the procedures above for constructing the lists.

7. Conclusion

Summary

Many of the aspects of structures in Icon are not original to it. Structures are first-class values in many programming languages. The run-time creation of structures whose sizes are not known at compile time also is supported by several programming languages. Pointer semantics is as old as LISP. Associative lookup dates back to SNOBOL4, and a general treatment of sets was pioneered by SETL.

Icon's judicious combination of these features, coupled with its expression-evaluation mechanism, give its data structuring facilities their power:

- The fusion of stacks, queues, and vectors supports different kinds of access to sequences of values in a coherent framework.
- The uniform representation of values and the use of pointer semantics provides heterogeneous structures, which in turn allow structures of different kinds to be used in combination.

- Generators provide a concise method of accessing all the elements of a structure. Since generators are a general aspect of expression evaluation, not just idiosyncratic to structures, processing structures fits naturally with other kinds of computation.
- The important concept of failure as distinct from error, combined with goal-directed evaluation, allows concise and natural formulation of many kinds of operations on structures.

Status

The current version of Icon, Version 7.5, was originally developed under UNIX and has been successfully installed on over 60 different kinds of UNIX systems, ranging from VAXes to the NeXT. There also are implementations of Icon for the Amiga, the Atari ST, the Macintosh, MS-DOS, MVS, VM/CMS, and VMS.

All of these implementations are in the public domain and are available from the Icon Project, Department of Computer Science, The University of Arizona, Tucson, AZ 85721.

Acknowledgements

The origins of the data structures in Icon lie primarily in SNOBOL4 [Griswold et al. 1971]. Many persons have participated in the design and implementation of data structures in Icon. Dave Hanson, Tim Korb, Cary Coutant, and Steve Wampler were major contributors.

References

[B], *The B Newsletter*, Informatics Department, Mathematical Centre, Amsterdam.

M. Cowlishaw, The Design of the REXX Language, *SIGPLAN Notices* **22**(2), pages 26-35 (Feb. 1987).

L. L. Fabisinski, *Computing with ProLex 1.0,* MetaLex Systems, Inc. (1989).

D. J. Farber, R. E. Griswold and I. P. Polonsky, SNOBOL, A String Manipulation Language, *J. ACM* **11**(1), pages 21-30 (Jan. 1964).

D. J. Farber, R. E. Griswold and I. P. Polonsky, The SNOBOL3 Programming Language, *Bell System Technical Journal* **XLV**(6), pages 895-944 (July-Aug. 1966).

O. R. Fonorow, Modelling Software Tools in Icon, *Proceedings of the 10th International Conference on Software Engineering*, Apr. 1988, pages 202-220.

J. F. Gimpel, *Algorithms in SNOBOL4,* John Wiley & Sons, New York, NY (1976).

R. E. Griswold, *String and List Processing in SNOBOL4; Techniques and Applications,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1975).

R. E. Griswold, String Analysis and Synthesis in SL5, *Proceedings of the ACM Annual Conference*, 1976, pages 410-414.

R. E. Griswold, String Scanning in the Icon Programming Language, *Computer J.*, to appear.

R. E. Griswold and M. T. Griswold, *The Icon Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1983).

R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language,* Princeton University Press (1986).

R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, NJ (second edition, 1971).

B. W. Kernighan and R. Pike, *The UNIX Programming Environment,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1984).

B. Liskov, *CLU Reference Manual,* Springer-Verlag (1981).

J. McCarthy, P. W. Abrahams, D. J. Edwards and M. I. Levin, in *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, MA (1962).

A. Newell, *Information Processing Language-V Manual,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1961).

R. P. Polivka and S. Pakin, *APL: The Language and Its Use,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1975).

J. T. Schwartz, R. B. K. Dewar, E. Dubinsky and E. Schonberg, *Programming with Sets: an Introduction to SETL,* Springer Verlag (1986).

G. L. Steele Jr., *Common LISP: The Language,* Digital Press (1984).

L. Sterling and E. Shapiro, *The Art of Prolog,* MIT Press, Cambridge, MA (1986).

K. Walker, A Type Inference System for Icon, The Univ. of Arizona Tech. Rep. 88-25 (1988).

[*submitted May 5, 1989; revised Aug. 21, 1989; accepted Sept. 1, 1989*]