The Icon Analyst

In-Depth Coverage of the Icon Programming Language

June 1991
Number 6

In this issue …

Modeling String Scanning

Like most other features of Icon, you can use string scanning without a "deep" understanding of what's going on. For example, you don't have to know how string scanning is actually implemented. Generally, that's just as well. But, like many other aspects of a programming language, you can use string scanning more effectively if you understand more about it.

The sure way to a deep understanding of a programming language feature is to implement it yourself. That's why the implementors of a programming language often have an advantage in using the language, and in using it in ways different from others — they *know* how features work; they don't have to puzzle them out or guess.

Of course, few Icon programmers would care to make the investment in re-implementing a language feature just to understand it better. There's an easier alternative in some cases: model the implementation of the feature at a high level, in the language itself. That's what this article is about; modeling string scanning in terms of Icon procedures. This model cannot handle all aspects of string scanning, but it does show how string scanning works at an operational level for which no number of words can substitute.

This model also allows experimental extensions to string scanning. From an implementor's view, such a model is an easy and flexible way to try out new and untested language features without the much larger investment of doing a real implementation.

String-scanning has the form

expr1 ? *expr2*

where *expr1* is the *subject expression* that provides a subject that is the focus of attention for the evaluation of the *analysis expression, expr2*. During scanning, the keyword &subject is the value being scanned and &pos is the position in &subject where matching expressions apply.

At first glance, you might think all you need to do is write a procedure, and model string scanning as

expr1 ? *expr2* → Scan(*expr1*,*expr2*)

But in a procedure call, all arguments are evaluated before the procedure is called. Consequently, *expr2* is evaluated before there is a chance to change the two "state variables" &subject and &pos so that scanning applies to the result of evaluating *expr1*.

String scanning is a control structure that does not follow the order of evaluation of procedure calls. The value of *expr1* provides the value of &subject, and &pos is set to one, so that scanning begins at the beginning of &subject. After these variables are set, *expr2* is evaluated. It typically examines the value of &subject and changes the value of &pos in the process. &subject and &pos are global variables; their values constitute an environment for scanning. The outcome of the scanning control structure is the outcome of *expr2*.

Even though *expr1* ? *expr2* is a control structure, it can be modeled using procedures. A naive model is

expr1 ? *expr2* → Bscan(*expr1*) & *expr2*

The procedure Bscan() intervenes in the evaluation process and sets the values of &subject and &pos before *expr2* is evaluated. Bscan() can be written as follows:

```
procedure Bscan(e1)
  &subject := e1
  &pos := 1
  return
end
```

The assignment to &pos is redundant and is included only for clarity.

This model illustrates how simple the string-scanning control structure is — any actual pattern matching takes place during the evaluation of *expr2* and the only function of the string-scanning expression itself is to set the values of the keywords in the scanning environment.

This naive model, which is equivalent to the mutual evaluation

(Bscan(*expr1*),*expr2*)

does not account for the fact that string-scanning control structures can be nested or that several of them can occur in mutual evaluation, so that several scanning environments can exist simultaneously at any point in program execution (see "String Scanning" in Issue 3 of the *Analyst*). In order for string scanning to behave in a useful and coherent way, the values of &subject and &pos are saved prior to assigning new values to them in a string-scanning control structure, and they are restored when the evaluation of the string-scanning control structure is complete. Furthermore, if the scanning expression suspends, the string-scanning control structure suspends. It may be resumed later to produce another result, so it is necessary to reset &subject and &pos if a scanning control structure is resumed. To accomplish this, a more general model is needed:

expr1 ? *expr2* → Escan(Bscan(*expr1*),*expr2*)

The operation of this model is illustrated by the following procedures in which records of type ScanEnvir hold the values of &subject and &pos for scanning environments:

```
record ScanEnvir(subject,pos)

procedure Bscan(e1)
  local OuterEnvir
  OuterEnvir := ScanEnvir(&subject,&pos)
  &subject := e1
  &pos := 1
  suspend OuterEnvir
  &subject := OuterEnvir.subject
  &pos := OuterEnvir.pos
  fail
end

procedure Escan(OuterEnvir,e2)
  local InnerEnvir
  InnerEnvir := ScanEnvir(&subject,&pos)
  &subject := OuterEnvir.subject
  &pos := OuterEnvir.pos
  suspend e2
  OuterEnvir.subject := &subject
  OuterEnvir.pos := &pos
  &subject := InnerEnvir.subject
  &pos := InnerEnvir.pos
  fail
end
```

In this formulation, *expr1* is evaluated first and provides the argument to Bscan() that is used for the new value of &subject. In Bscan(), the current values of &subject and &pos are saved in OuterEnvir before the new ones are set. Bscan() then suspends with OuterEnvir, which is passed on to Escan(). However, *expr2* is evaluated first and may change the values of the state variables before Escan() is called. If evaluation of *expr2* succeeds, Escan() is called with two arguments: the outer scanning environment that was in effect before Bscan() was called, and the result produced by the evaluation of *expr2*.

Escan() saves the scanning environment as it was left by the evaluation of *expr2* in InnerEnvir, restores the outer environment, and suspends with the result produced by the evaluation of *expr2*.

If the scanning control structure occurs in a context in which it is resumed, the evaluation of Escan() picks up after the suspend expression, and &subject and &pos are restored to the values they had when *expr2* produced its previous result. If *expr2* produces another result, as in

expr1 ? move(1 | 2)

Escan() is called again; the situation is the same as it was when *expr2* produced its previous result.

If *expr2* does not produce another result, Bscan() is resumed and picks up evaluation after its suspend expression. It restores the outer scanning environment and fails. At this point, *expr1* is resumed. If it produces another result, as in

(s1 | s2) ? *expr2*

Bscan() is called again and the process described above is repeated.

The reason the values in OuterEnvir are updated after Escan() is resumed is that they may have been changed while it was suspended. An example of this is

expr1 & (*expr2* ? *expr3*) & (&pos +:= 1) & find(s)

assuming &pos can be incremented but that find(s) fails. If this happens and *expr1* is eventually resumed, the change in &pos should be preserved.

It is important to note that OuterEnvir is shared by Escan() and Bscan() — that is, it is a pointer to a record. If this were not the case, changing the fields of OuterEnvir in Escan() would not affect the values that Bscan() restores before failing.

Pointer Semantics

Most programming languages provide a few different kinds of structures. Arrays and records are typical. Icon has a much richer repertoire of structures than most programming languages, but there's more to structures in Icon than just variety.

In most programming languages, structures are static objects. In Pascal, for example, an array is declared and its size and shape are fixed. While the values in an array can be changed during program execution, the array itself cannot be changed. Furthermore, an array isn't a value like an integer is. In fact, the idea of an array being a "value" may seem a bit bizarre.

But what *is* a value? In what sense is an integer a value but an array not one?

What is usually meant by a value in a programming language is an object that can be assigned to a variable, passed as an argument to a procedure, and returned as the result of a procedure. Clearly an integer is a value in this sense, while in most programming languages, an array is not. There's a similar situation with procedures. In most programming languages, a procedure is not a value, but in Icon it is. The term "first-class value", which originated in the Lisp community, is sometimes used to distinguish program objects than can be used as values in all generality as opposed to objects whose use is restricted.

What difference does it make whether or not a Pascal array or an Icon list is a value? For the most part, you can use lists in Icon like you use arrays in Pascal, without taking advantage of the "first-class" status of lists. Of course, you might suspect, if you don't already know, that there are useful things you can do with first-class values that you can't do otherwise.

Pointers

You may wonder how a list can be a first-class value. Lists can be large — there's no limit to the size of a list except the amount of memory available. How can an arbitrarily large value be assigned to a variable or passed to a function? And how does Icon know what to do, since the size of a list cannot, in general, be determined when a program is compiled? The answers to these questions are simple, but in them lurk both powerful and dangerous features of structures.

A structure is a collection of values. For example, a list is a collection of all the elements in it. These elements are logically (and physically) bound together by being in the list. This allows them to be treated collectively and separately from all other values. We've glossed over a subtle point, however. Consider a list produced by evaluating list(100). The value returned by list(100) is not the collection of 100 elements. Instead, it is a *pointer* to the collection.

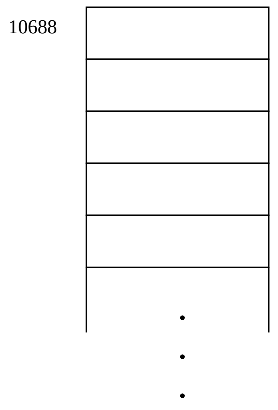
The concept of a pointer is important in many programming languages. Sometimes it is explicit, as it is in C. Sometimes it is implicit, as in Icon. The word "pointer" carries a connotation that is suggestive and generally accurate, but unless you've been involved in low-level programming, the idea may seem a bit slippery. In fact, it is almost impossible to explain what a pointer is without resorting to implementation details. Indeed, the concept of a pointer is derived from the way most computers access their memories. While most programming language concepts are derived from familiar aspects of mathematics or business transactions, the "real-world" analogies for pointers are a bit strained. Of course, if you know what a pointer is, all this is boring. If so, skip forward a bit.

If we can't do any better by way of explaining pointers than to allude to computer architecture, we'll at least own up to our limitations. A pointer is a computer memory address; a location where data is stored. From here on, we'll resort to pictures.

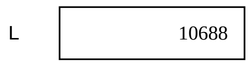
Consider the expression

```
L := list(100)
```

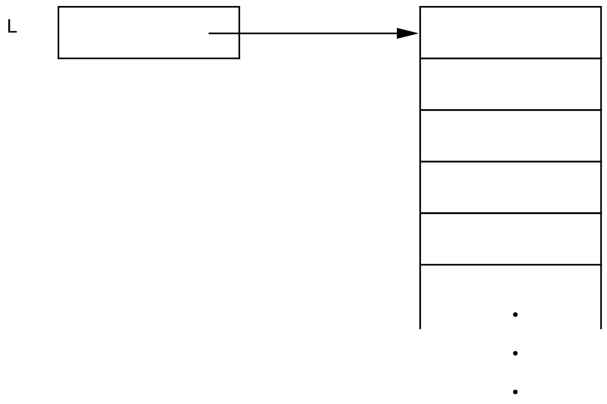
The result of evaluating this expression is to first create a list of 100 elements and then assign it to L. Suppose that the list is stored in memory starting at address 10,688. Then the list can be depicted as



Here, 10,688 is the address of the beginning of the list. The value returned by list(100) and assigned to L is this address, which can be depicted as



The relationship between this value and the list is, of course, the address. To emphasize this relationship, it helps to replace the address by an arrow (pointer):



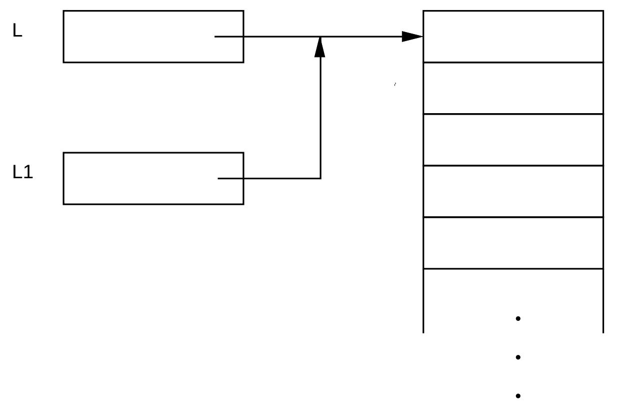
As you probably know, this diagram isn't accurate. It's an oversimplification; Icon's values and lists are more complicated than shown here. The diagram above, however, captures the idea and is correct as far as it goes. It's good enough for our concerns here.

Fine, you may say, but so what? The important point is that a list *value* (the value of L here) is quite simple. It doesn't matter how big the list is, the value is just an address (pointer). This simplicity has many ramifications.

What happens, for example, when a list value is assigned to another variable, as in

```
L1 := L
```

There are two plausible outcomes. One possibility is that the collection of elements pointed to by L could be copied and the address of the copy assigned to L1. Some programming languages do this. Icon takes a simpler approach: The *value* of L, the address of its collection of elements, is assigned to L1. Thus, for the example above, both L and L1 have the same value (10,688) after the assignment. Using pointers, this can be depicted as



That is, both L and L1 point to the same collection of elements — the *same* list.

This method of treating assignment, which also applies to passing arguments to procedures and returning values from them, has a major affect on how lists can be used in Icon.

Several aspects of "non-copying" assignment are immediately obvious:

- Assignment of a list value is fast.
- Assignment of a list does not create a new list.
- Several variables may point to the same list and hence share their elements.

There are other consequences, which we'll discuss later.

Allowing different list values to share elements is a mixed blessing. If it happens accidentally, it can lead to puzzling results and bugs. Referencing one list value can change the list pointed to by another. For example,

```
L1[10] := 0
```

changes the tenth element of the list pointed to by L as well.

The key phrase is *pointed to.* It's not that assigning a value to an element of L1 changes L. It doesn't. But it does change a value in the list that is pointed to by both L and L1.

Clearly, this is a problem if this is not the intended result. On the other hand, this aspect of lists can be very useful. For example, suppose you are dealing with lists of integers and you find you want to change all the negative values in such lists to zero. This is clearly a situation in which you'd like to be able to use a procedure. And you can:

```
procedure clip(L)
  every !L <:= 0
  return L
end
```

For example,

```
clip(range)
```

passes a pointer to the list range to clip(), which goes through the elements of this list and sets the negative values to zero. As a result, the list pointed to by L, which is the same list as the one pointed to by range, is modified.

Notice that clip() returns (a pointer to) the list on which it operates. Thus,

```
newrange := clip(range)
```

assigns the modified list to newrange; range and newrange now have the same value.

If you don't want the clip() to modify the list pointed to by its argument, you can use copy(), which creates a copy of the structure pointed to by its argument — a new structure and at a different address. For example,

```
procedure clip(L)
  L := copy(L)
  every !L <:= 0
  return L
end
```

makes a copy of the structure pointed to by its argument, which it then modifies and returns. In this case, the result of

```
newrange := clip(range)
```

is to assign to newrange a list that is different from the value of range; they point to different lists and the elements of range are not changed.

There's an important point about language design here. If assignment in Icon copied structures (instead of just pointers to them), there would be no way for two values to share the same structure (unless a special operation were provided for this purpose). We argue that copying pointers rather than entire structures is not only efficient but it also allows powerful programming techniques. Clearly we think this is the way to handle assignment of structures or we wouldn't have done it that way in the design of Icon. The point is that while efficiency is important, there is much more to this decision.

We'll start by looking at something simple that may be perplexing if you've not used structures in Icon:

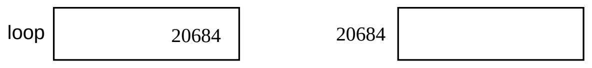
```
loop := list(1)
loop[1] := loop
```

What happens when we assign a list to an element of itself? The result is clearer if we're more careful of our language: What happens when we assign a *pointer to* a list to an element of itself?

Let's look at diagrams for the two steps. First

```
loop := list(1)
```

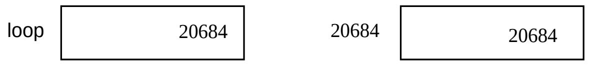
Suppose the one-element list is at address 20,684:



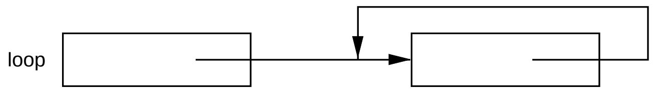
The box at address 20,684 represents the one-element list. Now consider

```
loop[1] := loop
```

Recall what assignment does:



In terms of pointers, this can be depicted as



That is, both loop and loop[1] point to the same place, the place where the list is. This should be what you expect as the result of evaluating

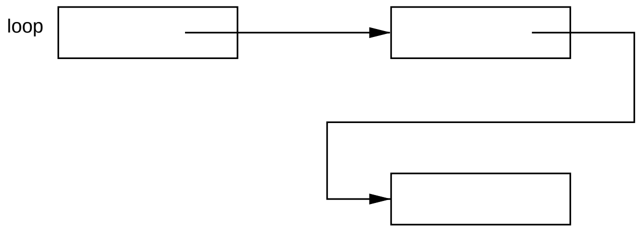
```
loop[1] := loop
```

That is, the values of loop and loop[1] are the same after the assignment.

Note that if assignment copied the structure pointed to by loop, the result would be entirely different. Icon's version of this is

```
loop := list(1)
loop[1] := copy(loop)
```

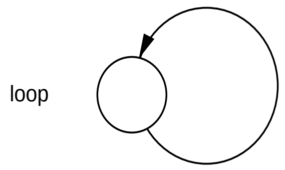
which produces the following result:



There is no loop, and there are two lists.

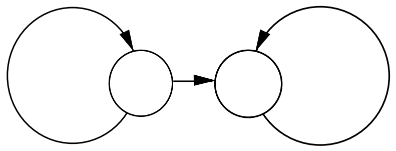
You might well ask what good all of this is. Who needs a loop in a structure anyway — it looks like big trouble. A full understanding of the potential here opens the door to several powerful programming techniques. The basic idea is simple: Because of the way that Icon handles structures and assignment (so-called "pointer semantics", as used for the title of this article), it is possible to build directed graphs that are common in many problems.

There is only a single step to make this more obvious: breaking away from a style of diagramming that emphasizes Icon's structures and using instead a more abstract representation. For the example above, a different (but equivalent) view is:



Before going on, it is important to understand that the examples given above for lists apply to all kinds of Icon structures as well: records, sets, and tables. For example, the value produced by table() is a pointer to a table structure (which is a bit more complicated and less intuitive than a list structure).

This diagram is a conventional representation of a graph with one node and an arc directed from the node to itself. The important point is that the arc is represented by the list *value* (a pointer), while the node is represented by the structure to which the list value points. It should be easy to see how to handle nodes that have more than one arc directed out. All that's needed is a list element for each arc. For example, the graph

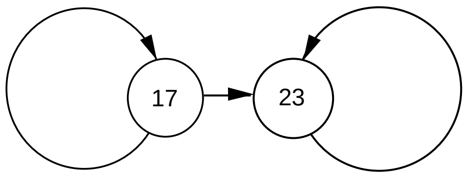


can be constructed as follows:

```
node1 := list(2)
node2 := list(1)
node1[1] := node1
node1[2] := node2
node2[1] := node2
```

Directed graphs often have a value associated with each node. A place for the value is easily provided by reserving a list element for it. Since the number of arcs out may vary, it's convenient to use the first element of a list for the value and the remaining elements for the arcs. In this way, the value of a node is always at the same place.

For example, the graph



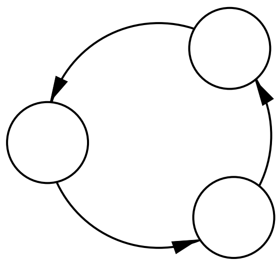
can be represented by

```
node1 := list(3)
node2 := list(2)
node1[1] := 17
node2[1] := 23
node1[2] := node1
node1[3] := node2
node2[2] := node2
```

An Example — Rings

Lists are convenient for representing graphs in which the number of arcs out varies from node to node. Records are useful when the number of arcs out of every node is the same. Records offer the additional advantage of providing field names that can be used as mnemonics to refer to node values and arcs.

Consider a ring, which consists of *n* nodes connected in a circle. For example, a ring with three nodes has the structure:



Since each node has only one arc directed from it, such a ring can be built with records. As above, if a value is associated with each node, another field is needed. A declaration for such a node is

```
record node(value,arc)
```

For example, a ring of three nodes with values "a", "b", and "c" can be constructed as follows:

```
node1 := node("a")
node2 := node("b")
node3 := node("c")

node1.arc := node3
node2.arc := node1
node3.arc := node2
```

More generality is provided by a procedure that constructs rings. For example, create_ring("a","b","c") constructs a ring such as the one above. Since the number of arguments varies from call to call, the arguments can be passed in a list:

```
procedure create_ring(value[])
  local first, current

  first := node(value[1]) | fail
  current := first
  every i := 2 to *value do
    current := node(value[i],current)
  first.arc := current
  return first
end
```

A pointer to the first node is kept in first so that the ring can be closed after the last node is created. The expression

```
current := node(value[i],current)
```

creates a new node with the ith value and directs its arc to the current node before setting the current node to the newly created one.

This procedure is adequate for creating a ring, but something more may be needed, depending on how rings are used. For example, if the designated “first” node in a ring may be changed, another pointer is needed to identify the first node — that is, a level of indirection in referring to the ring.

This “header” node can be provided by another record. By using a different record type, it is possible to distinguish between the header and the nodes in the ring itself:

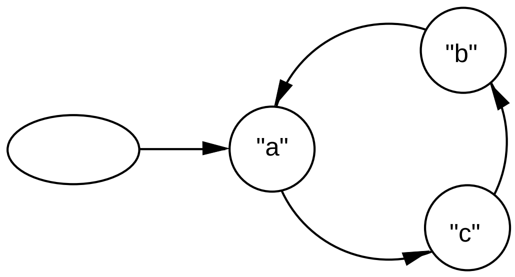
```
record ring(arc)
```

Not that both node and ring have an arc field. Icon allows this and it provides a handy mnemonic when different records have fields that are conceptually equivalent.

All that’s needed is a minor modification to create_arc() so that it returns a ring:

```
return ring(first)
```

The graph that results from create_ring("a","b","c") can be depicted as:



The oval at the left is drawn with a different shape to distinguish it from the nodes in the ring itself. You may find it handy when drawing graphs to use different shapes for nodes of different types. See Reference 1 for examples.

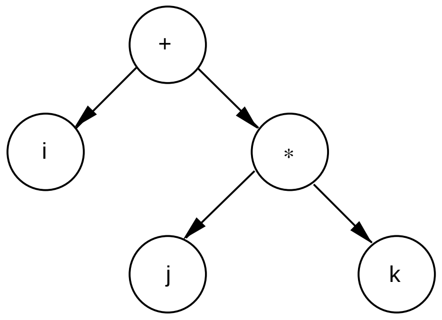
Finally, here's a procedure to generate the values from a ring, continuing around the ring indefinitely:

```
procedure ring_value(R)
  local n

  n := R.arc
  repeat {
    suspend n.value
    n := n.arc
    }
end
```

The Null "Pointer"

Sometimes it's useful to think of arcs as being "typed". For example, in a binary tree, a node has two arcs, one pointing to its left subtree and one pointing to its right subtree:



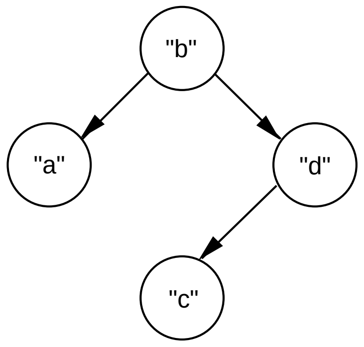
For such a structure, the nodes with arcs can be represented by a record in which the field names distinguish the two types of arcs:

```
record bnode(value,larc,rarc)
```

Since "leaf" nodes at the bottom of the tree have no arcs, they could be represented with a different kind of record:

```
record leaf(value)
```

In some kinds of binary trees, the situation is not so simple:



Here, one node does not have a right subtree. The idea above could be extended to provide two more kinds of nodes, one that has only a left subtree and one that has only a right subtree. Having four kinds of nodes in binary trees makes constructing and processing them very complicated, especially when such trees are built incrementally by adding subtrees.

A simpler approach is to use only one kind of node throughout the tree but to allow some arcs to be missing. For example, leaf nodes have both arcs missing. The question, then, is how to designate a missing arc.

The usual method of handling a missing arc is to use the null value in place of a pointer. The motivation for using the null value is simple: it's the default value, so if you create a record or other structure and don't specify a pointer value, the null value is already there. It's also easy and fast to check for the null value.

Using this idea, the binary tree shown above can be built as follows:

```
root := bnode("b")
root.larc := bnode("a")
root.rarc := bnode("d")
root.rarc.larc := bnode("c")
```

The nodes with values "a" and "c" have two null "pointers", while the node with value "d" has one. These values are provided by default, since no values for arcs are provided when the nodes are created and only the needed ones are filled in later.

Although we've written out the construction of this binary tree as four separate assignments, we could have done it with one big nested expression — simply because a tree has no loops and at most one arc pointing to any node.

It's easy to process such trees, recognizing when arcs are missing by testing for the null value. For example, here's a procedure that traverses a binary tree, writing the value of every node:

```
procedure write_values(T)
  write(T.value)
  write_values(\T.larc)
  write_values(\T.rarc)
end
```

The procedure calls itself recursively only when subtrees are present, as determined by the non-null test; a field that is non-null, must, by construction, contain a pointer to a node. The recursive calls trace the pointer structure of the tree.

Conclusions

Directed graphs are so useful and ubiquitous that you'd think most programming languages would provide easy ways of representing such graphs. It's hard to imagine anything more natural and general than pointers in this regard. But pointers are "considered dangerous" by some computer scientists, especially members of the "police-state school of programming". Pointers are, indeed, dangerous, as any C programmer knows — it's all too easy to wind up with a pointer to an inappropriate place. That's true in Icon too; think about what happens if a binary tree is constructed so that a nonnull field is not a pointer to a node. But living in a totally safe world is stupefyingly boring and it's very hard to do anything with tools that can't do damage.

How "dangerous" pointers are depends to a large degree on how a programming language handles them. You're less likely to get into trouble with pointers in Icon than in C. In Icon, you can at least determine the type of a value during program execution and tell whether it's a pointer (structure) or something else.

Admittedly, it's difficult to debug programs that use pointers extensively. Diagnostic tools are lacking; there's no way to convert a pointer to something concrete and printable. You might think Icon would at least provide you with a way to find out the memory address to which a pointer corresponds. While such a feature would not be hard to implement, it would have dubious merit. Recall from the article on memory monitoring in the first issue of the Analyst that Icon objects may move as the result of garbage collection. That is, the memory address corresponding to a pointer may change. A pointer whose initial address is 10,688 may be something different later on in program execution.

The best prescription for avoiding problems with pointers is to take extra care; a little self-discipline goes a long way in this area.

Further Reading

We've only touched on what you can do with pointers. Reference 2 has numerous examples of the use of Icon structures to build different kinds of directed graphs. There often are different ways of representing the same graph. The one you chose may make a lot of difference in ease of processing and getting correct results. Sets and tables, in particular, offer possibilities that you might not think of at first glance.

Reference 1 also provides numerous examples of the use of pointers to build directed graphs. The programming language used in that reference is SNOBOL4, but SNOBOL4 and Icon use the same semantics for pointers, so it's easy to transcribe programs that deal with structures from one language to the other.

References

1. *String and List Processing in SNOBOL4*, Ralph E. Griswold, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

2. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

Evaluation Sandwiches

There may be times when you'd like to do something when a particular expression is evaluated but not interfere with its evaluation. For example, if you suspect an expression contains a bug, you might want to associate some diagnostic output with it so that you can tell it's been evaluated, what value it produced, and so on.

Suppose, for example, the following expression is not producing the expected results:

```
text ? {
   every write(verb())
   }
```

Suppose you try adding a diagnostic statement such as

```
text ? {
   every {
      write(verb())
      write(&errout,"&pos = ",&pos)
     }
   }
```

This doesn't work, since the diagnostic expression interferes with the resumption of verb(). What you probably want is

```
text ? {
   every write(verb()) do
      write(&errout,"&pos = ",&pos)
   }
```

A less obvious alternative is conjunction:

```
text ? {
   every write(verb()) &
      write(&errout,"&pos = ",&pos)
   }
```

The difference between the two previous expressions is worth thinking about — it goes to the heart of expression evaluation in Icon. In the first case, write(verb()) is the first expression in a compound expression, and hence is bounded. Once it produces a result, its evaluation is complete and it never can be resumed. In the third case, write(verb()) is in conjunction with the diagnostic expression; although the diagnostic expression doesn't suspend, goal-directed evaluation resumes the suspension of verb().

While putting the diagnostic expression in the do clause solves the problem, you might wonder about the last alternative. Is it always really safe to put a diagnostic expression in conjunction with another expression?

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

FTP: cs.arizona.edu (cd /icon)

To avoid having to worry about all kinds of possibilities, it's worth thinking about a general way to include diagnostic expressions, either before or after the evaluation of an expression of interest, in such a way that the evaluation of the expression itself is not perturbed.

To get slightly more abstract, suppose *expr* denotes the expression of interest and *before* and *after* denote, respectively, expressions that are to be evaluated before and after *expr* is evaluated. Then a general method of approach is

before & *expr* & *after*

However, such an expression does not produce the result of *expr*, since the outcome of the entire conjunction is the outcome of *after*. Here's where mutual evaluation is useful; you can sandwich *expr* between *before* and *after*, but select *expr*:

2(*before*,*expr*,*after*)

Since the selection operation produces the outcome of the selected expression, which may be a variable, this "sandwich" can be used even in assignment expressions. For example,

```
L[i] := s
```

can be rewritten as

2(*before*,L[i],*after*) := s

and the assignment still is made to L[i].

When you use such a sandwich, you can't just have any kind of expressions for *before* and *after*. Obviously, *before* can't fail — you'd never get to *expr*. Similarly, in the general case, *after* can't fail either. In fact, both *before* and *after* must be monogenic (that is, produce exactly one result). Of course, if *before* and *after* are just expressions that write something out, there's no problem. You'd probably never think to have them be generators anyway. But what if *expr* is a generator? The sandwich still is correct — it generates the results of *expr*. But, while you're thinking about generators, suppose you want to know more about what *expr* does, such as whether it succeeds or fails, whether it produces a lot of results, how many times it is resumed, and so on. This can be done with the sandwich also, by providing expressions for *before* and *after* that do more than just write diagnostic messages.

While this more general form of monitoring *expr* can be done with suitable expressions for *before* and *after* directly in the sandwich, it's easier to understand if procedures are used instead:

2(before(),*expr*,after())

This leaves you free to write the procedures in various ways without changing the sandwich itself. In the simplest case, the procedures might just write some information of interest:

```
procedure before()
  write(…)
  return
end
```

```
procedure after()
  write(…)
  return
end
```

Here's a case where it's easy to make a mistake and forget to put in the return expressions. If you do, before() will fail and you'll never get to *expr*.

If you're interested in whether *expr* fails or generates a lot of results, the return expressions in before() and after() can be replaced by suspend expressions. An example is:

```
procedure before()
  write("about to evaluate")
  suspend
  write("no result produced")
  fail
end

procedure after()
  write("result produced")
  suspend
  write("resumed")
  fail
end
```

If you don't completely understand what's going on, take the time now to look at it closely — it's a good example of what's involved in the evaluation of an expression, the production of its results, suspension, and resumption.

Note that the message "no result produced" illustrates the problem with the concept of failure. As it stands, there is no way for before() to tell if it is being resumed because *expr* didn't produce any result at all, or if *expr* produced many results and eventually produced no more.

Suppose that you want after() to have access to the result that *expr* produces. You can manage that in several ways, including assignment to an auxiliary variable, but there's another form that also gives more insight into expression evaluation in Icon:

(before(),after(*expr*))

The order of evaluation still is correct: before() first, then *expr* (arguments are evaluated before functions are called), and then after(). Since after() gets the value produced by *expr*, it needs to produce that value also, which becomes the value of the entire expression. (In the absence of a selector, a mutual evaluation expression returns the result of its last argument.) Thus, after() should look something like this:

```
procedure after(x)
  write("value produced is ",image(x))
  suspend x
  write("resumed")
  fail
end
```

Be sure to convince yourself that this formulation still does what it's supposed to do. If after() is resumed, it fails. When a procedure call fails, its argument is resumed. Thus, *expr* is resumed. If it produces another result, after() is called again, writes the value produced, and suspends in turn.

If you've been following closely, you'll notice something slightly wrong with this last formulation. In the "sandwich" formulation, the sandwich produced the results produced by *expr*. The term *result* is used in Icon in the technical sense to mean either a value or a variable. In the last formulation, after() produces the value produced by *expr*. If *expr* produces a variable, only its value is produced. This happens because all arguments to procedures in Icon are passed by value; there is no way to get a variable into a procedure and hence no way for it to produce an argument variable as a result. Hence this formulation cannot be used in situations where *expr* produces a variable to which an assignment is made.

All this is fine, but if you want to use a lot of evaluation sandwiches (such as for every expression in a program), writing them is tedious and error-prone, if not a practical impossibility. There's a tool that makes the production of evaluation sandwiches (and many other Icon program transformations) easy. It's called a variant translator.

We'll have an article on variant translators in an upcoming issue of the Analyst.

Program Termination

If you're writing a program for someone else to use (an "application program"), the way your program terminates is important, both in terms of its user interface and in terms with its interaction with its environment. Generally speaking, there are two principal types of termination: normal and error.

Normal termination may occur because the application program has come to natural completion or because the user is finished with it and tells the program so.

There is a tendency for persons who write application programs to try to cast them in anthropomorphic terms and make them behave like human beings. When it comes to termination, such a program may engage in parting amenities. While such "good-bye" messages are a matter of style (which may depend on the user community and what it expects), remember that an application program may run from a script and write output to a file. Above all, remember that cute and verbose messages quickly become stale and annoying. And not everyone may think your jokes are funny.

The effect of program termination on its environment is of greater practical concern than whether or not a user likes your prose. In most operating systems, a program produces an exit code. The value of the exit code tells the operating system whether everything has gone well or there has been an error. Exit codes usually are small integers like 0 and 1 and can be thought of as what a program itself returns when it is done.

Unfortunately, there is not a complete agreement among operating systems as to what exit code values mean. Most systems use 0 to stand for normal termination (everything went well, as far as can be told) and 1 (or any nonzero value) to indicate that some kind of an error occurred during the execution of the program. There is a notable exception; VMS uses odd values for normal termination and even values for error termination. Since the codes themselves vary, we'll just use the terms "normal" and "error" and assume only two are significant.

The user of a program may or may not notice its exit code. A sophisticated user may arrange to be told the code by the operating system, but that is the exception. (You can use

The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
Gould-Simpson Building
The University of Arizona
Tucson, Arizona 85721
U.S.A.

(602) 621-8448

FAX: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

...{uunet,allegra,noao}!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona

this in writing Icon programs, since the value of system(s) is the exit code produced when the program specified by s terminates). However, other programs may take note of an error exit code and do something about it. For example, a command script may stop abruptly if a program produces an error exit code. That's fine if that's what you intended, but it's easy to return an error exit code unintentionally, so it's important to pay attention, when writing an application, to how it terminates.

There are four ways that an Icon program can terminate of its own accord:

- By returning from the initial call of the main procedure that started its execution.
- By evaluating stop(s).
- By evaluating exit(i).
- As the result of a run-time error.

Termination as a result of returning from the main procedure can occur as a result of an explicit return, fail, or suspend (although that is a bit bizarre) or implicitly by flowing off the end of the main procedure, which is equivalent to failure. In all of these cases, a normal exit is signaled. In particular, failure of the initial main procedure call does not produce an error exit code. That's fortunate, since most programs terminate normally by flowing off the end of the main procedure.

The usual reason for terminating program execution by stop(s) is because an error has been detected. stop(s) sometimes is used when there's no easy way to get back to the initial call of the main procedure from the point where the program should terminate. That's not to say it's impossible to write a program so that it can always get back to the main procedure, but it certainly can be difficult and awkward to do so.

If a program terminates as a result of stop(s), error termination is signaled. stop(s) should be used only if you want to indicate error termination, not just as a convenient way to get out. Furthermore, stop(s) always writes something — an empty line if s is omitted. This can be disconcerting to the user (the cursor on a terminal may jump unexpectedly), and if the output of the program is written to a file, it may add unexpected data.

The function exit(i) is more versatile, since it can be used to terminate execution from any point in a program and i can be used to provide the appropriate exit code. The default, if i is omitted, is the code for normal termination: 0 for most systems but 1 for VMS. exit(i) itself produces no output, but output can be provided separately if you want it. For writing application programs, a more flexible abstraction is provided by Exit(s,i), which writes s if it is nonnull and terminates with exit code i. This is easily provided by an Icon procedure:

```
procedure Exit(s,i)
   write(\s)
   exit(i)
end
```

For sophisticated applications, various values of i can be used to signal different flavors of termination, ranging from normal termination to the indication of various kinds of errors. Remember, however, that different systems interpret exit codes in different ways.

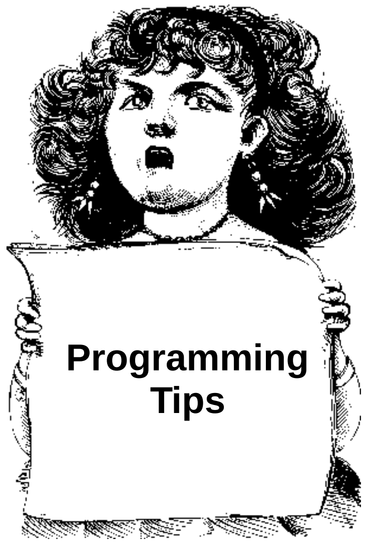
A run-time error, quite reasonably, indicates error termination and returns the same code as stop(). Generally speaking, error termination, either intentional or unintentional, should be carefully avoided in application programs. For one thing, the user of an application program should not need to know that the program is written in Icon, much less what Icon's various error messages mean and how they might relate to how the program works and what went wrong. Even if the error is the user's fault (the notorious "user error"), an application program should provide the user with useful information couched in terms of the application (not Icon). Mysterious messages tend to produce user hostility and sometimes guilt ("I broke it"). Besides, if you really want your program to behave in a hostile manner, you can do better than producing an Icon run-time error message.

We'll have more to say about this in a subsequent article on writing robust programs.

Renew Now!

If you subscribed to the Analyst for a year, starting with the first issue, your subscription ends with this issue and there's a renewal form enclosed with this mailing.

Don't miss an issue of the Analyst! Renew now!



One of the most common difficulties for persons learning Icon is the use of default values for tables. If the default value is an integer or the null value, as in `table(0)` and `table()`, it usually doesn't cause problems. But if the default value is a structure, most inexperienced Icon programmers run into puzzling problems.

A structure may be a useful default value in problems where the values associated with table keys are complex. For example, in writing a program to produce concordances, the value associated with a word may be the set of line numbers in which the word occurs. This sounds simple enough:

```
lines := table(set())
```

Then to add a new line number for a word, all that's needed is

```
insert(lines[word],number)
```

Except things come out all wrong. All the words come out with the same line numbers — in fact, all the line numbers. What's going on? It's not a bug in Icon (a common first reaction). The method used is wrong; the result of a conceptual misunderstanding.

The problem is that there's only *one* set that is used as the default value for all keys, not a different set for each different key. To see why this is the case, consider two formulations:

```
S := set()
lines := table(S)
```

and

```
lines := table(set())
```

These two formulations produce the same result. In fact, in `table(set())`, `set()` is evaluated first to produce a (single) set, which then becomes the argument for `table()`. This single set is then the default value for all new keys used to reference the table. No wonder the results are not as expected!

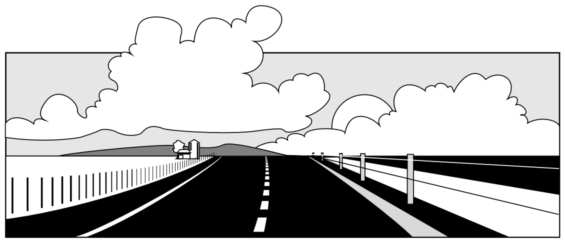
When Icon programmers discover what actually happens, their natural reaction is "But that's not what I *want*!". Unfortunately, there's no way to have a structure created automatically every time a table is referenced with a new key. It's necessary to check for a new key and explicitly create the structure.

One way to do this is to use a null value for the table default (and do not put the null values in the table):

```
lines := table()
```

Then when subscripting lines, do something like this:

```
/lines[word] := set()
insert(lines[word],number)
```



What's Coming Up

You're probably used to thinking of string scanning as a primarily analytic tool. Have you ever wondered what a similar mechanism for string synthesis might look like? One approach is to combine analysis and synthesis in a string transformation facility, using ideas similar to the ones in Icon's string scanning mechanism. In the next issue, we'll suggest what a string transformation facility might be like, using the idea of modeling as in the article in this issue.

We'll also have an article on something a bit different — variant translators. These are programs analogous to Icon's own translator, which converts Icon programs into virtual machine instructions that are then interpreted. But a variant translator changes the form of a source program instead of producing virtual machine code. The article on "Evaluation Sandwiches'" in the issue suggests an application for such a capability. In the next issue, we'll show how to go about this.

You're surely familiar with the fact that an Icon expression can produce a sequence of results. In programming, you probably think of the situations that cause an expression to produce more than one result and how the results are produced. Another approach is to consider a *result sequence* in a more abstract way as the capability that an expression has to produce results, even if it doesn't happen to produce all of them in a given situation. This view can provide new ways of thinking about programming. We'll have an article on this subject next time.