The Icon Analyst

In-Depth Coverage of the Icon Programming Language

August 1996
Number 37

In this issue ...

Expanded Analyst Format

For the first six years of the Analyst, we've maintained a 12-page format. In recent issues, with more images and diagrams, the 12-page format has been cramped and difficult to manage. Some topics, notably building visual interfaces, had to be broken up into many articles and it's taken a long time to cover the subject. And sometimes we've left out material because there wasn't room.

The printing method we use constrains us to publishing in a multiple of four pages. So, unless we send out blank paper, the next step beyond 12 is 16.

We've gone to 16 pages in this Analyst as an experiment. We expect to have 16-page issues from time to time in the future as the material we have warrants it.

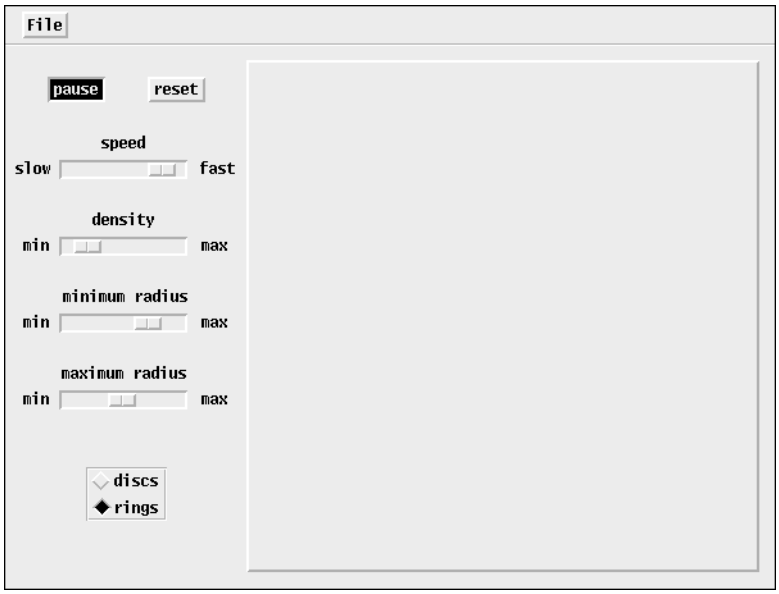
The subscription price for the Analyst will remain the same.

Building a Visual Interface

Prototyping the Interface

In the last article on building visual interfaces, we completed the interface with all the vidgets configured and positioned as we wanted them — at least provisionally.

We can see the interface "in action" without leaving VIB. Typing @P starts up a prototype of the application with functional vidgets. See the image at the left.

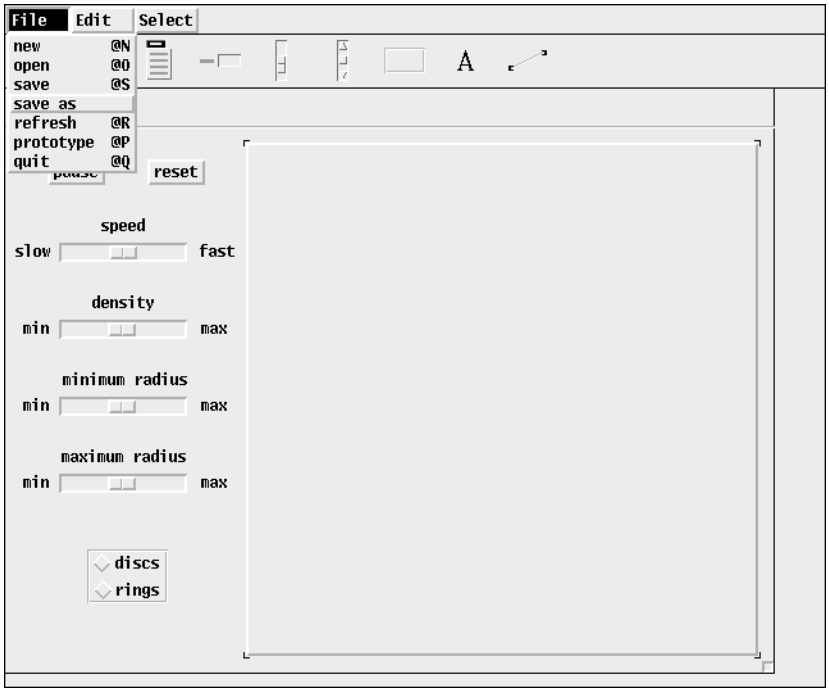


Protoyping the Kaleidoscope Application

The prototype comes up in a separate window. We can click on buttons, pull down the menu, move a slider thumb, and so forth. A listing of the activated vidgets and their callback values is written to standard output, where we can see if we're getting what we expected.

Here's an example of the output from the prototyping mode:

```
callback: id=reset, value=1
callback: id=pause, value=1
callback: id=pause, value=&null
callback: id=sld_speed, value=412
callback: id=sld_speed, value=128
callback: id=sld_density, value=82
```



The File Menu

who prefer them to memorizing keyboard shortcuts, and they contain reminders of what the shortcuts are. The menus also provide facilities that are not used frequently but nonetheless are important.

The File menu, shown at the left, provides for creating new interfaces, opening previously saved ones, saving the current interface, and so on.

It's wise to save an interface frequently while working on it — and it's very fast. The refresh item in the File menu redraws the application canvas in case something is drawn incorrectly, as sometimes happens.

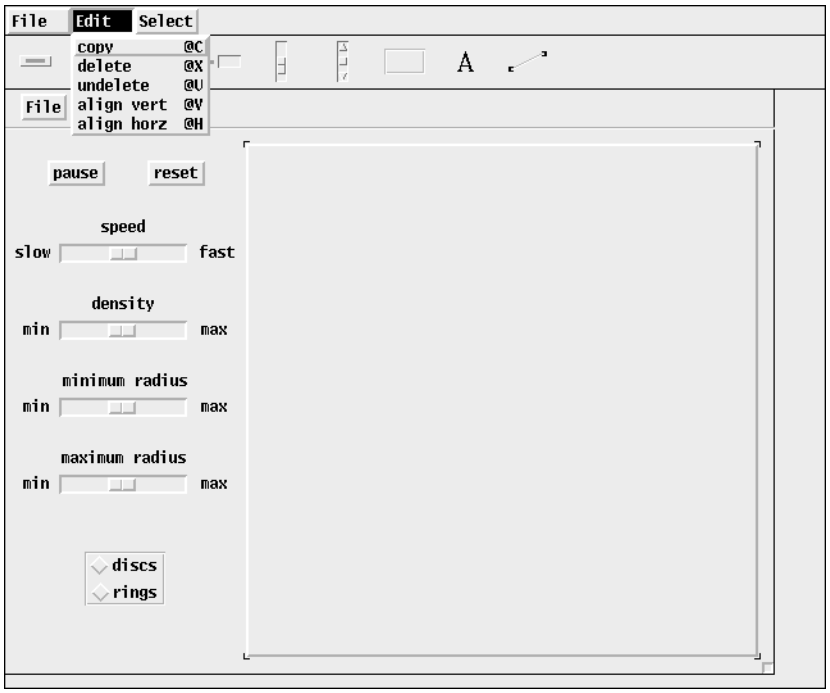
The Edit menu, shown below, provides for copying vidgets, deleting vidgets, "undeleting" the last deleted vidget, and aligning vidgets.

```
callback: id=sld_min_radius, value=84
callback: id=sld_min_radius, value=35
callback: id=sld_max_radius, value=192
callback: id=sld_shape, value="discs"
callback: id=sld_shape, value="rings"
callback: id=sld_shape, value="discs"
callback: id=file, value=["quit          @Q"]
callback: id=file, value=["snapshot  @S"]
```

Pressing q when the mouse cursor is not on any vidget dismisses the prototype and we can go back to VIB to make adjustments or just admire our work.

The VIB Menus

VIB has three menus to assist in building interfaces. We don't use them often, because most of their functionality can be obtained using keyboard shortcuts. The menus are useful for persons



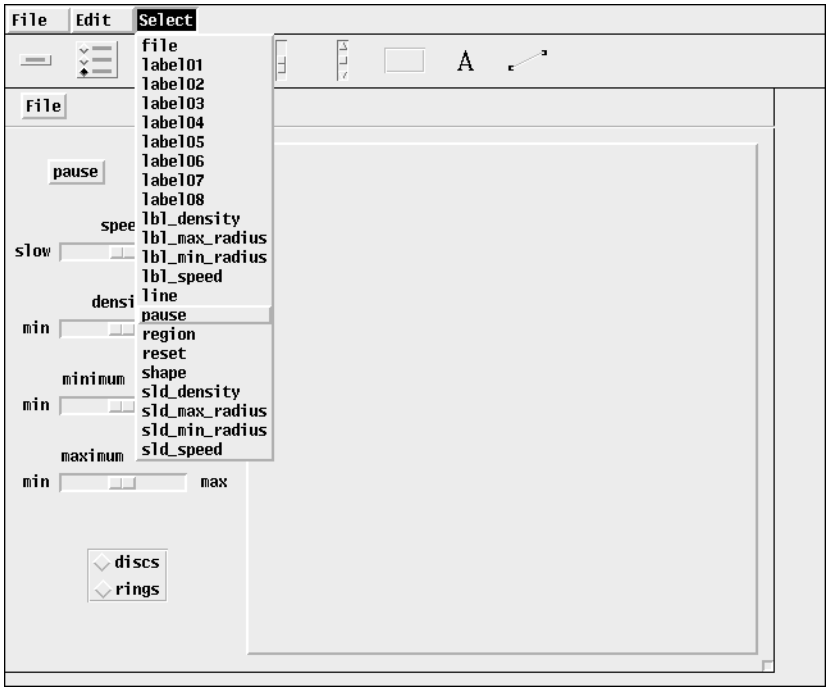
The Edit Menu

The Select menu allows a vidget to be selected by its ID as shown in the image at the top of the next page.

Ordinarily a vidget is selected by clicking on it. Sometimes, however, it is difficult to select a line, since it's only two pixels wide. A vidget also may have no visible appearance and can get "lost". The Select menu solves these problems. It also

Back Issues

Back issues of The Icon Analyst are available for $5 each. This price includes shipping in the United States, Canada, and Mexico. Add $2 per order for airmail postage to other countries.



The Select Menu

illustrates why it is important to choose good mnemonics for vidget IDs.

Next Time

That finishes up our description of using VIB. In the next issue of the Analyst, we'll show the code that VIB produces and then go on to writing the application itself.

Dynamic Analysis — A Different Approach

In past articles on dynamic analysis of Icon programs [1-4], we've used MT Icon and the instrumentation built into Icon's run-time system to capture information about running programs.

In this article, we'll look at a more conventional method for getting information about program execution — inserting instrumentation code into the source program itself.

Source-Code Instrumentation

Adding instrumentation code to programs has been used in many studies of program behavior. The best known results are in algorithm animation [5-6].

In this article, we'll study string concatenation. One approach is to replace expressions of the form s1 || s2 by calls to a library procedure, say moncat__(s1, s2), where moncat__() is a procedure that concatenates s1 and s2 but also performs other actions, such as reporting the length of the result. (Trailing underscores are used both as a convention to identify instrumentation code and to reduce the probability of a collision between names used for instrumentation and names used in the program being instrumented.)

Inserting instrumentation code by hand is error-prone, time-consuming, tedious, and may have to be redone if the program is modified.

Icon meta-translators [7] solve these problems by mechanizing the translation of an Icon program to, say, replace all concatenation expressions by calls to a monitoring procedure.

Meta-translators provide code-generation procedures for the various kinds of expressions and declarations that can occur in Icon programs. Default code-generation procedures are provided to echo the program being translated, producing an equivalent program that differs only in layout. Alternative translations are accomplished by modifying the default procedures. The code-generation procedures are themselves written in Icon, so no skill in another programming language is needed to craft a meta-translator.

For example, the default code generator for binary operations is:

```
procedure Binop(op, e1, e2)                # e1 op e2
   return cat("(", e1, " ", op, " ", e2, ")")
end
```

where op is the string name of the operator and e1 and e2 are the operand expressions. The procedure cat() from the Icon program library concatenates an arbitrary number of strings. For example, s1 || s2 is translated into (s1 || s2) leaving the expression unchanged except for the addition of parentheses to assure proper grouping.

To change the translation of concatenation to the form suggested earlier, it is only necessary to test for the operator "||" and produce an alternative translation for it:

```
procedure Binop(op, e1, e2)       # e1 op e2

   if op == "||" then
      return cat("moncat__(", e1, ",", e2, ")")
   else
      return cat("(", e1, " ", op, " ", e2, ")")

end
```

Notice that only the translation for concatenation has been changed.

Translation for augmented concatenation, ||:=, also needs to be specified. The code-generation procedure Asgnop() handles augmented assignment. With the change for concatenation, it is:

```
procedure Asgnop(op, e1, e2)              # e1 op e2

   if op == "||:=" then
      return cat(e1, " := moncat__(", e1, ",",  e2, ")")
   else return cat("(", e1, " ", op, " ", e2, ")")

end
```

It only remains to write moncat__() to monitor concatenation. There are many possibilities. A simple one is to just write out lengths of the concatenation, as in

```
procedure moncat__(s1, s2)
   local s

   s := s1 || s2

   write(*s)

   return s

end
```

Note that moncat__() must return the concatenation of its arguments so that the program being monitored will run properly.

Using the Results of Monitoring

In this simple example, the information written by moncat__() might be used to produce a summary report, as in previous articles on dynamic analysis. A more sophisticated approach would be to produce an on-the-fly, animated display of concatenation. We'll get to this very interesting possibility in the next article on dynamic analysis, but first we'll show some of the things that can be done with postmortem analysis.

In previous articles on dynamic analysis, we used a suite of 11 programs from the Icon program library. For reference, these are the programs:

program	*functionality*
csgen.icn	sentences from context-free grammars
deal.icn	randomly dealt bridge hands
fileprnt.icn	character display of files
genqueen.icn	solutions to the *n*-queens problem
iiencode.icn	text encoding for files
ipxref.icn	cross references for Icon programs
kwic.icn	keyword-in-context listings
press.icn	file compression
queens.icn	solutions to the *n*-queens problem
rsg.icn	sentences from context-free grammars
turing.icn	Turing machine simulation

One thing we easily can get from the data produced by moncat__() are counts of the number of concatenations each program performs:

csgen:	10568
deal:	11207
fileprnt:	2492
genqueen:	3168
iiencode:	0
ipxref:	2598
kwic:	10619
press:	14086
queens:	2
rsg:	0
turing:	2860
total:	57600

It's not surprising that there is a great deal of difference in the number of concatenations these programs perform. But the figures for iiencode and rsg *are* surprising, since both of these programs produce large amounts of output.

The reason for these apparent anomalies is that both programs use writes() to construct output on the fly without doing an actual concatenation [8]. (There are concatenation operators in rsg.icn, but they are used for special situations that do not arise in the way we test the program.)

Here are the figures for the total number of characters produced by concatenation:

csgen:	114620
deal:	70307
fileprnt:	50353
genqueen:	60192
iiencode:	0
ipxref:	351758
kwic:	674633
press:	46173

queens:	74
rsg:	0
turing:	22880
total:	1390990

The average number of characters per concatenation, omitting programs that do no concatenation, also is interesting:

csgen:	10.85
deal:	6.27
fileprnt:	20.21
genqueen:	19.00
ipxref:	135.40
kwic:	63.53
press:	3.28
queens:	37.00
turing:	8.00

These numbers tell us quite a bit about the test programs, but human beings have a hard time understanding comparative magnitudes when expressed as numbers. Lengths are easier to compare. Histograms for these figures, omitting the programs that do no concatenation, are shown below. Notice that the scales are different.

A Practical Problem

The monitoring procedure shown earlier writes the result lengths to standard output. Programs being monitored generally write to standard output also. Something needs to be done about this.

The obvious approach is to write monitoring output to a named file. This is somewhat cumbersome, however. It's simpler if the monitoring procedure writes to standard output as shown earlier and the data is redirected to an appropriate file by a script. This will work if the output of the program being monitored is suppressed. One way is to insert a line like

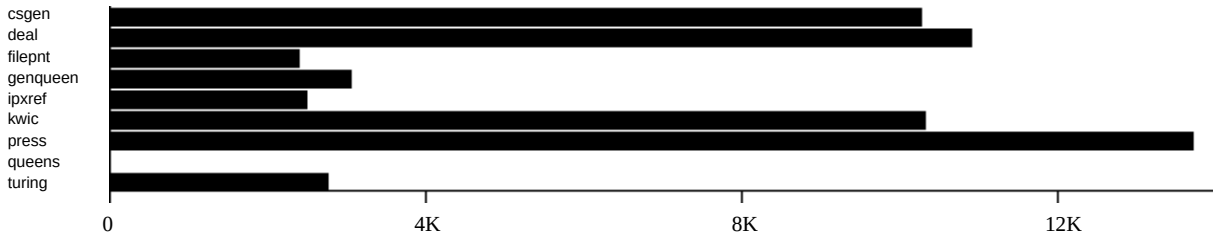
```
write := writes := –1
```

at the beginning of the program being monitored. This replaces the initial function values of the global variables write and writes by –1, so that a call of the form

```
write(s1, s2, ...)
```

becomes equivalent to

```
(–1)(s1, s2, ...)
```



Total Number of Concatenations

Total Number of Characters Produced by Concatenation

Average Number of Characters per Concatenation

The –1 selects the last argument in the argument list, which is what write and writes return after writing — except this way, nothing is written. Returning the last argument assures that expressions like

```
result := write(s1, s2, ...)
```

work properly. (There is a subtle problem with this, which we'll discuss in an article on crafting robust monitors.)

This method is not foolproof. For example, it doesn't handle situations in which the program itself changes write or writes, where the program provides a procedure by one of these names that does something other than write, or if the program relies on writing to a file and reading back the result.

You probably can think of other unusual situations that defeat this mechanism. Nonetheless, this approach is efficient and works with almost all programs. (There always will be programs that can't be monitored by any specific method. Designing monitor-proof programs that self-destruct when monitored is a fun game, as is finding ways of handling the various problems that can arise even for programs not out to foil monitoring.)

Editing a program to add a line to suppress output is not hard, but it's unnecessary. The code can be inserted by the meta-translator at the same time it replaces concatenation operators by calls to a monitoring procedure.

Here is the code-generation procedure that handles calls:

```
procedure Invoke(e, es[])                    # e(e1, e2, ...)
   local result

   if *es = 0 then return cat(e, "()")

   result := ""
   every result ||:= !es || ", "

   return cat(e, "(", result[1:–2], ")")

end
```

Here e is the procedure name, or more precisely, the expression applied to the argument list es.

All that's needed is to check for write and writes and replace them by (–1):

```
procedure Invoke(e, es[])                    # e(e1, e2, ...)
   local result

   if e == ("write" | "writes") then e := "(–1)"

   if *es = 0 then return cat(e, "()")

                  ...
```

Presto! Output from the program is gone. Note that the parentheses around –1 are needed, since

```
–1(s1, s2, ...)
```

parses as

```
–(1(s1, s2, ...))
```

which is almost certain to lead to disaster.

You may think of cases that this method doesn't handle that are handled by the manual insertion of

```
write := writes := –1
```

but, again, they are unlikely to occur in practice.

Other Uses for Monitoring Concatenation

We can study many things related to concatenation. We can, for example, write out the strings that result from concatenation instead of just their lengths. This allows us to study the strings produced by concatenation. And, of course, the lengths always can be determined from the strings. (A problem with writing strings may be the amount of data produced — our 11 test programs produce well over a megabyte of strings from concatenation. Writing the actual strings probably is not something to do unless they actually are needed.)

Writing the Results of Concatenation

Some care is needed in writing out the results of concatenation, since the strings may contain linefeeds and other characters that cause problems. The easy solution to is to use image(), which converts "nonprintable" characters to Icon escape sequences. The procedure moncat__() might look like this:

```
procedure moncat__(s1, s2)
   local s

   s := s1 || s2

   write(image(s))

   return s

end
```

Now we have the opposite problem: converting imaged strings back into the original strings. Icon program library to the rescue! It contains a

procedure ivalue() that does just what's needed. So a program to analyze imaged data might start like this:

```
link ivalue

procedure main()
   while s := ivalue(read()) do …
```

Using monitoring data of this kind, we counted the number of distinct strings each program produced by concatenation. Here are the results, with the total number of strings produced shown for comparison:

program	*total*	*distinct*
csgen:	10568	7625
deal:	11207	4962
fileprnt:	2492	1505
genqueen:	3168	1
iiencode:	0	0
ipxref:	2598	2281
kwic:	10619	523
press:	14086	2997
queens:	2	2
rsg:	0	0
turing:	2860	13
total:	57600	12909

Several things about these counts deserve mention. The reason turing produces so few different strings is that the test data consists of many repetitions of the same of the same Turing machine specification — not the best test. Note that this tabulation suggests this and points to a flaw in testing.

The two *n*-queens programs show a distinct contrast. Why should genqueen do so many more concatenations than queens? (Both produce essentially the same output.) We thought it might be a difference in the method of constructing the output, which displays a board for every solution for 9 queens, of which there are 352. Instead, the difference has a much simpler explanation.

Both programs construct a string representing a row with no queens and then insert Qs in places corresponding to a solution. Inserting the Qs is done by assignment to a subscripted position in the row, not by concatenation. The difference between the two programs is that genqueen builds a blank row for each row in each solution, while queens does it only once for the entire program (the board size is fixed for one execution of the program).

Here's the offending procedure from genqueen:

```
procedure writeboard ()
   local row, r, c
         …
   write (repl ("––", n), "–")

   every r := 1 to n do {
      c := rw [r]
      row := repl ("| ", n) || "|"
      row [2 * c] := "Q"
      write (row)
       write (repl ("––", n), "–")
      }

   write ()

end
```

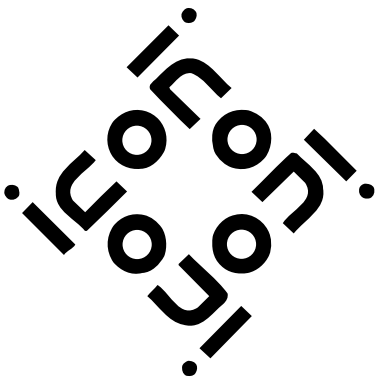
A simple change to use a static variable reduces the number of concatenations for the entire program to one:

```
procedure writeboard ()
   local row, r, c
   static bar

   initial bar := repl ("| ", n) || "|"

   write (repl ("––", n), "–")
         …
   every r := 1 to n do {
      row := bar
      c := rw [r]
      row [2 * c] := "Q"
         …
```

This one change reduces the execution time for the program by 7%.

A possible use for studying the number of distinct strings produced — or rather the number of duplicates produced — is in designing a strategy for string allocation.

Icon appends every newly created string to the end of the space already allocated for strings without any attempt to determine if the string already exists somewhere else among the strings previously allocated [8]. (There are a few heuris-



tics related to what's already at the end of the allocated space.)

The SNOBOL languages, ancestors of Icon, took a much different approach. They checked every newly created string to see if it already had been allocated. (This was done by using a hash table for storing strings.) As a result, they never allocated space for a string that already had been stored [9].

Hashing, of course, is a comparatively expensive process and it would seem that its cost would override the cost of duplicate allocation, which is paid for mostly in garbage collection. But there is little theoretical or empirical basis on which to decide whether one allocation strategy is better than another — or any other strategy. Having information about duplicates would be a start, although it would require knowing about all kinds of string creation, not just concatenation, and testing with a much larger suite of programs. Something to think about, perhaps.

The Operands of Concatenation

It's easy enough for moncat__() to provide information about operands, not just the results of concatenation. The histograms below show composite results for all 11 test programs.

As might be expected, there are more short right operands than left ones. This typically results from building up strings by appending successive values, as in

```
result := ""
every result ||:= expr
```

We'll show more evidence of this paradigm in another article.

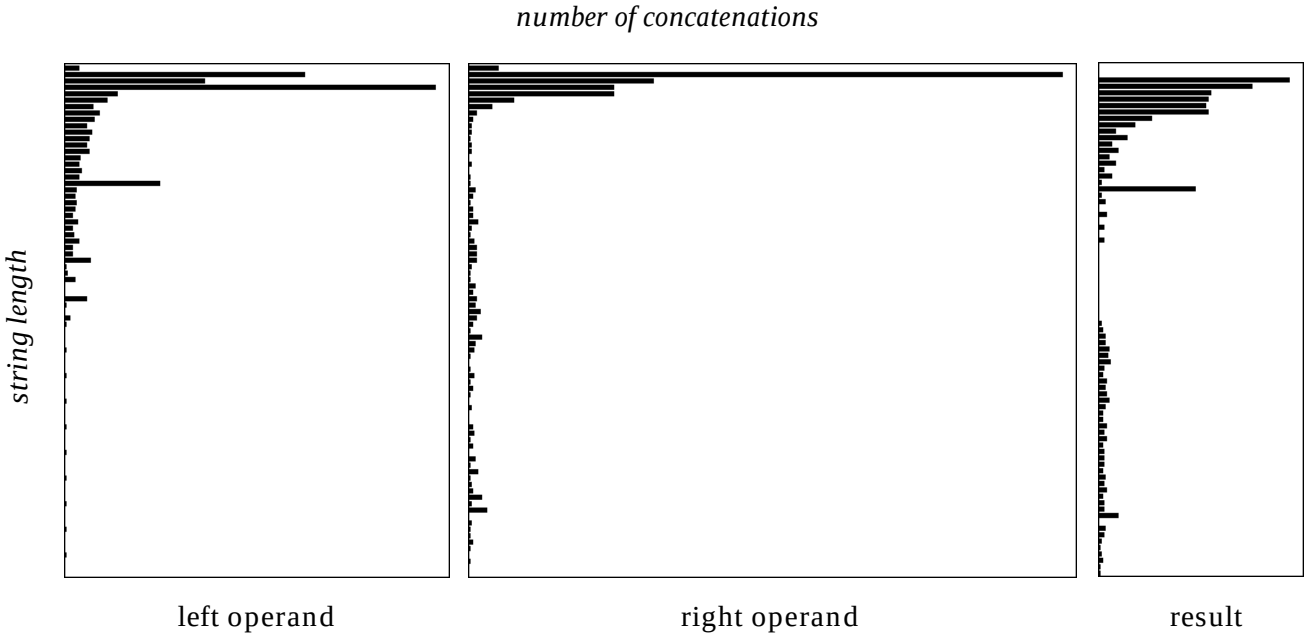
Other Possibilities

The procedure for monitoring concatenation could do many things. It could look for specific operands or results and, perhaps, alert a user to a situation that suggests a problem.

The procedure also could turn monitoring on or off under user control. Suppose you want to turn off monitoring after, say, 10,000 concatenations. One way is to simply bypass the output of monitoring information beyond that point. For the first version of moncat__() that we showed, this could be done as follows:

```
procedure moncat__(s1, s2)
   local s
   static count

   initial count := 0

   count +:= 1

   s := s1 || s2

   if count <= 10000 then write(*s)

   return s

end
```

There's a better way, and one that does not require calling moncat__() after the cutoff point:



```
procedure moncat__(s1, s2)
   local s
   static count

   initial count := 0

   count +:= 1

   s := s1 || s2

   write(*s)

   if count = 10000 then moncat__ := proc("||", 2)

   return s

end
```

In this version, the value of moncat__ is changed to the operator for concatenation, so that moncat__() itself never is called again.

Of course, you probably wouldn't want to build an arbitrary cutoff value into the procedure. What ways can you think of to specify a value at the time monitoring begins or interactively during monitoring?

Another approach to producing monitoring results is to have the monitoring procedure accumulate data internally rather than using post-processing as we've done for the results given in the article.

Next Time

Summary results show nothing about the *sequence* of events that occur during program execution. Yet it's often the sequence or a part of it that is most interesting and important in understanding program behavior. Furthermore, "seeing" concatenation as it occurs helps greatly in understanding what's going on.

We'll address animated visualizations in the next article.

References

1. "Dynamic Analysis of Icon Programs", Icon Analyst 28, pp. 9-11.

2 "Dynamic Analysis of Icon Programs", Icon Analyst 29, pp. 10-12.

3. "Dynamic Analysis", Icon Analyst 30, pp. 9-11.

4. "Dynamic Analysis", Icon Analyst 33, pp. 3-6.

5. *Algorithm Animation*, Mark H. Brown, The MIT Press, 1987.

6 "Perspectives on Algorithm Animation", Mark H. Brown, *Proceedings of the 1988 Conference on Human Factors in Computing Systems*, 1988, pp. 33-38.

7. "Meta-Variant Translators", Icon Analyst 23, pp. 8-10.

8 . "String Allocation", Icon Analyst 9, pp. 4-7.

9. *The Macro Implementation of SNOBOL4*, Ralph E. Griswold, W. H. Freeman and Company, 1972, pp. 96-100.

The Icon Analyst

Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend
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
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
U.S.A.

voice: (520) 621-6613

fax: (520) 621-4246

Electronic mail may be sent to:

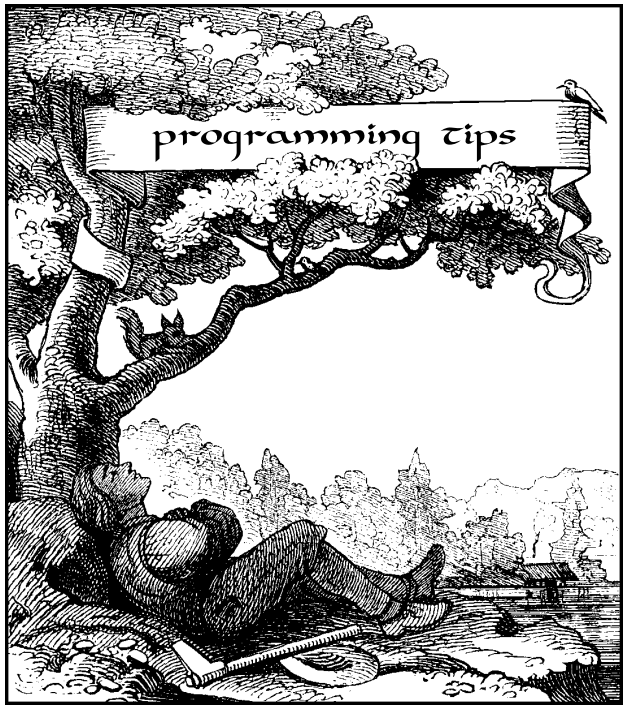
icon-project@cs.arizona.edu

THE UNIVERSITY OF
ARIZONA ®
TUCSON ARIZONA

and

Bright Forest Publishers

Tucson Arizona



Values in Rotation

In some situations a series of values is needed in rotation. Examples are cycling through colors for a display and servicing a number of queues.

If there are only a few values, they can be assigned to variables and rotated using Icon's exchange operation:

```
value1 :=: value2 :=: ... :=: valuen
```

Rather magically, it seems, every time this expression is evaluated, the values shift to the right and the former value of the last variable moves to the first variable.

To see how this works, suppose the value of x1 is 1, the value of x2 is 2, and the value of x3 is 3. All assignment operators associate to the right, so

```
x1 :=: x2 :=: x3
```

parses as

```
(x1 :=: (x2 :=: x3))
```

x2 :=: x3 therefore is evaluated first. As a result, the value of x3 is 2 and the value of x2 is 3. Since an assignment operation returns its left operand, the leftmost exchange operation is now equivalent to x1 :=: x2. After it is evaluated, x1 is 3, x2 is 1, and x3 is 2 — a circular right shift.

The same method works for the elements of a list:

```
value[1] :=: value[2] :=: ... :=: value[n]
```

Such expressions are tedious to compose if there are many values to be treated in this way. And, of course, they only rotate to the right, so for rotation to the left (the more common requirement), the values have to be arranged in reverse order. If the number of values is unknown when the program is written or changes during program execution, this method doesn't work at all.

The conventional way to rotate values to the left is to put the values in a list and increment an index into the list, using modular arithmetic to wrap around at the end:

```
n := *value
i := 1
repeat {
   # use value[i] ...
   i +:= 1
   i %:= n
   }
```

There is an easier (and niftier) way to do this in Icon:

```
repeat {
   v := get(value)
   # use v ...
   put(value, v)
   }
```

This method uses value as a queue, removing the first value and putting it back on the end.

To rotate to the right, all that's needed is:

```
repeat {
   v := pull(value)
   # use v ...
   push(value, v)
   }
```

Incidentally, you don't have to worry about these methods causing allocation and using up time and space — they don't. Icon lists are implemented as circular queues, and an internal index references the elements in a circular fashion. "Removing" an element from, say, the beginning of a list doesn't actually remove it unless it's the only element; it just changes an index. See Reference 1 for details of how this is done.

Reference:

1. *The Implementation of the Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Princeton University Press, Princeton, New Jersey, 1986, pp. 80-91.

Versum Predecessors

In the article on equivalent versum sequences [1], we noted that the same versum number can have two quite different predecessors. For example, 1111 has the predecessors 209 and 1010. Thus, both 209 and 1010 are seeds of the same versum sequence.

We, however, defined equivalence for seeds so that it applies only to numbers with the same number of digits. By this definition, 209 and 1010 are not equivalent seeds, although they produce equivalent sequences. The justification for this definition was based on two observations: (1) such "inequivalent" seeds are relatively rare and (2) working with numbers having the same number of digits produces more meaningful results. We can "take care of" this awkwardness by introducing the concept of *weak equivalence* to include all numbers that have the same reverse sums. The procedure vpred() given in the last article [2] generates the weakly equivalent primary predecessors of a number.

Now we will look at this problem from a different perspective. In the first place, we'll define *equivalent versum numbers* in the same way as we defined equivalent seeds — after all, all versum numbers are seeds, and whether or not a versum number is considered as a seed or just as a number in a versum sequence is immaterial.

To simplify the following discussion, we'll define the *versum successor* of a number to be the result of its reverse addition. Thus, 1111 is the successor of both 209 and 1010.

Since the versum equivalents of a predecessor of a number are also predecessors of that number, we'll define the *primary predecessors* of a versum number to be the primaries of its predecessors. For example, 209 is the primary of the equivalence class containing 209, 308, 407, 506, 605, 704, 803, and 902, while 1010 is the primary of the equivalence class containing 1010 and 1100. In other words, 1111 has (at least) two different primary predecessors.

Now we can frame the questions given at the end of the last article on versum numbers [2] in a more precise way:

1. What is the maximum number of primary predecessors a versum number can have?

2. How many versum numbers have more than one primary predecessor?

Some versum numbers have only one primary predecessor (11 is an example) and others have at least two, as shown above. It's obvious that the predecessor of an *n*-digit versum number can only have *n* digits (with no carry from the lead digit on reverse addition) or (*n*–1) digits (with a carry on reverse addition to produce an *n*-digit successor). But by definition, all the *n*-digit numbers that have the same successor are in an equivalence class, and similarly for (*n*–1)-digit numbers. So a versum number can have at most two primary predecessors and we can dispense with the first question above. We can safely refer to versum numbers with more than one primary predecessor as *bimorphs*.

All bimorphs must begin with a 1, since their (*n*–1)-digit predecessors produce a carry on reverse addition. But their *n*-digit predecessors must start with a 1 and end in a 0 to produce an initial 1 without a carry on reverse addition. Thus bimorphs have the form 1*x*1, their *n*-digit predecessors have the form 1*x*′1, and their primary (*n*–1)-digit predecessors have the form 2*x*′′9.

How do we produce bimorphs? A brute-force approach can be used at the start, hoping that the results will suggest a better way of constructing more.

A naive approach is to generate all 1*x*1 versum numbers from 1*x*′0 and 2*x*′′9 primary versum numbers and look for duplicates:

```
# Program 1

link pvseeds

procedure main(args)
   local n, candidate, k, all, bimorphs

   n := (0 < integer(args[1])) |
     stop("*** invalid specification")

   all := set()
   bimorphs := set()

   every k := (n – 1 | n) do
      every candidate := pvseeds(k) do {
         candidate +:= reverse(candidate)
         if *candidate > n then break next   # too far
         if *candidate = n & check(candidate) then {
            if member(all, candidate) then
               insert(bimorphs, candidate)
            else insert(all, candidate)
            }
         }

   every write(!sort(bimorphs))

end
```

```
procedure check(candidate)

   if candidate[1] == "1" == candidate[–1] then return
   else fail

end
```

The procedure check() determines if a candidate is a bimorph. A procedure is used to make changes easier in subsequent programs.

This approach works well for small values of *n*, but the number of 1*x*1 versum numbers that have to be kept in the set all becomes impossibly large as *n* increases. Various methods can be used to improve performance, but the approach is hopeless.

We can trade memory for file space, which generally is more plentiful than memory, by writing out the 1*x*1 versum numbers and looking for duplicates later. This sounds good, but again the number of 1*x*1 versum numbers becomes overwhelming and finding duplicates in huge files is not that easy.

An alternative approach is to generate all 1*x*1 numbers (we know no way to generate only 1*x*1 versum numbers [2]) and use vpred() to find bimorphs:

```
link vpred

# Program 2

procedure main(args)
   local n, candidate, bimorph

   n := digits(args, 2)

   bimorphs := set()

   every candidate := "1" ||
      right((0 to (10 ^ n) – 1), n, "0") || "1" do
         insert(bimorphs, check(candidate))

   every write(!sort(bimorphs))

end

procedure digits(args, m)
   local n

   if (n := integer(args[1]) – m) & (n >= 0) then
      return n
   else stop("*** invalid specification")

end

procedure check(candidate)
   local count

   count := 0

   every vpred(candidate) do
      count +:= 1
   if count = 2 then return candidate else fail

end
```

The procedure digits() serves to reduce the number of digits for which a "core" must be generated exhaustively. Notice that the procedure check() is different from the earlier one.

There are of course, a very large number of candidates and this approach becomes hopelessly time-consuming as *n* grows large.

By a closer examination of how bimorphs are formed and a rather tedious argument about different cases, we can show that bimorphs must have one of these forms:

10*x*01	10*x*11	10*x*21
11*x*01	11*x*11	11*x*21
12*x*01	12*x*11	12*x*21

Using this information, we can reduce the search space considerably:

```
# Program 3

procedure main(args)
   local n, core, candidate, bimorphs

   n := digits(args, 4)

   bimorphs := set()

   every core := right((0 to (10 ^ n) – 1), n, "0") do
      every candidate := (("10" | "11" | "12") || core ||
       ("01" | "11" | "21")) do
          insert(bimorphs, check(candidate))

   every write(!sort(bimorphs))

end
```

This is better, but the computation still is hopelessly time-consuming for even moderate values of *n*.

An alternative approach is to use 2*x*'9 predecessors to produce 1*x*1 candidates. This has the advantage of reducing the number of digits for the core by 1:

```
# Program 4

procedure main(args)
   local n, pred, candidate, bimorphs

   n:= digits(args, 3)

   bimorphs := set()

   every pred := "2" || right((0 to (10 ^ n) – 1), n, "0") ||
      "9" do {
```

```
      candidate := pred + reverse(pred)
      insert(bimorphs, check(candidate))
      }
  every write(!sort(bimorphs))
end

procedure check(candidate)
  if candidate[1:3] == ("10" | "11" | "12") &
    candidate[–2:0] == ("01" | "11" | "21") then return
  else fail
end
```

We can make further improvements by determining that the 2*x*′9 predecessors must have one of the following forms:

20*x*09 20*x*99 21*x*99 22*x*99

```
# Program 5

procedure main(args)
  local n, pred, core, candidate, bimorphs
  n := digits(args, 5)
  bimorphs := set()
  every core := right((0 to (10 ^ n) – 1), n, "0") do {
    every pred := ("20" || core || ("09" | "99")) |
      (("21" | "22") || core || "99") do {
        candidate := pred + reverse(pred)
            insert(bimorphs, check(candidate))
        }
    }
  every write(!sort(bimorphs))
end
```

We can do even better than this by noting that the 1*x*′′0 bimorph predecessors for even *n* are divisible by 110 (11 × 10). The factor of 10 is obvious, since their last digit is 0. The factor of 11 may seem mysterious, but we'll show why it's there in a subsequent article on the factors of versum numbers.

Combining this information with the possible patterns of numbers of the form (1*x*′′0)/110 that can have bimorph successors, yields this program:

```
# Program 6

procedure main(args)
  local n, core, candidate, bimorphs
   n := digits(args, 7)
  bimorphs := set()
  every core := right((0 to (10 ^ n) – 1), n, "0") do {
    every candidate :=
      ("90" || core || ("01" | "02" | "91")) |
      ("91" || core || ("81" | "82" | "91" | "92")) |
      ("92" || core || "81") |
      ("99" || core || "90") * 110 do {
        insert(bimorphs,
          check(candidate + reverse(candidate)))
        }
    }
  every write(!sort(bimorphs))
end
```

Comparative timings, in seconds on a DEC Alpha 200 4/233 show how much is gained by increasingly sophisticated approaches:

	program					
n	1	2	3	4	5	6
4	0.05	0.09	0.02	0.02	–	–
5	0.20	0.89	0.10	0.05	0.02	–
6	0.62	18.33	1.90	0.76	0.10	–
7	3.69	218.40	23.70	10.60	1.41	0.04
8	12.20	?	?	222.76	26.39	–
9	74.57	?	?	?	432.58	6.69

The dashes indicate values for which the program does not apply. The question marks indicate values for which timings were not done — for the obvious reason. Note that although Program 1 compares favorably with others on the issue of speed, it is not usable because of the amount of memory it requires.

There are several things we could do to improve the performance of these programs: put the procedure code in line, write a procedure to replace vpred() that would just test for two predecessors, instead of generating them, and so on.

We could improve performance by further refining the possible patterns, but we haven't done that (yet). There is, however, a heuristic we did try: An examination of bimorphs and their predecessors shows that none contain a 3, 4, 5, or 6. Using this heuristic reduces the time for Program 6 and *n*=9 to 2.66 seconds. We haven't tried to *prove* that these digits don't appear, but using the heuristic gives the correct results through *n* = 16.

All that having been said, here are counts of bimorphs through *n* = 16. We've included counts of versum numbers of the form 1*x* and 1*x*1 for comparison:

versum numbers

n	*all*	1*x*	1*x*1	*bimorphs*
1	4	0	0	0
2	14	6	1	0
3	93	13	5	1
4	256	104	19	1
5	1793	273	112	2
6	4872	1984	369	2
7	34107	5227	2159	7
8	92590	37718	7033	7
9	648154	99434	41133	21
10	1759313	716745	133730	21
11	$\sim 1.2\times10^7$	1889589	781861	65
12	$\sim 3.3\times10^7$	$\sim 1.4\times10^7$	$\sim 2.5\times10^6$	65
13	$\sim 2.3\times10^8$	$\sim 3.6\times10^7$	$\sim 1.5\times10^7$	200
14	$\sim 6.3\times10^8$	$\sim 2.6\times10^8$	$\sim 4.8\times10^7$	200
15	$\sim 4.4\times10^9$	$\sim 6.8\times10^8$	$\sim 2.8\times10^8$	616
16	$\sim 1.2\times10^{10}$	$\sim 4.9\times10^9$	$\sim 9.2\times10^8$	616

The approximate counts given in exponent form were obtained by recurrences like the one given in the last article [2]. Note that the number of bimorphs for successive odd / even numbers of digits are the same.

As you can see, the number of versum bimorphs is indeed quite small even compared to only those versum numbers that begin and end with a 1.

As in other aspects of versum numbers we've studied, there are evident digit patterns in bimorphs. Here are bimorphs and their predecessors though *n* = 9:

	bimorphs	*predecessors*	
n=3:	121	29	110
n=4:	1111	209	1010
n=5:	11011	2009	10010
	12221	2299	10120
n=6:	110011	20009	100010
	121121	22099	100120
n=7:	1100011	200009	1000010
	1112111	202909	1001110
	1197801	200799	1098900
	1208911	210899	1009910
	1210121	220099	1000120
	1211111	211999	1020910
	1222221	222999	1001220
n=8:	11000011	2000009	10000010
	11111111	2020909	10001110
	11988801	2009799	10989900
	12098911	2109899	10099910
	12100121	2200099	10000120
	12101111	2110999	10200910
	12211221	2220999	10001220
n=9:	110000011	20000009	100000010
	110121011	20029009	100011010
	111089011	20108909	100099010
	111101111	20200909	100001110
	111111011	20119909	100209010
	111222111	20229909	100012110
	119777801	20007799	109789900
	119898801	20099799	109899900
	120877911	21007899	100889910
	120989011	21008999	101098910
	120998911	21099899	100999910
	121000121	22000099	100000120
	121001111	21100999	102000910
	121011011	21019999	101109910
	121121121	22029099	100011120
	121122111	21129999	102011910
	121978021	22007999	100989020
	122089121	22108999	100099120
	122101221	22200999	100001220
	122111121	22119999	100209120
	122222221	22229999	100012220

Despite the patterns, we haven't found a rule for generating bimorphs. If you find one, please let us know.

One thing that our knowledge about the nature of bimorphs provides is improvements to vpred(). Just knowing that there are at most two primary predecessors of a versum number, which can happen only for versum numbers of the form 1*x*1, can be used to make the procedure considerably more efficient. The original procedure, with minor typographical changes, is:

```
procedure vpred(i)
   local j, preds

   if i < 1 then fail

   preds := set()

   every j := integer(vpred_(i)) do {
      if (j + reverse(j)) = i then
         insert(preds, vprimary(j))
         }

   suspend !sort(preds)

end
```

Using the observations above, this procedure can be rewritten as follows:

```
procedure vpred(i)
   local j, preds

   if i < 1 then fail

   if i[1] == "1" == i[–1] then {          # may be two
```

```
      every j := integer(vpred_(i)) do {
        if (j + reverse(j)) = i then
          insert(preds, vprimary(j))
        if *preds = 2 then break
        }
      suspend !sort(preds)
      }
    else {
      every j := integer(vpred_(i)) do {
        if (j + reverse(j)) = i then
          return vprimary(j)
        }
      fail                                  # none
      }
end
```

The set for numbers of the form 1*x*1 can be eliminated by keeping track of the first primary predecessor and comparing it with subsequent ones:

```
    local j, firstp
          …
    if i[1] == "1" == i[–1] then {          # may be two
      every j := integer(vpred_(i)) do {
        if (j + reverse(j)) = i then {
          j := vprimary(j)
          (/firstp := j) | {
            if j ~= firstp then {
              suspend firstp
              return j
              }
            }
          }
        }
      return \firstp                        # may be none
          …
```

An open question is whether there is a direct way to compute one predecessor of a bimorph from the other. If there is, vpred() could be considerably simplified and made to run much faster.

Next Time

Given past history, it should be no surprise that there are more articles on versum numbers in the works. We have more material on versum bimorphs. After that, we'll discuss the factors of versum numbers and versum primes.

References

1. "Equivalent Versum Sequences", Icon Analyst 32, pp. 1-6.

2. "Versum Numbers", Icon Analyst 35, pp. 5-11.

Icon Glossary

This is the last section of the glossary of Icon terms that we're compiling. We have a few things to add and a number of things to clean up. When we've completed that, we'll put it all together and publish it as an Icon Project document. We'll include a copy of it with an upcoming issue of the Analyst.

path: a specification for the location of a file. Paths are used for locating **library modules** and **include files**.

polymorphous operation: an **operation** that applies to more than one **data type**. The size **operator**, *X, is an example.

procedure: a computational unit whose definition is cast in the form of an **identifier**, which names the procedure, followed by a list of **parameters** to be used in the computation. The term procedure includes both **built-in** procedures (also called **functions**) and **declared** procedures, but sometimes it is used in the more restricted sense of the latter.

procedure return: leaving the **invocation** of a **procedure**. When a **procedure return**s, it may produce a **result** or **fail**. A **procedure** also may **suspend** with a **result**, in which case the **procedure** can be **resumed**. **Variables** that are **local** to the **procedure** remain intact when it **suspends** and are available if the **procedure** is **resumed**.

record constructor: a **function** that creates an instance of a **record**. A record constructor is provided automatically for every **record declaration**.

record declaration: a declaration that defines a **record**.

run-time error: an **error** that occurs during program **execution**. Run-time errors cause program **termination** unless **error conversion** is enabled.

scope: the extent in time and location in which a **variable** is accessible. There are three kinds of scope: **global**, **local**, and **static**.

storage: see **memory**.

string image: A **string** that describes a value.

string: a sequence of **characters.** Strings are values in their own right, not arrays of **characters**.

string invocation: the **invocation** of a **function**, **procedure**, or **operator** by its **string name**.

string name: a **string** that identifies a **function**, **procedure**, or **operator**. The string name for a **function** or **procedure** is just the name by which it is used. The string name for an **operator** resembles the symbols that designate the **operator**.

string scanning: high-level **string** analysis using the concepts of a **subject string** and movement of a **cursor** position in it.

substring: a **string** within a **string**.

success: evaluation of an expression that produces a **result**; the opposite of **failure**.

suspension: interruption of the evaluation of a **generator** when a **result** is produced. See also **resumption**.

syntax error: a grammatical **error** in a program. Syntax errors are detected during **translation**.

table: a **data structure** composed of **key/value** pairs, in which **keys** are distinct. Tables can be **subscripted** by **keys** to assign or access corresponding values. Table **subscripting** produces **variables**.

table lookup: referencing a **table** by a **key** to produce the corresponding value. If the **table** does not contain the **key**, the **default table value** is produced.

termination: the end of **execution**.

thrashing: a situation in which **garbage collection** occurs frequently because the amount of available **memory** is small.

transmission: passing a value to a **co-expression** when it is **activated**.

traceback: a listing of **procedure calls** leading to the evaluation of the current **expression**. A traceback is provided when a program **terminates** with a **run-time error**, or in any event if **termination dumping** is enabled.

translation: the process of converting Icon source code to code for an imaginary machine (**virtual machine**). The result of translation of a source code file is a pair of **ucode** files. See also **compilation**.

translator: the program that translates Icon source code into **ucode**.

type: see **data type**.

type conversion: converting a value from one **data type** to another. Type conversion occurs automatically when a value is not of the expected **type**; this is called **implicit type conversion** or **coercion**. Type conversion can be performed explicitly by type-conversion **functions**. If **implicit type conversion** cannot be done, a **run-time error** occurs. If **explicit type conversion** cannot be done, the type-conversion **function fails**.

ucode: the result of **translating** Icon source code into code for a **virtual machine**. Ucode files are readable text.

unary operator: an **operator** with one **operand**. See also **prefix operator**.

undefine directive: a **preprocessor directive** that removes a **preprocessor definition**. See also **define directive**.

variable: a reference to a value and to which **assignment** can be made. There are several kinds of variables, including **identifiers**, some **keywords**, the **elements** of **records** and **lists**, and **table subscripts**. See also **dereferencing**.

What's Coming Up

We have a lot on deck. We'll continue the series on building applications with visual interfaces with an article about the code VIB produces for an interface and how to connect it to the code that runs the application itself.

We'll also continue with dynamic analysis, showing how concatenation can be visualized.

We have more on versum bimorphs in the wings and an interesting article on random number generation that came by way of a letter from a subscriber.

It looks like the next Analyst will run 16 pages again.