The Icon Analyst

In-Depth Coverage of the Icon Programming Language

April 1998
Number 47

In this issue …

Assault on Mount Versum 1
Exploring Carpet Space 5
Graphics Corner 10
From the Library 13
Tricky Business 14
What's Coming Up 16
Feedback? 16

Assault on Mount Versum

The main impediment to studying versum numbers has been the difficulty of identifying them: determining whether or not a number is versum — the sum of a number and its reversal.

Some numbers have digit patterns, notably palindromes, that are easy to handle and a determination can be made quickly, independent of the length of the number. Some numbers also can be rejected quickly using the "first-last" test [1] — and again in time independent of their length.

For the rest, we have been left to use a procedure, vpred(), that in the worst case tries possible solutions working from the ends all the way to the middle, recursively. This procedure does more than is needed to make a determination — it generates the predecessors. Generating predecessors does not, however, appreciably increase the time required to test for versumness, since only one predecessor is needed for this.

The time required to test for versumness in this fashion increases exponentially with the length of the string for worst-case data. This has been observed experimentally; mathematical analysis is intractable. In other words, the time required has the form

$$t = c \times k^n$$

where c is a constant that depends on the platform, k is a platform-independent constant that is about 2.02. Figure 1 shows why this time complexity limits versum testing in this manner to relatively small integers.

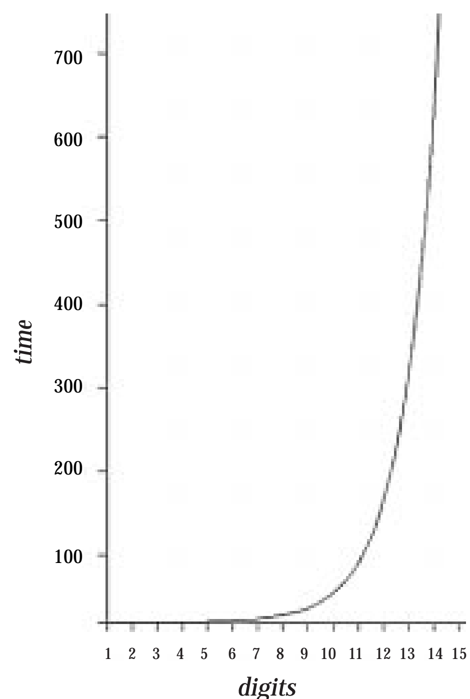


Figure 1. Testing for Versumness

On the platform we're using, versum testing for $n > 25$ is impractical. For particular numbers of interest, n can be somewhat larger — we just wait longer — but 25 is the practical limit for general testing.

The ideal solution to this problem is to find a better algorithm — one with a better time complexity. We have not found one and we doubt a fast algorithm exists.

So we decided to start by seeing what we could do to improve the performance of vpred() without fundamentally changing the method it uses. Note that this effort does not propose to change the time complexity; the most it can do is reduce the constants. In other words, it might

increase the value of *n* for which the time required is tolerable. However successful this effort might be, the fundamental problem remains.

What follows is a description of what we did — a case history. The approach we took was not the ideal one, perhaps. One can even argue that in the face of the known time complexity, it was not worth doing.

The Structure of the Procedures

In order to understand what follows, it's necessary to know a little about the approach and the procedures involved.

The approach is to segregate numbers according to their initial digit: 0, 1, and 2-9. An initial 0 arises in internal calculations and requires special processing. Numbers that begin with 1 require a separate approach because of the possibility that the initial 1 is the result of a carry on reverse addition. Numbers that begin with a digit greater than 1 all can be treated the same way [1].

Except for numbers with initial digit 0, the approach is to look for possible predecessors for the middle portion between the first and last digits — working on the middle portion of a string that is two digits less than the original one:

first-digit middle-portion last-digit

The procedures called depend on the first digit. Figure 2 shows the call graph.

vpred() is the "root" procedure — a wrapper for the procedures that do the actual work. All vpred() does is make some simple checks before calling vpred_() and then verifies its results.

vpred_() calls other procedures depending on the value of its argument.

vpred_1(), vpred_2(), and vpred_3() produce the predecessors of 1-, 2-, and 3-digit numbers, respectively. These "ground" the computation.

vpred_i0(), vpred_i1, and vpred_i2() handle larger numbers whose initial digit is 0, 1, or 2–9, respectively

These procedures in turn call vpred_noinc() and vpred_inc() to look for predecessors for middle portions that do not and do increase their length, respectively. The latter correspond to predecessors that produce a carry. These procedures call vpred_(), which is where the recursion arises.

Dynamic Analysis

The first thing we did was to apply some of dynamic analysis tools described in earlier Analyst articles [2-5] to try to get a better understanding what goes on during the computation.

We started by looking at procedure calls. Here is a typical example for a 20-digit number:

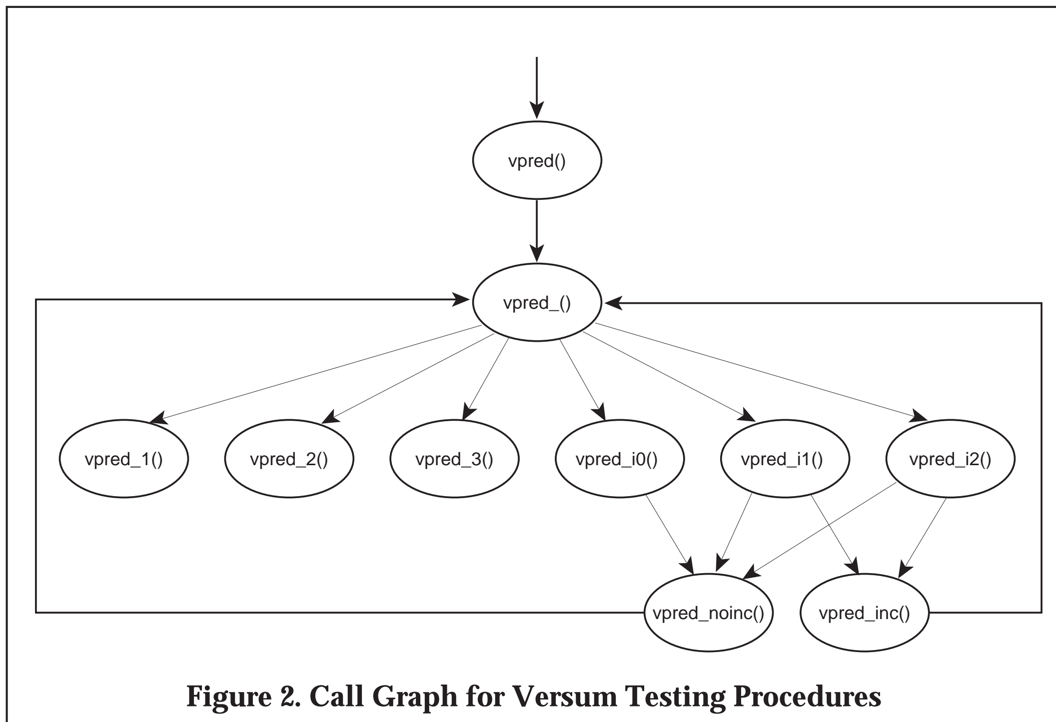


Figure 2. Call Graph for Versum Testing Procedures

procedure	calls	rets	fails	susps	resums	removes
vpred	3	0	3	0	0	0
vpred_	47503	0	47503	18032	18032	0
vpred_1	4536	0	4536	0	0	0
vpred_2	16968	4536	12432	0	0	0
vpred_3	11592	0	11592	0	0	0
vpred_i1	11794	0	11794	13432	13432	0
vpred_i2	2613	0	2613	64	64	0
vpred_inc	23714	0	23714	13384	13384	0
vpred_noinc	23786	0	23786	84	84	0

The main observation is that there are a lot of procedure calls, with vpred_() being the "winner" as expected.

Our expectations confirmed, we overlooked the obvious and put procedure calls aside temporarily to look at resource usage.

When looking at the program, type conversions stood out. The procedures use both string and integer representations of numbers, requiring conversion back and forth. In addition, when we first wrote the procedures, we were rather cavalier about type conversion, letting it happen automatically and unnecessarily. For example, there were instances of expressions such as this:

```
last || vpred_inc(1 || middle) || 0
```

To avoid an unnecessary type conversion, this obviously should be

```
last || vpred_inc("1" || middle) || "0"
```

We systematically revised the code to avoid this kind of problem. We also observed that some repetitious automatic (implicit) type conversions could be avoided by explicit conversion, as in string(i). Here are counts of type conversions before and after our modifications for the same number used in analyzing procedure calls:

before

Explicit successful conversions:	ii	112127
Implicit successful conversions:	is	158036
Implicit conversion no–ops:	ss	93613

after

Explicit successful conversions:	ss	192
Explicit conversion no–ops:	ss	588
Implicit successful conversions:	is	1048
Implicit conversion no–ops:	ss	6587

The types before and after conversion are given by letter pairs. For example, is indicates the conversion of an integer to a string. The "no-ops" occur when there is an implicit attempt to convert a value to the type it already has.

To fully understand what's going on, you'd have to examine the program in detail. The overall picture is clear: There are far fewer type conversions as the result of the changes. The changes affect execution time somewhat; it improved about 6%.

An Accidental Discovery

While working on improving the usage of types, we found a variety of other "little things" that needed fixing and we went over the program thoroughly, looking for code that could be improved.

Having done this, we decided some testing was in order to be sure we hadn't broken anything.

Our usual tests went without a hitch. Then, thinking of the article on versum primes in the last issue of the Analyst [6], we tried vpred() on the (then) largest known prime (a Mersenne prime with 895,932 digits). This was "safe", since the prime begins with a 6 and ends with a 1, and hence fails the first-last test and should be rejected immediately.

When we tried this, nothing happened. The program just sat there, apparently hung. The obvious possibility was something wrong with the first-last test, since without it, the program would run for an astronomical amount of time.

Checking the code, which is very simple, showed no problem. We briefly suspected something was wrong with Icon (a suspicion we know we should ignore, especially in cases like this).

We turned on tracing and to our amazement, the program "hung" in vpred() — it didn't even get to vpred_(). How could that be?

The first line of vpred() is a test to make sure the number is positive:

```
if i < 1 then fail
```

Aha! In this case i was a string (read from a file). The seemly innocuous comparison requires converting the integer to a string. That's a slow process for a really large integer — quadratic in the number of digits. In other words, the program was working; working hard.

We changed the test to this rather awkward expression:

```
if i[1] == ("–" | "0") then fail
```

and, sure enough, “instant rejection”.

Since we hope to test large integers for versumness, we had to consider the fundamental problem. We found many cases in which arithmetic was performed on strings. However, we couldn’t reverse course and make integers the preferred form, since there were many more string operations than integer operations, and conversion between large integers and strings is slow either way. We did what we could do easily, but there still are places, such as the division of strings by 2, that need to be replaced by symbolic operations. Later. In any event, the improvement in speed for some very large numbers (notably ones that are rejected) was improved vastly.

A Design Change

For reasons we’ve forgotten, the last thing vpred() does when it finds a versum predecessor is to convert it the corresponding versum primary [7]. This has the advantage of producing canonical forms, but it also can be very time consuming. Since the caller of vpred() can always compute versum primaries from what vpred() produces, we eliminated the conversion and got another noticeable improvement in speed (for numbers that are versum).

More Dynamic Analysis

As mentioned earlier, we tabulated procedure calls. This told us there were a lot of them, but it gave no insight as to the pattern or depth of recursion of calls.

At this point we resorted to tracing and analyzing the output, which is voluminous for anything but small numbers or quick rejections. Typical output looks like this:

```
vpred.icn :    37 | | vpred_("1276478784635844...")
vpred.icn :    75 | | | vpred_i1("1276478784635844...")
vpred.icn :   142 | | | | vpred_noinc("2764787846358441...")
vpred.icn :   192 | | | | | vpred_("2764787846358441...")
vpred.icn :    76 | | | | | | vpred_in("2764787846358441...")
vpred.icn :   178 | | | | | | vpred_in failed
vpred.icn :    80 | | | | | vpred_ failed
vpred.icn :   194 | | | | vpred_noinc failed
vpred.icn :   143 | | | | vpred_noinc("7647878463584414...")
vpred.icn :   192 | | | | | vpred_("7647878463584414...")
vpred.icn :    76 | | | | | | vpred_in("7647878463584414...")
vpred.icn :   176 | | | | | | | vpred_inc("1647878463584414")
vpred.icn :   185 | | | | | | | | vpred_("1647878463584414")
vpred.icn :    75 | | | | | | | | | vpred_i1("1647878463584414")
vpred.icn :   157 | | | | | | | | | | vpred_noinc("4787846358440")
vpred.icn :   192 | | | | | | | | | | | vpred_("4787846358440")
vpred.icn :    76 | | | | | | | | | | | | vpred_in("4787846358440")
vpred.icn :   178 | | | | | | | | | | | | vpred_in failed
vpred.icn :    80 | | | | | | | | | | | vpred_ failed
vpred.icn :   194 | | | | | | | | | | vpred_noinc failed
vpred.icn :   158 | | | | | | | | | | vpred_noinc(14787846358440)
vpred.icn :   192 | | | | | | | | | | | vpred_(14787846358440)
vpred.icn :    75 | | | | | | | | | | | | vpred_i1(14787846358440)
vpred.icn :   149 | | | | | | | | | | | | | vpred_noinc("78784635843")
vpred.icn :   192 | | | | | | | | | | | | | | vpred_("78784635843")
vpred.icn :    76 | | | | | | | | | | | | | | | vpred_in("78784635843")
                                  …
```

We decided that the depth of recursion was worth looking at, so we processed trace output with this simple program to produce depth numbers:

```
procedure main()

   while line := read() do
      line ? {
         i := 0
         every tab(upto('|')) do
            i +:= 1
         write(i)
         }

end
```

The numbers, although less voluminous than the trace output, were, if anything, less comprehensible. To understand the data, we needed a visual representation of it. Note that the indentation of the trace output provides a visual representation, although in that form it is hard to see the “big picture”.

We piped the numbers into a simple program that displays a scrolling histogram. Figure 3 shows what a (small) segment of the output looks like:

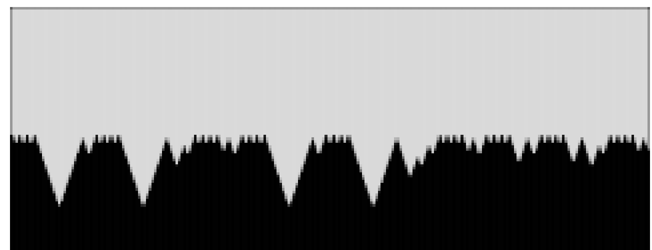


Figure 3. Visualization of Call Depth

The “peaks” and “valleys” suggest long sequences of recursive calls to vpred_() that ulti-

mately are futile. Although this short segment doesn't show it, the scrolling histogram gives the impression of repeated patterns.

In other words, it might be that the same computation is being performed repeatedly. We analyzed the trace output to tabulate the arguments with which vpred_() was called using this simple program:

```
procedure main()
   local count, line, arg, n

   count := table(0)

   while line := read() do
      line ? {
         tab(find("vpred_(") + 7) | next
         ="\""              # remove quote if string
         count[integer(tab(upto('")')))] +:= 1
         }

   count := sort(count, 4)

   while n := pull(count) do
     write(left(pull(count), 25), right(n, 8))

end
```

Comment: Programs like this show the value of a high-level language with good string-processing capabilities. This program took only a few minutes to write. We probably wouldn't have bothered doing it in C.

As expected, we found vpred_() called repeatedly with the same argument. An example is:

argument	*calls*
15	9072
145	6720
16	4536
5	4536
146	3360
45	3360
1462	3024
18462	2240
462	1512
1463	1512
18463	1120
8462	1120

...

It finally occurred to us to do what we should have considered doing earlier: Add memory to store results and then check arguments to see if they already had been processed, thus avoiding redundant computations [8-9].

Next Time

We'll continue the assault on Mount Versum in the next Analyst, starting by adding memory to the procedures.

References

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

2. "Dynamic Analysis of Icon Programs", Icon Analyst 28, pp. 9-12.

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

4. "Dynamic Analysis", Icon Analyst 30, pp. 6-11.

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

6. "Versum Primes", Icon Analyst 46, pp. 12-16.

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

8. "Procedures with Memory", Icon Analyst 21, pp. 8-12.

9. "Procedures with Memory", Icon Analyst 22, pp. 1-4.

Exploring Carpet Space

When we were developing the article on numerical carpets [1], we found situations in which we wanted to vary a parameter to see what effect that would have. For example, we varied the modulus from 2 through 17 with "lone-one" initialization to show how strikingly the modulus affects the pattern. We did this manually, creating 16 carpets, one by one.

When we found that moduli 4 and 8 produced interesting carpets with prime-sequence initialization, we tried other moduli, starting with ones of the form 2^n. This didn't turn up anything interesting, so we thought about trying a lot of moduli, such as 2 through 100. Doing that one by one, however, would have required more time and

effort — not to mention tedium — than we were willing to tolerate.

This is a case where the flexibility and generality of the carpet-specification program wasn't needed but another kind of programming help was.

The concept is simple: Something like

```
every Modulus := 2 to 100 do
   # create carpet
```

In fact, we created an *ad hoc* program to generate carpets in the background and used it to create many carpets in which one parameter of the specification took on a range of values.

For what it's worth, we didn't find anything interesting for prime-sequence initialization, but the idea of being able to explore *carpet space* was intriguing. This led to the application described in this article.

Carpet Space

There's more to exploring carpet space than just varying the modulus. Carpet space, as we have characterized it, is 7-dimensional:

width
height
modulus
top initializer
left initializer
neighbors
colors

The first three are simply numerical and their likely ranges are comparatively small. The remaining four, however, are more complex and can reasonably take on a vast number of values. The idea of even describing portions of 7-dimensional carpet space is daunting; concepts and strategies are needed. And, of course, programming help is necessary.

An Approach to Exploring Carpet Space

What we decided to do was to design a pair of programs using the general idea of the carpet-specification and carpet-generation programs [1], one program to produce specifications for portions of carpet space and the other to produce corresponding individual carpet specifications.

This is a grand idea, but the question is how to accomplish it in a manageable way. We settled for an application that allows the specification of alternative values for parameters. The result is a "meta-specification" from which a database of carpet specifications is obtained from combinations of parameter alternatives .

Conceptually, a meta-specification has a form such as this:

```
Width := 64 | 128
Height := 64 | 128
Modulus := 2  to 64
      …
```

This example illustrates several problems. If specifications are produced for every possible combination of alternatives, the example above would produce carpets of size 64×64, 64×128, 128×64, and 128×128. That might be what is wanted, but if only 64×64 and 128×128 are wanted, there is no way to specify that. With the 63 different moduli, there would be 252 carpets, half of which might be unwanted. And, of course, for other parameters with their alternatives, the number could become astronomical.

There are certain natural couplings of parameters: width and height, top and left initializers, and modulus and number of colors. By taking alternatives in parallel from these couplings, carpet space is effectively reduced to 4 dimensions. On the other hand, enforcing such couplings would make exploring some parts of carpet space impractically difficult.

We decided to provide optional couplings. Note that while these couplings often occur in practice and have a degree of naturalness to them, they are not the only ones possible. We simply decided that more flexibility would make the application too complicated to use effectively.

What coupling means in terms of generating specifications from a meta-specification is parallel evaluation instead of the built-in cross-product evaluation [2]. For example, coupling width and height for

```
Width := 64 | 128
Height := 64 | 128
```

would produce carpet sizes 64×64 and 128×128, while

```
Width := 128 | 64
Height := 64 | 128
```

would produce carpet sizes 128×64 and 64×128.

In couplings where the number of alternatives is not the same, we chose to stop when one of

a coupled pair of alternatives runs out. For example,

```
Width := 64
Height := 64 | 128
```

would produce only the carpet size 64×64. It might seem like such a specification would be a mistake, but it could be useful when turning couplings on and off .

Coupling of coupled pairs is possible by evaluating pairs in parallel. Consider, for example,

```
Width := 64 | 128
                              # coupled pair
Height := 64 | 128
                              ##### pairs coupled
Modulus := 5 | 6 | 7
                              # coupled pair
Colors := "g5" | "g6" | "g7"
```

(The alternatives for colors are the names of Icon palettes.) If widths and heights are coupled and the moduli and colors also are coupled, but separately, there are two carpet sizes and three modulus-colors combinations — six carpets in all. If the two couples are coupled, there are only two carpets: one 64×64 with modulus 5 and palette "g5" and another 128×128 with modulus 6 and palette "g6". The third alternates for the modulus and colors are not used, since there are only two alternatives for the size.

Although parallel evaluation terminates when one of the coupled pair of alternatives runs out, repeated alternation can be used to “fill out” coupled alternatives, as in

```
Width := |64
Height := |64
Modulus := 5 | 6 | 7
Colors := "g5" | "g6" | "g7"
```

which results in three 64×64 carpets.

Another problem with exploring carpet space is that it is all too easy to produce a meta-specification that corresponds to a huge or even unlimited number of carpets. For example,

```
Modulus := seq(2)
```

specifies an unlimited number of moduli. An obvious solution is to use

```
modulus := seq(2) \ limit
```

but an option to limit the total number of specifications produced from a meta-specification is needed also.

The next question is how all of this can be represented in an interactive application.

The Carpet Meta-Specification Program

Figure 1 shows the rather odd-looking interface for the meta-specification program.

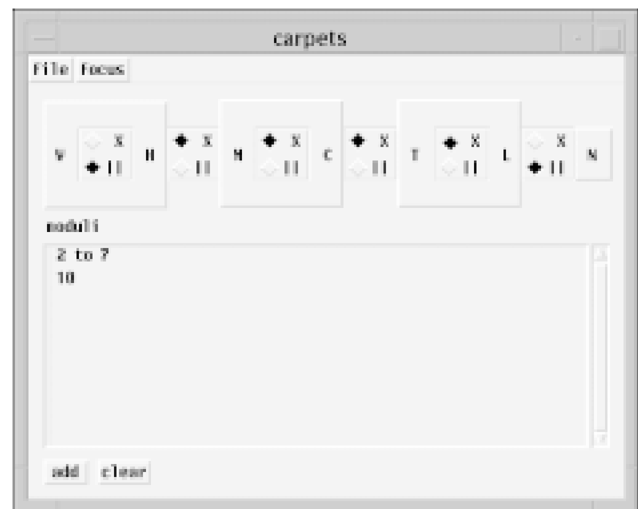


Figure 1. The Application Interface

Below the menu bar are radio buttons for specifying parallel (||) or cross-product (X) evaluation. (Parallel evaluations correspond to coupling in the sense we described earlier.) The labels W, H, M, C, T, L, and N identify the seven parameters. Thus, in Figure 1, widths and heights are coupled, and all other evaluations are cross evaluations, except that the neighbors are evaluated in parallel with all else.

Below the radio buttons is a text-list [3] that shows the alternatives that have been entered for a parameter. This window serves as a focus for one parameter at a time, rather than having seven text-lists. In Figure 1, the focus is on the moduli. The focus can be changed by using the Focus menu or keyboard shortcuts.

Alternatives can be given on separate lines or by expressions that themselves have alternatives.

Icon on the Web

Information about Icon is available on the World Wide Web at

http://www.cs.arizona.edu/icon/

In Figure 1, there are two lines, one specifying moduli 2 through 7 and the other specifying 10. These alternatives could, of course, be given on one line.

The add button below the text list brings up a dialog for adding a line to the current list, while the clear button removes all lines.

Selecting a line in the text-list brings up a dialog that allows the line to be edited or deleted. See Figure 2.

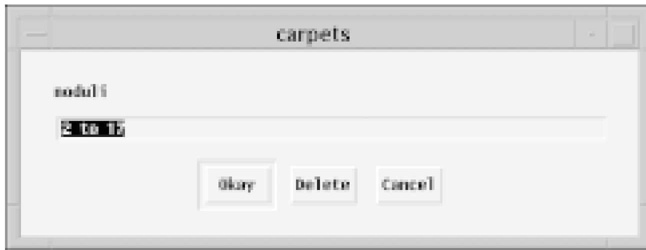


Figure 2. The Dialog for a Line

The File menu provides for loading and saving meta-specifications, as well as invoking the companion program to expand the meta-specification.

The Implementation

The implementation of the meta-specification program and its companion meta-expander show some interesting aspects of programming in Icon.

The meta-specification program itself is, for the most part, much like other interactive applications with visual interfaces. It's too long to show here, but it's available on our Web site (see page 14) and one important aspect of it is described in the **Tricky Business** article that starts on page 14.

The meta-specification program incorporates the functionality described in the previous section. It writes out a meta-specification as a file of preprocessor definitions for the parameter alternatives and couplings. It then compiles and executes the meta-expander, which includes the meta-specification file: The same technique used for basic carpet specification and generation.

The meta-expander is more interesting. If it weren't for parallel evaluation, the expansion of a meta-specification would be comparatively simple. It might look like this:

```
link carputil                # for record declaration
link xcode                   # to encode database as file

$include "plorincl.icn"      # meta-specification file

procedure main()

   i := –1

   database := table()

   rec := carpet()            # starting carpet

   every {
      rec.width := Width &    # generate all alternatives
      rec.height := Height &
      rec.modulus := Modulus &
      rec.colors := Colors &
      rec.top := Top &
      rec.left := Left &
      rec.neighbors := Neighbors
         } \ Limit do {
         rec.name := Name || right(i +:= 1, 3, "0")
         database[rec.name] := rec
         rec := copy(rec)     # start with last fields intact
         }

   xencode(database)          # write to standard output

end
```

In this program, database is a table for the specifications and carpet() is a record that contains the parameters for a single carpet specification. At the end of expansion, the database is written out using xencode() so that it can be loaded by a program to explore the expanded space or generate the individual carpets in the background.

The portion of the program that generates the parameters can be written with mutual evaluation instead of conjunction, a form that will be useful later:

```
every (
   rec.width := Width,
   rec.height := Height,
   rec.modulus := Modulus,
   rec.colors := Colors,
   rec.top := Top,
   rec.left := Left,
   rec.neighbors := Neighbors
     ) \ Limit do {   …
```

In order to perform parallel evaluation, co-expressions are necessary. Programmer-defined control operations [3-4] are designed for just such a situation. Here's a procedure that performs parallel evaluation, terminating when one of the expressions runs out of alternatives:

```
procedure Parallel_(L)      # Parallel_{expr1, expr2}

   while @L[1] do {
      @L[2] | fail
      suspend
      }

end
```

To see how this procedure fits into the expansion process, consider only the coupling of the width and height, which can be done as

```
every (
     Parallel_ {
        rec.width := Width,
        rec.height := Height
        },
     rec.modulus := Modulus,
     rec.colors := Colors,
     rec.top := Top,
     rec.left := Left,
     rec.neighbors := Neighbors
        ) \ Limit do {   …
```

Of course, for cross-product evaluation to be possible, Parallel_{} can't be hard-wired into the program. Instead, an identifier is used whose value is defined in the meta-specification:

```
every (
     wh{
        rec.width := Width,
        rec.height := Height
     },
     …
```

where for parallel evaluation the meta-specification file contains

```
$define wh Parallel_
```

To use this method for cross evaluation, we need a programmer-defined control operation for cross-product evaluation also:

```
procedure Cross_(L)        # Cross_{expr1, expr2}

   while @L[1] do {
      while @L[2] do
         suspend
      L[2] := ^L[2]
      }

end
```

To get cross-produce evaluation for the width and height, the definition in the meta-specification file would be

```
$define wh Cross_
```

In all there are six places where there may be either cross-product or parallel evaluation, depending on the meta-specification. The evaluation portion of the meta-expander is:

```
every {
   c3{
      c2{
         c1{
            wh{
               rec.width := Width,
               rec.height := Height
               },
            mc{
               rec.modulus := Modulus,
               rec.colors := Colors
               }
            },
         tl{
            rec.top := Top,
            rec.left := Left
            }
         },
      rec.neighbors := Neighbors
      }
   } \ Limit do {   …
```

where c1, c2, and c3 determine the evaluation mode of the parameter pairs.

References

1. "Anatomy of a Program — Numerical Carpets", *Icon Analyst* 45, pp. 1-10.

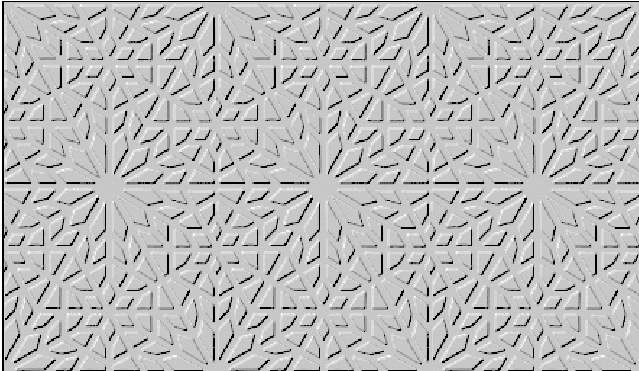
2. "Result Sequences", *Icon Analyst* 7, pp. 5-8.

3. "Programmer-Defined Control Operations",

Icon Analyst 22, pp 8-12.

4. "Programmer-Defined Control Operations", Icon Analyst 23, pp 1-4.

5. "Text-List Vidgets", Icon Analyst 46, pp. 1-4.

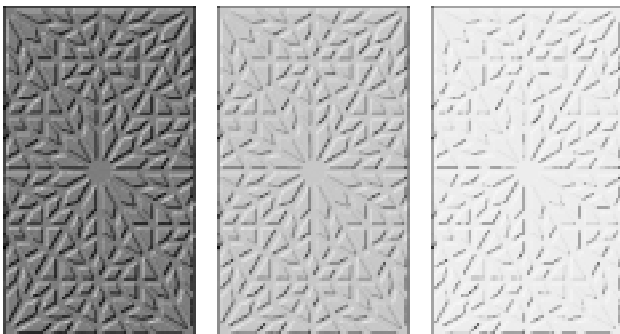


Graphics Corner

Gamma Adjustment

In the last issue of the Analyst [1], we described how the gamma value can be changed to adjust the appearance of an image displayed on a monitor.

In this article, we describe an application that allows the gamma value to be adjusted automatically. We use the term adjustment rather than correction, since the application can be used to produce images that do not have a "correct" appearance but which may nonetheless be useful. For example, the gamma value can be increased to lighten an image to make it suitable as a background for a Web page. Changing the gamma value also can be used to get other effects. See Figure 1.



$\gamma = 0.8$ normal γ $\gamma = 8.0$

Figure 1. The Effects of Different γ Settings

The Application

The idea behind the application is simple: Read an image from a file into a window with an altered value of γ, reset γ to its normal value, and write the image out:

```
gamma := WAttrib("gamma")        # normal gamma
WAttrib("gamma=5.0")             # new value
ReadImage( … )
WAttrib("gamma=" || gamma)       # restore normal
WriteImage( … )
```

The interface for the application is shown in Figure 2.

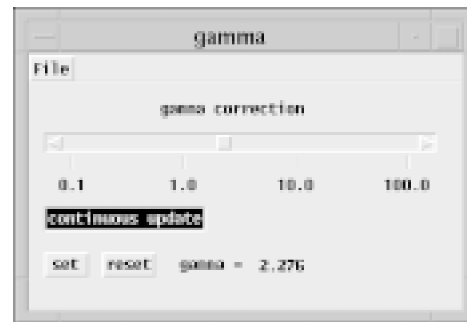


Figure 2. The Application Interface

The File menu provides for loading and saving images and, as usual, for quitting the application. When an image is loaded, it is displayed in a separate window.

The slider allows the value of γ to be set in the range of 0.1 to 100.0 on a logarithmic scale. At the extremes, moderately saturated colors become almost black and white, respectively. Fully unsaturated colors (intensity 0.0) and fully saturated colors (intensity 1.0) are unaffected as the formula shows:

$$B = I^{\gamma}$$

The current value of γ is shown in lower portion of the window. Initially it is the normal value for the monitor.

Since it's not possible to set a precise value with a slider, the set button is provided to produce

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

ftp.cs.arizona.edu (cd /icon)

a dialog for entering the γ value numerically. See Figure 3.

Figure 3. The Dialog for Setting γ

The reset button sets γ back to its initial value.

The continuous display toggle controls the effect of moving the slider. If it's on, as shown in Figure 2, the appearance of the image changes as the thumb of the slider is moved. If it's off, the appearance of the image only changes when the thumb is released. The reason for this option is that the image is read every time there is a callback from the slider. If a callback occurs whenever the thumb is moved, continuously updating for a large image may be too slow and lag the thumb movement. By filtering the slider events, a callback only occurs then the thumb is released [2].

Aside: Ideally, a copy of the image being modified would be read into a hidden window and copied to a visible window using CopyArea(). This would be fast and avoid the need for filtering events for large images. Unfortunately, CopyArea() does not apply γ correction when copying between windows with different values of γ. This is an inconsistency that was overlooked in the design and implementation of CopyArea().

The Program

The program is relatively simple and short, so we'll show it all here. See the notes that follow the listing.

```
link interact
link vfilter
link vsetup

global continuous_vidget        # update toggle
global gamma                    # gamma value
global gamma_vidget             # gamma vidget
global default_gamma            # default gamma
global name                     # name of image file
global pane                     # window for image
global vidgets                  # table of vidgets

procedure main()

   vidgets := ui()

   continuous_vidget := vidgets["continuous"]
   gamma_vidget := vidgets["gamma"]

   VSetState(continuous_vidget, "1")

   default_gamma := WAttrib("gamma")
   set_gamma(default_gamma)

   GetEvents(vidgets["root"], , shortcuts)

end

procedure continuous_cb(vidget, value)

   if \value then VSetFilter(gamma_vidget, &null) else
      VSetFilter(gamma_vidget, "1")

   return

end

procedure file_cb(vidget, value)

   case value[1] of {
      "load @L"   :  load_image()
      "quit @Q"   :  exit()
      "save @S"   :  save_image()
      }

   return

end

procedure gamma_cb(vidget, value)

   set_gamma(10.0 ^ value)

   return

end

procedure load_image()

   WClose(\pane)

   repeat {
      if OpenDialog("Load image file:") ==
         "Cancel" then fail
     pane := WOpen("label=" || dialog_value, "image=" ||
       dialog_value, "gamma=" || gamma) | {
          Notice("Cannot open image file.")
          next
          }
      name := dialog_value
      Raise()
      return
      }

end
```

```
procedure reset_cb()

  set_gamma(default_gamma)

  return

end

procedure save_image()

  WAttrib(\pane, "gamma=" || default_gamma) | {
    Notice("No image loaded.")
    fail
    }
  snapshot(pane)
  WAttrib(pane, "gamma=" || gamma)

  return

end

procedure set_cb()

  repeat {
   if OpenDialog("Set gamma value:", gamma, 10) ==
     "Cancel" then fail
    if 0.0 <= numeric(dialog_value) <= 100.0 then {
      set_gamma(dialog_value)
      return
      }
    else {
      Notice("Invalid gamma value.")
      next
      }
    }

end

procedure set_gamma(value)

  gamma := value

  WAttrib(\pane, "gamma=" || gamma)
  VSetState(gamma_vidget, log(value, 10))
  show_gamma()
  ReadImage(\pane, name)
  Raise()

  return

end

procedure shortcuts(value)

  if &meta then case map(value) of {
    "l"   :  load_image()
    "q"   :  exit()
    "r"   :  set_gamma(default_gamma)
    "s"   :  save_image()
    }

  return

end

procedure show_gamma()
  static old_gamma, x, y

  initial {
    old_gamma := ""
    x := vidgets["placeholder"].ax
    y := vidgets["placeholder"].ay
    }

  WAttrib("drawop=reverse")
  DrawString(x, y, old_gamma)
  DrawString(x, y, gamma)
  WAttrib("drawop=copy")

  old_gamma := gamma

  return

end

#===<<vib:begin>>=== modify using vib
procedure ui_atts()
  return ["size=337,201", "bg=gray–white"]
end

procedure ui(win, cbk)
return vsetup(win, cbk,
  [":Sizer:::0,0,337,201:",],
  ["10:Label:::109,97,21,13:1.0",],
  ["20:Label:::193,97,28,13:10.0",],
  ["3:Label:::23,97,21,13:0.1",],
  ["continuous:Button:regular:1:12,120,126,20:_
    continuous update",continuous_cb],
  ["file:Menu:pull::0,2,36,21:File",file_cb,
    ["load @L","save @S","quit @Q"]],
    ["gamma:Scrollbar:h::12,62,305,16:–_
      1.0,2.0,2.0",gamma_cb],
  ["glabel:Label:::102,37,112,13:gamma correction",],
  ["label1:Label:::276,97,35,13:100.0",],
  ["label2:Label:::117,159,56,13:gamma = ",],
  ["line1:Line:::0,23,336,23:",],
  ["line2:Line:::34,80,34,90:",],
  ["line3:Line:::209,80,209,90:",],
  ["line4:Line:::121,80,121,90:",],
  ["line5:Line:::295,80,295,90:",],
  ["reset:Button:regular::57,156,42,20:reset",reset_cb],
  ["set:Button:regular::12,156,35,20:set",set_cb],
  ["placeholder:Button:regularno::179,171,35,20:",],
  )
end
#===<<vib:end>>=== end of section maintained by vib
```

Notes

The placeholder vidget (the last vidget in the VIB code section above) is an invisible button — it

has no label and no outline. It serves only to identify a place on the canvas to show the current γ setting.

The procedure show_gamma() uses the upper-left corner of placeholder to identify the place the current gamma value is drawn whenever it is changed. The placeholder vidget was positioned experimentally to give the desired results. Note that show_gamma() uses reversible drawing to erase the previously displayed gamma value.

The procedure continuous_cb() changes the filter attribute of the slider vidget using

```
VSetFilter(vidget, value)
```

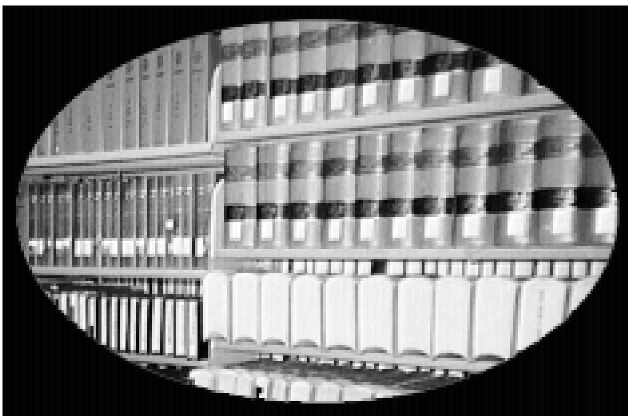
There is no filtering of slider events if value is null but filtering if it is nonnull.

Note: Prior to the development of this application, there was no way to change the filter attribute of a slider or scrollbar during program execution. We had to add VSetFilter() to do this. This procedure is not in the current Icon program library (Version 9.3.1). It will appear in the next release and will be sent in the next subscriber update to the library.

References

1. "Graphics Corner — Gamma Correction", Icon Analyst 46, pp. 5-7.

2. "Building a Visual Interface", Icon Analyst 36, pp. 1-4.



From The Library

In past articles on the Icon program library, we've concentrated on programs and procedures that are particularly useful. The library also contains recreations. Here's one of the best.

Concentration, also known as the Memory Game or Pelmanism, is a card game involving matching. A deck of cards is dealt face down on a table, and players take turns exposing pairs of cards, one after the other. A pair of equal rank (such as two queens) is removed and scored for the player, who is then awarded another turn. An unmatched pair is turned back over and ends the turn. Play continues until all cards have been removed.

A good memory of previously seen cards is crucial for success. Strategy also counts; sometimes it is best to choose a losing card that has been seen before. Ian Stewart discusses two-person Concentration in the October, 1991, issue of *Scientific American* magazine.

The Icon program library contains a solitaire version of Concentration in the file gprogs/concen.icn. This program displays an array of cards, initially face down, that are manipulated using the mouse. See Figure 1.

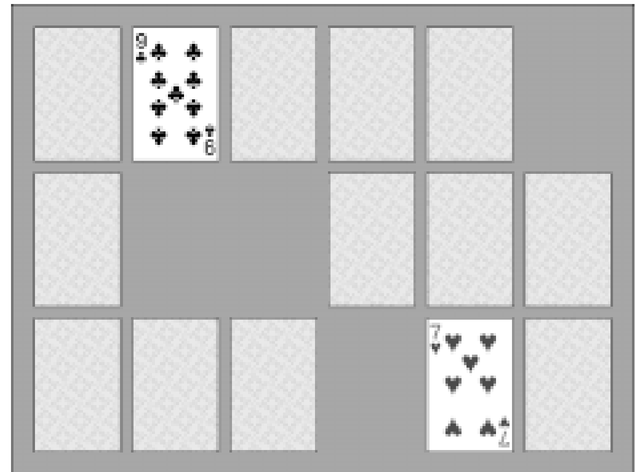


Figure 1. The Game of Concentration

Clicking on a card "turns it over" to reveal its face. Another click elsewhere exposes a second card, as seen in the screen snapshot. A third click (anywhere) ends the turn: the two cards are removed if they match or turned back over if they do not.

When all the cards have been paired, the game is over, and the full set of cards is displayed face up. Clicking again reinitializes for a new game. The program can be exited at any time by pressing the Q key.

Pairing up a full deck of cards can seem daunting at first. The Concentration program accepts a

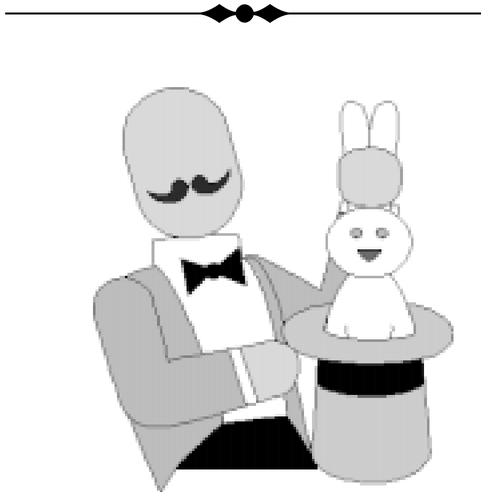
command-line argument to control the number of cards. For example,

```
concen 12
```

displays just twelve cards, a good number for learning how the game works.

The card images used in the Concentration game are some of the nicest we've seen. They came originally from a free implementation of Spider, an excellent double-deck solitaire game. Notice that we've put a repeat pattern with the Icon logo on the back of the cards, although it's difficult to see it in Figure 1.

The Concentration card game even inspired a television show by the same name. Hugh Downs hosted this NBC game show, which ran from 1958 to 1973. Contestants matched pairs of hidden prizes and won the game by solving a rebus.



Tricky Business

If you've looked through Icon's complete computational repertoire, you've probably found things that you couldn't imagine you'd ever use.

One of these might be variable(), which was added to Icon a few versions back. If s is the string name of a variable, variable(s) returns the corresponding variable, to which assignment can be made. If s is not the name of a variable, variable(s) fails.

One of the main uses of this function is in MT Icon [1], in which variable() allows one thread to access a variable in another thread. When writing the third edition of the Icon language book, we were hard pressed to find a meaningful example of the use of variable() in ordinary programming [2].

It's been our experience that unanticipated uses of language features are discovered, sometimes long after the features are introduced. Of course, we have no way of knowing how Icon programmers use various features unless they tell us.

We recently discovered a use for variable() that we had not thought of before. It basically involves a relationship between program structure and the data the program processes — where strings in the data are the names of variables in the program. This may come about by structuring the program to fit the data or vice versa.

An example occurs in the meta-carpet specification program described in an article that begins on page 5.

In this application, there is a single text-list [3] for displaying and modifying the alternatives for one of seven parameters. See Figure 1 on page 7.

The parameter of interest is chosen by the user through a keyboard shortcut or the Focus menu. See Figure 1 below.

Figure 1. The Focus Menu

Supplementary Material

Supplementary material for this issue of the Analyst, including color images and Web links, is available on the Web. The URL is

http://www.cs.arizona.edu/icon/analyst/iasub/ia47/ia47sub.htm

The callback procedure for this menu might look like this:

```
procedure focus_cb(vidget, value)

   case value[1] of {
      "widths     @W"     :  focus("widths")
      "heights    @H"     :  focus("heights")
      "moduli     @M"     :  focus("moduli")
                  …
      }

   return

end
```

where focus() loads the text-list vidget with a list that corresponds to its argument. Similarly, the keyboard shortcuts for setting the focus just call focus() in the same way.

We can simplify this by getting the string from the menu item.

```
procedure focus_cb(vidget, value)

   focus(value[1] ? tab(upto(' ')))

   return

end
```

The procedure focus() then could have this form:

```
procedure focus(param)

   case param of {
      "widths"   :  VSetItems(display, widths)
      "heights"  :  VSetItems(display, heights)
      "moduli"   :  VSetItems(display, moduli)
                  …
      }
         …
   return

end
```

where display is the text-list vidget and widths, heights, moduli, … are global variables whose values are the respective lists. See Reference 3 for an explanation of VSetItems().

You might think at this point, and rightly so, that there is a lot of redundancy in the code above — needed to get from strings to variables of the same name. That's where variable() comes in:

```
procedure focus(param)

   VSetItems(display, variable(param))
```

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 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

© 1998 by Ralph E. Griswold, Madge T. Griswold, and Gregg M. Townsend

All rights reserved.

```
      …
    return

  end
```

This takes care of all the parameters.

Note that the names of the variables for the lists were chosen in coordination with the items in the Focus menu. In other words, using variable() was taken into account in the selection of names.

The technique can be extended to simplify other aspects of the application. For example, by adding a global variable current and adding the following line to focus(), the string name of the current parameter is available throughout the application:

```
    current := param
```

The value of current can be used, for example, in the callback procedure that adds a line to the list of alternatives for the current parameter:

```
  procedure add_cb()

    if TextDialog(
      "Add line to " || current || ":",        # caption
      "alternatives",                          # label
      "",                                      # default
      ExprWidth                                # field width
      ) == "Cancel" then fail                  # quick exit

    put(variable(current), dialog_value[1])

    return

  end
```

One nice aspect of this design is that adding another parameter does not require major revisions throughout the program; in fact, the procedures shown here do not need to be changed at all.

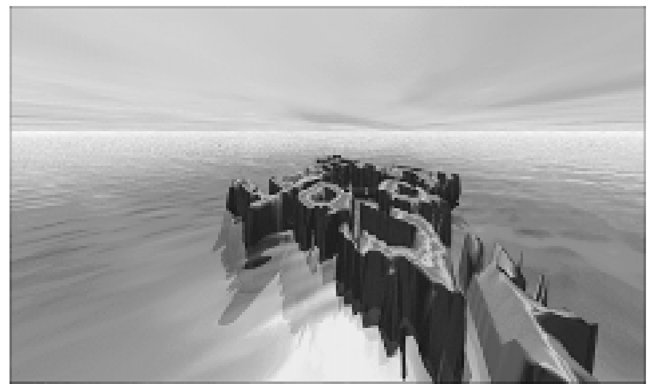
The technique described in this article is, of course, not limited to interactive applications. It can, for example, be used in a program that processes data with known keywords whose occurrence requires actions by the program.

References

1. "Multi-Thread Icon", Icon Analyst 14, pp. 8-12.

2. *The Icon Programming Language*, 3rd edition, Ralph E. Griswold and Madge T. Griswold, Peer-to-Peer Communications, Inc., 1996, p. 199.

3. "Text-List Vidgets", Icon Analyst 46, pp. 1-4.



What's Coming Up

As usual, we have a lot of things on deck and aren't sure yet which will be in the next issue of the Analyst.

The second article on the assault on Mount Versum is scheduled. Also in the works are articles on three-dimensional paths, digit patterns, and sorting. There'll also likely be another **Graphics Corner**.

Feedback?

The next issue marks eight years of publication for the Analyst.

When we started we had some ideas about what we wanted to do but no clear idea where they might lead in time. And we would not have predicted that the Analyst would continue for so long.

Our initial intention was to provide material for all levels of Icon programmers from novices to experts. Over time, there has been less and less material for beginning Icon programmers. For one thing, much of the basics was covered in early issues. For another, most of our subscribers were experienced Icon programmers, possibly because the Analyst did not appeal to beginning programmers. Another factor was our own interests.

In recent years, there has been more specialized material about application areas, such as the apparently never-ending series on versum numbers. And the introduction of graphics in Icon has had a major impact on the content of the Analyst.

We get very little feedback from our readers. We would like to have more. We'd like to know what you like and don't like, what we might do that we're not doing, how you feel about the on-line component of the Analyst, and so on. It's easy; just send a message to icon-project@cs.arizona.edu.