The Icon Analyst

In-Depth Coverage of the Icon Programming Language

October 1996
Number 38

In this issue ...

Random Numbers Revisited

Editors' note: The article that follows is taken from a letter from Carl Sturtivant.

This letter is in response to your articles on the Icon random number generator in Issues 28 and 29 of the Analyst and in particular, to your request for an explanation of the regularities shown in Issue 29.

A Glitch

When I first saw the "Rolling Your Own" section of the article in Analyst 28 [1], I was struck by the fact that the procedure rand_int(i) has been given a scale factor of $1/(2^{31}-1)$.

When &random is equal to its maximum value of $2^{31}-1$, its scaled value will be 1.0 (provided floating point arithmetic is conducted to sufficient accuracy). Thus the value returned by rand_int(i) will be integer(i * 1.0) + 1 which is i + 1. Thus ?2 could evaluate to 3, for example. To test this hypothesis, I ran the following program.

```
procedure main()

   &random := 1276559117  # Seed before 2^31 – 1
   writes(?2, "/")
   write(&random – (2 ^ 31 – 1))

end
```

The output of this program was *not* 3/0 as I expected, but 2/0. The zero means that &random was $2^{31}-1$ at the point where scaling occurred, and yet the overflow did not occur as predicted. Perhaps the scaling algorithm in Icon was not the same as rand_int(i)? To test this hypothesis, I ran the following program.

```
link analyst            # from "Rolling Your Own"

procedure main()

   &random := random := 1276559117

   writes(?2, "/")
   writes(&random – (2 ^ 31 – 1), ",")
   writes(rand_int(2), "/")
   writes(random – (2 ^ 31 – 1))

end
```

The output of this program was 2/0,3/0 as expected. Thus, rand_int(2) can return 3. The following program detects the source of the trouble in the true Icon random number generator.

```
procedure main()

   &random := 1276559117
   write(?0)

end
```

This program should write 1.0 if scaling is occurring as stated, but in fact the output for my implementation of Icon is less than one — 0.99999999672599.

Examination of the Icon source file rmacros.h shows that RandScale is defined to be 4.65661286e–10 with a comment that this is equal to $1/(2^{31}-1)$. However, this is not correct! It is more like $1/(2^{31}+6)$. The last digit should be an 8 at the accuracy given. Furthermore, in order that RandScale give 1.0 or more when multiplied by $2^{31}-1$, but not when multiplied by $2^{31}-2$, more decimal places need to be specified. On my implementation of Icon, it is necessary to specify RandScale to eleven places for this to be true.

What the above shows, however, is that $1/(2^{31}-1)$ is not the correct scale factor anyway, as it can lead to ?i evaluating to i + 1, albeit with very small probability. A better scale factor is 2^{-31}.

Changing the scale factor slightly does not make a great deal of difference for practical purposes. (The sequence of values that &random takes on remains unchanged.) Examination of the algorithm for computing ?i reveals why.

The algorithm computes ?i by placing equally spaced boundaries in the real interval [0,1) giving i sub–intervals that we can number in order from 1 to i. These sub–intervals are closed on the left and open on the right (because of the way that the Icon integer function works). Then a new value of &random is computed, and scaled to the interval [0,1). The number of the sub–interval this new value falls into is the result of evaluating ?i.

Most of the time the scaled value of &random does not fall near an interval boundary, provided that i is much smaller than the number of different values that &random can take on. Thus, a small change in the scale factor will not move the scaled value across a sub–interval boundary most of the time.

This intuitive argument can easily be quantified. Consider changing RandScale from its present value in the Icon source to an accurate representation of 2^{-31}. (The decimal expansion of 2^{-31} is *exactly* $4.656612873077392578125\times10^{-10}$.) For i < 32, the probability of choosing &random so that ?i would have a different value were RandScale changed is less than 10^{-7}.

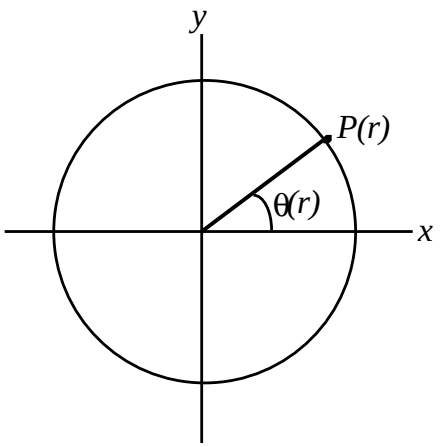
"Curiosity or Problem?"

Hereafter, we assume that the scale factor is exactly 2^{-31}. The modulus $m = 2^{31}$, so this means the scale factor is exactly $1/m$. This is important for the following construction to work properly.

The random number algorithm is more intuitively stated by using a circle rather than the interval [0,1), because a circle reflects the modular arithmetic naturally in its geometry.

Informally, we regard the sequence of values taken on by &random as scaled into the interval [0,1), which is stretched in length by a factor of 2π, and bent nose to tail to become the unit circle. (Yet another reason why the value 1.0 should *not* be present.)

This transformation is properly done using complex exponentials. If r is a possible value of &random, then the corresponding point $P(r)$ on the unit circle in the complex plane is $e^{2\pi ir/m}$. Here $\theta(r) = 2\pi r/m$ is the angle (in radians) illustrated in the following diagram.



This is the same as taking the unit circle, and placing 2^{31} equally spaced points around its circumference beginning on the positive x axis, corresponding in order to the values $0 \ldots 2^{31} - 1$ for &random.

Addition is naturally modulo $m = 2^{31}$, and corresponds to adding the angles of the two numbers, that is, $\theta((r_1 + r_2) \bmod m) = \theta(r_1) + \theta(r_2)$. (This also is the same as multiplying the corresponding complex numbers, that is $P((r_1 + r_2) \bmod m) = P(r_1) \cdot P(r_2)$.)

Now let's take a geometrical view of the "Curiosity or Problem?". We know that the values of &random are iterated using the following relation:

$$r_{k+1} = (ar_k + c) \bmod m$$

where $a = 1103515245$ and $c = 453816694$.

Interpreting these values as angles as above (and converting to degrees), we have $\theta(a) \approx 185°$ (and $\theta(c) \approx 76° \approx 75°$). This value interlocks with 360° in a coincidental way that we will show leads to the pattern in the first column of the output of the "Curiosity or Problem" program. (The conversion factor from integer values to angles is now $360°/m$ rather than $2\pi/m$.)

Now to convert the recurrence relation to operations on angles, we apply the function θ to both sides, and use the rule for applying θ to a sum (given above):

$$\theta(r_{k+1}) = \theta((ar_k + c) \bmod m) = \theta(ar_k \bmod m) + \theta(c)$$

It remains to simplify the term involving the application of θ to a product, modulo m. Since $\theta(x) = (360°/m)x$ (modulo 360°) we have $\theta(ar_k \bmod m) = \theta(a)r_k$ (modulo 360°). This just says that you add the angle $\theta(a)$ around the circle r_k times — its repeated addition for angles. So here's the outcome:

$$\theta(r_{k+1}) = \theta(a)r_k + \theta(c) \approx r_k\,185° + 75°$$

Now, what happens if we start with &random = $r_0 = 0, 2, 4, \ldots$ as in the given program? Some of the importance of r_0 being even is now apparent. It means that $r_0 185°$ is just a little over a multiple of 360°, that is, $r_0 185°$ is a "smallish" angle. In fact $r_0 185°$ increases by about 10° each time r_0 is increased by 2. Thus $\theta(r_1)$ takes on roughly the values 75°, 85°, 95°, 105°, 115°, … . Note that $\theta(c)$ is unimportant here: it just moves the regularity around so that it starts in a different place.

These numbers are somewhat similar: ("nearly" constant), so we might informally refer to them as having "approximate period 1". Similarly, the numbers in column 8 of the output of the given program have "approximate period 4".

The program computes ?20, so that the unit circle is partitioned like a dart board into 20 equal segments of 18°, numbered (unlike a dart board) in order 1 through 20. Thus the first column of the output is "explained" by the observation that $\theta(a)$ is very close to 180°.

The above argument relies upon there being one application of a linear congruential function to a sequence of seed values with constant small differences. (The seed values themselves may be large.) After the first iteration, the new seed sequence is no longer of this form, and the argument cannot be repeated.

However, the composition of a linear congruential function with another linear congruential function is once again a linear congruential function. Thus, for example, column 8 of the output can be computed from the seeds $r_0 = 0, 2, 4, 6, \ldots$ as follows: $r_8 = (a_8 r_0 + c_8) \bmod m$, for some constants a_8 and c_8. (This function is the linear congruential function corresponding to the usual linear congruential function composed with itself eight times.)

As the output in column 8 has "approximate period 4", no doubt we will find that $\theta(a_8)$ is near to 45°, so that increasing r_0 by 2 moves the result through near to 90°; increasing r_0 by 2 four times moves the result approximately to the same value.

The relations (if any) between the columns in the output of the "Curiosity or Problem?" program remain unexplained at this point. We need a more global view to understand this feature.

Regularities in General

Suppose we choose a general linear congruential random number generator, given by the recurrence $r_{k+1} = (ar_k + c) \bmod m$, where random numbers in the range 1 to n are extracted by the function $h(n, r_k) = \lfloor n\, r_k / m \rfloor + 1$, and $m = 2^j$. (That is, until further notice, a, c, $m = 2^j$ are perfectly general.)

It is not difficult to see that the solution to this recurrence is $r_k = a^k r_0 + c_k$ where:

$$c_k = \left(c \sum_{l=0}^{k-1} a^l\right)$$

(all using modulo m arithmetic). Therefore, column k of the output of an analog of the "Curiosity or Problem" program using the above general random number generator would be determined from the initial seeds $r_0 = 0, 1, 2, 3, \ldots$ by multiplying by $a^k \bmod m$, adding c_k and applying $h(n, \ldots)$.

So to understand column k in general, we need to know what possible values $\theta(a^k \bmod m)$ can take on. (Once again, as in the previous section, the value of $\theta(c_k)$ is unimportant, since it merely rotates any regularities.)

To understand $a^k \bmod m$, where $m = 2^j$, we need a little number theory. We will assume that $j \geq 3$, and that a satisfies the criterion $a \bmod 8 = 5$ (which is criterion 3 in Icon Analyst 28). We need the following fact about modulo 2^j arithmetic:

if $S \bmod 4 = 1$ then $S = 5^l \bmod 2^j$
for some $0 \leq l < 2^{j-2}$

What this amounts to is that exactly half of the odd numbers ($S = 1, 5, 9, 13, 17, \ldots, 2^j - 3$), have a discrete logarithm l in the base 5. These logarithms will add modulo 2^{j-2} when the corresponding numbers are multiplied modulo 2^j. Furthermore, $a \bmod 8 = 5$ implies that $a \bmod 4 = 1$, so that any valid choice of a will have such a logarithm.

To compute this logarithm, I constructed the following procedures. The procedure power(x, k, m) computes $x^k \bmod m$ by "repeated squaring". It is called by the procedure log5(S, j) which computes the unique value $0 \leq l < 2^{j-2}$ such that $S = 5^l \bmod 2^j$.

```
procedure power(x, k, m)

   if k = 0 then return 1
   if k % 2 = 0 then return power(x, k / 2, m) ^  2 % m
   else return x * power(x, k – 1, m) % m

end

procedure log5(S, j)

   if S % 4 ~= 1 | j < 3 then fail

   L := 0
   m := 4
   every 1 to j – 2 do {
      m *:= 2
      if power(5, L, m) ~= S % m
      then L +:= m / 8
      }

   return L

end
```

The procedure log5(S, j) computes the logarithm bit by bit. At each step, the modulus is doubled, and the old value of the logarithm is tested to see if it is correct with the new modulus. If so then the new uppermost bit of the logarithm must be a zero. If not then it must be a one, and an appropriate power of two is added on to correct the logarithm for the new modulus.

If $a \bmod 8 = 5$ then the bottom bit of the logarithm of a will be a one, that is, the logarithm will be odd. Consequently, it will always have a multiplicative inverse modulo 2^{j-2}.

Let α be this inverse. We have $a = 5^l \bmod 2^j$, where l is the logarithm, and is odd, and $1 = l\alpha \bmod 2^{j-2}$. Raising both sides of $a = 5^l \bmod 2^j$ to the power α gives $a^\alpha \bmod 2^j = 5$, and since 5 can be expressed as a power of a, then any number S satisfying $S = 4 \bmod 1$ can be expressed as a power of a. In other words, a is a legitimate base for logarithms, just like 5.

Let S satisfy $S = 4 \bmod 1$ and let $\sigma = \log_5(S, j)$. Then $S = 5^\sigma \bmod 2^j$. But $a^\alpha \bmod 2^j = 5$, so $S = a^{\alpha\sigma} \bmod 2^j$. In other words, $\log_a(S, j) = \alpha\log_5(S, j) \bmod 2^{j-2}$ where $\alpha = (\log_5(a))^{-1} \bmod 2^{j-2}$.

A concrete calculation will illustrate the mechanics for $a = 1103515245$, the usual Icon random number generator constant. A call of log5(1103515245, 31) returns 290333047, showing that $a = 1103515245$ is in fact $5^{29033047} \bmod 2^{31}$. A calculation with the extended Euclidean algorithm shows that $(290333047)^{-1} \bmod 2^{29} = 17190347$. This may easily be verified because $171903047 \times 290333047 \bmod 2^{29} = 1$. So, $5 = 1103515245^{17190347} \bmod 2^{31}$, and $\log_{1103515245}(S, 31) = 171903047 \log_5(S, 31) \bmod 2^{29}$.

What we've shown above is that any a satisfying $a \bmod 8 = 5$ will suffice as a base for logarithms (and we've given a means of calculating $\log_a(S, j)$ for any S satisfying $S \bmod 4 = 1$). *This means that $a_k \bmod 2^j$ may take on any value S satisfying $S \bmod 4 = 1$, simply by varying k.*

Specifically, we can choose a value S (satisfying $S \bmod 4 = 1$) that has $\theta(S)$ very close to a "coincidental" angle (for example, 180°, 45°, or 60°). Then there always exists $k = \log_a(S, j)$ such that $a^k \bmod 2^j = S$. Thus, the k^{th} column of output of programs like the one in "Curiosity or Problem?" would then (in principle) exhibit striking near periodicity of almost any kind desired.

Of course k could be rather large. But what we have established is that the kinds of columns output by the "Curiosity or Problem?" program are by no means untypical, and are not specific to the choice of constants made by the Icon random number generator, but are inherent whenever the modulus is a power of 2.

As an example, let's choose $\theta(S) \approx 90°$ with the standard Icon random number generator. Then $S = 2^{29} + 1$ is very close, and satisfies $S \bmod 4 = 1$. A call of log5(2 ^ 29 + 1, 31) gives 402653184. From four paragraphs back, we have $\log_{1103515245}(S, 31) = 171903047 \log_5(S, 31) \bmod 2^{29}$, and substituting, we have $\log_{1103515245}(S, 31) = 134217728$. Thus, in column 134217728 we should have approximate periodicity 4 (with minuscule drift from exact periodicity), as in the following program

```
procedure main()

   k := 134217728
   i := 1000

  every &random := 0 to 19 do {
      every 1 to k – 1 do ?1
      write(?i)
      }

end
```

The output from this program on my system is as follows. (It took an overnight run to collect these numbers.)

```
125
375
625
875
126
```

```
376
626
876
126
376
626
876
126
376
626
876
126
376
626
876
```

Conclusion

We have shown that the regularities in the columns of output from the "Curiosity or Problem?" program are an instance of a general phenomenon of column regularity. There is an enormous amount of regularity in the output of the Icon random number generator, so it's not surprising that some of it wound up in the first few columns. One may choose any period, with smaller or larger amounts of "drift", and by the procedure exemplified above, find a column exhibiting that behavior. Furthermore, this is inherent to all linear congruential random number generators using a modulus that is a power of two.

One possibility that you might wish to consider is having two random number generators in Icon with a keyword variable as a switch. Then you could keep the original generator (the default), and provide a more sophisticated source if required. (Perhaps the sophisticated source could also be seeded by the clock at the point it was switched to? If this feature was not required, it could always be avoided by assigning to &random.)

Of course the problem of finding a "good" source of pseudo-random numbers is notorious for its apparent trade-off between "goodness" and computational effort. An aggressive definition of "good" is roughly that no polynomial-time algorithm can distinguish the source from a truly random source.

This statement can be made precise, and is the standard of randomness used in theoretical studies of cryptography. We do not need such a drastic definition, but it's worth looking at the consequences intuitively, to see that the idea of "high degree" (see below) is important.

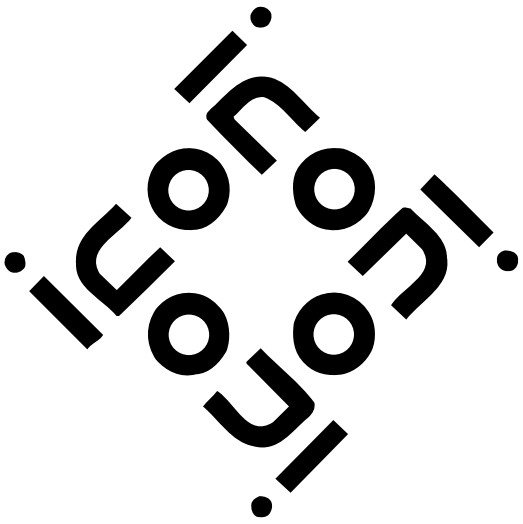
The existence of such a source is an open question, and is equivalent to the existence of (a natural class of) one-way functions. In fact it is one these one-way functions that would be used in place of the linear congruential function in such a pseudo–random source. An existence proof would imply $P \neq NP$ and much more.

One–way functions (if they exist) are functions of "high degree" (this has a proper definition, but take it intuitively), and consequently are much more painful to compute than a linear congruential function.

However, a "high-degree" function (chosen carefully) is likely not to possess any simple (or "simple-ish") algebraic properties (unlike low–degree functions) that give rise to patterns when iterated. And if there is a proof that iterating such a function eventually cycles through all the values in range, then it may well suffice for practical purposes provided that there is a reasonably efficient way to compute it.

Editors' note: When working with "random" numbers, it's well to keep in mind the following remark by John von Neumann:

> *Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.*



Icon on the Web

Information about Icon is available on the World Wide Web at

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

Visualizing Concatenation

In the last issue of the Analyst, we looked at collecting information for dynamic analysis and visualization. The problem is not just how to get the information — we've already done that for concatenation — but how to display it. Just writing a stream of numbers to a user's monitor is likely to be worse than useless.

For concatenation, a start would be to display the size of strings produced as successive bars whose heights correspond to size. For most programs, and especially for those of the greatest interest, there are far too many concatenations in most programs to display them all on the largest of monitors, even if the bars are reduced to one-pixel width.

This problem can be overcome by scrolling, so that the display shows only the most recent portion of the data.

Such a display is quite easy to create. Here's a procedure that takes a number as its argument and displays a vertical line of the corresponding height. For successive calls, new lines appear at the right and previous lines are scrolled to the left, eventually disappearing:

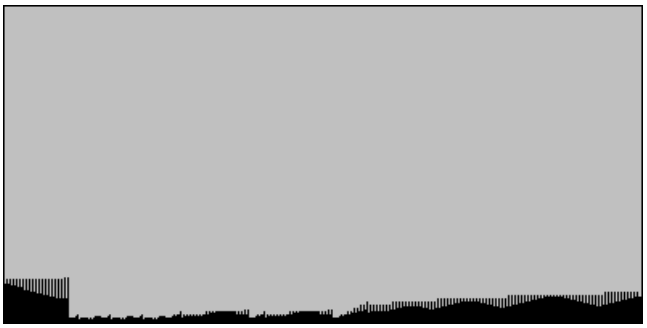
```
link graphics

$define Width       500
$define Height      200

procedure scroll_strip(n)

   initial {
      WOpen("size=" || Width || "," || Height,
        "bg=dark gray") |
          stop("*** cannot open window")

   CopyArea(1, 0, Width – 1, Height, 0, 0)
   EraseArea(Width – 1, 0, Width, Height)
   DrawLine(Width – 1, Height – n, Width – 1, Height)

   return

end
```

It takes only a little more to check for user events before returning:

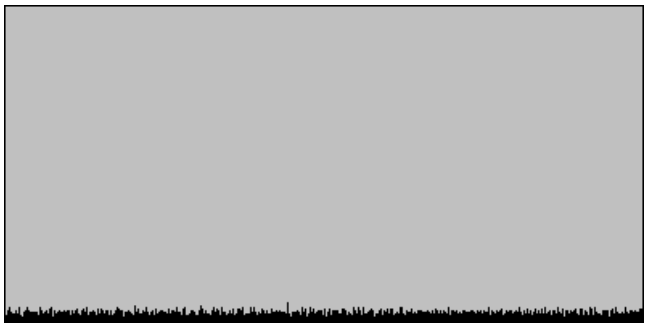
```
link interact
                    …
   while *Pending() > 0 then
      case Event() of {
         "p":  until Event() == "c"
         "s":  snapshot()
         "q":  exit()
         }

   return
```

If the user enters p, the visualization pauses until the user enters a c to continue it. An s produces a snapshot of the window, saved in an image file. The procedure snapshot() is part of the library module interact. It provides a dialog in which the user can specify the name of the image file. Finally, if the user enters q, the visualization terminates.

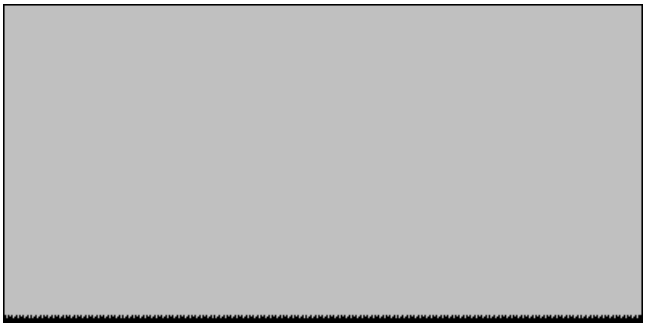
Snapshots from using this visualization technique for the eight test programs that do a significant amount of concatenation follow.



csgen



deal



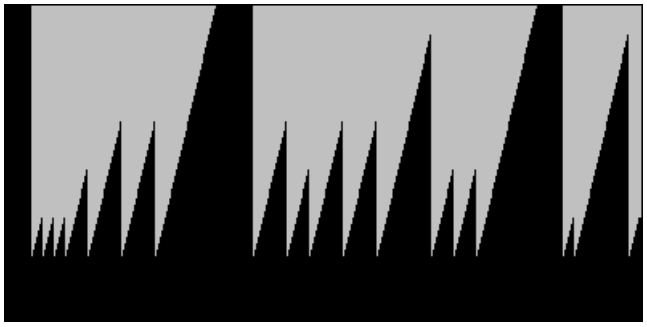
fileprnt

Downloading Icon Material

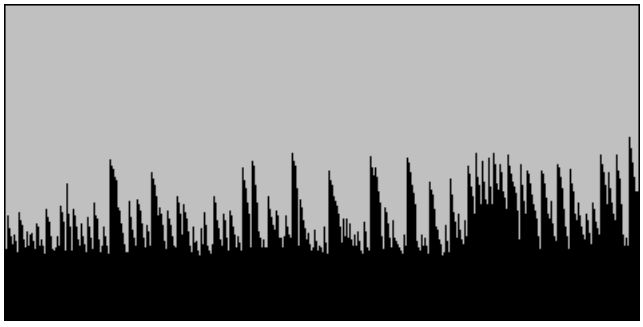
Most implementations of Icon are available for downloading via FTP:

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

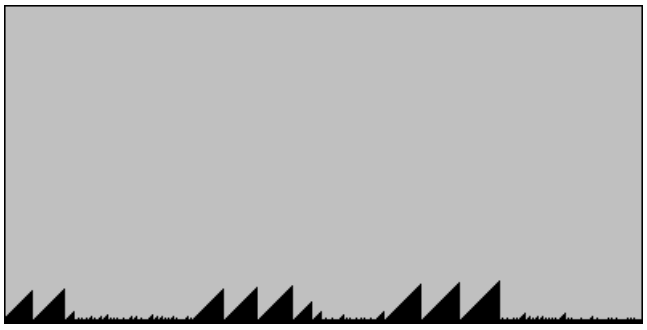
genqueen



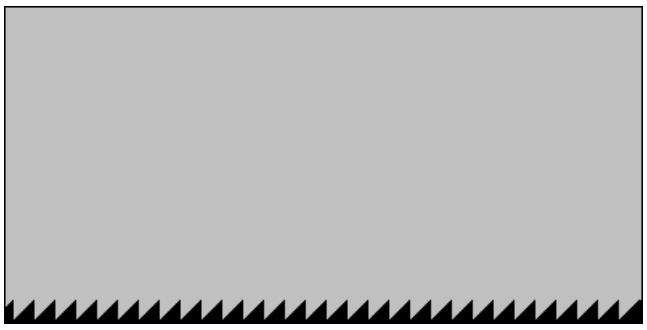
ipxref



kwic



press



turing

Some aspects of these images are worth noting. The image for genqueen shows that all concatenations are of the same length and hints at the problem we described in the last article on dynamic analysis. The monotonous pattern produced by turing also suggests how it was tested.

The tooth/sail/fin motif that appears in ipxref, press, and turing most likely is due to building up strings in a loop as described in the last article on dynamic analysis.

Scaling is an obvious problem, which we'll address later.

Another thing we can do in obtaining information about concatenation is to distinguish concatenation itself (||) from augmented concatenation (||:=).

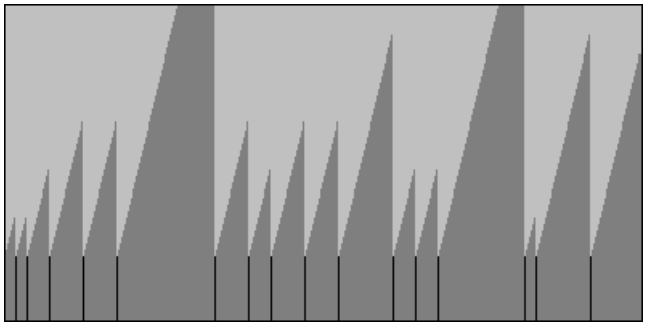
Different monitoring procedures could be used for the translation of these two operators, but an easier approach is to use the same procedure with an additional argument that's different for the two cases.

We chose "a" to indicate augmented concatenation and the null value for regular concatenation:

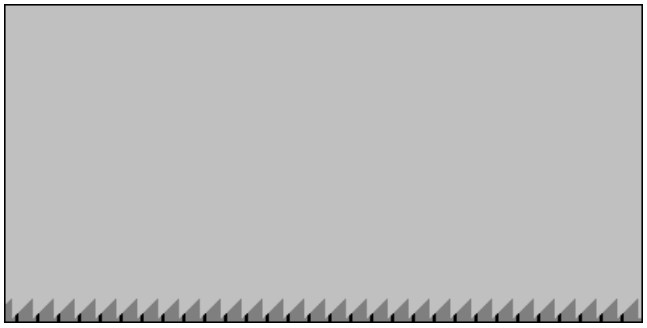
```
procedure Asgnop(op, e1, e2)            # e1 op e2

   if op == "||:=" then
      return cat(e1, " := cat__(", e1, ",", e2,
         ", \"a\")")
   else return cat("(", e1, " ", op, " ", e2, ")")

end
```

By putting the differentiating argument in the trailing position, a version of concat__() that doesn't distinguish the two can ignore the argument.

One way to distinguish the two concatenation operations in visualization is to assign different colors to them, say black for regular concatenation and gray for augmented concatenation. For the two test programs that use augmented concatenation, snapshots look like this:



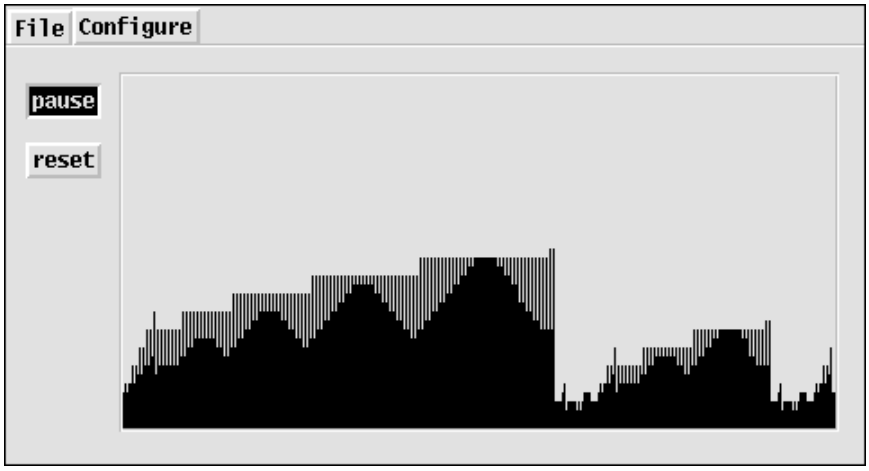
ipxref



turing

Incidentally, inspection of the other test programs shows that only these two could have used augmented concatenation, although others could have used a form of augmented concatenation that prepended rather than appended to the evolving string.

Additional functionality could be added to our simple visualization tool by using command-line arguments, but a better and more flexible approach is to provide a visual interface that allows the user to interact with the running visualization. The application shown below was created using VIB.



The pause button is a toggle that can be used to stop the display temporarily. The reset button clears the display region. The File menu provides for taking a snapshot and quitting the application. The Configure menu provides for scaling the display. In the image shown at the right, the scaling factor is 5.5. And, of course, there are keyboard shortcuts.

Since we've not yet finished the series of articles on building visual interfaces, we won't show the code for this application, but there's not much to it — only 114 lines for the program itself and 19 lines for VIB interface code.

Next Time

There are many things related to dynamic analysis and program visualization that we could do next. Before going on in this area, however, we'll have an article describing a framework for dynamic analysis.

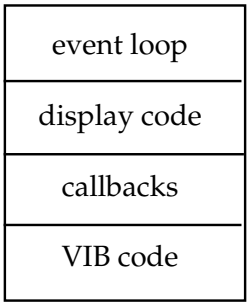
The Kaleidoscope

In previous articles, we've described how to build the visual interface for the kaleidoscope application. Now it's time to look at the rest the application.

There are two main parts to the job: writing the code for the kaleidoscopic display and handling user events.

We could, of course, have started with a program that displayed kaleidoscopic images without any interface at all. In fact, that is how the program started. It's not unusual to take an existing program and add a visual interface; in some cases this may be the best path to follow.

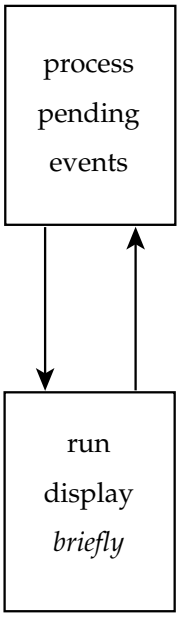
The kaleidoscope application can be viewed as having four main sections:



We'll leave the display portion to last, since it's largely independent of the rest of the applica-

tion, and first focus on the event loop and callbacks.

Understanding the logic of control flow in such a program is important. It can be viewed in this way:



Processing user events is the priority. Only when there are no pending user events does control pass to running the display. The "briefly" aspect of running the display is important; if control is not returned quickly to processing user events, the interface may be unresponsive and annoying.

It's worth noting that there are different kinds of applications with visual interfaces. Some, like the kaleidoscopic display, work "in the background" when the user is not producing events. Other applications are entirely "event driven" and only perform computations in response to user events. Editors are typical event-driven applications.

Of course, the phrase "process user events" is deceptively simple; this part of the program may be complex, there may be many things to handle, and the logic may be intricate.

When dealing with a visual interface produced by VIB, events are not handled by Event() but rather by a higher-level procedure ProcessEvent(). This procedure knows about vidget callbacks.

A typical event-processing loop for a program that does background work looks like this:

```
repeat {
  while *Pending() > 0 do
    ProcessEvent(root)
    # do a little background work
  }
```

where root is the root vidget. We'll show how to get this vidget later.

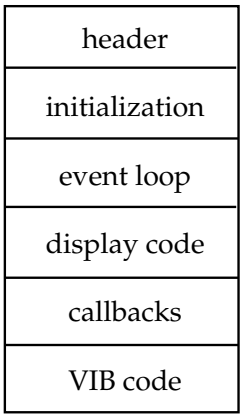
The expression

```
*Pending > 0
```

checks to see if any events are queued. If there are, they are processed. Control only goes to background work (in our case, the kaleidoscopic display) if there are no events pending.

For an entirely event-driven application, there is an alternative procedure, GetEvents(), which takes over control and just processes user events.

At this point we can add two sections to our program layout:



The header section consists of link and global declarations.

Programs with visual interfaces need to share values between callback procedures and other parts of the program. For this reason, such programs tend to have many global variables. Here are the ones for the kaleidoscope application:

```
link interact          # to take snapshots
link random            # to randomize display
link vsetup            # VIB support and graphics

#  Interface globals

global vidgets         # table of vidgets
global root            # the root vidget
```

```
global size             # size of view area
global half             # half size of view area
global pane             # graphics context for viewing

#  Parameters that can be set from the interface

global delayval         # delay between
global density          # number of circles
global draw_proc        # drawing procedure
global max_off          # maximum offset of circle
global min_off          # minimum offset of circle
global max_radius       # maximum radius of circle
global min_radius       # minimum radius of circle

#  State information

global draw_list        # pending drawing parameters
global reset            # view area needs resetting
global state            # null when display is running
```

We'll have more to say about the global variables as we describe other parts of the program.

The main procedure is simple; it just calls a procedure to initialize the program and then a procedure to run the display:

```
procedure main(args)

   init()

   kaleidoscope()

end
```

Here's the initialization code:

```
procedure init()

   vidgets := ui()

   root := vidgets["root"]
   size := vidgets["region"].uw
   if vidgets["region"].uh ~= size then
      stop("*** improper interface layout")

#  Produce different display on every execution.

   randomize()

#  Set initial values.

   delayval := 0
   density := 30
   max_radius := size / 4
   min_radius := 1
   draw_proc := FillCircle

   state := &null

#  Initialize vidget values.

   VSetState(vidgets["sld_speed"], delayval)
   VSetState(vidgets["sld_density"], density)
   VSetState(vidgets["sld_min_radius"], min_radius)
   VSetState(vidgets["sld_max_radius"], max_radius)
   VSetState(vidgets["sld_shape"], "discs")

#  Get graphics context for drawing.

   half := size / 2

   pane := Clone("bg=black", "dx=" ||
      (vidgets["region"].ux + half),
      "dy=" || (vidgets["region"].uy + half),
      "drawop=reverse")

end
```

The call of ui(), which is in the VIB section of the program, opens a window for the interface, draws the vidgets, initializes them, and returns a table of the vidgets, which are keyed by their IDs as given in VIB. The root vidget, which encloses and controls all others, has the ID "root". It is needed for processing events, as shown earlier.

The ID for the display region is "region", as shown when the interface was constructed. Its uw field gives the usable width of the region. A check is made against the usable height to be sure they are the same (the display requires a square region).

Next, randomize() is called to assure different displays for different runs. Then the initial parameters for the display are set: no delay, at most 30 circles at one time, a maximum circle radius that is one-fourth the size of the display region, a minimum that is one pixel, and filled circles (note that FillCircle is a procedure in Icon's graphics repertoire).

The state is set to null, corresponding to running as opposed to paused. Since state is initially null by default, it is not necessary to set it explicitly; doing so simply provides documentation.

The next section of code uses VSetVidget() to set the initial states of the vidgets. Note that the program must know the vidget IDs.

Establishing a graphics context for the display region finishes the initialization. (The fields ux and uy specify the upper-left corner of the usable part of the display region.) Because of the symmetrical geometry of the display, it's easiest to draw with the origin in the center, as shown. The drawop=reverse mode is used so that circles can be erased by redrawing them.

Here's the procedure that displays the kaleidoscope and handles user events:

```
procedure kaleidoscope()

  #  Each time through this loop, the display is
  #  cleared and a new drawing is started.

  repeat {

    EraseArea(pane, –half, –half, size, size)
    draw_list := []
    reset := &null

    #  In this loop a new circle is drawn and an old
    #  one erased, once the specified density has
    #  been reached. This maintains a steady state.

    repeat {
      while (*Pending() > 0) | \state do {
        ProcessEvent(root, , shortcuts)
        if \reset then break break next
        }
      putcircle()
      WDelay(delayval)

      #  Don't start clearing circles until the
      #  specified density has reached. (The
      #  drawing list has 4 elements for each circle.)

      if *draw_list > (4 * density) then clrcircle()
      }
    }

end
```

This procedure requires some explanation, because it depends both on the way the display is handled and on the way events are handled.

The procedure consists of a main repeat loop from which there is no direct exit — that's handled by the quit item in the File menu, which terminates program exeution. This loop initializes the display and is executed once again for each time the display is reset.

An empty list for drawing information is created, and reset is set to null, indicating that the display is initialized.

The inner repeat loop processes events and draws circles. The event processing loop is a bit more complicated than the one shown earlier:

```
while (*Pending() > 0) | \state do {
  ProcessEvent(root, , shortcuts)
  if \reset then break break next
  }
```

If there is a pending event, or if the display is paused, indicated by a nonnull value of state, ProcessEvent() is called. If an event is pending, it is processed. Otherwise, ProcessEvent() waits for one. (When the display is paused, there is nothing to do but to wait for an event.) The argument shortcuts is a procedure that handles keyboard shortcuts as opposed to vidget events.

Next reset is checked; if it is nonnull, something has happened that requires the display to be reinitialized. You need to look at the code to see that there are two loops to break out of: an inner while loop and its enclosing repeat loop. The next is not needed, since the end of the inner repeat loop also is the end of the outer repeat loop, but it helps clarify the control flow.

Once out of this loop, we're finally able to draw a circle: no events are pending, the state is set for a running display, and the display has been reinitialized if necessary. Here, again, is the drawing code:

```
putcircle()
WDelay(delayval)
if *draw_list > (4 * density) then clrcircle()
```

The delay is provided to allow the user to slow the display; on a fast platform, the display may just be a blur unless there are delays.

The procedure putcircle() "puts" one circle; drawing it in symmetric positions and adding four elements to draw_list for its description so that it can be redrawn later to erase it. If the size of the drawing list indicates the maximum density has been reached, clrcircle() is called to erase the oldest circle and remove its specifications from draw_list.

Here is the procedure putcircle():

```
procedure putcircle()
  local off1, off2, radius, color
  static colors

  initial colors := PaletteChars("c1")

  # get a random center point and radius

  off1 := ?size % half
  off2 := ?size % half
  radius := ((max_radius – min_radius) *
    ?0 + min_radius) % (half – ((off1 < off2) | off1))

  color := PaletteColor("c1", ?colors)

  put(draw_list, off1, off2, radius, color)

  outcircle(off1, off2, radius, color)

  return

end
```

The static variable colors is assigned a string of characters from palette c1, which has 90 colors. This provides an easy way to get colors that can be selected at random. We chose c1 after trying various other palettes.

The offsets for the centers of the circles are picked with an element of randomness, as are the radii and colors. The four values that characterize a circle are put on the end of draw_list, and outcircles() is called to do the actual drawing:

```
procedure outcircle(off1, off2, radius, color)

   Fg(pane, color)

   # Draw in symmetric positions.

   draw_proc(pane, off1, off2, radius)
   draw_proc(pane, off1, –off2, radius)
   draw_proc(pane, –off1, off2, radius)
   draw_proc(pane, –off1,–off2, radius)
   draw_proc(pane, off2, off1, radius)
   draw_proc(pane, off2, –off1, radius)
   draw_proc(pane, –off2, off1, radius)
   draw_proc(pane, –off2, –off1, radius)

   return

end
```

The procedure clrcircles() also draws circles, but it gets the specification of the oldest circle from draw_list and removes it in the process:

```
procedure clrcircle()

   outcircle(
     get(draw_list),                    # off1
     get(draw_list),                    # off2
     get(draw_list),                    # radius
     get(draw_list)                     # color
     )

   return

end
```

That takes care of the functionality of the application proper. Now we can move on to the callbacks.

The callback for the pause button is quite simple: It just sets state to the vidget value. (Of course, we designed the handling of the state for this.)

```
procedure pause_cb(vidget, value)

   state := value

   return

end
```

The callback for the reset button is even simpler:

```
procedure reset_cb(vidget, value)

   reset := 1

   return

end
```

The callback to change the speed of the display also is simple. It just sets the global variable delayval:

```
procedure speed_cb(vidget, value)

   delayval := sqrt(value)

   return

end
```

The square root of the value produced by the slider vidget is used to provide a more intuitive result for the user.

Setting the density of the display (the maximum number of circles displayed simultaneously, also involves setting a global variable. When this global variable is changed, however, the display must be re-initialized, so reset is set to a nonnull value:

```
procedure density_cb(vidget, value)

   density := value

   reset := 1

end
```

The callback to change the shape used for the display also just amounts to setting a variable. The callback value is the name of the button chosen from the radio-button vidget, and the value assigned to draw_proc is the corresponding procedure. The display must be re-initialized for this change also:

```
procedure shape_cb(vidget, value)

   draw_proc := case value of {
     "discs":  FillCircle
     "rings":  DrawCircle
     }

   reset := 1
```

```
    return
end
```

Next Time

That's all the space we have. In the next article, we'll show the rest of the callbacks and the VIB interface code.

From the Library

We've been asked how large the programs in the Icon program library are. When asked why this was a concern, the response was "obviously, big programs have more functionality than small ones."

That's generally true in some sense, assuming you include the size of linked procedures, but it's often the small programs that are the most useful. It also does not follow that all small programs are simple or easy to design and write.

The program ifilter for the Icon program library is an example of a small but very useful program. We described this program in an earlier Analyst article on string invocation [1]. We'll present the program here from a somewhat different perspective, but recapitulate the earlier material in case you didn't read that article or have forgotten it.

Here's the program as it appears in Version 9.2 of the Icon program library:

```
procedure main(args)
   local op
   op := proc(args[1], 2 | 1 | 3) |
      stop("*** invalid or missing operation")
   while args[1] := read() do
      every write(op ! args)
end
```

Just glancing at the code, it may not be at all obvious what this program does. Given what it does, it may not be obvious how useful it is.

What ifilter Does

The command line with which ifilter is called is all-important. The first command-line argument, which is required, is taken to be the string name of an Icon function or operator. The remaining arguments, which are optional, are taken as trailing arguments to which the function or operator is applied. These arguments are fixed and serve as parameters for any one use of ifilter.

The first argument for the function or operator, on the other hand, is obtained from standard input. So, for example,

```
ifilter map aeiou AEIOU
```

results in the evaluation of

```
map(read(), "aeiou", "AEIOU")
```

and the result is written out. This continues for all lines of input, "filtering" from standard input to standard output, replacing all instances of a, e, i, o, and u by their uppercase versions.

Of course, the function used need not require parameters:

```
ifilter trim
```

filters standard input, removing trailing blanks. (The second argument of trim() is effectively omitted and defaults to ' '.)

As suggested above, operators can be used in filtering also:

```
ifilter "%" 11
```

produces the residue modulo 11 of integers given in standard input — something we've found useful in working with versum sequences.

As written, ifilter always puts lines from standard input in the first argument position. This generally works well for functions, since Icon is designed so that in most cases the first argument of functions vary, while the rest of the arguments

often are constants that serve as parameters. This is not always the way functions are used and operators generally don't have this property. We'll come back to this point later.

How Does ifilter Work?

ifilter first uses proc() to check that the first command-line argument is indeed the name of a function or operator and, if so, to convert it to the function or operator so that string invocation is not needed in the processing loop.

The second argument to proc() is not used for functions, but for operators it distinguishes among unary and binary operators with the same string name and is necessary for ternary operators. (i to j by k is a ternary operator whose string name is "..."; can you think of any other ternary operators?) The order of alternation for the second argument of proc() selects binary operators instead of unary operators when there is a choice. For example, "*" is taken as the multiplication operator, not the size operator (more on this later).

If the first argument passes muster, it is called in the form

```
op ! args
```

There's a little trickery here; args consists of a list whose elements are the command-line arguments. For example, in

```
ifilter map aeiou AEIOU
```

args is

```
["map", "aeiou", "AEIOU"]
```

to get the list for invoking map(), we need only replace the first argument of args by the result of reading. The effect is

```
op ! [read(), "aeiou", "AEIOU"]
```

Finally

```
every write(op ! args))
```

is used to produce all the results of op in case it is a generator.

Procedures can be linked with ifilter to add them to the program, as in

```
icont –o sifilter ifilter strings.u
```

which produces a program sifilter that includes the procedures in the program library module strings.icn. To use these procedures, however,

```
invocable all
```

must be added to ifilter.icn. Otherwise the procedures in strings.icn would be deleted by the linker, since they are not referenced in the program itself.

Embellishments

As we've gone through the discussion of ifilter, we've mentioned some limitations and problems:

- The strings to be processed only can appear in the first-argument position.
- For unary and binary operators that have the same string name, only the binary one is available.
- All the results of a generator are produced even if all are not wanted. Think about

```
ifilter seq
```

We can take care of all of these limitations by adding command-line options to ifilter and rewriting parts of the program to make it more general.

The first question is design: How should these facilities be cast and what should the defaults be? In particular, are the ways these issues presently are handled by ifilter the best ones if more control is provided to the user?

We would argue that the first argument position is the best default for the reasons explained already. We also would argue that the choice of a binary operator over a unary one with the same string name is the best default. The string names in question are:

name	*unary operator*	*binary operator*
"+"	numeric value	addition
"–"	negative	subtraction
"="	tab(match())	equality
"@"	activation	transmission
"*"	size	multiplication
"/"	null test	division

It seems to us that the only questionable case is "*". Consistency demands a binary interpretation as the default for all.

The situation on generation is different. In the absence of being able to control generation, it seems the best choice is not to limit it; otherwise there is no way to get more than the first result of a genera-

tor. Given an ability to control generation, it might be better to limit it to one result as the default. The trade-off is between having to specify an extra command-line option to get generation as opposed to unexpected or runaway output. Our choice is to limit generators to one result in the absence of a command-line option to get more results.

To the user, the command-line interface looks like this:

–a *i*	argument position *i* for input strings; default 1
–o *i*	interpret ambiguous operator string name: *i* = 1 or 2; default 2
–l *i*	limit generation to *i* results with non-positive value indicating unlimited generation; default 1

Using options(), which we advocate in situations like this, does most of the work needed in processing the command line. Note that options() removes command-line options and their values from the list of command-line arguments but leaves anything else intact — specifically the operator or function name and any arguments. Thus, the beginning of the program might look like this:

```
link options
        …
   opts := options(args, "a+o+l+")
   i := \opts["a"] | 1
   limit := \opts["l"] | 1
   if limit < 1 then limit := 2 ^ 31

   if opts["o"] === (&null | 2) then {
      op := proc(pop(args), 2 | 1 | 3) |
         stop("*** invalid or missing operation")
      }
   else if opts["o"] = 1 then {
       op := proc(pop(args), 1 | 2 | 3) |
         stop("*** invalid or missing operation")
      }
   else stop("*** invalid –o option")
                  …
```

The way of specifying unlimited generation is a bit awkward, but it works and saves a special case later.

The handling of ambiguous operator string names now looks like this:

```
op := proc(pop(args), first | second | 3) |
   stop("*** invalid or missing operation")
```

and limiting generation is easy:

```
every write(op ! args) \ limit
```

Placing the strings being read in the right place is a bit more difficult:

```
lextend(args, i – 1)
args := args[1:i] ||| [&null] ||| args[i:0]
```

The procedure lextend(), which is from the Icon program library module lists, assures args is long enough. Then a list is created with a place holder for the strings to be read. The rest is easy:

```
while args[i] := read() do
   every write(op ! args) \ limit
```


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

For reference, here is the complete program:

```
invocable all

link lists
link options

procedure main(args)
   local op, opts, i, interp, limit, first, second

   opts := options(args, "a+o+l+")
   i := \opts["a"] | 1

   case opts["o"] of {
      &null | 2:   {
         first := 2
         second := 1
         }
      1:   {
         first := 1
         second := 2
         }
      default:   stop("*** invalid value for –o")
      }

   limit := \opts["l"] | 1
   if limit < 1 then limit := 2 ^ 31

   op := proc(pop(args), first | second | 3) |
      stop("*** invalid or missing operation")

   lextend(args, i – 1, "")
   args := args[1:i] ||| [&null] ||| args[i:0]

   while args[i] := read() do
      every write(op ! args) \ limit

end
```

Comments

Granted, the embellished program itself is nearly four times the size of the original one. If you add in the procedures that it links, which gives the true size of the program, it's more than 13 times the size of the original program: 95 lines of source code altogether. And the final program certainly has more functionality. It's not clear that it's more than three times as useful as the original program, that the additional features will be used often, or that anyone will remember they're there when they are needed.

These are common aspects of additional program functionality. Extra features increase program size and often are rarely used, but when they are needed, they can be very useful indeed.

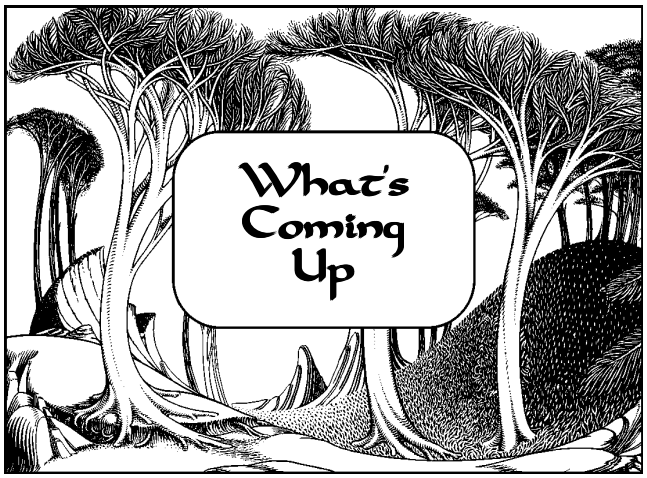
One question is knowing when to stop — knowing whether additional features are worthwhile or just contributing to creeping featurism that weighs down the program and eventually may render it difficult to use. (If you're familiar with the evolution of commercial word processors, you know exactly what we mean.)

Good design and (especially) good defaults, are important. Ideally, a user who does not need extra features should not have to know about them.

Finally, an admission. ifilter has been around in one form or another for a long time. We've reinvented it more than once, having forgotten it already existed. And the new features here came about only as a result of writing this article and being forced, by the exposition, to address its deficiencies.

Reference

1 . "Applications of String Invocation", Icon Analyst 29, pp. 3-6.



We'll finally finish the kaleidoscope application in the next issue of the Analyst, but that's not the end of the series on visual interfaces: We have custom dialogs to cover also.

We have articles on versum sequences backed up, and we'll try to get one into the next issue.

We'll shift direction in our series of articles on dynamic program analysis and describe a framework within which various kinds of analyses can be cast in a consistent fashion.

We have other articles for **From the Library** and several candidates for **Programming Tips**. As usual, we have to see how everything fits together.