The Icon Analyst

In-Depth Coverage of the Icon Programming Language

August 1992
Number 13

In this issue …

Face Lift for the Analyst

The Analyst is now beginning its third year of publication. Starting with this issue, we've made a few changes in the typographical design to make the Analyst more readable and, we hope, more attractive.

Until now, we've used Times for the body text in which articles are set and Helvetica for captions and program material. These type faces have their merits and both are widely used — so widely used, in fact, as to be painful for persons interested in typographic design.

We've decided that the Analyst needed a body face that is more "open" than Times. We tried several alternatives and finally picked Palatino for its grace and legibility.

The new sans-serif face that replaces Helvetica is Frutiger. Frutiger actually is very similar to Helvetica — it takes careful examination to find the small differences in characters between the two faces. The characters in Frutiger are, however, a little wider than in Helvetica, which makes Frutiger easier to read.

We've also increased the type size and now are setting 11 points on 13 points instead of the former 10 on 12.

In addition to the changes in type faces, we've changed the style of the Analyst's title slightly to give it a crisper look. For this, we used the "Inline" effect from Effects Specialist.

Solutions to Exercises

In the last issue of the Analyst, we posed some problems involving expressions to generate sequences of values. Our solutions follow. We've added some discussion, since there are different ways of solving most of the problems, as well as different ways of formulating the same solution in Icon.

If you produce better or more interesting solutions than the ones that follow, please send them to us. We'll include them in a future issue of the Analyst.

Problem 1. A sequence consisting of the names of the months of the year: "January", "February", … "December".

This one is easy and straightforward. A simple alternation is all that's needed:

```
"January" | "February" | "March" | "April" |
  "May" | "June" | "July" | "August" |
  "September" | "October" | "November" |
  "December"
```

On the other hand, if you need a list of the names of the months for some other reason, such a list could serve double duty:

```
months := [
  "January", "February", "March", "April",
  "May", "June", "July", "August",
  "September", "October", "November",
  "December"]
```

Then !months can be used to generate the names.

Problem 2. A sequence consisting of the lowercase letters in increasing alphabetical order.

You could use a simple alternation here too, but there's a much more concise method:

```
!&lcase
```

Although &lcase is a cset, !&lcase generates one-character strings.

Problem 3. A sequence consisting of the lowercase letters in decreasing alphabetical order.

To get the letters in decreasing alphabetical order, you could do something like this:

```
&lcase[26 to 1 by –1]
```

We prefer

```
!reverse(&lcase)
```

It's worth noting that reverse() is only evaluated once in this expression.

Problem 4. An infinite sequence consisting of the lowercase letters in increasing alphabetical order, repeatedly.

The word "repeatedly" should suggest repeated alternation:

```
|!&lcase
```

You need to watch the els and vertical bars here — they look very much alike in a sans-serif face. In fact, we've set the vertical bars in a different face for this article to make them somewhat easier to distinguish from els.

Problem 5. An infinite sequence consisting of the lowercase letters in decreasing alphabetical order, repeatedly.

The same idea applies here:

```
|!reverse(&lcase)
```

Problem 6. A sequence consisting of strings representing the times in minutes in the 24-hour day, starting midnight and ending at the minute before midnight: "00:00", "00:01", … "00:59", "01:00", … "23.59".

The solution to this problem has two components: a generator for the hours and a generator for the minutes.

0 to 23 generates the hours and 0 to 59 generates the minutes, but the results need to be padded with zeros:

```
right(0 to 23, 2, "0")
right(0 to 59, 2, "0")
```

All that remains is a concatenation with colons added as separators:

```
right(0 to 23, 2, "0") || ":" ||
  right(0 to 59, 2, "0")
```

Problem 7. An infinite sequence consisting of the digit 1: 1, 1, 1, 1, … .

This one is simpler than the previous repeated sequences, although it may seem strange at first sight:

```
|1
```

Problem 8. An infinite sequence of randomly distributed strings "H" and "T".

The expression ?s produces a randomly selected character from s, so ?"HT" produces an "H" or a "T" at random. Again, repeated alternation provides the desired sequence:

```
|?"HT"
```

Problem 9. An infinite sequence consisting of randomly selected digits.

The same idea works here:

```
|?&digits
```

To generate integers instead of strings, you could do this:

```
integer(|?&digits)
```

or this:

```
|(?10 – 1)
```

Problem 10. An infinite sequence consisting of randomly selected characters.

"Characters" here includes all of the possible 256 characters, so a solution is:

```
|?&cset
```

Another possibility is:

```
|char(?256 – 1)
```

Problem 11. An infinite sequence consisting of the squares of the positive integers: 1, 4, 9, 16, … .

The general approach to the problem of generating a sequence of values based on the sequence of positive integers is to generate the positive integers and then apply a function to the results to get the desired sequence.

One model for this is

```
i := 0 & |(i +:= 1) & f(i)
```

where f(i) produces the desired value (which need not be an integer, of course). This also can be written as

```
(i := 0, |(i +:= 1), f(i))
```

For the squares, this becomes

```
(i := 0, |(i +:= 1), i ^ 2)
```

There are lots of variations on this theme, such as using

```
(i := 1) | |(i +:= 1)
```

to produce the positive integers. There is an easier way: `seq()`. Using it, an expression to generate the squares is just

```
seq() ^ 2
```

Note that something like

```
(1 to 1000000) ^ 2
```

does not satisfy the problem specification, since it only generates a finite sequence of squares.

Problem 12. An infinite sequence consisting of the Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, … .

It's hard to keep the Fibonacci numbers out of exercises like this, since they have so many fascination properties and there are so many ways to generate them.

Fibonacci numbers are defined by the recurrence relation:

$$\begin{array}{ll} f(i) = 1 & i = 1, 2 \\ f(i) = f(i-1) + f(i-2) & i > 2 \end{array}$$

It's easy to formulate a recursive procedure based on this recurrence relation, but since procedures are not allowed in solutions to these exercises, a recursive procedure surely can't help. An iterative procedure suggests a method:

```
procedure fibseq()

  suspend (i | j) := 1
  repeat {
    i :=: j
    suspend j +:= i
    }

end
```

It takes a little cleverness to get from a procedure to a procedure-free expression, but once you see the method, it's not all that hard:

```
((i | j) := 1) | |((i :=: j) & (j +:= i))
```

We've added more parentheses than are necessary to be sure the grouping is clear.

The left expression in the main alternation initializes the two variables and also produces the first two values in the Fibonacci sequence. After that, all the "action" is in the right expression in the main alternation.

There's an entirely different approach that uses a "closed form" from which the *i*th Fibonacci number can be computed directly:

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}}$$

This is rather complicated, especially for repeated computation in a sequence. There's a simpler form [1]:

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^i}{\sqrt{5}} \quad \textit{rounded to the nearest integer}$$

Pre-computing the constants, this can be cast in the form of the following Icon expression:

```
integer(0.5 + 0.4472136 * 1.618033989 ^ seq())
```

This works correctly for the first few dozen Fibonacci numbers, but at some point floating-point arithmetic in Icon lacks the precision necessary to yield the correct integer value.

Problem 13. An infinite sequence consisting of the factorials of the positive integers: 1, 2, 6, 24, 120, 720, … .

After the Fibonacci numbers, the factorials are easy:

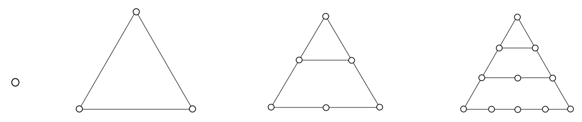
```
(i := 1) & |(i *:= seq())
```

The repeated alternation can be moved inside the parentheses, since all that's necessary is for the variable to be generated repeatedly:

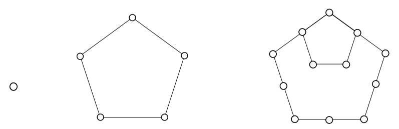
```
(i := 1) & (|i *:= seq())
```

Problem 14. An infinite sequence consisting of the "triangular numbers": 1, 3, 6, 10, 15, 21, … .

The triangular numbers are an instance of the *polygonal numbers*, the two-dimensional part of the more general *figurate numbers* [2]. As the name suggests, these numbers have geometrical interpretations. The first four triangular numbers are given by the number of nodes in the following figures:



Similarly, the first three pentagonal numbers are given by the following figures:



We'll leave it to you to construct additional figures in this sequence.

There are formulas for the figurate numbers. For example, the *i*th triangular and pentagonal numbers are given by the formulas

$$i(i + 1)/2 \quad \text{and} \quad i(3i - 1)/2$$

The sequence for the triangular numbers therefore can be produced simply by using the first formula, incrementing *i* at each step as in previous expressions. There's an easier approach, however, and one that's recommended for unknown sequences: Take the first difference of successive terms, to see if it suggests something. In the case here, the first difference yields 2, 3, 4, 5, 6, … . In other words, if $t(i)$ is the *i*th triangular number, then

$$t(i) = t(i - 1) + i$$

Of course, we haven't proved this, but it's easy enough using the formula above.

From this, a sequence to generate the triangular numbers is just

```
(i := 1) | (i +:= seq(2))
```

Problem 15. An infinite sequence consisting of the prime numbers: 2, 3, 5, 7, 11, 13 … .

Once you're working with integer sequences, it's natural to wonder about the sequence of the most mysterious numbers of all — the prime numbers. While there is no known method for computing primes efficiently in sequence, a brute-force method is simple — just generate all the integers and filter out those that are not prime. The trivial observation that 2 is the only even prime leads to the following expression:

```
2 | ((i := seq(3, 2)) &
   (not(i = (2 to i) * (2 to i))) & i)
```

The second operand of the conjunction,

```
not(i = (2 to i) * (2 to i))
```

fails if i can not be represented as the product of two integers. Otherwise, the result of the expression is just i, the third operand of the conjunction.

It is, of course, not necessary to check all the way to i * i. A much better test is

```
not(i = (k := (3 to sqrt(i) by 2)) * (i / k))
```

There are all kinds of other possibilities, which we leave as exercises.

Problem 16. An infinite sequence consisting of *n* copies of each positive integer *n*: 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, … .

This one is entirely different from the previous exercises. This solution uses limitation in combination with repeated alternation:

```
i := seq() & (|i \ i)
```

Acknowledgment

Gregg Townsend provided the last method given here for computing the Fibonacci numbers. Andrew Appel provided the Icon expression on which the first method of generating the primes numbers given above is based. Bill Griswold and Gregg Townsend provided the basis for the second one. See References 3 and 4 for more discussion of ways to compute prime numbers in Icon.

References

1. Knuth, Donald E. *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, Addison-Wesley, Reading, MA., 1969, pp. 78-83.

2. *A Handbook of Integer Sequences*, N. J. A. Sloane, Academic Press, 1973.

3. "Programming Corner", *The Icon Newsletter* 25, pp. 10-11.

4. "Programming Corner", *The Icon Newsletter* 35, p. 4.

An Introduction to X-Icon

X-Icon is the name we use for a version of Icon that supports extended display facilities via the X Window System [1].

The X Window System, which we'll just call X from now on, is a large and complicated package that supports windows, graphics, displayed text, interaction with input devices such as a keyboard and mouse, and so on. There does not seem to be a single word that aptly describes all of these capabilities. We'll use "graphics" subsequently with the understanding that the term as used here includes all that goes along with displays.

X-Icon is designed to provide graphical capabilities for Icon in a way that avoids many of the details and tedious programming tasks that are necessary when, for example, accessing X directly from C.

With X-Icon you can display windows; draw points, lines, and various shapes; display text in a variety of sizes and type faces; accept input directly from the keyboard; determine the position of the mouse when buttons are pressed and released; and so forth.

In many cases, X-Icon provides access to the underlying X capabilities in a relatively direct manner. (If you are familiar with X, X-Icon interfaces X at the functional, Xlib level, not at the Xt toolkit level). In other respects, however, X-Icon handles matters automatically that otherwise would require intricate and error-prone coding.

For example, X mandates an event-driven paradigm in which interaction between the user and the program must be handled at every instant by the program. In X-Icon, on the other hand, this event-driven model is optional. It is possible to write many graphic applications in X-Icon with very ordinary looking code.

Similarly, the display is maintained automatically by X-Icon. You don't have to worry about redrawing the screen if a window is opened, moved, or closed.

X-Icon Capabilities

From the perspective of a programmer, X-Icon offers the following kinds of capabilities:

• Windows can be opened and closed as desired, using the functions open() and close().

• Text in arbitrary type faces and sizes, including proportional-width faces, can be displayed using write() and writes() in the same manner as strings are written to files.

• Individual characters entered from the keyboard can be processed as they are typed without having to wait for a return character.

• Points, lines, arcs, smooth curves, and polygons can be freely intermixed with text.

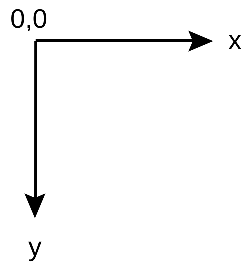
• Colors can be used for both text and graphics.

As you might imagine, it takes quite a bit of mechanism to accomplish all of this. (No one has succeeded in finding a simple way of dealing with graphics, and all present systems are painfully large and complex.) In addition to what X-Icon does automatically, it provides 39 functions and 12 keywords associated with graphics.

Obviously, we can't describe everything about X-Icon here. We'll just attempt to illustrate, mostly by example, what it's like to program in X-Icon.

Basic Window Operations

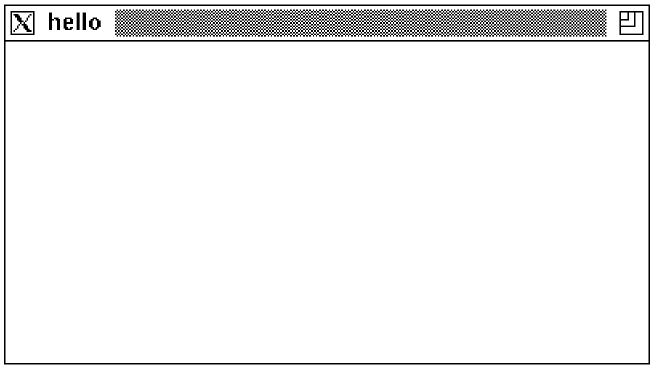
In order to use X, you need to know something about its coordinate system. The screen and each window are treated as portions of an x-y plane of pixels, with the origin (0,0) at the upper-left corner. Pixel positions increase in the x direction to the right and downward in the y direction:



Suppose you want to create a 400-by-200 pixel window on the screen with the upper-left corner of the window at x-y position (10,20). This is done in X-Icon with the open() function, giving a name (title) for the window as the first argument (much as the name of a file is given), and with "x" as the second argument to indicate that an X window is to be opened. Subsequent arguments give the initial values of attributes associated with the window. In the case above, this might be:

```
example := open(
   "hello",
   "x",
   "pos=10,20",
   "width=400",
   "height=200"
   ) | stop("*** cannot open window")
```

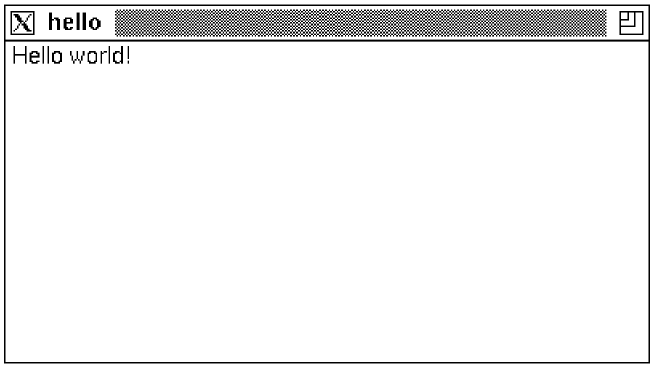
The value returned by open() in this case is assigned to a variable, example, to provide a way to refer to this widow subsequently. The usual alternative is provided to terminate program execution with an error message in case it is not possible to open the window. The result of the open() is a blank window:



You now can write text in the window, as in

```
write(example, " Hello world!")
```

which produces:

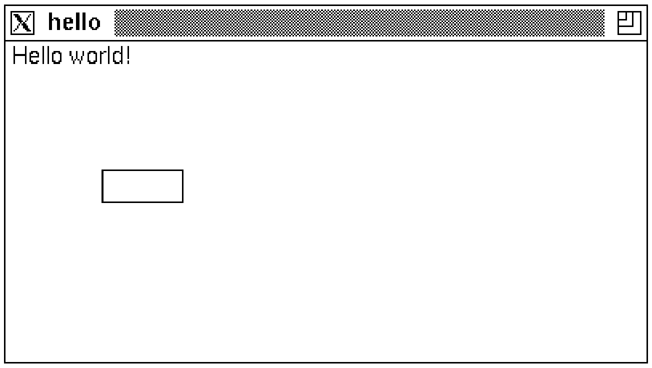


```
Hello world!
```

Drawing (lines, shapes, and so forth) is done with functions. For example, the following function call draws a rectangle 50 pixels wide and 20 pixels high with its upper-left corner at position (60,80) in the window:

```
XDrawRectangle(example, 60, 80, 50, 20)
```

The result is:



```
Hello world!
```

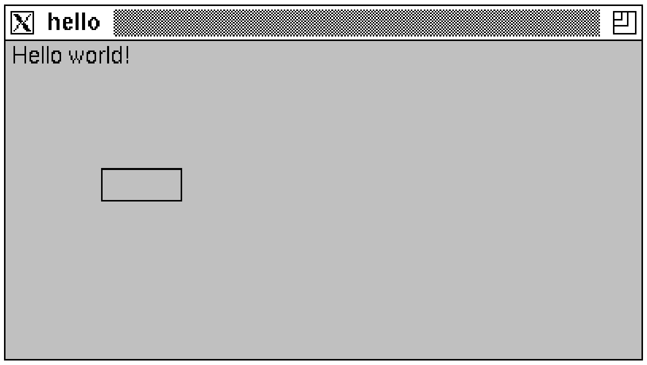
The names of most X-Icon functions start with an X (to distinguish them from similar functions that might be written for other graphical systems, such as Microsoft Windows). The rest of the name usually is derived from the name of the corresponding Xlib function. The result often is rather long and cumbersome, but at least it is descriptive, which becomes important with such a large repertoire of functions.

Although it's not shown here, several rectangles can be drawn with one call of XDrawRectangle(), which takes an arbitrary number of arguments that specify successive quadruples of x-y coordinates, width, and height. This is true for most drawing functions.

Window Attributes

As suggested by the example of open() given earlier, an X window has numerous attributes. X-Icon supports 44 attributes in all, with default values for attributes that are not explicitly specified.

Two important attributes are the background and foreground colors of a window. A window is filled initially with the background color when it is opened. Text, points, and lines are rendered in the foreground color. As indicated in the preceding example, the default background color is white and the default foreground is black. If the window had been opened on a color monitor with the additional attribute "bg=gray", the subsequent operations would have produced a window that looks like this:



The attributes associated with a window can be changed after the window is opened. For example,

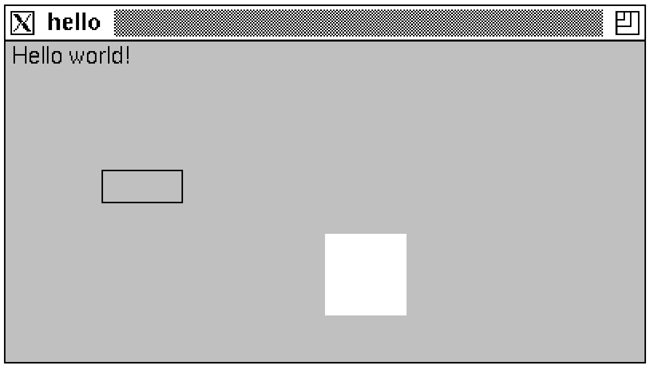
```
XFg(example, "white")
```

changes the foreground color of example to white.

Some functions draw shapes that are filled in the foreground color. For example,

```
XFillRectangle(example, 200, 100, 50, 50)
```

draws a solid white square:



Example — Random Shapes

What we've described so far is enough to write a simple X-Icon program to display shapes of randomly selected colors and sizes – a (poor) sort of "modern" art. As with any such program, there are a lot of features you might imagine. We'll start something simple and inflexible.

Let's specify a window 500 pixels wide and 300 pixels high. Allowing the size to be specified on the command line is left as an exercise [2]. We'll draw filled rectangles first, and generalize this later. And we'll select colors from a small palette.

The dimensions of the rectangles will be selected randomly from between one pixel and the window dimensions. Their positions will be randomly selected also.

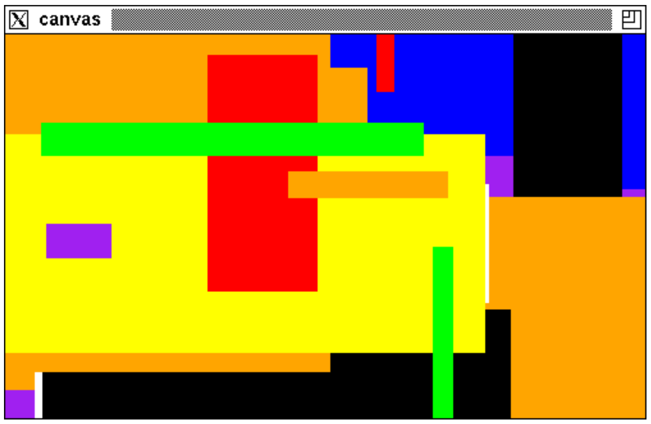
Here's a first cut at the program:

```
procedure main(arg)
  height := 500
  width := 300
  colors := [
    "red",
    "blue",
    "green",
    "yellow",
    "purple",
    "white",
    "black"
    ]
  canvas := open(
    "canvas",
    "x",
    "height=" || height,
    "width=" || width
    ) | stop("*** cannot open canvas")
  repeat {                  # drawing loop
# select dimensions
    w := ?width
    h := ?height
# select center
    x := ?width – w / 2
    y := ?height – h / 2
# select color
    XFg(canvas, ?colors)
# draw a rectangle
    XFillRectangle(canvas, x, y, w, h)
# pause to reflect
    delay(100)
    }                       # continue
end
```

The delay is provided to prevent the drawing from proceeding too rapidly.

Incidentally, when the sizes and positions of the rectangles are selected in this way, portions of them may fall outside the window. Such portions are "clipped" and not drawn.

A typical display from this program is:



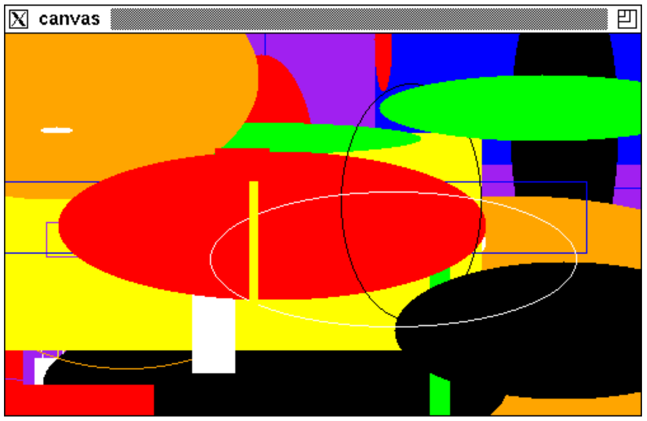
It's also possible to draw filled ellipses, as well as unfilled shapes. All the X-Icon functions for doing these kinds of things take the same arguments, so it's easy to generalize this program by specifying a list of functions

```
shape := [
  XFillRectangle,
  XFillArc,
  XDrawRectangle,
  XDrawArc
  ]
```

and replacing the call above to select a drawing function randomly:

```
(?shape)(canvas, x, y, w, h)
```

Now a typical display looks like this:



Events

When the mouse pointer is in an X-Icon window, pressing a key, pressing a mouse button, dragging the mouse with a button pressed, and releasing a mouse button cause "events" in that window.

Events are queued so that they are not lost if it takes a while for the program to get around to processing them. The queue is an Icon list that is the value of XPending(window). For example,

```
*XPending(window) > 0
```

succeeds if there is an event pending in window.

The function XEvent(window) produces the next event for window and removes it from the queue. If there is no pending event, XEvent() simply waits for one. When XEvent() removes an event from the queue, the position on the screen at which the event occurred also is recorded in Icon keywords.

The value of a keyboard event is a one-character string corresponding to the key pressed. Mouse events are integers for which there are corresponding keywords. For example, &rpress and &rrelease are the values for the events that occur when the right mouse button is pressed and released, respectively.

Events can be used to control a program. For example, pressing and releasing the right mouse button could be used to cause the drawing program given earlier to stop and start. Similarly, pressing the q key on the keyboard could be used to cause the program to terminate.

To illustrate this, a call to a procedure

```
checkevent(canvas)
```

could be added at the end of the repeat loop. The procedure might look like this:

```
procedure checkevent(window)
# process pending events
  while *XPending(window) > 0 do {
    case XEvent(window) of {
      "q":  exit()   # quit
      &rpress:  {    # pause
        until XEvent(window) === &rrelease
        return       # resume
        }
      }
    }
  return
end
```

The while loop continues as long as there is an event pending in window. If the pending event is a q, the program execution terminates occurs via exit(). (The window is closed and vanishes in such a case.) If the right mouse button is pressed, control drops into another loop waiting for the button to be released. Note that any other events that occur are ignored.

More About X-Icon

As suggested by X-Icon's 39 functions and the 44 window attributes it supports, we've only begun to touch on X-Icon's capabilities here.

We'll have more to say about X-Icon in future issues of the Analyst and also have some articles on some of the applications we've written using X-Icon.

In the meantime, if you're interested in knowing more about X-Icon, we have a 41-page technical report that describes the full range of its capabilities [3]. As a subscriber to the Analyst, you can get a free copy of this report. Just ask for the "X-Icon TR" and be sure to mention that you are a subscriber to the Analyst.

Where X-Icon is Going

X-Icon presently runs on several UNIX platforms. The code itself has no particular dependency on UNIX, and it should be possible to get X-Icon running on other platforms that support X.

The X in X-Icon of course refers to its use of the X Window System. But the X also could be taken to reflect its somewhat experimental nature.

We've learned a lot about incorporating graphic capabilities in a high-level programming language since we started to develop X-Icon. We now know better ways of handling some things. And, of course, X is not the only graphics game in town. We only chose it for our work because it is widely available and readily accessible in our environment.

Similar kinds of facilities could be provided, for example, on the Amiga, the Atari ST, the Macintosh, Microsoft Windows, and OS/2 using their own graphic systems. Such graphic systems all differ in various respects. Ideally, graphical capabilities for Icon should be independent of any particular underlying graphic system and be portable across a wide range of platforms.

This is a tall order — a capability of one graphic system (and one that users of that system view as important) often is lacking in another system. Portability and commonality across diverse platforms tend to exclude some features on individual platforms or else require considerable work to add to platforms that don't already support them.

There's also the problem that users of a specific platform expect applications to support the "look-and-feel" of that platform.

No one has yet produced a satisfactory universal system for graphics and we don't have any pretensions that we can do this. We think the proper approach is to develop higher-level ways of using graphics in Icon based on X and then to explore other graphics systems.

The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
Editors

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

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

voice: (602) 621-8448

fax: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

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

The University of Arizona
Tucson Arizona

and

The Bright Forest Company
Tucson Arizona

Acknowledgments

X-Icon is the inspiration of Clint Jeffery, who did most of the design and implementation. Some of the material in this article was adapted from an earlier paper [4].

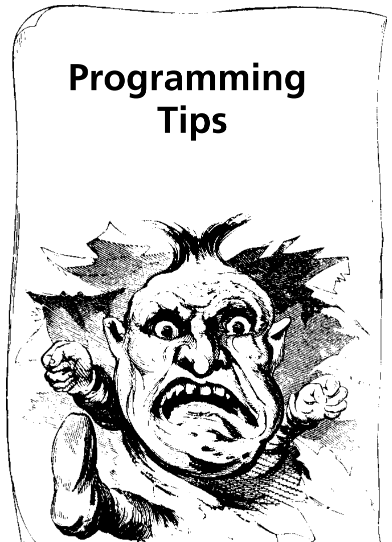
References

1. "The X Window System", R. W. Scheifler and J. Gettys, *ACM Communications on Graphics*, Vol. 5, April 1986, pp. 79-109.

2. "Command-Line Arguments", The Icon Analyst 11, pp. 7-10.

3. *X-Icon: An Icon Window Interface*, Clinton L. Jeffery, technical report TR91-1, Department of Computer Science, The University of Arizona, 1991.

4. "X-Icon: An Icon Window Interface", Clinton L. Jeffery and Ralph E. Griswold, *Proceedings of the Fifth International Conference on Symbolic and Logical Computing*, April, 1991, pp. 19-38.



Recursive Generators

Recursion is a powerful and well-known tool for formulating solutions to problems in which data or algorithms are defined in terms of themselves. The ability of an Icon expression to generate a sequence of values often can be used to provide concise and elegant formulations of computations that otherwise would be tedious and error-prone. The combination of recursion and generation is powerful indeed.

The term "recursive generator" refers to a situation in which a procedure suspends with a call to itself (perhaps though intermediate calls). Several examples of recursive generators are given in the Icon language book [1].

Like many other "light-bulb" experiences, it's often difficult to see how to formulate a recursive generator, but when you do, the insight may be a revelation.

We recently had such an experience. We were trying to come up with a *nom de plume* for the author of a projected mystery novel. We had lists of first names (Mary, George, Constance ...) and last names (Glenn, Roberts, Hankle ...) and we were trying various combinations. We finally decided that it would be fun, and possibly useful, to try all possible combinations. There were enough possible combinations that a program was needed.

This kind of thing is easy enough to do in Icon:

```
first := ["Mary", "George", "Constance", ... ]
last := ["Glenn", "Roberts", "Hankle", ... ]

every write(!first, " ", !last)
```

or, for something more complicated than just printing,

```
every name := !first || " " || !last do
  process(name)
```

We then asked the usual question programmers ask: "How do we generalize this to *n* lists?"

There's clearly no problem with formulating a solution for three lists, four lists, and so on for any specific value of *n*, but suppose we don't know how many lists there will be? The problem is that every generating expression has to be written down in the program, and if you don't know how many there are in advance, there's no obvious way to do this.

In such a situation, a good approach is to think of recursion to handle dynamically what cannot be handled statically. That is, devise a procedure that explicitly handles one of the *n* components of the problem and then call it recursively to handle the remaining *n*–1 components.

So we started by postulating a procedure

```
allcat(L1, L2, L3, ... )
```

to generate all possible concatenations of the elements of L1, L2, L3, ... We expected allcat() to contain something like this

```
!L1 || allcat(L2, L3, ... )
```

With this in hand, only the details remain, as they say. (And this is where many well-conceived plans end in failure.)

A good way to proceed is first to find *a* solution and then refine it. In the refinement you may find a model that will work in similar situations in the future without need for the intermediate solutions that come from a first effort.

In the case here, the question is how to pass an arbitrary number of lists to allcat(). An easy answer is to put them in a list:

```
allcat([L1, L2, L3, … ])
```

Note that the list of lists need not appear explicitly in a program — it could be built up prior to the call to allcat().

The procedure allcat() then might start out like this:

```
procedure allcat(L)
   L1 := get(L)      # oops; get() might fail
   # now compute !L1 || allcat(L)
                     …
```

At this point, three things need to be considered:

1. How to handle the case when L is empty.
2. How to stop the recursion.
3. How to return the concatenations.

L might be empty because allcat() is called with an empty list initially. While that might not be expected, it needs to be handled. The natural interpretation of a empty list is that there are no items to concatenate. In this case allcat() should produce no results — that is, it should fail:

```
L1 := get(L) | fail
```

It's also possible for L to become empty as a result of taking a list off of it as suggested in the code fragment above. This occurs when allcat() is called with a list that contains a single element. The "concatenations" from a single list consist of just the elements in it, which suggests

```
L1 := get(L) | fail
if *L = 0 then suspend !L1
```

The suspend is used, of course, since allcat() is supposed to *generate* concatenations. Since the argument of suspend generates all the elements of L1, a call of the procedure in which this expression occurs also generates all the elements of L1.

The final step is to handle the general case in which allcat() is called recursively:

```
else suspend !L1 || allcat(L)
```

That is, all the results generated by allcat() for the remainder of the list L are concatenated onto each of the strings generated by !L1.

But there's a bug here. Every time allcat() is called, it removes an element from L. Because of Icon's pointer semantics [2], all calls of allcat() are working on the *same* list. L is entirely consumed in the concatenations for the first result produced by !L1. The formulation above won't work at all because of this. The thing to do is pass a copy of L. With this, the procedure becomes:

```
procedure allcat(L)
   L1 := get(L) | fail
   if *L = 0 then suspend !L1
   else suspend !L1 || allcat(copy(L))
end
```

At this point, you might note that L does not have to be a list of lists — it could be a list of any values for which element generation produces strings or values convertible to strings. For example,

```
allcat([&lcase, &ucase])
```

generates

```
aA
aB
…
zA
zB
…
zZ
```

Similarly, the names as discussed at the beginning of this article are generated by

```
allcat([file, " ", last])
```

This works as desired because !" " generates a single " ".

This kind of usage suggests changing the identifier L1 to x:

```
procedure allcat(L)
   x := get(L) | fail
   if *L = 0 then suspend !x
   else suspend !x || allcat(copy(L))
end
```

As indicated above, this is *a* solution. It works, but it's not the best possible formulation. The awkward part of this solution is the way that allcat() has to be called — with a list argument. This means the person using allcat() has to construct a list. A better approach is to allow the user to call allcat() with as many arguments as are needed, as is possible for built-in functions such as write(). That is, instead of having to write

```
allcat([x1, x2, x3, … ])
```

the user can write

```
allcat(x1, x2, x3, … )
```

Icon has a feature just for this purpose: A procedure can be declared with an argument that is a list that automatically gatherers up all the arguments in a call of the procedure. In the case here, the procedure header is

```
procedure allcat(L[ ])
```

where the brackets indicate this method for handling the arguments. Thus, a call of the form

```
allcat(x1, x2, x3, … )
```

results in L in the procedure in having a list value as if

```
L := [x1, x2, x3, … ]
```

had been evaluated.

That takes care of the way allcat() is called, but there's now a problem when it calls itself recursively: allcat() now expects several arguments, and those arguments are now in an Icon list, L.

Again, there is a feature for handling this problem: list invocation. The expression

```
allcat ! L
```

which, following the previous formulation, is equivalent to

```
allcat(x2, x3, … )
```

since x1 has been removed from L at the point of the recursive call.

The entire procedure therefore is

```
procedure allcat(L[])

   x := get(L) | fail

   if *L = 0 then suspend !x
   else suspend !x || (allcat ! L)

end
```

The parentheses in the last expression are provided for clarity; they are not needed, since the list invocation operator has high precedence. Notice that L is not copied in the recursive call. Copying is not necessary, since a new list is created automatically when a procedure with a variable-length argument list is called.

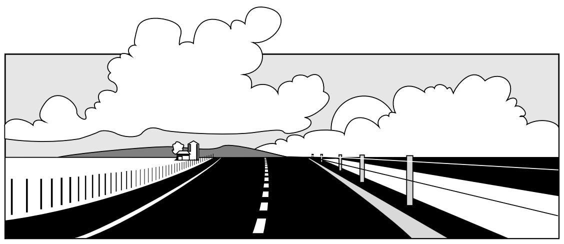
If you aren't accustomed to using procedures that can have a variable number of arguments in combination with list invocation, it may take you a while to get used to this kind of formulation. But once you see what's going on, you should be able to find many uses for the technique.

By the way, notice that we've constructed a recursive generator. It happened quite naturally, not by specific design. That's usually the way recursive generators come about.

References

1. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, p. 170.

2. "Pointer Semantics", The Icon Analyst 6, pp. 2-8.



What's Coming Up

All programming languages have idioms that are peculiar to the specific syntax and semantics of the language. Icon, with its rich expression-evaluation mechanism, has a large and interesting set of idioms. In the next issue of the Analyst, we'll have the first of two articles on idiomatic usages in Icon — full of things we think you'll find interesting.

We'll also have an article on using arrays in Icon, and a description of a new experimental version of the Icon interpreter than allows several programs to run under the same invocation of the interpreter and communicate with each other.

A summary of responses to the questionnaire from Analyst 12 is slated for the next issue also. If you haven't sent back your questionnaire yet, please take a few minutes now to do so.