The Icon Analyst

In-Depth Coverage of the Icon Programming Language

December 1993
Number 21

In this issue …

Returning Multiple Values

Sometimes it's useful and convenient to produce several values as the result of one computation. For example, the maximum, minimum, and average of a set of numerical values can be computed in the same loop. This leads to the idea of a procedure that can return more than one value.

Some programming languages provide such a facility directly. It typically looks like this:

```
procedure p()

   …                # perform computation

   return result1, result2, result3

end
```

where result1, result2, and result3 are assigned values in the computation above. Some provision is needed for obtaining multiple values returned from such a procedure. One syntactic notation is

```
x, y, z := p()
```

which assigns result1 to x, result2 to y, and result3 to z. There are, of course, endless variations on this; we'll stick with this example for concreteness in the discussion that follows.

In some programming languages, call-by-reference can be used as a substitute for returning multiple values. In such languages, references to the variables to which the values are to be assigned are passed to the procedure, as in

```
procedure p(ref:x, ref:y, ref:z)

   …                # perform computation

  x := result1
  y := result2
  z := result3

  return

end
```

Icon supports neither the direct return of multiple values nor call-by-reference, which raises the issue of how multiple values *can* be returned from a procedure in Icon. There are several ways, which we'll discuss here.

Generation: Obviously an Icon procedure can *generate* multiple values. In fact, this capability is one of the most powerful features of Icon. When values are needed one after another, nothing could be more natural than

```
procedure p()

   …                # perform computation

  suspend result1
  suspend result2
  suspend result3

end
```

This also can be written as

```
procedure p()
   ...                # perform computation
  suspend (result1 | result2 | result3)
end
```

since when p() is resumed, the argument of suspend is resumed.

The trouble with generating multiple values is that it's not always convenient to use values produced one after another. For the example above, where the values are to be assigned to different variables, generation is anything but convenient. You might try

```
(x | y | z) := p()
```

but this only assigns a value to x; there's nothing that resumes p() to produce another value. If you try

```
every (x | y | z) := p()
```

the result is even worse. Since generators are resumed in a last-in, first-out fashion, p() is repeatedly resumed to assign values to x, the first variable in the alternation. The final value assigned to x is the third value generated by p(). The same thing then happens for y and z — p() is called three times and generates nine values! The result is hardly what's wanted.

There's no good way around this problem, although we'll say more about it at the end of this article.

Global Variables: One solution to the multiple-value problem is to use global variables in place of call-by-reference:

```
global x, y, z

procedure p()
   ...                # perform computation
  x := result1
  y := result2
  z := result3
  return
end
```

This is not an attractive solution. The procedure and the code that calls it both must know which global variables are used and the same ones must be used everywhere. This is, in fact, one of the worst kinds of usages of global variables and illustrates why they are disparaged. There are appropriate uses for global variables, as when there really is a global program state that many procedures need to access. But that's not the case here.

Multiple Calls: While we're dispensing bad solutions to this problem, here's another unattractive one that sometimes is used to produce the effect of coroutines in programming languages that do not support them directly.

The idea is to keep state information in the procedure so that the first time it is called, it computes all the values needed but returns only the first. The second time it is called, it returns the second value, and so on. This approach looks like this

```
procedure p()
  static state, result1, result2, result3
  initial state := 1

  case state of {
    1: {
         ...          # perform computation
         state := 2
         return result1
         }
    2: {
         state := 3
         return result2
         }
    3: {
         state := 1
         return result3
         }
    }
end
```

Thus,

```
x := p()
y := p()
z := p()
```

assigns the three values to the desired variables.

Aside from the complicated form of the procedure and the opaque nature of the calls to it (which could be made clearer by providing the state as an argument), this approach clearly is dangerous, since an incorrect number of calls results in a disaster.

Co-Expressions: One straightforward solution to the multiple-value problem is to return a co-expression:

```
procedure p()
   ...               # perform computation
  return create (result1 | result2 | result3)

end
```

Then the co-expression can be activated to produce the results as needed:

```
results := p()
x := @results
y := @results
z := @results
```

A less obvious but more "Icon-ish" way is:

```
every (x | y | z) := @p()
```

Not many Icon programmers would use co-expressions to solve the multiple-value problem. Many Icon programmers are wary of co-expressions for reasons that are both valid and invalid. Co-expressions generally are not well understood. To many Icon programmers, co-expressions seem unnatural and not in the spirit of the rest of the language. Some worry that co-expressions may not function properly (a concern that has some foundation). Others worry about how much space and time co-expressions use. And it's certainly true that the use of co-expressions limits the portability of a program, since co-expressions are not supported by all implementations of Icon. We won't go into these matters here, although we plan to discuss them in a future article.

Structures: We've deliberately presented the least attractive ways of handling the multiple-value problem first, leaving the best solution to last.

In most cases, returning a structure that contains all the values computed by a procedure is the best solution. A list is the easiest and more general structure to use:

```
procedure p()
   ...      # perform computation
  return [result1, result2, result3]
end
```

The elements of the list that is returned then can be accessed by position:

```
results := p()
x := results[1]
y := results[2]
z := results[3]
```

In some situations, records may be more natural than lists:

```
record stats(max, min, aver)
procedure p()
   ...      # perform computation
  return stats(result1, result2, result3)
end
```

with

```
results := p()
x := results.max
y := results.min
z := results.aver
```

One concern with structures is the amount of storage that they use. In the examples above, a list or record is created every time p() returns. In most situations the space that is allocated is transient and is reclaimed, if necessary, by garbage collection. Normally this does not cause a problem in program performance. However, if program execution speed is critical, a single structure can be reused repeatedly:

```
procedure p()
  static results
  initial results := list(3)
    ...      # perform computation
  results[1] := result1      # or assign above
  results[2] := result2
  results[3] := result3
  return results
end
```

This technique works when callers of the procedure expect to extract the results the list that is returned by p() but do not save the list for later use or modify it. For example,

```
results := p()
x := get(results)
y := get(results)
z := get(results)
```

has disastrous consequences for the next call of p().

Since the caller of the procedure can wreak havoc with the procedure, this technique should be used only in time-critical applications and even then it should be clearly identified.

Incidentally, a record is somewhat smaller than a list with the same number of values [1]. In critical applications in which the overhead of storage allocation and garbage collection is a problem, performance can be improved somewhat by using records in place of lists. Since records can be subscripted by position, the caller of the procedure need not know that a record is returned. For the previous example of records,

```
results[1]
```

and

```
results.max
```

produce the same value.

Generators Versus Non-Generators: As mentioned above, values produced in sequence by a generator sometimes are awkward to handle. There is no good solution to this problem, but there are several methods for working around it that parallel the solutions given above to the multiple-value problem.

One approach is to put the results produced by a generator in a list:

```
results := []
every put(results, p())
```

and access the resulting list elements as shown previously. Of course, this approach is applicable only to generators that produce a finite number of values and even then it is appropriate only for generators that produce a modest number of values.

Co-expressions also can be used:

```
results := create p()
```

and the values obtained by successive activations of results. This technique is applicable even for generators that can produce an infinite number of values.

We've saved the most awkward approach to handling generators to last. It parallels the multiple-call solution to the multiple-value problem:

```
count := 0

every result := p() do {
  count +:= 1
  case count of {
    1:  x := result
    2:  y := result
    3:  z := result
    default:        ...        # out of bounds
    }
  }
```

We say "most awkward" on the basis of structure and complexity. This method, however, doesn't allocate any storage and it's portable.

Reference

1. "Memory Utilization", The Icon Analyst 4, pp. 7-10.

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!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona

Graphic Contexts in X-Icon

In previous articles we've presented a somewhat superficial view of what an X-Icon window is: a rectangular array of pixels together with various attributes that determine, for example, the color in which drawing is done and the font in which text is written.

An X-Icon window (an object of type window) actually consists of a binding between two other objects: a *canvas*, which is what you see on-screen, and a *graphic context*.

Some attributes, like the window dimensions, its label, and whether it's iconified or not, are associated with the canvas. Other attributes, such as the foreground color, background color, and font, are associated with the graphic context.

When you create a window with open(), the result is a binding of a new canvas with a new graphic context. This situation is shown in Figure 1.

For many purposes, you don't need to know that what X-Icon calls a window is a binding of a canvas and a graphic context. For example, XAttrib() works with any attribute, querying it or setting it in the canvas or the graphic context, depending on where the particular attribute resides. The reason that the underlying structure is important is that other graphic contexts can be created and bound with canvases in different ways. Figure 2 illustrates a possibility.

Since graphic contexts contain the attributes, like colors and fonts, that determine the appearance of drawing and text, different effects can be produced in the same canvas by performing operations on different bindings with the canvas. For the bindings shown in Figure 2 on the next page,

```
write(level1, "Here are some choices:")
every write(level2, "    ", !choices)
```

write a heading in red followed by a list of items in blue.

Of course, the same effect could be obtained by changing attributes in a single graphic context. Graphic contexts offer several advantages over changing attributes: (1) a particular set of attributes can be established and encapsulated in a graphic context, (2) once graphic contexts are established, less code is required to change the effects of operations on windows, and (3) there is less likelihood of programming errors (such as failing to restore the value of an attribute after it has been changed).

It's also possible to bind the same graphic context with several different canvases, as shown in Figure 3. There are, of course, many other possible ways that canvases and graphic contexts can be bound.

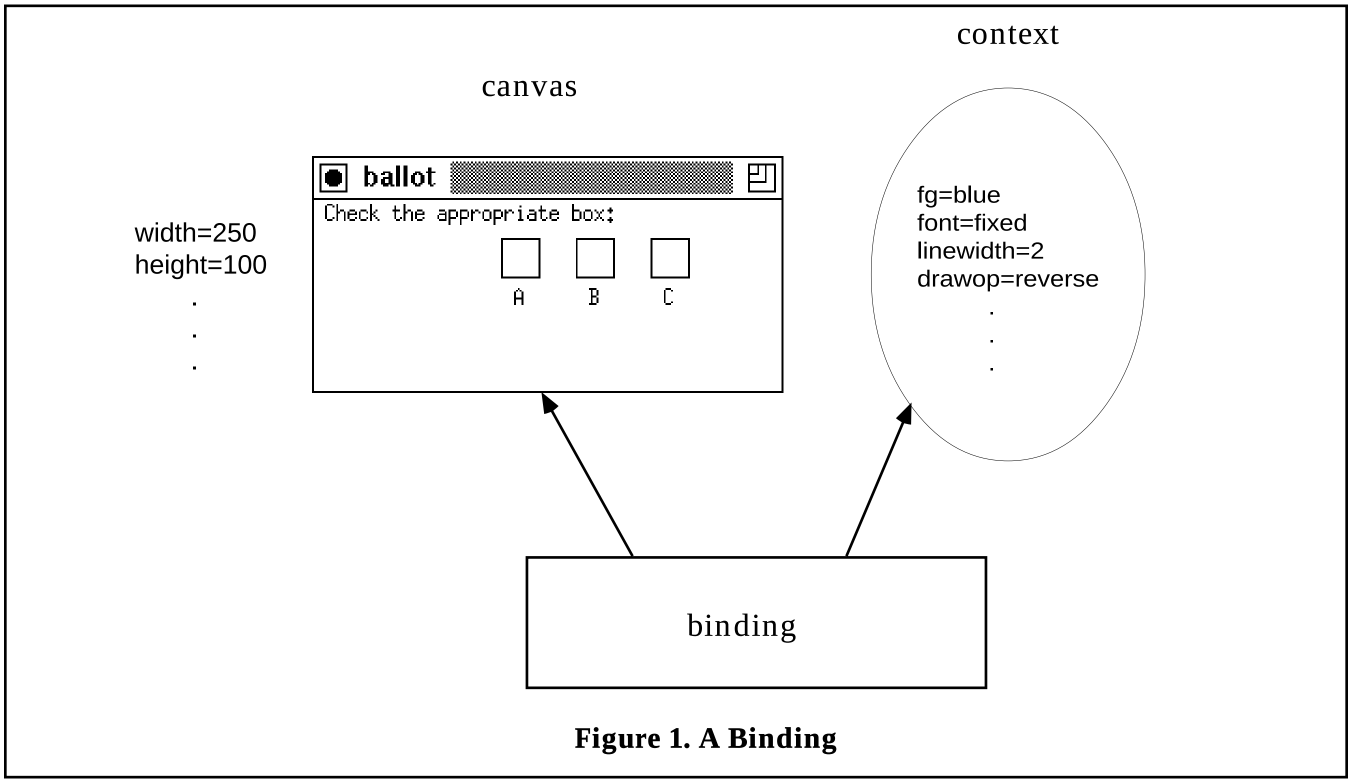


Figure 1. A Binding

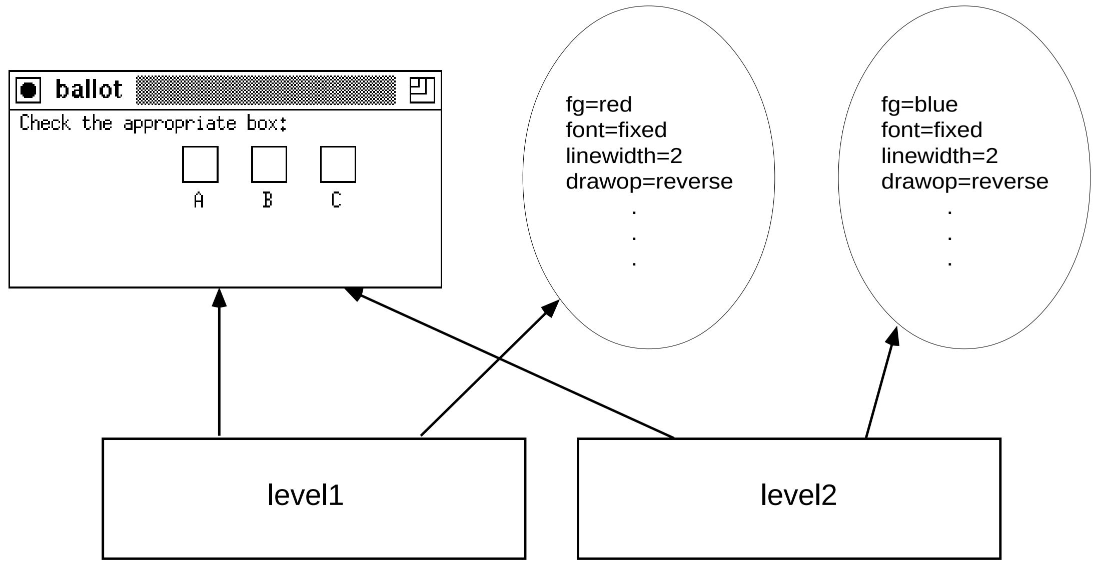


Figure 2. Two Bindings to the Same Canvas

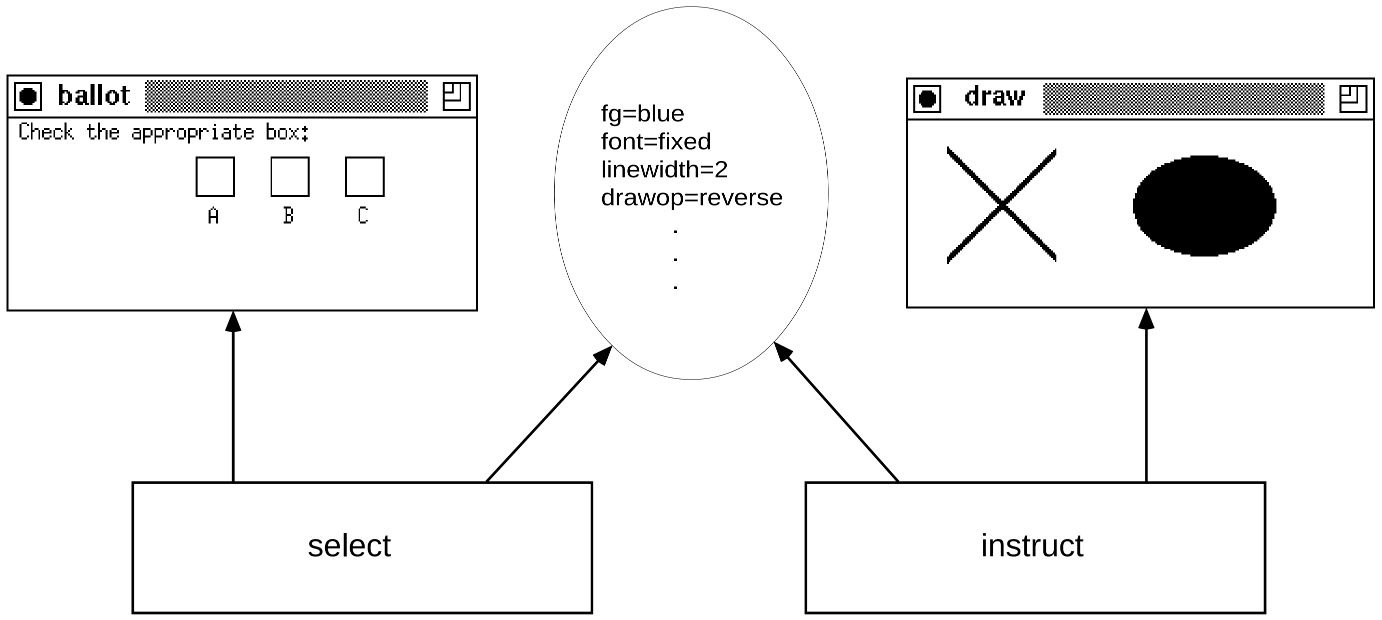


Figure 3. Two Bindings to the Same Graphic Context

As mentioned above, open() creates a canvas and a graphic context and returns a binding between them. The function XBind() can be used to create other bindings and new graphic contexts.

XBind(w1, w2) creates a new binding consisting of the canvas associated with w1 and the graphic context associated with w2. The value produced by XBind(w1, w2) is this binding — a value of type window.

XBind(w), with no second argument, creates a new graphic context bound to the canvas for w. The initial values of the attributes for this new graphic context are the same as those for the graphic context for w (note that w is a binding). Although a canvas may be bound to several different graphic contexts, a binding always has a single graphic context.

If both window arguments to XBind() are omitted, a binding is produced between a new, but invisible, canvas and a new graphic context. This invisible canvas is called a *pixmap*. A pixmap behaves in all respects like a visible canvas except that it has no on-screen manifestation. You can draw on a pixmap or write text to it, but nothing changes on the screen.

Despite the fact that pixmaps are invisible, they are very useful; in fact, they are indispensable for some operations. Images can be built up in pixmaps and then copied to visible canvases using XCopyArea(). This technique can be used to "pan" over a portion of a large pixmap as if portions of it were seen through a window, to scroll, and so on. Similarly, an image in a visible canvas can be copied to a pixmap to save it for possible later display.

In X, pixmaps, like visible windows, reside in the memory of the server, so the extent to which they can be used is limited by amount of memory on the server. While this is true for visible canvases also, it's just more tempting to create a huge pixmap that may be larger than the entire screen.

The first two arguments of XBind() are interpreted as windows with default behavior as mentioned above. &window is not a default for this function and must be given explicitly. Additional arguments can be used to establish the values for attributes for the graphic context related to the new binding. For example,

```
alert := XBind(&window, "fg=red")
```

creates a new binding to the canvas for &window whose graphic context has the same attributes as the graphic context for &window, except with a red foreground.

Coordinate Translation

A graphic context contains two attributes, dx and dy, that are used to translate the position at which output is placed in a window in the x and y directions, respectively. For example, as a result of

```
XAttrib("dx=10", "dy=20")
```

output to &window is offset by 10 pixels in the x direction and 20 pixels in the y direction.

Coordinate translation only applies to output to the window. It does not affect input values such as &x and &y, which are set by events.

Clipping

The function XClip(x, y, w, h) clips output to the specified rectangular area. The defaults are the same as those for other such functions. Thus,

```
XClip(20, 200)
```

clips the output to &window so that any output to the left of the x position 20 and above the y position 200 is not drawn.

The extent of clipping also can be queried or set using the attributes clipx, clipy, clipw, and cliph.

Attributes Associated with Graphic Contexts

So far, we've just mentioned that some attributes are associated with canvases and some are associated with graphic contexts. Here are the attributes that reside in ga raphic context:

colors: fg, bg, reverse, drawop
text: font, fheight, fwidth, ascent, descent, leading
drawing: fillstyle, linestyle, linewidth, pattern
clipping: clipx, clipy, clipw, cliph
translation: dx, dy

All other attributes are associated with the canvas.

Next Time

In the next issue of the Analyst, we'll start dealing with a large and difficult subject: color.

Procedures with Memory

There are situations in which a procedure is called many times with the same argument values and always returns the same value for those arguments. Obviously, re-computation is expensive in terms of time and in some cases in terms of space. In some cases, which we will describe later, the amount of such redundant computation is so great that it limits what is possible to compute.

This matter is discussed briefly in the Icon language book [1], where the problem is dealt with by adding "memory" to procedures so that computed values are saved and not re-computed on subsequent calls with the same arguments. We will look at this problem more closely here and describe ways of handling the technical problems that arise.

The Problem

The problem with redundant computation is particularly serious when a computation is defined recursively, so that one call may lead to many other calls. The examples of this situation that are easy to understand are somewhat artificial in the sense that they aren't likely to occur in "real" programs. Nevertheless, the basic problem *is* real and can occur when searching databases, traversing structures, and so forth.

The Fibonacci numbers provide a particularly nice, if somewhat artificial, example. They occur in so many contexts that it seems they must be part of the fabric of the (mathematical) universe. Although you're not likely to have to compute the Fibonacci numbers, they are fascinating in their own right.

The Fibonacci numbers are defined by the recurrence relation:

$$f(i) = 1 \qquad i = 1, 2$$
$$f(i) = f(i-1) + f(i-2) \qquad i > 2$$

The sequence therefore is 1, 1, 2, 3, 5, 8, 13, 21, 34, … . This looks innocuous enough, but the numbers start to get large quickly. For example, $f(35)$ is 9227465.

A straightforward procedure for computing the *i*th Fibonacci number follows from the recurrence:

```
procedure f(i)
  if i = (1 | 2) then return 1
  else return f(i – 1) + f(i – 2)
end
```

The problem with this kind of a formulation is not that any one call is expensive. The problem is that one call leads to another, and many of the calls have the same argument and hence result in repeated computation. For example, f(8) calls f(7) and f(6), f(7) calls f(6) and f(5), and so on. The way the calls add up is illustrated by the computation of f(10), where the numbers of calls are

f(10)	1
f(9)	1
f(8)	2
f(7)	3
f(6)	5
f(5)	8
f(4)	13
f(3)	21
f(2)	34
f(1)	21
total	109

Note that except for f(1), the number of calls itself forms the Fibonacci sequence. This isn't a coincidence, and you can see where it leads, considering how rapidly the Fibonacci numbers grow.

You might well argue that computing the Fibonacci numbers recursively is simply the wrong approach. An efficient iterative method is easy to formulate and there's even a closed form from which the *i*th Fibonacci number can be computed directly [2].

However, you don't have to go far to find a recurrence relation that doesn't lend itself to such approaches. Ackermann's function is the most famous example:

$$a(i, j) = j + 1 \qquad i = 0, j \geq 0$$
$$a(i, j) = a(i-1, 1) \qquad i > 0, j = 0$$
$$a(i, j) = a(i-1, a(i, j-1)) \qquad i > 0, j > 0$$

A corresponding procedure is:

```
procedure a(i, j)
  if i = 0 then return j + 1
  else if j = 0 then return a(i – 1, 1)
  else return a(i – 1, a(i, j – 1))
end
```

Note that this recurrence is "nested". That is, the recurrence is defined not only in terms of previous values but also in terms of the recurrence applied to previous values.

The value of $a(i, j)$ grows astronomically as i gets larger:

$$a(0, j) = j + 1$$
$$a(1, j) = j + 2$$
$$a(2, j) = 2j + 3$$
$$a(3, j) = 2^{j+3} - 3$$

while for $i = 4$, the value cannot even be represented with ordinary mathematical operations:

$$a(4, j) = \left. 2^{2^{2^{\cdot^{\cdot^{2}}}}} \right\} j + 3 \text{ times} - 3$$

For example, $a(4, 1)$ is $2^{2^{2^{2}}} - 3$, or 65,533. We're talking about really big numbers as j grows. And we don't even know of a formulation of $i > 4$.

Ackermann's function is famous as an example of a general recursive function that is not primitive recursive. Getting into recursive function theory and explaining what this means would take us far afield from the topic of this article. But if you need a topic of conversation during a dull baseball game, you can mention this fact. If you're asked what it means (and don't know), you can just say that the value of Ackermann's function grows faster in the values of its arguments than any primitive recursive function could.

One point in mentioning Ackermann's function here is to dispel a common misunderstanding about recursive functions. It is sometimes said that it's impossible to compute some recursive functions by iterative means. That's not true. Kleene, in fact, gives a general method of converting any recursive function to an iterative one [3]. Although his construction is wildly impractical, just knowing that iterative methods exist gives some hope in finding practical ones. For Ackermann's function, there is a comparatively short iterative method [4]. Figure 1 shows an Icon version (the original was written in FORTRAN).

Don't ask us to explain how this works. The published version was presented without explanation except for a few comments in the program that we've carried over. You'll probably agree, in any event, that this approach is not obvious.

Note that this iterative method requires a small amount of storage in the form of two lists. (We can say small, since for $i > 6$, the value of Ackermann's function is so huge as to be impractical to compute.) The amount of space used by the

```
procedure a(i, j)

   if i = 0 then return j + 1

   value := list(i + 1)
   place := list(i + 1)

   value[1] := 1
   place[1] := 0

   repeat {                                      # new value[1]
      value[1] +:= 1
      place[1] +:= 1
      every k := 1 to i do {                     # propagate value
         if place[k] = 1 then {                  # initiate new level
            value[k + 1] := value[1]
            place[k + 1] := 0
            if k ~= i then break next
            }
         else {
            if place[k] = value[k + 1] then {
               value[k + 1] := value[1]
               place[k + 1] +:= 1
               }
            else break next
            }
         }
         if place[i + 1] = j then return value[1]     # check for end
      }

end
```

Figure 1. Computing Ackermann's Function Iteratively

iterative method certainly is tiny compared to the amount of stack space needed for the recursive version.

It isn't necessary for a recurrence to have more than one argument to be interesting; it's really the nesting that adds richness to the a recurrence. Here's a particularly interesting nested recurrence that produces the "chaotic sequence" [5]:

$$q(i) = 1 \qquad i = 1, 2$$
$$q(i) = q(i - q(i-1)) + q(i - q(i-2)) \qquad i > 2$$

The sequence for this recurrence is chaotic in the sense that values don't just get progressively larger. Instead, a value sometimes is less than the preceding one, although on the average the values increase. Here's how the sequence starts: 1, 1, 2, 3, 3, 4, 5, 5, 6, 6, 6, 8, 8, 8, 10, <u>9</u>, 10, 11, 11, 12, 12, 12, 16, <u>14</u>, 14, 16, ... The underscores show values that are less than their predecessors.

Such a sequence seems to offer no hope for a closed form, as there is for the Fibonacci sequence, or even an explicit representation, as there is for the first few values of *j* for Ackermann's function. In fact, it's not even clear how to get an idea of what the sequence does. Does it ever produce a negative value? Is it, in fact, well-defined? (That is, does the recurrence ever require trying to compute *q(i)* for *i* less than one?)

Analytic attacks on such a beast are hard [6, 7] and beyond most of us. But if we can compute a lot of values in the sequence, we might see suggestive patterns.

A recursive procedure based on the recurrence above is trivial to formulate:

```
procedure q(i)
   if i = (1 | 2) then return 1
   else return q(i – q(i – 1)) + q(i – q(i – 2))
end
```

Trying to use this procedure brings us squarely back to the topic of this article. Although redundant computation is clearly a problem in the recursive computation of the Fibonacci sequence, it's a much more serious problem here. We can get a hint of the problem by looking at the calls for q(10):

q(10)	1
q(9)	1
q(8)	2
q(7)	3
q(6)	5
q(5)	9
q(4)	22
q(3)	77
q(2)	284
q(1)	77
total	481

For q(11), q(12), and q(13) this way, the total numbers of calls are 813, 1,393, and 2,325, respectively. In fact, if you try to compute the chaotic sequence using the recursive procedure given above, you'll quickly discover you're in trouble. For example, it takes nearly 9 hours to compute q(36) using Icon on a Sparc IPX. Looking at the way the time increases for each successive value, q(37) probably will take about 14.5 hours. We decided not to bother. (The computation time appears to increase in the "Fibonacci manner"; computing any one value takes about as long as computing the preceding two.)

Of course, the time would be considerably less if the computation were done in C or (heaven forbid) in assembly language and run on a Cray-2. No matter what language or computer you use, however, you can't hope to get a great many values in the sequence, and that's just what you need to see patterns. 1,000 or even 10,000 values might be needed. This is what we meant when we said earlier that redundant computation can limit what it's possible to compute. You might think we really meant *practical* to compute, assuming it would take longer than we'd want to wait. In fact, there are many problems, including this one, where you won't get far with the theoretically fastest computer in the expected lifetime of the universe [8].

Perhaps at this point you're actually interested in seeing what patterns may exist in the chaotic sequence. The question is how to get enough values in the sequence to see such patterns.

Adding Memory to Procedures

It's clear that redundant computation must be avoided if there's to be any hope of computing many values in such a sequence. The obvious way to avoid duplicate procedure calls is to keep track of calls and record their values. Then, when a call occurs that has occurred before, its value can be produced without performing the computation again. In other words, add memory to the procedure. This idea is not new [9], and it's an important component of dynamic programming [10].

The basic idea is simple: Provide memory in form of a data structure that is subscripted by argument value. When a procedure is called, the memory is checked. If there's a value for the argument in the memory, the value is returned. If there isn't, the computation is performed as before and the resulting value is added to the memory before the procedure returns.

For Icon, tables provide a particularly convenient form of memory. They can be subscripted by any value, it's not necessary to worry about out-of-range references, and they grow in size automatically as information is added to them.

Here's how the procedure for computing the chaotic sequence looks with memory added:

```
procedure q(i)
  static memory

  initial {
    memory := table()
    memory[1] := memory[2] := 1
    }

  if value := \memory[i] then return value
  else return memory[i] :=
    q(i – q(i – 1)) + q(i – q(i – 2))

end
```

The use of a static variable allows all calls of q() to access the memory but prevents other procedures from accessing it — after all, it's q()'s memory.

The first time q() is called, memory is set up and the two initial values in the sequence are placed in it so that it's not necessary to check for them in subsequent calls. Since the default value for the table is null, it's easy to check if an argument is in memory. If it isn't, the computed value is added to memory before returning. That's all there is to it.

The main part of the procedure can be written more concisely as

```
/memory[i] := q(i – q(i – 1)) + q(i – q(i – 2))
return memory[i]
```

This formulation also eliminates the local variable used in the previous version of this procedure.

Another possible formulation is

```
return \memory[i] |
  (memory[i] := q(i – q(i – 1)) + q(i – q(i – 2)))
```

Incidentally, don't try something like

```
return /memory[i] :=
  q(i – q(i – 1)) + q(i – q(i – 2))
```

This returns whether or not the null test fails. If the value for i has been computed before, the procedure call fails.

The question remains as to how effective the addition of memory to q() is. The improvement in performance is startling. As mentioned earlier, it takes about 9 hours to compute q(36) on a Sparc IPX using the straight recursive method. Using the version with memory, it takes about 12 milliseconds to compute q(36). In fact, using memory, the time to compute values in sequence is nearly linear; q(1000) takes only 333 milliseconds. Calling q(i) for a specific value of i requires the computation of all smaller values that have not already been computed. Computing the first 1,000 values in the chaotic sequence takes about 400 milliseconds.

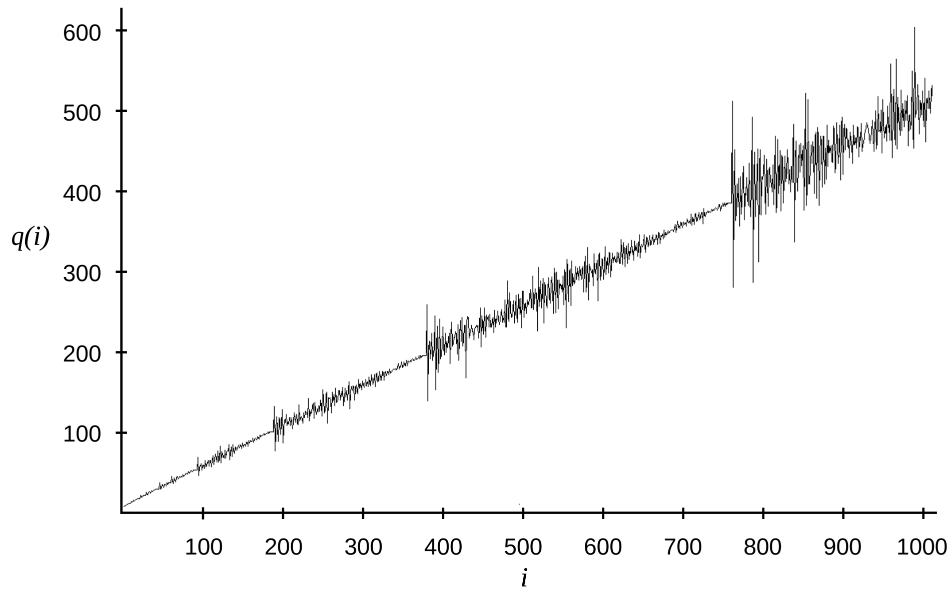


Figure 2. The Chaotic Sequence

Who needs a faster language or platform?

A thousand values are more than enough to reveal some startling properties in the chaotic sequence. See Figure 2. While such a plot doesn't prove anything, it certainly looks like the average value of *q(i)* is close to *i* / 2. Bursts of oscillation around this apparent average also are evident, and they seem approximately to double in maximum magnitude and duration as *i* increases. Perhaps most intriguing are the self-similarities that are associated with fractals. It's also clear that none of these properties of the chaotic sequence could be discovered from only a few values. Up to q(36), which is where we gave up with the purely recursive computation, it was evident that the sequence was unusual, but there was not hint of just how unusual.

There's no question that memory is very effective in this kind of computation. But there are technical questions. Memory takes space. Tables are convenient, but they require more space than the obvious alternative, lists. And what about adding memory to procedures that have more than one argument? Another interesting question is adding memory to recursive generators. We'll consider these matters in the next issue of the Analyst.

References

1. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, pp. 97-98.

2. "Solutions to Exercises", The Icon Analyst 13, p. 3.

3. *Introduction to Metamathematics*, Stephen C. Kleene, D. Van Nostrand Company, Inc., Princeton, New Jersey, 1952.

4. "Recursion and Iteration", H. Gordon Rice, *Communications of the ACM*, Vol. 8, No. 2 (February, 1965), pp. 114-115.

5. *Gödel, Escher, Bach: an Eternal Golden Braid*, Douglas R. Hofstadter, Basic Books, New York, 1979, pp. 137-138.

6. *Fractals, Chaos, Power Laws*, Manfred Schroeder, W. H. Freeman and Company, New York, 1991, p. 59.

7. "Conway's Challenging Sequence", Colin Mallows, *American Mathematical Monthly*, Vol. 98, No. 1, January, 1991, pp. 5-20.

8. "Complexity and Transcomputability", Hans J. Bremermann, in *The Encyclopaedia of Ignorance*, Pergamon Press, New York, 1977.

9. *Memo Functions: A Language Feature with Rote Learning Properties*, D. Mitchie, DMIP Memorandum MIP-R-29, Edinburgh, 1967.

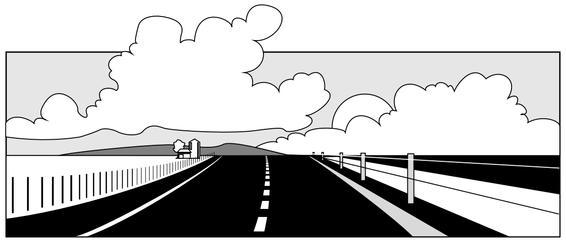
10. *Fundamentals of Computer Algorithms*, Ellis Horowitz and Sartaj Sahni, Computer Science Press, Potamac, Maryland, 1978, pp. 198-247.

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

RBBS: (602) 621-2283

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



What's Coming Up

We'll wrap up the subject of procedures with memory in the next issue of the Analyst. In that issue, we'll also present the first of two articles on programmer-defined control operations, one of the more arcane features of Icon.

And we'll continue the series on X-Icon with the first of two articles on color.