The Icon Analyst

In-Depth Coverage of the Icon Programming Language

December 1997
Number 45

In this issue …

Anatomy of a Program — Numerical Carpets

It has long been known that modular arithmetic often produces interesting numerical patterns. These patterns can be made easier to comprehend and more interesting by rendering them as images with colors associated with numerical values.

The example most often cited is Pascal's Triangle, which exhibits the binomial coefficients. If you take the coefficients modulo *m* for various *m*, you get different but interesting patterns. Figures 1 through 3 show a portion of Pascal's Triangle for *m* = 2, 3, and 5, using *m* uniformly spaced shades of gray from black (for 0) to white (for *m*–1).

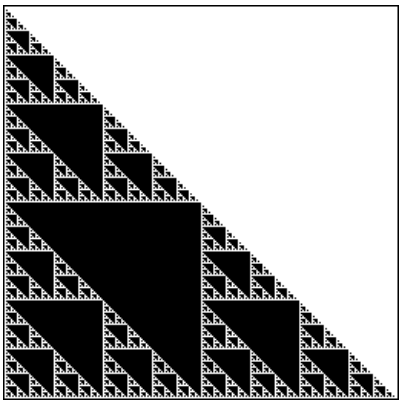


Figure 1. Pascal's Triangle Modulo 2

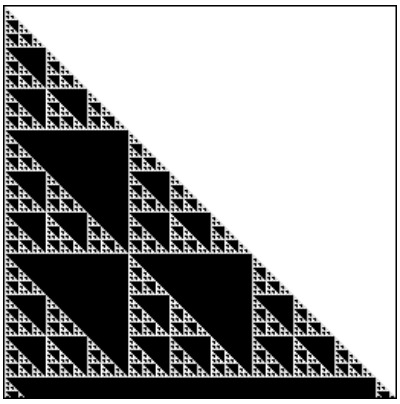


Figure 2. Pascal's Triangle Modulo 3

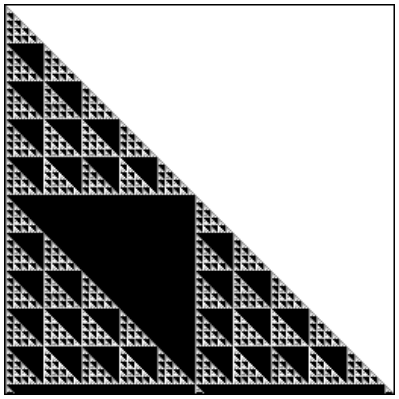


Figure 3. Pascal's Triangle Modulo 5

See References 1 and 2 for extensive treatments of this subject.

Our interest in the subject came from a paper entitled "Carpets and Rugs: An Exercise in Numbers" [3]. This brief and informal paper describes the results of generating integers on a square array according to a "neighborhood" rule. The results are reduced modulo *m* and colored according to the resulting values.

Computing the values of the elements of an array according to the values of their neighbors is the basis for cellular automata [4,5]. Cellular automata may be one dimensional, two dimensional, or of higher dimension. The two-dimensional Game of Life is the best known [6].

Cellular automata are characterized by three properties:

- *Parallelism:* The values of all cells are updated at the same time at discrete intervals $t_1, t_2, t_3, \ldots$.
- *Locality:* The value of a cell at time t_{n+1} depends only on the value of the cell and its neighbors at time t_n.
- *Homogeneity:* The same update rules apply to all cells.

Various neighborhoods can be used. For two-dimensional cellular automata, the neighborhood of a cell typically consists of cell and its eight physically adjacent neighbors, as shown in Figure 4:

nw	*n*	*ne*
w	*c*	*e*
sw	*s*	*se*

Figure 4. The Neighborhood of a Cell

Here we have labeled the cells according to compass points with the center cell labeled *c*. Which of the cells in the neighborhood contribute to the value of the center cell varies with the automaton.

The "Carpets and Rugs" article we mentioned above differs from cellular automata in that the values of the array elements are not computed in parallel, but rather one after another in a regular way in which previously computed values potentially affect newly computed values. Once a value is computed, it is not changed.

An $n \times n$ array is initialized along the top row and down the left column with all other values being zero initially. The rule used to compute new values is very simple:

$$a_{i,j} = (a_{j,i-1} + a_{i-1,j-1} + a_{i-1,j}) \bmod m \quad 2 \le i, j \le n$$

Note that the initialized cells are not changed.

The neighborhood used is shown in Figure 5, where the value of the center cell is the sum of the values in the gray cells.

nw	*n*	*ne*
w	*c*	*e*
sw	*s*	*se*

Figure 5. The Carpet Neighborhood

Thus, in terms of the labeling, $c = n + nw + w$. Note that the new value of *c* does not depend on its prior value.

The paper does not mention the order in which the array is traversed ("woven" seems apt for carpets), but with this neighborhood, the same results can be obtained either by a row-primary traversal, left to right, top to bottom, or by a column-primary traversal.

Having filled in the array, a color is assigned to each integer based on its value and the values of neighboring cells. The result then is displayed as a computer-generated image. The paper does not say how the colors are assigned, but simply assigning a different color to each integer $0 \le i \le m-1$ produces images similar to the ones shown in the paper.

Only two initialization schemes are described in the paper: (1) all ones across the top row and down the left column, and (2) alternating zeros and ones across the top and down the left side.

Even with these simple initialization schemes and the simple rule for computing values, the results for different moduli are fascinating. Figures 6 and 7 show "carpets" similar to the ones in the paper.

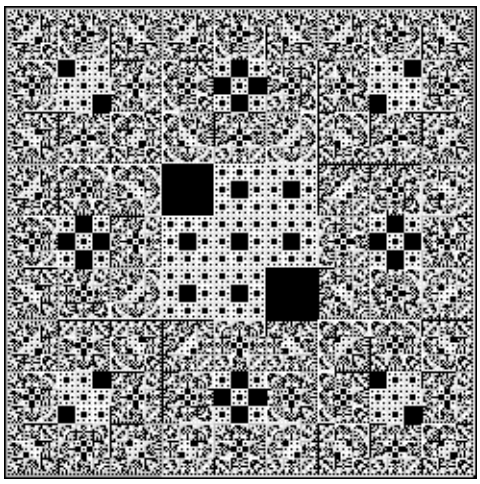


Figure 6. Carpet from All-Ones Initialization

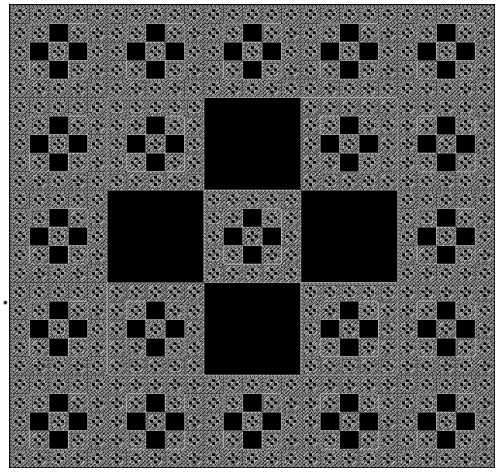


Figure 7. Carpet from One-Zero Initialization

Other moduli produce similar images.

The paper raises more questions than it answers. Some of the more obvious ones are:

- What is the effect of different initialization values?
- What is the effect of initializing different portions of the array?
- What happens if the array is not square?
- What happens if the neighborhood computation is different?
- How do different color schemes affect the visual result?
- What happens if different traversal paths are used?
- What if …

The first issue to resolve is whether any such changes yield results that are both significantly different from those using only the method given in the paper and also are interesting. A few simple experiments answered this question, at least for us. See Figures 8 and 9. (Such images cannot be produced using only the methods described in the paper.)

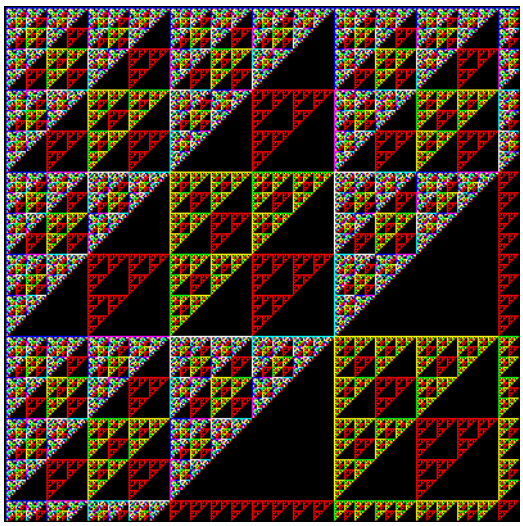


Figure 8. "Pascal" Carpet

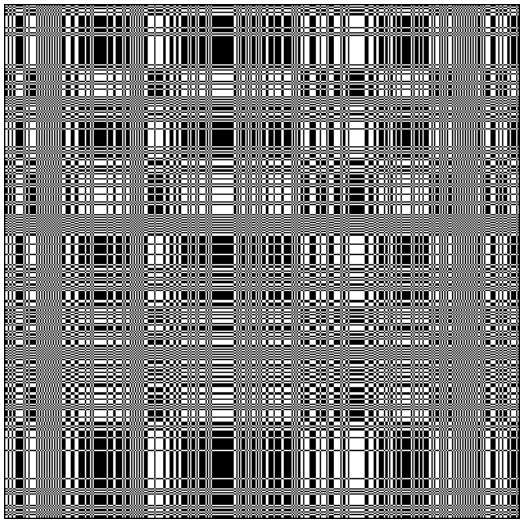


Figure 9. "Open Weave" Carpet

With so many independent variables, some of which offer not only endless but very different possibilities, two things are obvious: (1) an experi-

mental approach is appropriate, and (2) a general, flexible, and easily used tool is needed.

This leads us to "programmable numerical carpets" in which a user can use programming techniques to experiment and explore. A visual interface in which various possibilities can be tried and evaluated interactively adds to power and ease of use.

There are, of course, *too* many independent variables posed by the preceding questions. We decided to stay within the confines of the method described in the paper with only a few extensions that do not affect the underlying ideas:

- separate specifications for carpet width and height (length)
- specification of different neighborhood computations (but using only the *n*, *nw*, and *w* cells)
- separate specifications for row and column initialization
- specification of various initialization values

The width, height, and modulus are just constants that the user can specify. The challenging issue is initialization. Providing the user with a choice among a list of predefined initializations is easy, but it obviously is very limiting. Instead, the user should be able to specify the sequences of numbers for initialization.

We used the word "sequence" in the last sentence for a purpose. We could have used other words, such as "list", to convey the idea of order. But in Icon, the concept of sequence runs so deeply and is such a powerful programming technique that *thinking sequences* is something that should come naturally.

For example, the initializations used in the paper can be represented as sequences generated by the expressions |1 and |(0 | 1). Now think of all the other possibilities! Possibilities such as seq(), which generates 1, 2, 3, … and fibseq() from the Icon program library, which generates the Fibonacci sequence 1, 1, 2, 3, 5, 8, … , and many more.

But this idea takes us into deep water. It implies the ability to evaluate an arbitrary Icon expression during program execution. It is, of course, possible to write a program in which the initialization expressions are edited before the program is compiled and run. But this is too laborious and time-consuming for exploring the vast space of numerical carpets.

How *can* you evaluate an arbitrary Icon expression within a running program? You can't. But you can accomplish the equivalent.

One method is to write out a file consisting of the expression wrapped in a declaration for a main procedure. For example, to "evaluate" seq(), the file might look like this:

```
procedure main()
   every write(seq())
end
```

If the file is to be named expr.icn, a procedure to produce the file is just:

```
procedure expr_file(expr)
   local output

   output := open("expr.icn", "w") |
      stop("*** cannot open file for expression")

   write(output, "procedure main()")
   write(otuput, "    every write(", expr, ")")
   write(output, "end")

   close(output)

   system("icont –s expr –x")

   return

end
```

The –s option suppresses informative output from icont, while the option –x causes the program to be executed after compilation.

There is one thing very wrong with expr.icn: an expression like seq() is an infinite generator; output continues until something intervenes. That's easily fixed by limiting the generator. For the initialization of the top row, this might be used:

```
every write(seq()) \ width
```

where width is the width of the array.

Before doing this, however, there is the question of how to get the output of expr back into the program that created it. One way would be to write it to a known file and read from the file when expr completes execution.

An alternative is to open the command line as a pipe instead of using system():

```
input := open("icont –s expr –x", "pw")
```

This has the same effect as the use of system() above, except it creates a pipe, input, from which the values produced by expr can be read one at a time as needed. Using this method, it's not neces-

sary to add limitation to expr.icn. Depending on the operating system, expr may produce a few more values than are ever used, but in most situations, this is not a problem. Of course, the operating system must support pipes.

Note that pipes have to be created for every expression that needs to be evaluated to produce a carpet. There are at least three, one for each initializer and one for the neighborhood computation. We also found it helpful for the user to be able to specify the modulus as an expression, such as &phi ^ 2, and it just might be useful to allow the dimensions to be specified by expressions.

We have used this monolithic approach successfully, using exprfile.icn from the Icon program library to manage the details. We prefer a different approach, however; one that is simpler and more flexible. In this approach, a carpet-configuration program writes a file that contains preprocessor definitions for the various carpet parameters and expressions. It then uses system() to compile and execute a carpet-generation program that includes the definition file and constructs the carpet.

The advantage of this approach is that it's easy to write the preprocessor definitions and they are "magically" there when the carpet-generation program is compiled.

Separating the construction into two applications has the additional advantage of separating two quite different functionalities into two programs as opposed to packing them all into one program.

There are downsides to the separation. Since carpet generation is done by a separate application, the user needs to shift attention to this application to view the image it produces. Another problem is that the carpet-configuration program must know the location of the source code for the carpet-generation program.

Less obvious, perhaps, is error checking. In the monolithic approach, a user syntax error in an initialization expression can be detected before carpet construction begins. With the "duolithic" approach, that can't be done without "evaluating" expressions in the carpet-configuration application, which would defeat the purpose of the separation. Instead, the syntax error does not occur until the carpet-generation program is compiled.

But this is, after all, an application for Icon programmers; they don't make mistakes. Or, if they do, they know intuitively what is wrong and how to fix it ….

In case you are wondering about speed, the "duolithic" approach is faster.

The interface for the carpet-specification program is simple: It consists of menus for file operations and setting specifications, a single button to create a carpet, and a "logo" for decoration. See Figure 10.

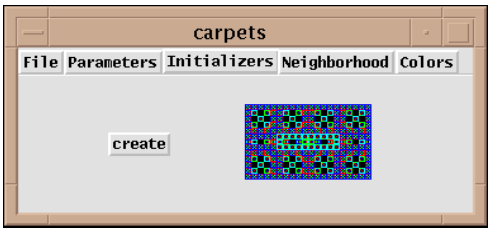


Figure 10. Carpet-Specification Interface

Figure 11 shows the dialog for entering and editing initializers.

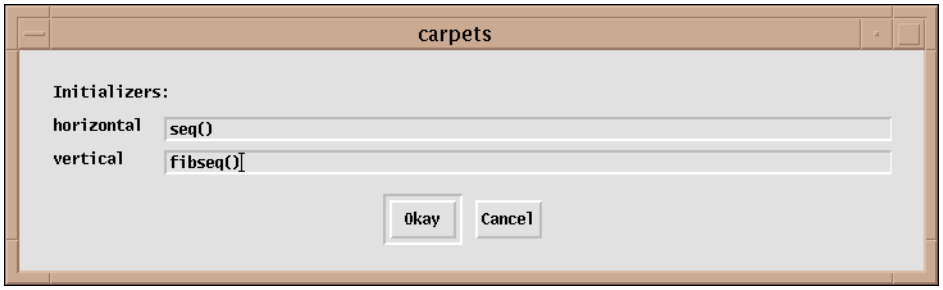


Figure 11. The Initializer Dialog

The text-entry fields are long to allow complicated expressions to be entered

Figure 12 shows the dialog for entering and editing the neighborhood expression.

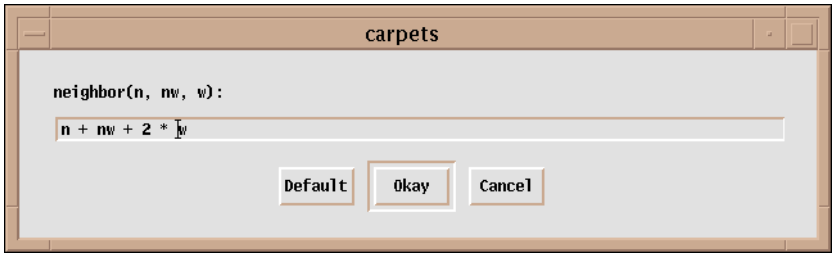


Figure 12. The Neighborhood Dialog

Note that the variables n, nw, and w are used to refer to the cells relative to the current one when the carpet is generated. The values of these variables are supplied in the carpet-generation program. The Default button restores the expression to $n + nw + w$.

Here is an example of a definition file produced by the carpet-specification program.

```
$define Comments "October 14, 1997"
$define Name "untitled"
$define Width (128)
$define Height (128)
$define Modulus (5)
$define Hexpr (seq())
$define Vexpr (fibseq())
$define Nexpr (n + nw + 2 * w)
$define Colors "c2"
```

The definition for Colors is the name of a color palette.

Carpet Specifications

Dimensions

The size of a carpet usually affects its "completeness" as shown in Figure 13.

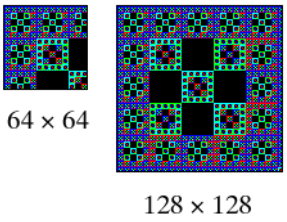


Figure 13. One Effect of Carpet Size

In some cases, the dimensions may affect the scale of the pattern. The patterns for carpets that are not square usually resemble the patterns for square ones.

In most cases, modest dimensions, such as 64 × 64 give an indication of the nature of the carpet. Considerable time can be saved by starting with small sizes to find promising candidates for larger carpets.

It is, of course, possible to contrive specifications that produce very different patterns depending on the dimensions of the carpet. Consider, for example,

```
(|0 \ 100) | 1
```

for both initializers. Since the first 100 cells are zero, carpet with dimensions less than 101 × 101 will be a solid color, while a 200 × 200 carpet is as shown in Figure 14.

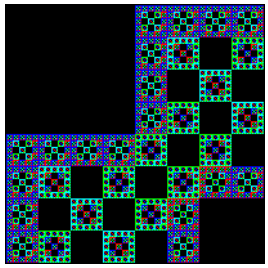


Figure 14. Another Effect of Carpet Size

Moduli

The patterns produced vary considerably in appearance depending on the modulus. Even the simplest initializer, a lone one on the upper-left corner, produces interesting patterns for different moduli. The results for a 128 × 128 array with moduli from 2 through 17 are shown in Figure 15.

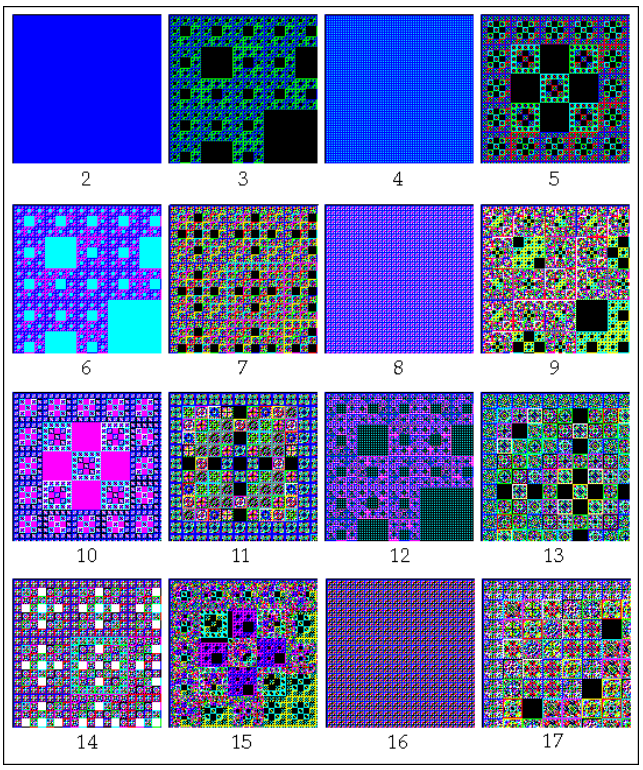


Figure 15. Effects of the Modulus

The patterns that result from different moduli often show significant differences between prime and composite moduli. There is, of course, a strong interaction between the modulus and the initializers.

Initializers

Initializers provide the most fertile ground for designing interesting carpets. There are endless possibilities, which is a problem in itself.

Even the simplest initializers often produce interesting results, as shown in Figure 15. If the initializers for the top row and left column are the same and the default neighborhood computation is used, the resulting carpet is symmetrical around the diagonal from the upper-left corner to the lower-right one. The result often is more attractive than if different initializers are used for the top and left edges, but there are endless exceptions.

Icon's generators offer an easy way to experiment. Even simple generators like |(1 to 5) produce interesting carpets.

Some numerical sequences, when used as initializers, produce interesting patterns. Rather surprisingly, the prime numbers produce interesting carpets for moduli 4 and 8. Figure 16 shows the carpet for modulus 4. The carpet for modulus 16 is similar.

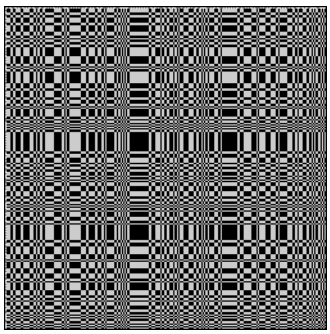


Figure 16. The Primes with Modulus 4

On the other hand, for other moduli at least through 100, the carpets for primes are chaotic and show little structure. The carpet for modulus 3, shown in Figure 17, is typical.

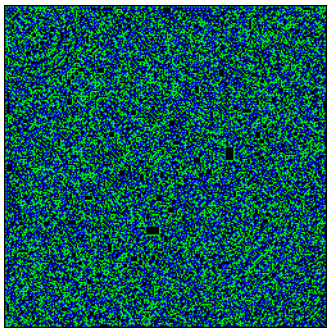


Figure 17. The Primes with Modulus 3

It's worth noting that the Icon program library contains a large number of procedure that generate various numerical sequences. See genrfncs.icn. The module pdco.icn contains programmer-defined control operations [7,8] that allow sequences to be composed in various ways, such as interleaving results from several sequences.

Neighborhoods

Neighborhoods are tricky. Most expressions other than the default one produce degenerate or chaotic carpets. Scaling values sometimes produce interesting results. For example, $3 * n + nw + 2 * w$, with modulus 5 and lone-one initializers, produces the carpet shown in Figure 18.

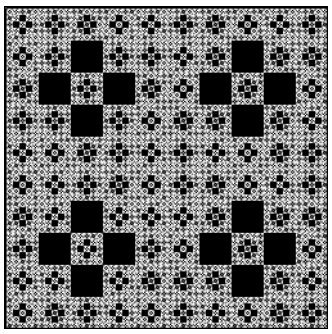


Figure 18. A Non-Standard Neighborhood

If you look closely, you'll see that this carpet is not symmetric around the diagonal.

Colors

Lists of colors are used in displaying carpets. They may come from Icon's built-in palettes, or from color lists provided by the carpet-specification program, or they can be supplied by the user.

Different color lists, of course, may make marked differences in the visual appearances of the same carpet. Contrasting colors may make patterns easier to discern, but they may not produce the most attractive results.

There is a strong correlation between the modulus and the colors used. The number of colors need not be the same as the modulus, but if the number of colors exceeds the modulus, some colors will not be used. A more interesting situation occurs when the number of colors is less than the modulus. In this case the carpet-generation program "wraps around", taking values greater than the number of color modulo the number of colors.

An interesting possibility exists for using color lists in which colors are duplicated, thus mapping

different values into the same color. We have not explored this yet.

The Programs

The carpet-specification program, named carport, is a simple VIB application. Most of the code is routine and we'll only show three procedures.

The procedure init() initializes the interface and sets up the default carpet specifications:

```
procedure init()
   vidgets := ui()              # initialize interface
   # Set up carpet defaults.
   comments := ""
   name := "untitled"
   width := 128
   height := 128
   modulus := 5
   hexpr := "|1"
   vexpr := "|1"
   nexpr := "n + nw + w"
   colors := "c2"
   return
end
```

The procedure that is called to edit the initializers shows how simple the process is: There is no error checking; whatever the user enters is passed along to the carpet-generation program:

```
procedure initers()
   if TextDialog("Initializers:", ["horizontal", "vertical"],
      [hexpr, vexpr], 80) == "Cancel" then fail
   hexpr := dialog_value[1]
   vexpr := dialog_value[2]
   return
end
```

The procedure create_cb() writes the definition file for the carpet-generation program and then compiles and executes the program, which is named carplay:

```
procedure create_cb()
   local out
   output := open("carpincl.icn", "w") | fail
   write(out, "$define Comments ", image(comments))
   write(out, "$define Name ", image(name))
   write(out, "$define Width (", width, ")")
   write(out, "$define Height (", height, ")")
   write(out, "$define Modulus (", modulus, ")")
   write(out, "$define Hexpr (", hexpr, ")")
   write(out, "$define Vexpr (", vexpr, ")")
   write(out, "$define Nexpr (", nexpr, ")")
   write(out, "$define Colors ", image(colors))
   close(output)
   # compile and run
   system("icont –s carplay –x")
   return
end
```

Note that image() is used to place quotation marks around strings and that expressions are surrounded by parentheses to prevent misinterpretation when they are substituted for their names in the carpet-generation program.

The carpet-generation program, shown on the next page, is a bit more interesting.

The file containing the preprocessor definitions, carpincl.icn, is included before the actual code. The main procedure simply calls carpet() to produce the carpet and then waits for the user to save the image if desired before quitting. The interface is primitive to avoid linking lots of code that would be necessary for a more sophisticated interface, since the user of carport must wait for carplay to compile and link, the time saved is worth the inconvenience.

The procedure carpet() uses the symbols defined in carpincl.icn (distinguished by initial uppercase letters). There are some subtleties here. Modulus, Width, and Height might be expressions, not just numbers. Their assignment to variables, which are used subsequently, assures that expressions are not evaluated repeatedly. Note that the number of colors, assigned to cmod, may be different from the modulus.

First the left and top edges are initialized, using the expressions from carpincl.icn. The edges are colored before going on. Next, the carpet is created by traversing the matrix. Note that negative values and real numbers are allowed in specifications. Real numbers are converted to integers and the absolute value is used in determining the color to assign to a cell.

The procedure neighbor() is called with the three neighbors of interest. It simply returns whatever Nexpr specifies. Notice that the computation in neighbor() does not have access to the matrix or the other local variables in carpet(); this effectively

```
link carpcolr                    # color handler
link matrix
link genrfncs                    # procedures for initializers
link wopen

$include "carpincl.icn"

procedure main()

   carpet()

   repeat case Event() of {
      "q":  exit()
      "s":  WriteImage(Name || ".gif")
      }

end

procedure carpet()
   local m, n, colors, v, modulus, cmod, array, height, width

   colors := carpcolr(Colors)

   cmod := *colors

   modulus := Modulus
   width := Width
   height := Height

   array := create_matrix(height, width, 0)

   WOpen("size=" || width || "," || height) |
      stop("*** cannot open window")

   m := 0
   every v := (Vexpr \ height) do
      array[m +:= 1, 1] := v % modulus
   n := 0
   every v := (Hexpr \ width) do
      array[1, n +:= 1] := v % modulus

   every m := 1 to height do {          # color left edge
      Fg(colors[(abs(integer(array[m, 1])) % cmod) + 1])
      DrawPoint(m – 1, 0)
      }
   every n := 1 to width do {           #  color top edge
      Fg(colors[(abs(integer(array[1, n])) % cmod) + 1])
      DrawPoint(0, n – 1)
      }

   every m := 2 to height do {          # compute and color
      every n := 2 to width do {
         array[m, n] := neighbor(array[m, n – 1],
            array[m – 1, n – 1], array[m – 1, n]) % modulus
         Fg(colors[(abs(integer(array[m, n])) % cmod) + 1])
         DrawPoint(n – 1, m – 1)
         }
      }

   return

end

procedure neighbor(n, nw, w)

   return Nexpr

end
```

The Carpet-Generation Program

confines the neighborhood computation to the values of the three neighboring cells — it can't "reach out" and access other cells.

That's all there is to it. Of course, other features could be added to carplay to, for example, tile the carpet image so the user can see how it looks used in that way.

Carpets on Our Web Site

The Web page for this issue of the Analyst contains numerous examples of the fruits of our (carpet) labors and well as some interesting results of programming errors. We also hope to put up the programs described in this article, but that may take a while.

The color images there are much more interesting than the grayscale versions here. If you are interested in numerical carpets, you should visit this page.

Next Time

The programs given here allow easy experimentation. If, however, you find an interesting specification and want to explore variations on it, it's tedious. For example, in looking for interesting carpets with prime initializers, we covered a range of moduli in hopes there might be interesting results for moduli other than 4 or 8. Doing this one at a time using the programs described in this article just wasn't feasible.

In another article on numerical carpets we'll describe applications that allow generating carpets in the background for a range of specifications.

There are many more things that can be done to increase the capability of the carpet programs. These include:

- specification of different ways to initialize the carpet, instead of just along the top and right edges
- specification of different paths for traversing the carpet, instead of just row primary or column primary
- specification of neighborhoods using cells other than *n*, *nw*, and *w*
- "reweaving" a carpet to use its final values as initialization for another traversal

Doing this, especially the specification of arbitrary paths on an array, involves solving both

conceptual and programming problems.

We'll probably have an article on specifying paths before we apply them to carpets.

References

1."On Computer Graphics and the Aesthetics of Sierpinksi Gaskets Formed from Large Pascal's Triangles", Clifford A. Pickover, in *The Visual Mind: Art and Mathematics*, Michele Emmer, ed., pp. 125-133.

2. *Chaos and Fractals; New Frontiers of Science*, Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe, Springer-Verlag, 1992.

3."Carpets and Rugs: An Exercise in Numbers", Dann E. Passoja and Akhlesh Lakhtakia, in *The Visual Mind: Art and Mathematics*, Michele Emmer, ed., pp. 121-123.

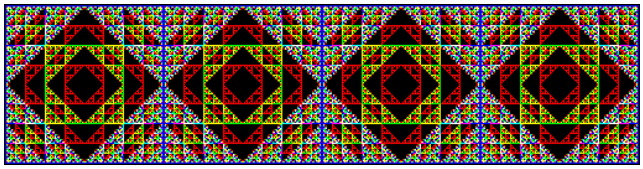
4. *Cellular Automata and Complexity; Collected Papers*, Stephen Wolfram, Addison-Wesley, 1994.

5. *Cellular Automata Machines*, Tommaso Toffoli and Norman Margolus, The MIT Press, 1991.

6. *The Recursive Universe; Cosmic Complexity and the Limits of Scientific Knowledge*, William Poundstone, Contemporary Books, 1985.

7."Programmer-Defined Control Operations", Icon Analyst 22, pp. 8-12.

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



Graphics Corner

Seamless Tiling

Seamless tiling, in which the borders between adjacent generating tiles (GTs) are not visually evident, is important in many applications. Seamless tiling is used extensively in decoration, fashion, and architecture — in fact, if you look around, you'll see seamless tiles almost everywhere in man-made objects.

Computer graphics have greatly expanded the use and demand for seamless tilings — for computer "wallpaper", Web page backgrounds, multimedia backgrounds, and textures for the surfaces of three-dimensional models.

Seamless tiles of bewildering variety are available for the taking from many Web sites, but see the **Copyright Issues** sidebar on the next page. Many commercial collections are available on CD-ROM. Some image-manipulation programs can create seamless tilings, and several applications are dedicated to just this task.

Motifs, of course, may contain elements that give the appearance of seams when they are tiled. We won't consider that here. The question is whether the process of tiling creates visual discontinuities at the boundaries of adjacent tiles.

There are several techniques for creating motifs that tile seamlessly. Some of these are artistic in nature and some of them have mathematical bases.

One frequently used technique that is to mirror a motif use the *p2mm* symmetry ("prickly pear" in quilting terminology). This is illustrated in Figures 1 and 2.

Figure 1. A Motif

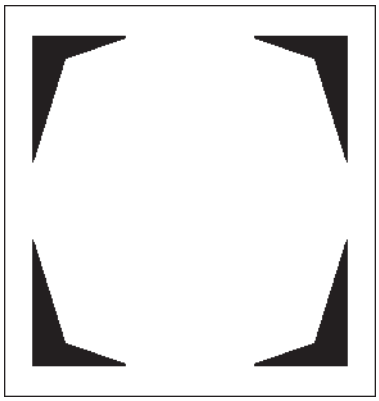


Figure 2. The Mirrored Motif

Copyright Issues

Under the current copyright law, new works, including images, are "born copyrighted". They are copyrighted when they come into existence, and there is no requirement to register them or even put a copyright mark on them. There are some limitations on what can be copyrighted — it must be original and have substance — but almost any original writing, program, image, and performance is automatically copyrighted when it is created.

What this means is that to be safe, you should presume any document, image, sound clip, movie, VRML world — whatever — that you encounter on the Web is copyrighted. Furthermore, even if someone tells you it's all right to use, say, one of their images on your home page, you have no way of knowing if they have the authority to grant your use. To assume otherwise is paranoid, perhaps, but you can be sure that large companies with substantial exposure go to great lengths to be sure they are not infringing on a copyright.

Modifying a copyrighted work does not get you off the hook. You've simply created a "derivative work", for which permission is required.

This does not mean you can't use copyrighted material. The law provides a concept of "fair use", which generally allows copying for scholarly or even personal use. The ultimate test for liability is the damages that may be claimed for infringement. They generally are based on the loss of value to the copyright owner because of the infringement. (Damage awards for copyright infringement can be very substantial.)

In any event, you can't copy and distribute a copyrighted work without permission of the copyright owner without infringing on the copyright. Putting material on the Web constitutes distribution.

We have to be careful about the images we use in our publications. Usually, they either are original or from purchased clip art.

Disclaimer: We are not lawyers and the remarks above merely reflect our understanding of the situation concerning copyright.

As shown in Figure 2, the *p2mm* symmetry results from reflecting the motif horizontally and vertically. As a result, the opposite outside edges are identical and hence tile seamlessly. Incidentally, the *p2mm* symmetry is the only one of the 17 plane symmetries that is rectangular and also tiles seamlessly.

Some three-dimensional modeling programs mirror images to be tiled over surface by default to reduce visual artifacts. And many otherwise uninteresting motifs can be mirrored to produce interesting seamless tiles. For example, numerical carpets, described in the article beginning on page 1, often are much more attractive if they are mirrored. The carpet shown in Figure 3 is interesting, but increasing its size only produces larger rectangles to the right and down. Mirroring it, however, as shown in Figure 4, produces a more interesting "optical-art" image.

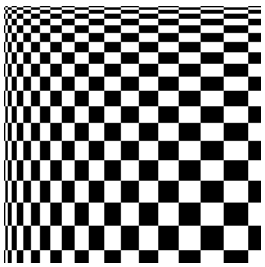


Figure 3. A Checkered Carpet

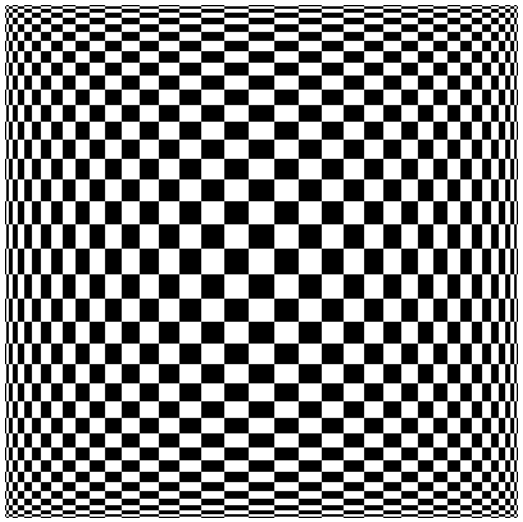


Figure 4. The Checkered Carpet Mirrored

The mirroring can be done entirely using CopyArea(), which is very fast. Here's a procedure to mirror a window:

```
link wopen

procedure mirror(win, x, y, w, h)
   local width, height, sym, x1, y1

   /win := &window
   /x := 0
   /y := 0
   /w := WAttrib(win, "width")
   /h := WAttrib(win, "height")

   if w < 0 then {          # handle negative dimensions
      w := –w
      x –:= w
      }
   if h < 0 then {
      h := –h
      y –:= h
      }

   width := 2 * w
   height := 2 * h

   sym := WOpen("size=" || width || "," || height) | fail

   CopyArea(win, sym, x, y, w, h)

   every x := 0 to w – 1 do
      CopyArea(sym, sym, x, 0, 1, h, width – x – 1, 0)

   every y := 0 to h – 1 do
      CopyArea(sym, sym, 0, y, width, 1, 0,
         height – y – 1)

   return sym

end
```

The image to be mirrored first is copied to the upper-left corner of a target window whose dimensions are twice those of the original image. One-pixel-wide columns from this copy are then copied to their opposite positions at the right side. Once the top half is complete, one-pixel-wide rows are copied to their opposite positions at the bottom.

Links

Here are some links to seamless tiles on the Web. This is only a sample; there are hundreds of URLs at which you'll find seamless tiles.

Note: These links and others are on the Web page for this issue of the Analyst. Please realize that Web sites come and go and also move around. The links given here have been stable for some time.

Julianne's Background Textures:

http://www.sfsu.edu/~jtolson/textures/textures.htm

The Backgrounds Archive:

http://the-tech.mit.edu/KPT/bgs.html

The Background Sampler:

http://home.netscape.com/assist/net_sites/bg/backgrounds.html

Netcreations:

http://www.netcreations.com/patternland/

The Wallpaper Machine:

http://www.cacr.caltech.edu/cgi-bin/wallpaper.pl

Josh's Tilable Image Gallery:

http://jcomm.uoregon.edu/~josh/tile_gallery/

Background Images Archive:

http://www.ist.net/clipart/uwa/bkgs/bkg_menu.html

Axem Textures:

http://axem2.simplenet.com/heading.htm

Realm Graphics:

http://www.ender-design.com/rg/backidx.html

Truman Brown's Texture Woild:

http://www.websharx.com/~ttbrown/txtwoild.html

Texture Station:

http://www.aimnet.com/~bosman/Frame_setup.htm

Versum Numbers as Factors

You'll recall that a versum number is the result of adding the digit reversal of a number to itself. A versum sequence results from repeating the process.

In two previous articles [1,2] we presented some results about the factors *of* versum numbers. In this article, we'll explore versum factors *as* factors. We warn you at the outset that there are no spectacular results, but some of the programming problems may be of interest.

Factoring

We need to say something about factoring before going on. Looking for specific factors, like 11, is easy. But factoring an arbitrary large integer may not be. Stephen Wolfram puts it this way in the reference volume for *Mathematica*, a system for doing mathematics on computers [3]:

> "You should realize that according to current mathematical thinking, integer factoring is a fundamentally difficult computational problem. As a result, you can easily type in an integer that *Mathematica* will not be able to factor in anything short of an astronomical length of time."

This, of course, its not a problem with *Mathematica*; it's a fundamental problem. By the way, although you will see claims that factoring *is* fundamentally hard, that has not been proven. There is an albeit very remote possibility that an entirely new, fast method will be discovered. Such a development would, of course, have a major impact on public-key encryption systems, which are based on the assumption that factoring is a hard problem.

Note that some very large numbers are easily factored. 1000! is an example.

For this article, we had to factor a lot of numbers. Many methods of factoring have been developed, especially for numbers with special properties. The simplest method, called "baby division", is just to try division by successive primes. In Icon, it looks like this:

```
procedure factorseq(i)
   local j, p

   j := sqrt(i)              # as far as needed

   every p := prime() do {
      if p > j then return i
      while i % p = 0 do {
         suspend p
         i /:= p
         }
      if i = 1 then fail
      }

end
```

This procedure generates prime factors in increasing order with repeated factors given multiple times.

Baby division works very nicely for small numbers and even for huge numbers, such as 1000!, that have only small factors. It's hopeless, however, for numbers that have really large factors. And, of course, if you want to do much factoring, you want to use the fastest language, the fastest algorithm, and the best implementation.

We used three programs for factoring. One was a UNIX program, *factor*, that is very fast. Its output shows multiple factors with exponents. The format is illustrated by the output for 24:

```
24 = 2^3 3
```

Unfortunately, this program can't handle numbers larger that $2^{16}-1$ and gives erroneous results for such numbers.

Another program was *Mathematica*. It is extremely fast and can handle numbers of any size, subject to the limitations mentioned earlier. We only used *Mathematica* when we had to; most of our work for this article was done on a UNIX platform, while our copy of *Mathematica* is on a Macintosh. Getting data back and forth was awkward.

We used baby division written in Icon for some numbers that *factor* couldn't handle but weren't large enough to require *Mathematica*.

We converted output from *Mathematica* and Icon's baby division to *factor* format, so that all data was in the same form for use in other programs.

Prime Factors and Divisors

There are two subjects of potential interest: prime factors and divisors. Divisors include all the numbers that divide a number evenly, including 1 and the number itself. For example, the prime factors of 24 are 2, 2, 2, and 3, while the divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

The divisors of *i* can be obtained simply by trying every number from 1 to *i*; although 1 and *i* divide *i*, it's hardly worth making special cases for them:

Additional Material

Additional material related to this issue of the Analyst, including color images and Web links, is available on the Web. The URL is

http://www.cs.arizona.edu/icon/analyst/iasub/ia45/ia45sub.htm

```
procedure divisors(i)
   every j := 1 to i do
      if i % j = 0 then suspend j
end
```

This method is hopelessly slow except for small numbers. A more reasonable alternative is to generate the divisors of a number from its prime factors. If a number has a prime factorization of the form

$$a^i \times b^j \times c^k \times \ldots$$

then its divisors are generated by

```
suspend (a ^ (0 to i)) * (b ^ (0 to j)) * (c ^ (0 to k)) * ...
```

It's relatively easy to convert the output from *factor* to an Icon program that produces the divisors:

```
procedure main()
   local line, expr, number, factors, factor, term, exp
   write("procedure main()")
   while line := read() do {
      expr := ""
      line ? {
         write("writes(", image(tab(find(" = ") + 2)), ")")
         move(1)
         factors := tab(0)
         }
      factors ? {
         while term := tab(upto(' ') | 0) do {
            term ? {
               if factor := tab(upto('^')) then {
                  move(1)
                  exp := tab(0)
                  }
               else {
                  factor := tab(0)
                  exp := "1"
                  }
               }
            expr ||:= " * (" || factor || " ^ (0 to " || exp || "))"
            move(1) | break
            }
         write("every writes(\" \", ", expr[3:0], ")")
         write("write()")
         }
   write("end")
end
```

Typical output from this program is:

```
procedure main()
writes("100 =")
every writes(" ", (2 ^ (0 to 2)) * (5 ^ (0 to 2)))
write()
writes("101 =")
every writes(" ", (101 ^ (0 to 1)))
write()
writes("102 =")
every writes(" ", (2 ^ (0 to 1)) * (3 ^ (0 to 1)) * (17 ^ (0 to 1)))
write()
writes("103 =")
every writes(" ", (103 ^ (0 to 1)))
write()
                    ...
```

Divisors produced this way are not necessarily in numerical order. That can be taken care of easily, but it wasn't necessary for what we were doing.

Unfortunately, this approach is not workable for getting the divisors of a large number of numbers. There are, for example, 9,000,000 7-digit numbers. A program for generating their divisors would be more than 27,000,000 lines long. Granted, the output could be broken up into pieces, but that leads to considerable clerical complexity.

Fortunately, it's not necessary to generate Icon code and compile and execute it. Input in *factors* format can be used to produce the divisors directly. The obvious problem is that the number of terms is not known and is, in fact, not bounded.

In a situation where there is a computation involving an unknown number of components, recursion should come to mind. Here's a program to convert prime factors to divisors:

```
procedure divisors(factors)
   local term, factor, exp
   factors ? {
      term := tab(upto(' ') | 0)
      term ? {
         if factor := tab(upto('^')) then {
            move(1)
            exp := tab(0)
            }
```

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

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

```
      else {
        factor := tab(0)
        exp := 1
        }
      }
    if pos(0) then suspend alts(factor, exp)
    else suspend {
      move(1)
      alts(factor, exp) * divisors(tab(0))
      }
    }
end
```


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


```
procedure alts(factor, exp)
  suspend factor ^ (0 to exp)
end
```

Armed with the factoring programs and this one for finding divisors, we can investigate versum numbers as factors.

A few observations are in order before showing the results:

- All numbers have at least two divisors, 1 and the number itself.
- Since 1 is not a versum number, some numbers have no versum divisors. (A prime versum number, of course, has one divisor.)
- Since 2 is a versum number, half of all numbers have a least one versum divisor.

Here are the prime factor results for *n*-digit numbers, $1 \le n \le 7$; *vf* stands for versum prime factors and *nvf* stands for non-versum prime factors:

n	*vf*	*nvf*	*ratio*
1	7	7	1.000
2	97	125	0.776
3	1017	1619	0.628
4	10318	18788	0.594
5	103582	208045	0.498
6	1036923	2246080	0.462
7	10374384	28360244	0.435

Prime Factors of *n*-Digit Numbers

The fact that the percentage of versum prime factors drops off as *n* increases should not be surprising. Large numbers have larger factors and the percentage of versum numbers among all numbers drops off sharply as *n* increases [4].

We find it interesting that the form of the number of versum factors as *n* increases — the ratio of the number of versum factors for *n* and *n*+1

Icon on the Web

Information about Icon is available on the World Wide Web at

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

becomes very close to 10. (Of course, we're dealing with small values of *n*.)

Since for every increase in *n*, the number of numbers increases by 10, this means that the *increase* in the number of versum factors is approaching 0. In fact, the same is true of all factors, although it's not as marked or obvious. Here are the ratios:

n	*vf*	*all*
2	13.857	15.857
3	10.484	11.873
4	10.145	11.873
5	10.038	10.706
6	10.010	10.535
7	10.004	10.427

Increase in Number of Prime Factors

We would think that larger numbers would have more factors on average. These figures suggest that, if anything, the increase is very small. We've not been able to find anything on this subject. If you can provide a pointer to the literature, please let us know.

Now on to divisors:

n	*vd*	*nvd*	*ratio*
1	8	16	0.500
2	151	299	0.505
3	1716	4864	0.352
4	18239	68351	0.266
5	186702	886369	0.210
6	1887058	10916213	0.173
7	18953594	129801721	0.146

Divisors of *n*-Digit Numbers

The increase in the number of versum dividers again is approaching 10, but it's not clear that's true for all divisors, although the amount of increase decreases:

n	*vd*	*all*
2	18.875	19.750
3	11.364	14.622
4	10.628	13.159
5	10.236	12.392
6	10.107	11.931
7	10.043	11.618

Increase in Number of Divisors

Next Time

We could go on with factors and divisors, looking at numbers with special properties, such as polygonal numbers, Fibonacci numbers, ... There would be no end to this. This is not, however, The Journal of Versum Numbers.

In the next article on versum numbers, we'll move on to versum primes. Note that we have already touched on that subject in this article by looking a versum prime factors. There's much more to tell, however.

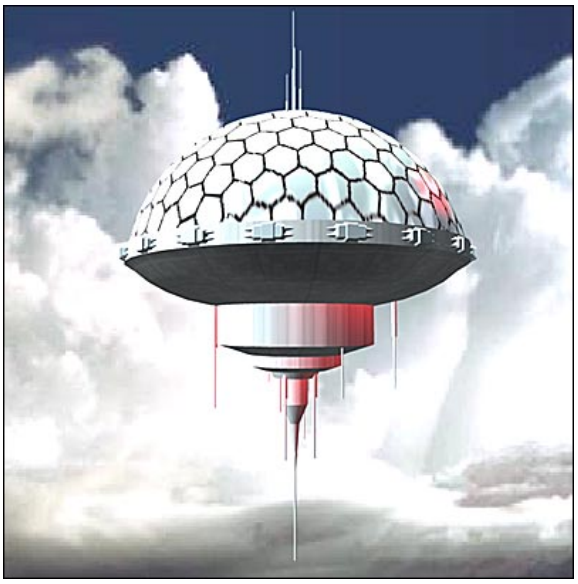
References

1."Factors of Versum Numbers", Icon Analyst 40, pp. 9-14.

2."Factors of Versum Numbers", Icon Analyst 43, pp. 9-14.

3. *The Mathematica Book*, Stephen Wolfram, 3rd ed., Wolfram Media and Cambridge University Press, 1996.

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



What's Coming Up

We have lots in the works: an article on an Icon debugger that we had hoped to complete for this issue, as well as articles on sorting, paths, versum primes, and using Icon's preprocessor.

We also have material for the **Graphics Corner**, **Tricky Business**, and **Programming Tips**.