Numerical Carpets

Ralph E. Griswold
Department of Computer Science, The University of Arizona

Introduction

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).

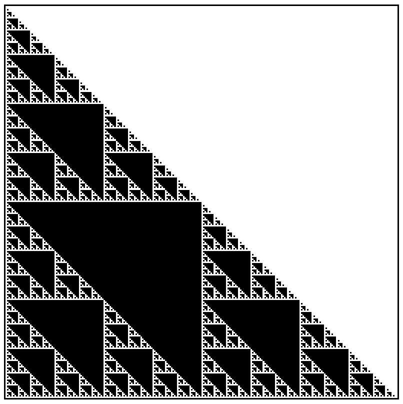


Figure 1. Pascal's Triangle Modulo 2

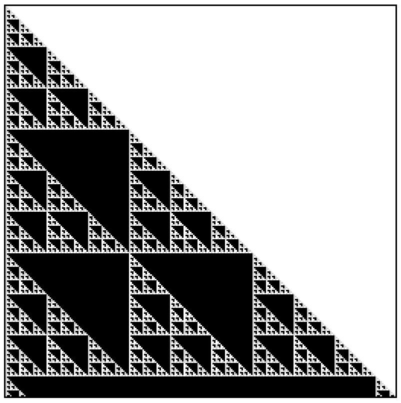


Figure 2. Pascal's Triangle Modulo 3

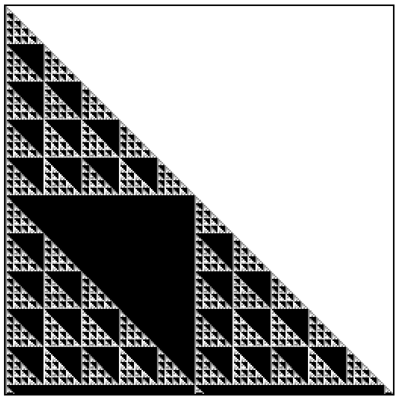


Figure 3. Pascal's Triangle Modulo 5

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,… .
- *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 the cell and the 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.

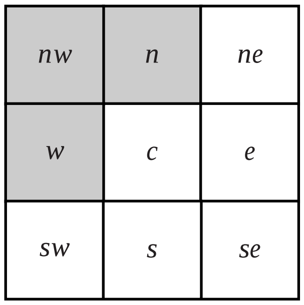


Figure 5. The Carpet Neighborhood

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.

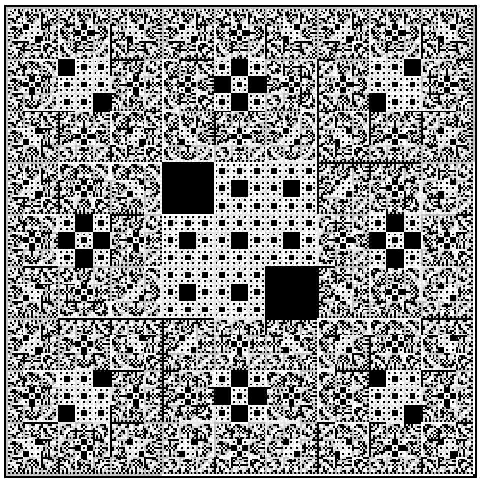


Figure 6. Carpet from All-Ones Initialization

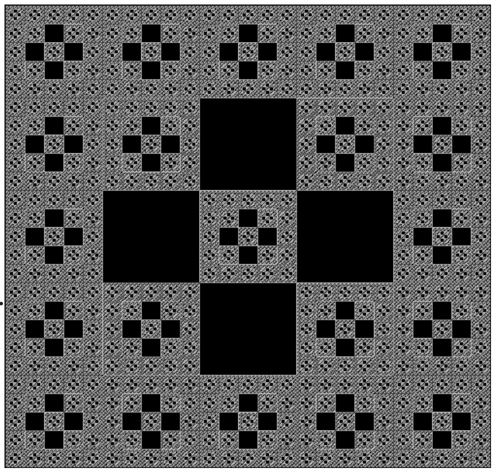


Figure 7. Carpet from One-Zero Initialization

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.)

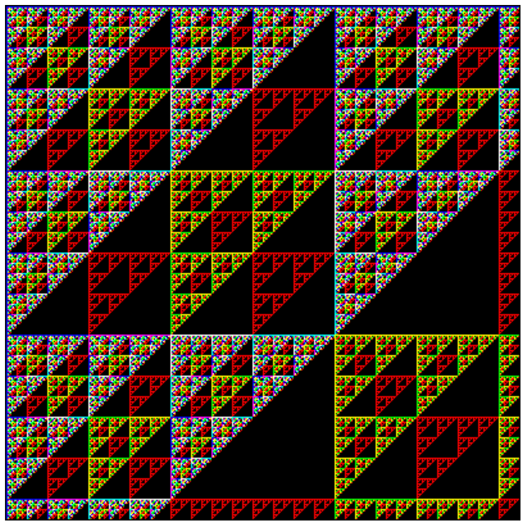


Figure 8. "Pascal" Carpet

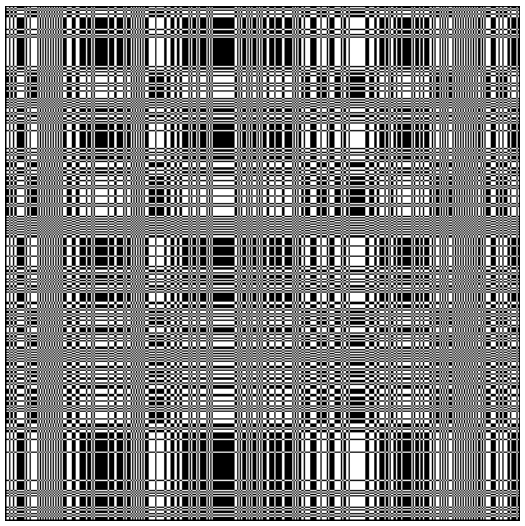


Figure 9. "Open Weave" Carpet

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

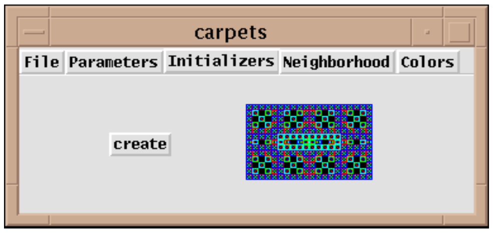
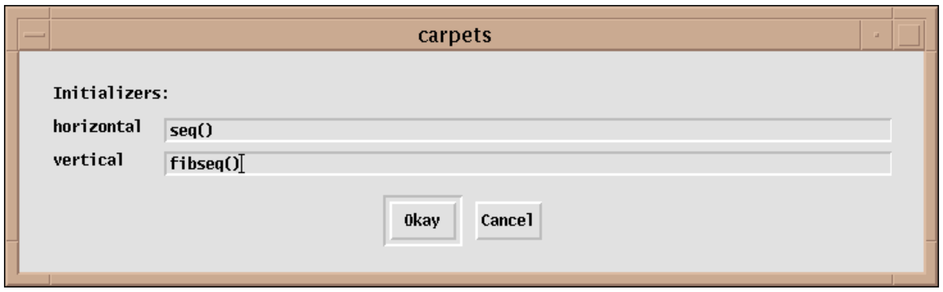


Figure 10. Carpet-Specification Interface

Figure 11 shows the dialog for entering and editing initializers.



Initializers:

horizontal seq()

vertical fibseq()

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.

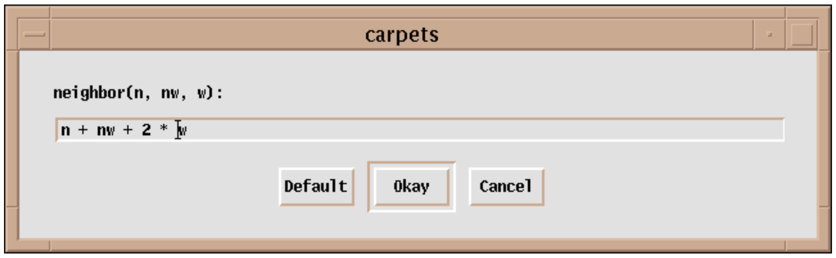


Figure 12. The Neighborhood Dialog

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.

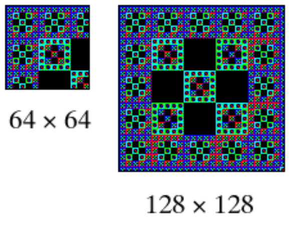


Figure 13. One Effect of Carpet Size

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.

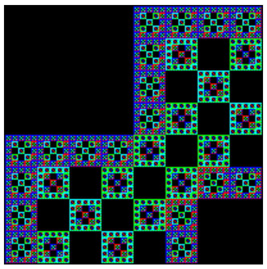


Figure 14. Another Effect of Carpet Size

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.

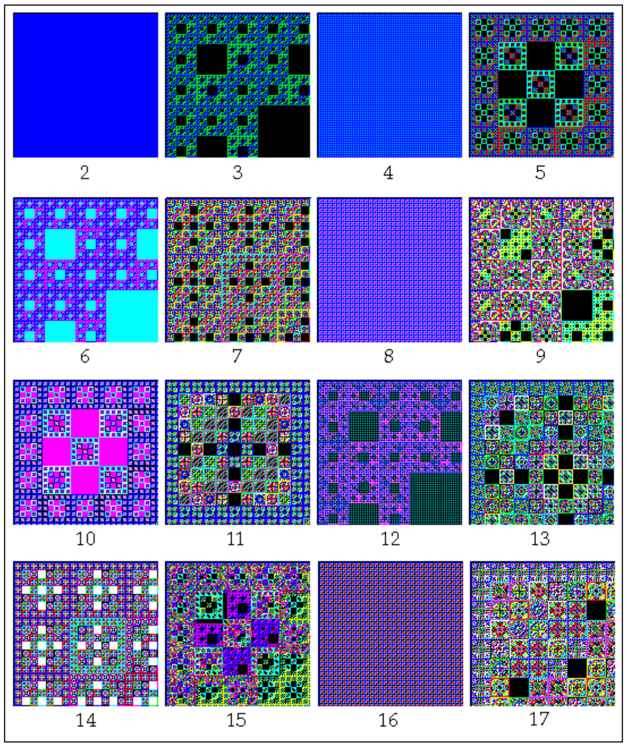


Figure 15. Effects of the Modulus

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.

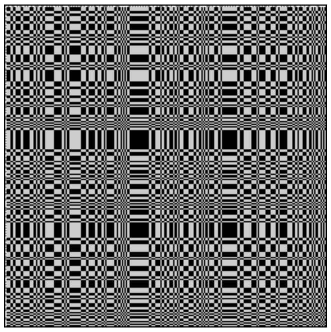


Figure 16. The Primes with Modulus 4

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.

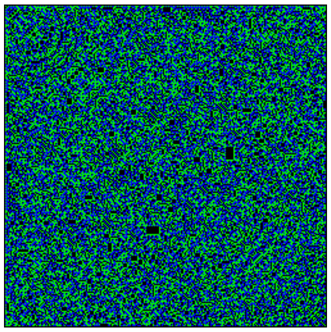


Figure 17. The Primes with Modulus 3

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.

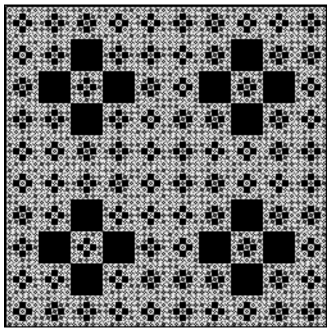


Figure 18. A Non-Standard Neighborhood

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 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.

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.

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.