The Icon Analyst

In-Depth Coverage of the Icon Programming Language

February 2000
Number 58

In this issue

Twills 1
From the Library — Complex Arithmetic 2
Residue Sequences 4
Weavable Color Patterns 7
Graphics Corner — Custom Palettes 10
Name Drafts Revealed 15
What's Coming Up 16

Twills

In the article on name drafting in the last issue of the Analyst [1], we mentioned that name drafts usually are woven in overshot, which uses twill tie-ups.

Twills typically have a diagonal texture. See Figures 1 and 2.

Figure 1. A Twill Weave

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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

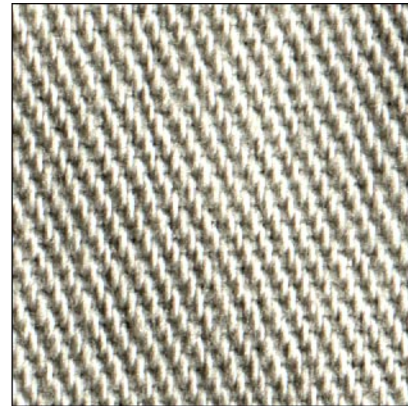


Figure 2. Twill Weave Enlarged

The diagonal effect is the result of an interlacing in which threads pass over or under two or more perpendicular threads in a shifting pattern. This is accomplished by tie-ups that have corresponding shift. The effect of a tie-up like this is shown clearly when the threading and treadling are straight draws. See Figure 3.

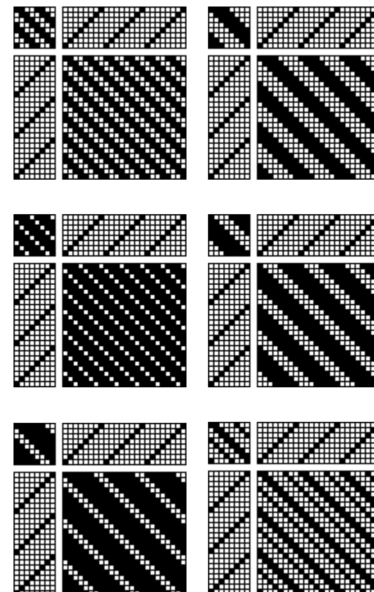


Figure 3. Twill Drafts

A twill tie-up is characterized by a row pattern that rotates left or right by one square for

successive rows. For example, the top row of the tie-up for the bottom-right draft in Figure 3 has the pattern 2-over, 3-under, 1-over, 2-under.

Weavers have several ways of representing such patterns. One is exemplified by

$$\frac{2 \quad 1}{3 \quad 2}$$

This notation, although graphically suggestive, is difficult to produce typographically. At the present time, the notation

/2/3/1/2

is most commonly used. Incidentally, this notation indicates the number of treadles, and unless otherwise stated, the number of shafts is the same.

Here's a procedure that converts this kind of notation to a bi-level image string:

```
procedure twill_tieup(pattern, shift)
  local row, i, rows, count

  count := 1              # odd/even over/under toggle

  row := ""

  pattern ? {
    while ="/" do {
      i := tab(many(&digits)) | fail
      row ||:= repl(count, i)
      count +:= 1
      count %:= 2
      }
    if not pos(0) then fail
    }

  if *row < 2 then fail

  rows := []

  put(rows, row)

  every i := 1 to *row – 1 do
    put(rows, row := rotate(row, shift))

  return rows2pat(rows)

end
```

The procedure rows2pat() is from the library module patutils.

As shown in Figure 3, the over/under pattern has a marked effect on the appearance of weaves. With different threadings and treadlings, as shown in the article on name drafts, the effects can be even more varied.

The subject of twills is enormous. All kinds of variations and devices are used, and there are entire books devoted to the subject [2-6]. Most of this material is beyond the scope of the Analyst, although it may crop up from time to time in future articles.

References

1. "Name Drafting", Icon Analyst 57, pp. 11-14.

2. *A Twill of Your Choice*, Paul R. O'Connor, Interweave Press, 1981.

3. *Fascination of Twills (Fourshafts), Master Weaver Library* ,Vol. 9, S. A. Zielinski, Nilus Leclerc, 1981.

4 *Fascination of Twills (Multishafts), Master Weaver Library* ,Vol. 10, S. A. Zielinski, Nilus Leclerc, 1981.

5. *Extended Divided Twill Weaves*, Virginia Harvey, Shuttle Craft Guild Monograph 39, Shuttle-Craft Books, 1988.

6. *Extended Manifold Twill Weaves*, Virginia Harvey, Shuttle Craft Guild Monograph 40, Shuttle-Craft Books, 1989.

From the Library — Complex Arithmetic

For books are more than books. They are the life, the very heart and core of ages past, the reason why men lived and worked and died, the essence and quintessence of their lives.

— Amy Lowell

Complex numbers are familiar to anyone who has taken courses in basic mathematics. Although they often are introduced in particular application contexts, it's worth remembering that complex numbers are just pairs of numbers. Gauss introduced the concept for integers — "Gaussian integers" — although most modern applications are for pairs of real numbers.

The usual representation of complex numbers is as

a + b*i*

where a is the *real part*, b is the *imaginary part*, and $i = \sqrt{-1}$.

The main use of complex numbers is to represent points in the "complex plane" as illustrated in Figure 1.

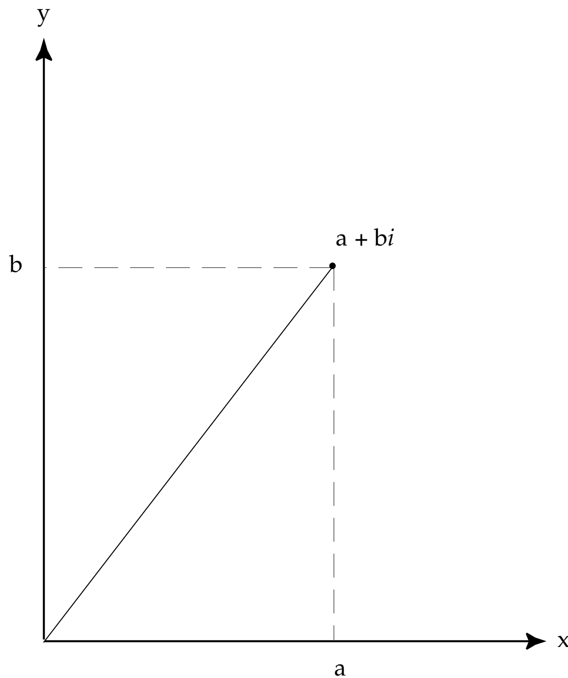


Figure 1. The Complex Plane

Complex numbers provide a way of manipulating quantities in the plane. Despite the terminology, this is just arithmetic in the x-y plane.

The rules of arithmetic for complex numbers reflect the geometry:

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

$$(a + bi) \times (c + di) = ((a \times c) - (b \times d)) + ((a \times d) + (b \times c))i$$

$$(a + bi) / (c + di) = ((a \times c) + (b \times d)) / (c^2 + d^2) + (((b \times c) - (a \times d)) / (c^2 + d^2))i$$

Two functions defined for complex numbers are useful: the complex conjugate and absolute value:

$$conj(a + bi) = a - bi$$

$$abs(a + bi) = \sqrt{a^2 + b^2}$$

Library Module

The Icon program library module complex provides complex arithmetic. The implementation is similar in style to that for rational numbers [1].

The string representation of a complex number is exemplified by "(3.0+5.25i)". The internal representation uses records:

```
record complex(rpart, ipart)
```

There are seven procedures for manipulating complex numbers:

strcpx(s)	convert string to record
cpxstr(z)	convert record to string
cpxadd(z1, z2)	add complex numbers
cpymul(z1, z2)	multiply complex numbers
cpxdiv(z1, z2)	divide complex numbers
cpxconj(z)	form complex conjugate
cpxabs(z)	form absolute value

The implementation of these procedures is straightforward. An example is

```
procedure cpxadd(z1, z2)

   return complex(z1.rpart + z2.rpart,
      z1.ipart + z2.ipart)

end
```

The current version of complex.icn is on the Web page for this issue of the Analyst.

Reference

1. "From the Library—Rational Arithmetic", Icon Analyst 57, pp. 19-20.

Residue Sequences

Given a sequence $S = \{t_n\}$, its residue sequence modulo m is $\{t_n \bmod m\}$. For example, the residue sequence of the squares mod 7, $\{n^2 \bmod 7\}$, is

1, 4, 2, 2, 4, 1, 0, 1, 4, 2, 2, 4, 1, 0, 1, 4, 2, 2, 4, 1, 0, 1, 4, 2, 2, 4, 1, 0, 1, 4, 2, 2, 4, 1, …

while the residue sequence of the squares mod 11 is

1, 4, 9, 5, 3, 3, 5, 9, 4, 1, 0, 1, 4, 9, 5, 3, 3, 5, 9, 4, 1, 0, 1, 4, 9, 5, 3, 3, 5, 9, 4, 1, 0, 1, ...

If the t_n are all nonnegative we can compute the terms of the residue sequence using `tn % m`; otherwise we need to use the library procedure `residue(tn)` [1]. For example,

```
(seq() ^ 2) % 11
```

generates the residue sequence of the squares mod 11.

Residue sequences are of interest to us for two reasons:

- The magnitudes of their terms are bounded.
- Many residue sequences are periodic.

Having all the terms within a fixed range — 0 to $m - 1$ or 1 to m for shaft arithmetic [1] — obviously is useful for threading and treadling sequences in weaving. Periodic sequences provide the natural basis for repeats in weaves and other kinds of designs. Periodic sequences also have finite data representations.

If you look at the two sequences shown above, you'll note the residue sequence of the squares mod 7 is periodic with period 7 and the residue sequence of the squares mod 11 is periodic with period 11. This is no accident, but don't jump to the conclusion that all residue sequences of the squares mod m are periodic with period m. They all are periodic, but not all with period m. For example, for $m = 12$, the period is 6.

We'll come back to this and related issues in another article.

Modular Arithmetic

Modular arithmetic, invented by Gauss, has many important uses, including random number generation, hardware design, and cryptography.

Two integers a and b are equal modulo m, written

$$a \equiv b \bmod m$$

if the remainder of a divided by m is equal to the remainder of b divided by m. Put another way, $a \equiv b \bmod m$ if and only if $a - b$ is a multiple of m.

There are numerous laws and relations in modular arithmetic. The most important for us are

if $a \equiv b$ and $c \equiv d \bmod m$ then

$$a + c \equiv b + d \bmod m$$
$$a - c \equiv b - d \bmod m$$
$$a \times c \equiv b \times d \bmod m$$

Cancellation (division) does not hold in general in modular arithmetic. We'll come back to this point in a later article.

These laws enable us to compute many residue sequences more efficiently than computing terms of the sequence and then taking their residues. For example, for the Fibonacci sequence given by the recurrence relation

$$t_1 = t_2 = 1$$
$$t_n = t_{n-1} + t_{n-2} \qquad n > 2$$

Its residue sequence modulo m can be computed by

$$t_1 = t_2 = 1$$
$$t_n = (t_{n-1} + t_{n-2}) \bmod m \qquad n > 2$$

This avoids computation with very large integers (on a computer with 32-bit words, the forty-seventh term in the Fibonacci sequence, 2,971,215,073, exceeds the word size). Thus many residue sequences can be computed using programming languages that don't support large-integer arithmetic and avoiding the slower computation and storage allocation for those that do.

There are many ways to cast the computation of the Fibonacci sequence in Icon. Here's one that you may not have seen before:

```
procedure fibseq()
  local terms

  terms := [1, 1]

  repeat {
    put(terms, terms[1] + terms[2])
    suspend get(terms)
    }

end
```

The use of a list instead of specific identifiers

allows easy generalization to other recurrence relations.

The computation of the residue sequences of the Fibonacci sequence can be cast as

```
procedure fibmodseq(m)
   local terms

   terms := [1, 1]

   repeat {
      put(terms, (terms[1] + terms[2]) % m)
      suspend get(terms)
      }

end
```

Properties of Residue Sequences

There are several properties of residue sequences that may be of interest:

- Are they periodic?
- If so, what are their periods?
- If periodic, do they have pre-periodic parts?
- What residues are present?
- What is distribution of the residues?

Contrary to an ill-considered claim we've seen that all residue sequences are periodic, some are not. Examples of non-periodic residue sequences are the primes and the digits of the decimal (or any other base) expansion of π. Another sequence we've mentioned before is the "multi" sequence which consists of n copies of n:

1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, …

Its residue sequences are not periodic because the span of equal numbers constantly increases.

Some sequences have periodic residue sequences only for certain moduli. For example, versum sequences always have periodic residue sequences for moduli 9 and 11 but not for most other moduli. (If you've followed the articles on versum sequences, the reason for the periodicity for modulus 11 should be obvious; 9 is another story that we'll get to later.)

Whether or not a residue sequence is periodic, the residues it contains are important. In weaving, for example, missing residues correspond to unused shafts or treadles. For example, for the squares mod 7, only four of the seven possible residues are present: 0, 1, 2, and 4, while for 11, only six of the 11 possible residues are present: 0, 1, 3, 4, 5, and 9.

Of the questions posed above, few general results are known. Even when it's known *a priori* that a residue sequence is periodic, predicting its period is difficult if not intractable.

For some sequences, however, a considerable amount is known. The Fibonacci sequence is the prime example [2-4].

All Fibonacci residue sequences are purely periodic. The periods do not exceed $6 \times m$, and periods of $6 \times m$ are achieved only for $m = 2 \times 5^n$ for $n > 0$ — 10, 50, 250, and so on.

We cannot find any information, however, on the residues present in these sequences or on their distribution. These matters may never have been considered.

Figure 1 on the next page shows the Fibonacci residue sequences for m = 2 through 18.

Recurrence Relations

The following sequences are known to have periodic residue sequences for all moduli:

$\{ n^k \}$

$\{ k^n \}$

$\{ n \times (n + 1) / 2 \}$ (the triangular numbers)

What do these have in common? They all can be represented by linear recurrence relations with constant coefficients — that is, by relations of the form

$$t_n = c_1 \times t_{n-1} + c_2 \times t_{n-2} + \ldots + c_k \times t_{n-k}$$

where each term is determined by a fixed number of previous terms.

The subject of recurrence relations is enormous. We'll start to take up part of it in the next issue of the Analyst.

References

1. "Shaft Arithmetic", Icon Analyst 57, pp. 1-5.

2. *Fibonacci Numbers*, N. N. Vorob'ev, Pergamon Press, 1961.

3. *Linear Recursion and Fibonacci Sequences*, Brother Alfred Brousseau, Fibonacci Association, 1971.

4. *Fibonacci and Related Number Theoretic Tables*, Brother Alfred Brousseau, Fibonacci Association, 1972.

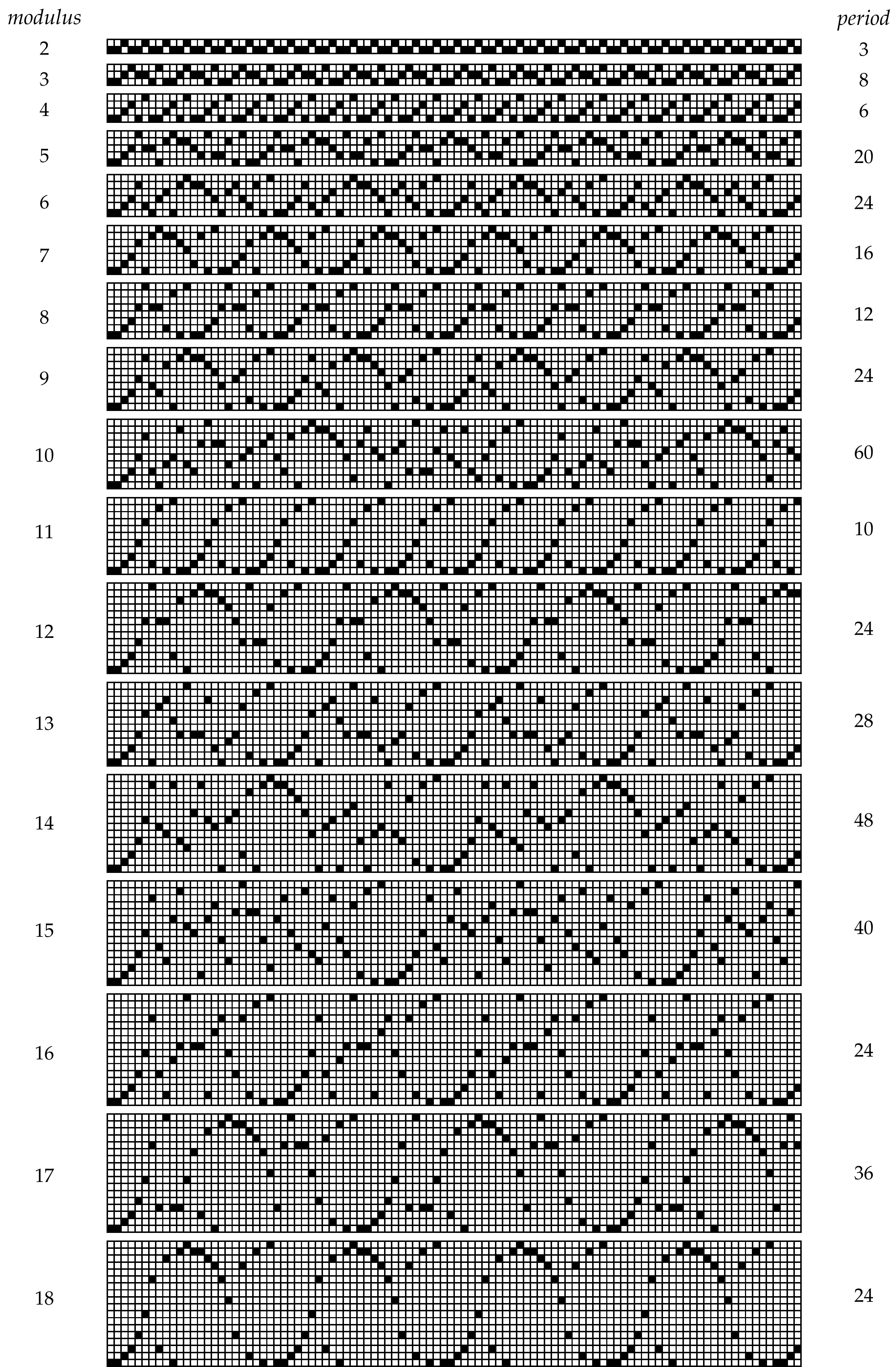


Figure 1. Fibonacci Residue Sequences

Weavable Color Patterns

Nature uses only the longest threads to weave her patterns, so each small piece of her fabric reveals the organization of the entire tapestry.

— Richard Feynman [1]

Suppose you see a color image that strikes your fancy and think “Hey! That would make great place mats. I'll ask Aunt B to weave some for me.” This is not as simple as it sounds. In the first place, Aunt B would need a weaving draft. We've shown how to create drafts from two-color images [2], but for more colors, the problem is considerably more difficult. And there are color patterns that can't be woven. In fact, most can't. This article explores the problem of determining if a color pattern can be woven.

The Problem

From a weaving point of view, every pixel in a color image corresponds to a point of interlacement between a vertical (warp) thread and horizontal (weft) thread. Therefore, either the warp thread or the weft thread must be the color of the pixel.

We can consider a colored image as a rectangular grid of squares that contain characters standing for colors. Columns correspond to warp threads and rows correspond to weft threads. In order for the grid to be weavable, the columns and rows must be labeled in a way that the label for every square is its column label or its row label— “satisfied”. Figure 1 shows an example grid.

	c_1	c_2	c_3
r_1	A	B	C
r_2	C	B	A

Figure 1. A Labeled Grid

Back Issues

Back issues of The Icon Analyst are available for $5 each. This price includes shipping in the United States, Canada, and Mexico. Add $2 per order for airmail postage to other countries.

We can assign column and row labels as follows. Starting with square (c_1, r_1), either c_1 or r_1 must be A. Arbitrarily pick c_1 to be A. This forces r_2 to be C. Figure 2 shows the labeling to this point.

		A		
		c_1	c_2	c_3
	r_1	A	B	C
C	r_2	C	B	A

Figure 2. First Labeling Step

We now see that c_3 must be A. This requires r_1 to be C and hence c_2 must be B. Figure 3 shows the final labeling.

		A	B	A
		c_1	c_2	c_3
C	r_1	A	B	C
C	r_2	C	B	A

Figure 3. The Final Labeling

So far, so good. But what about the grid shown in Figure 4?

	c_1	c_2	c_3
r_1	A	B	C
r_2	C	A	B

Figure 4. Another Grid

We can start as we did before, assigning A to c_1. This forces r_2 to be C, which in turn forces c_2 and c_3 to be A and B, respectively, as shown in Figure 5.

		A	A	B
		c_1	c_2	c_3
	r_1	A	B	C
C	r_2	C	A	B

Figure 5. First Step in Labeling

Now we're stuck: r_1 cannot be both B and C. If we start anywhere else and try any other combination of labelings, we find it's not possible to satisfy

the grid, and the pattern cannot be woven.

Note that if a larger pattern contains such a subpattern, the larger pattern cannot be woven either.

This is a very small pattern by weaving standards and it has only three colors. What then of more colors and larger patterns?

The Number of Colors

Suppose a pattern has k colors. We already know that for $k = 2$, all patterns can be woven [2] — assign one color to all columns (warp threads) and the other color to all rows (weft threads) and pick one or the other depending on the color at every intersection.

We've illustrated by example that for $k = 3$, there are some patterns that cannot be woven. For larger k there is a more fundamental problem. If a pattern has m columns and n rows, there are only $m + n$ colors available. If k is greater than $m + n$, then the pattern can't be woven at all. Thus, there are 2×3 patterns that can't be woven for this reason. See Figure 6.

	c_1	c_2	c_3
r_1	A	B	C
r_2	D	E	F

Figure 6. A Pattern with Too Many Colors

For what follows, we'll assume $k \leq m + n$.

Approaches to Solving the Problem

There are several possible ways the problem might be solved.

One way would be to write out the constraints a pattern imposes and cast them as an expression with mutual evaluation. For Figure 1, a program might look like this:

```
procedure main()
  local r1, r2, c1, c2, c3

  every {
    (r1 := "A" | "B" | "C") &
    (r2 := "C" | "B" | "A") &
    (c1 := "A" | "C") &
    (c2 := "B") &
    (c3 := "C" | "A") &
    (\r1 | \c1) == "A" &
    (\r1 | \c2) == "B" &
    (\r1 | \c3) == "C" &
    (\r2 | \c1) == "C" &
    (\r2 | \c2) == "B" &
    (\r2 | \c3) == "A"
    }
  do {
    write("rows=", r1, r2)
    write("cols=", c1, c2, c3)
    write()
    }
end
```

This program works as expected and produces two solutions, the second of which is the one we showed earlier:

```
rows=AA
cols=CBC

rows=CC
cols=ABA
```

It's not that hard to write a "compiler" that reads in a grid and produces such a program:

```
procedure main()
  local rows, col, i, j

  rows := []

  while put(rows, read())      # build array of rows

  write("procedure main()")
  write("   every {")
  every i := 1 to *rows do
    write("      (r", i, " := !",
      image(string(cset(rows[i]))), ") & ")
  every j := 1 to *rows[1] do {
    col := ""
    every col ||:= (!rows)[j]
    write("      (c", j, " := !",
      image(string(cset(col))), ") & ")
    }
  every i := 1 to *rows do
    every j := 1 to *rows[1] do
      write("      (r", i, " | c", j, ") == ",
        image(rows[i,j]), " &")
  write("      &null")               # terminate conjunction
  write("      }")
  write("   do {")
  write("      writes(\"rows=\")")
  every i := 1 to *rows do
    write("      writes(r", i, ")")
  write("      write()")
  write("      writes(\"cols=\")")
```

```
    every j := 1 to *rows[1] do
      write("     writes(c", j, ")")
    write("     write()")
    write("     write()")
    write("     }")
    write("end")

  end
```

A more compact program can be produced by using an array in place of individual variables, but we won't bother, since this solution method takes far too long for all but simple small patterns. The reason is obvious: In general, all possible combinations must be tried until there is a solution.

We have two viable solutions. One recognizes the problem as an instance of the 2-satisifiabilty (2-SAT) problem, for which there is a known algorithm.

The other solution is heuristic in nature. We'll describe the heuristic solution here for several reasons:

- It's original as far as we know.
- It's interesting.
- It's fast for most patterns.
- It illustrates an approach that is worth considering for other problems.

The Heuristic Solution

A Word About Heuristics

A word about heuristics is in order, since they often are misunderstood. Heuristics use insights into the nature of a problem and intelligent guesses to build a solution method tailored to the problem.

Using heuristics doesn't mean wild guessing or proceeding blindly, just hoping to find a solution. Nor need a heuristic solution give incorrect answers, although proving a heuristic method is correct and terminates — and hence is an algorithm — may be difficult.

Heuristics can be used in many ways. For the problem here, one possibility would be look for a fast way to reject a pattern because it contains an unsolvable subpattern (such as the ones shown earlier). Of course, the absence of a known unweavable subpattern does not prove the whole pattern is weavable — so that problem would still exist.

Checking for special cases such as this one often takes more time on average than it saves. A notable example occurred in an implementation of SNOBOL4, in which a heuristic was used with the intention of saving time in storage management. Instead, it wasted time on average [3].

Since it's difficult — even impractical — to analyze the effects of such heuristics without implementing them and doing performance testing, such heuristics should be viewed with skepticism.

A good heuristic method relies on understanding the nature of the problem and, if possible, breaking the problem down into smaller, more tractable, subproblems.

Insights into Color Weavability

For the color weavability problem, the following observations are particularly useful.

- If a row or column is all one color, that color can be assigned to the corresponding row or column without affecting the rest of the problem. Hence such rows and columns can be eliminated from further consideration.
- Duplicate rows and columns can be eliminated for the same reason.
- The pattern can be rotated without changing the problem; in this sense, there is no difference between rows and columns.
- Rows can be interchanged (rearranged) without changing the problem, and the same is true of columns.

To get ideas for the heuristic approach to the problem, we can look at small patterns and see what implications they have for the pattern as a whole. Consider the 3-colored 2×2 patterns shown in Figure 7.

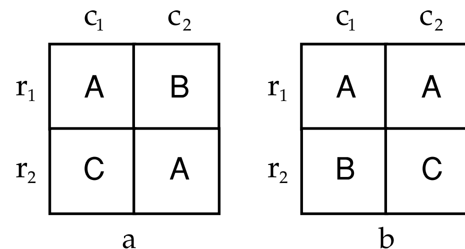


Figure 7. Three-Colored 2×2 Patterns

These are the only distinct 3-colored 2×2 patterns; all others are equivalent to these by rotation or row and column interchange.

Figure 7a imposes some constraints on any larger pattern in which it is embedded: c_1 must be A or C, c_2 must be B or A, and similarly for the two

rows. For Figure 7b, however, c_1, c_2, and r_2 are not constrained but r_1 is completely determined and must be A for the entire pattern in which this subpattern is embedded. This particular subpattern turns out to provide a sufficient basis for a heuristic solution; no others need be considered.

We'll leave you with that thought and describe the program in the next article.

References

1. *The Character of Physical Law*, Richard Feynman, MIT Press, 1967.

2. "Drawups", Icon Analyst 56, pp. 18-20.

3. "Performance of Storage Management in an Implementation of SNOBOL4", Ralph E. Griswold, David R. Hanson, and David G. Ripley. *IEEE Transactions on Software Engineering*, Vol SE-4, No. 2 (1978), pp. 130-137.

Graphics Corner—Custom Palettes

If you love the intense cloud, pour into every image its warm summer blood.

— Paul Eluard

Background

We described Icon's built-in palettes in an earlier article on image strings [1] and complete documentation is available in the Icon graphics book [2].

Built-in palettes are adequate for most images and uses. It's more typical to have problems because of the limitation to 256 different colors in an image. The colors provided by built-in palettes, however, are fixed and in some situations they give poor or even misleading results.

One situation the built-in palettes cannot handle is many shades of the same color, such renderings of illuminated reflective surfaces. While the built-in grayscale palettes can handle most "black-and-white" images, there is no corresponding capability for, say, shades of green.

The images shown in Figure 1 illustrate this problem. The top-left image is the result of ray tracing a dimly illuminated glass sphere. The image to its right is the result of reducing the number of colors to 256 to get a GIF. You probably can't tell the difference between the two, given the way the Analyst is printed, but if you look at the images on our Web site, you'll see the consequences of the reduction in the number of colors. The remaining six images are the results of reading the GIF image with Icon's color palettes.

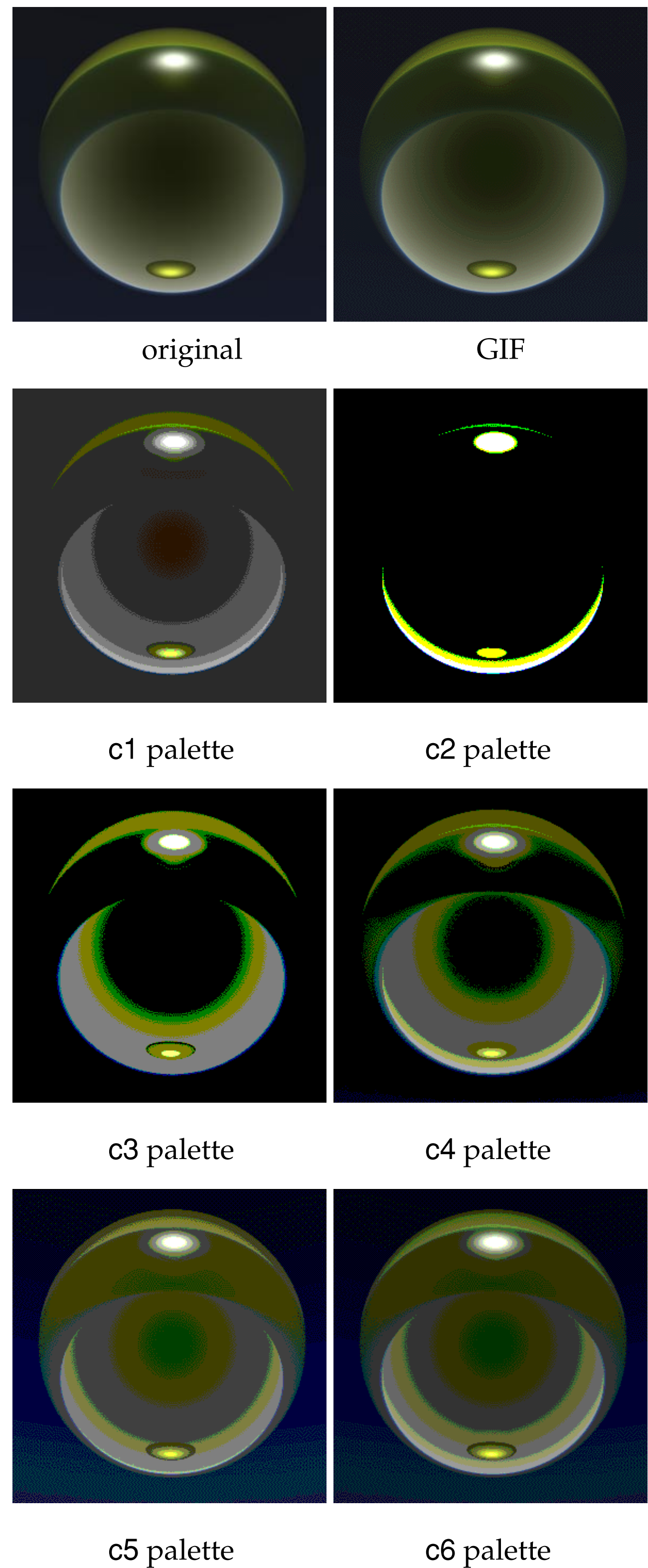


Figure 1. Interpreting an Image Using Palettes

The c1 palette is a general-purpose palette with 90 colors based on Icon's color naming system. It works well for most color images but not for this one. Palettes c2 through c6 have equally spaced colors in RGB space (plus extra grays) and range from 9 to 241 colors. You'd expect c2 to give poor results, but you might hope for a good rendering with the color palettes with more colors. This is not the case. In fact, only 32 of the 241 colors from the c6 palette are usable — the colors in the image are packed close together in small regions of the RGB cube where there are only a few colors in the c6 palette.

The Palette Mechanism

For reference, here's a brief description of Icon's palette mechanism:

DrawImage(x, y, ims) draws the image corresponding to the specified image string.

PaletteChars(name) returns the keys in the named palette.

PaletteColor(name, key) returns the color in the named palette for the given key.

PaletteGrays(name) returns the keys for shades of gray in the named palette.

PaletteKey(name, color) returns a key in named palette for a color that is close to the given color.

ReadImage(file, x, y, name) reads the specified image file using only the colors in the named palette.

Custom Palettes

To overcome the limitations of built-in palettes, we've added a library module for programmer-defined, custom palettes.

Design Considerations

One design requirement was that custom palettes be a transparent addition to the built-in mechanism. That is, it should be possible to create and use color palettes in all contexts that built-in palettes can be used, and the user should not have to know whether a built-in or custom palette is being used.

In addition, to make color palettes as useful as possible, they should be able to represent any color that Icon's color system can handle. And, unlike built-in palettes, custom palettes should allow duplicate colors (we'll show uses for this capability in a later article).

The user should be allowed to chose the name for a color palette, except possibly excluding the names of the built-in palettes, and to specify the keys.

Finally, there should be no limit to the number of custom palettes that can be used except the memory required for them.

Data Structures

A record is used to encapsulate the information associated with a custom palette:

```
record Palette_(name, keys, table)
```

where name is the name of the palette, keys is a list of the palette keys, and table is a table. The table is indexed by the palette keys whose corresponding values are the colors. Note that keys is redundant; it is provided to avoid repeated computation during use.

Here's a simple example, constructed manually:

```
rgb_palette := Palette_("rgb", "012", table())

rgb_palette.table["0"] := "red"
rgb_palette.table["1"] := "green"
rgb_palette.table["2"] := "blue"
```

One difficulty with managing custom palettes is that they must be accessible to programs that use them. In addition, different sets of custom palettes may be needed in different situations.

To handle these needs, custom palettes are accessed through "databases" that are tables indexed by the palette names whose associated values are Palette_() records. A palette database can be saved in a file using xencode() and loaded into a program using xdecode() as shown in previous articles.

The current palette database is the value of the global variable PDB_. The use of a global variable is awkward, but it works and there seems to be no better way to interface the custom palette mechanism.

Procedures

The module palettes links two other modules and declares PDB_ and two records:

```
link imrutils
link lists

global PDB_

record Palette_(name, keys, table)
```

```
record Color_(r, g, b)
```

Underscores are used to reduce the probability of collision with names in programs that use this module.

It's obviously necessary to be able to distinguish between built-in palettes and custom palettes. For this, we use a bit of trickery:

```
procedure BuiltinPalette(name)

  BuiltinPalette := proc("PaletteChars", 0)

  return BuiltinPalette(name)

end
```

This procedure takes advantage of the fact that the built-in palette functions fail if given a name that is not the name of a built-in palette. Here BuiltinPalette is assigned the built-in function PaletteChars, which is called in the first invocation of the procedure and then subsequently.

Creating a custom palette is relatively straightforward:

```
procedure CreatePalette(name, keys, colors)
  local i, k, t

  initial InitializePalettes()

  if BuiltinPalette(name) then fail

  if *labels ~= *cset(keys) then fail      # duplicate
  if *labels ~= *colors then fail          # mismatch

  t := table()

  every i := 1 to *colors do
    t[keys[i]] := ColorValue(colors[i]) | fail

  PDB_[name] := Palettes_(name, keys, t)

  return PDB_[name]

end
```

Here name is the palette name, keys is a string of key characters, and colors is a list of color specifications.

The procedure InitializePalette() is called by all procedures that deal with custom palettes before they do anything else. It doesn't do much, but it provides the necessary abstraction and a place other work could be done:

```
procedure InitializePalettes()

  /PDB_ := table()

  InitializePalettes := 1          # make it a  no–op

  return

end
```

Note that InitializePalettes() changes itself into an operation that does nothing, so that only the first call of it is effective.

CreatePalette() allows — and requires — the caller to specify the keys. In most cases, the specific keys used in a palette are not of interest or even known to the user. The following utility procedure provides keys automatically to facilitate the creation of custom palettes:

```
procedure makepalette(name, clist)
  local keys
  static alphan

  initial alphan := &digits || &letters

  if *clist = 0 then fail

  keys := if *clist < *alphan then alphan else &cset

  return CreatePalette(name, keys[1+:*clist ], clist)

end
```

Some procedures overload built-in ones of the same names. The simplest are:

```
procedure PaletteChars(args[])
  local name
  static palette_chars

  initial {
    InitializePalettes()
    palette_chars := proc("PaletteChars", 0)
    }

  if type(args[1]) == "window" then get(args)

  name := args[1]

  if BuiltinPalette(name) then
    return palette_chars(name)
  else
    return (\PDB_[name]).keys

end
```

and

```
procedure PaletteColor(args[])
  local palette_lookup, name, s
  static palette_color

  initial {
    InitializePalettes()
    palette_color := proc("PaletteColor", 0)
    }

  if type(args[1]) == "window" then get(args)
```

```
  name := args[1]
  s := args[2]

  if BuiltinPalette(name) then
    return palette_color(name, s)

  palette_lookup := (\PDB_[name]).table | fail

  return \palette_lookup[s]
end
```

These procedures illustrate a minor complication that has to be handled. All the built-in palette functions have an optional window first argument. This argument is not used (although a window must be open to allow access to platform-dependent color names). In the procedures above, if the first argument is a window, it is discarded.

Note that the built-in versions of PaletteChars() and PaletteColor() are assigned to static variables and used if the palette name is a built-in one.

The procedure PaletteKey(name, color) returns the key for a color in the named palette that is close to the specified color. For custom palettes, this procedure is implemented by finding a palette color that is the minimum distance from the given color in RGB space:

```
procedure PaletteKey(args[])
  local name, s
  static palette_key

  initial {
    InitializePalettes()
    palette_key := proc("PaletteKey", 0)
    }

  if type(args[1]) == "window" then get(args)

  name := args[1]
  s := args[2]

  if BuiltinPalette(name)
    then return palette_key(name, s)
    else return NearColor(name, s)

end
```

NearColor() returns a color near to s:

```
procedure NearColor(name, s)
  local palette_lookup, k, measure, close_key, color

  measure := 3 * (2 ^ 16 – 1) ^ 2        # maximum

  color := ColorValue(s) | fail

  palette_lookup := (\PDB_[name]).table | fail

  every k := key(palette_lookup) do{
    if measure >:= Measure(palette_lookup[k], color)
    then {
      close_key := k
      if measure = 0 then break          # exact match
      }
    }

  return \close_key

end
```

Measure() produces a value that determines how far apart the colors are. The distance between two colors is given by

```
sqrt((color1.r – color2.r) ^ 2 +
  (color1.g – color2.g) ^ 2 + (color1.b – color2.b) ^ 2)
```

It is not necessary, however, to take the square root to compare values for different colors:

```
procedure Measure(s1, s2)
  local color1, color2

  color1 := RGB(s1)
  color2 := RGB(s2)

  return (color1.r – color2.r) ^ 2 +
    (color1.g – color2.g) ^ 2 + (color1.b – color2.b) ^ 2

end
```

The procedure RGB() parses the color value and returns a record:

```
procedure RGB(s)
  local color

  color := Color_()

  ColorValue(s) ? {
    color.r := tab(upto(',')) &
    move(1) &
    color.g := tab(upto(',')) &
    move(1) &
    color.b := tab(0)
    } | fail

  return color

end
```

The procedure DrawImage() is the most challenging to implement. Since it requires drawing individual pixels, efficiency is important. The approach is to create a table with the same keys as the custom palette but with corresponding values that are lists of the coordinates at which the keys appear in the image string. Once all the coordinates have been computed, all the pixels for one key are writ-

ten in a single call of DrawPoint().

```
procedure DrawImage(args[])
  local palette_pixels, palette_lookup
  local keys, c, x, y, row, imr
  static draw_image

  initial {
    InitializePalettes()
    draw_image := proc("DrawImage", 0)
    }

  if type(args[1]) ~== "window" then
    push(args, &window)

  imr := imstoimr(args[4]) | return draw_image ! args

  if BuiltinPalette(imr.palette) then
    return draw_image ! args

  palette_lookup := (\PDB_[imr.palette]).table | fail
  palette_pixels := copy(palette_lookup)

  keys := cset(imr.pixels)

  every !palette_pixels := [args[1]]

  x := args[2]

  every c := !keys do {
    y := args[3]
    imr.pixels ? {
      while row := move(imr.width) do {
        row ? {
          every put(palette_pixels[c],
            x + upto(c) – 1, y)
          }
        y +:= 1
        }
      }
    }

  every c := !keys do {
    Fg(args[1], palette_lookup[c]) | fail
    DrawPoint ! palette_pixels[c]
    }

  return

end
```

This procedure must deal with the case in which DrawImage() is called with a bi-level pattern [3]. This possibility is detected (more or less) by the failure of imstoimr() when given a bi-level pattern. For this case, the built-in version of DrawImage() is used. The procedure imstoimr() is from the imrutils module [4].

Loose Ends

Three palette capabilities have not yet been implemented: transparency, PaletteGrays(), and the specification of a palette in ReadImage(). When these are done, they will be added to the palettes module, which is on the Web site for this issue of the Analyst.

The Next Step

The palettes module provides the necessary infrastructure for creating and using custom palettes. Creating custom palettes from scratch is, however, cumbersome.

In a subsequent article, we'll describe programs that facilitate the creation of custom palettes, including one for creating custom palettes from images and an interactive application that allows the construction and modification of custom palettes in various ways.

References

1. "Graphics Corner — Drawing Images", Icon Analyst 49, pp. 11-13.

2. *Graphics Programming in Icon*, Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend, Peer-to-Peer Communications, 1998.

3. "Graphics Corner — Bi-Level Patterns", Icon Analyst 56, pp. 4-5.

4. "Graphics Corner — Fun with Image Strings", Icon Analyst 50, pp. 10-13.

Supplementary Material

Supplementary material for this issue of the Analyst, including images and program material, is available on the Web. The URL is

http://www.cs.arizona.edu/icon/analyst/iasub/ia58/

Name Drafts Revealed

In the article on name drafts [1], we showed eight woven images based on name drafts and said we'd show the strings and tie-ups used in a subsequent article. It is of course reasonable if not obligatory to provide this information.

We obtained the woven images by relatively uncontrolled experimentation. From perhaps 25 images, we selected eight that we thought were the most interesting and visually distinctive.

When we offered to "reveal all", we'd forgotten what silly things we'd tried. Now we're confronted with a more embarrassing revelation than we expected. Nonetheless, here it is.

The general model for producing the threading and treadling sequences is

```
s1 := reflect(s)
OddEvenPDCO{genmapshafts(s1, p(s1))}
```

where s is the string used, reflect() creates a pattern palindrome, and p is a procedure that determines how the characters of s are mapped onto shaft numbers. See the previous article for details [1].

Figure 1 shows the woven images again for reference and Figure 2 on the next page lists the corresponding parameters. The columns *map1* and *map2* in Figure 2 give the maps for the threading and treadling sequences, respectively (which are the same except for image d). See the article about twills that starts on page 1 for an explanation of the tie-up notation.

Because we were experimenting and not always thinking through what we were doing, there are some unusual effects. For example, fchars(s) produces the characters of s in decreasing order of frequency. It does not take into account the order in s in which characters of the same frequency occur, and because of the way it is coded (which might deserve changing), it scrambles the order of the characters that appear equally frequently. For example, fchars(&letters) produces

```
"kKxhXHueUErbRBoOlLyiYIvfVFscSCpPmMzjZJ_
  wgWGtdTDqaQAnN"
```

This is an interesting effect, but we weren't aware of it until we wrote this article.

As to the results, one thing we found particularly interesting was the striking difference in appearance between images a and b, although they only differ in their tie-ups and number of shafts used.

We said we'd reveal all and we have. Don't you wish we hadn't?

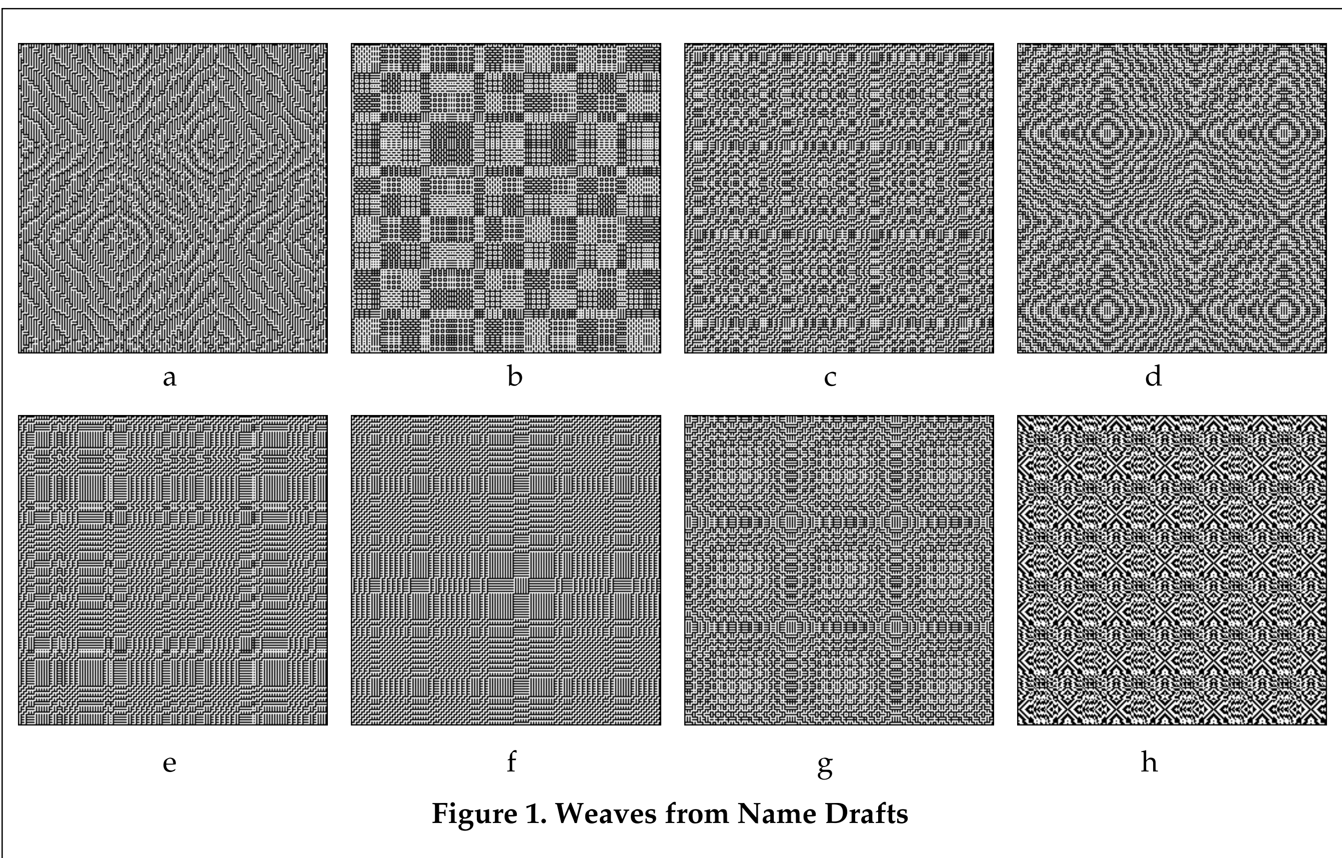


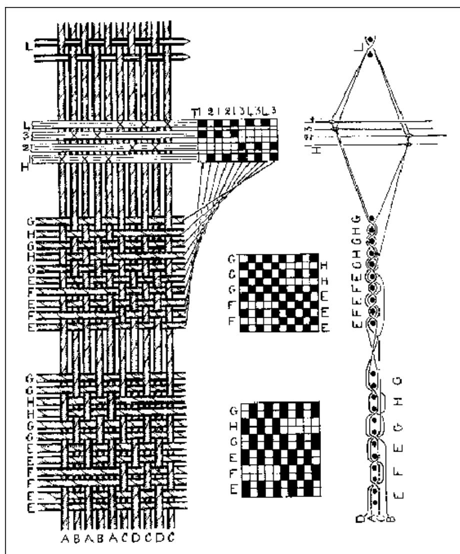
Figure 1. Weaves from Name Drafts

image	*string*	*map1*	*map2*	*tie-up*
a	&letters	fchars()	fchars()	/2/1/2
b	&letters	fchars()	fchars()	/1/2/1
c	"Metropolitan Lace Pudding"	fchars()	fchars()	/1/2/1
d	&letters ++ &digits	fchars()	cset()	/1/2/1
e	"Spring has sprung. The grass has ris. I wonder where the flowers is."	csort()	csort()	/1/2/1
f	"Moses supposes his toeses are roses, but Moses supposes erroneously."	fchars()	fchars()	/1/2/1
g	"Long, long, ago, far, far, away."	fchars()	fchars()	/1/2/1
h	"The Icon Project"	ochars()	ochars()	/3/1

Figure 2. Name-Draft Parameters

Reference

1. "Name Drafting", Icon Analyst 57, pp. 11-14.



What's Coming Up

We'll continue our series on sequences in the next issue of the Analyst with an article on recurrence relations.

We'll follow up the article on weavable color patterns in this issue with the description of the program itself. We also plan an article on tie-ups.

Once again we failed to make a start on the planned series of articles on classical cryptography. We do, however, have the first article in draft form and it may make it into the next issue of the Analyst.

More directly related to Icon, we'll have an article that explains the Icon linker and how to avoid some problems that arise from its removal of unreferenced procedures.

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-analyst@cs.arizona.edu

THE UNIVERSITY OF
ARIZONA®
TUCSON ARIZONA
and

Bright Forest Publishers
Tucson Arizona

© 2000 by Ralph E. Griswold, Madge T. Griswold, and Gregg M. Townsend

All rights reserved.