The Icon Analyst

In-Depth Coverage of the Icon Programming Language

August 1999
Number 55

In this issue …

Correction

When we were completing the last issue of the Analyst, we made some last-minute changes to the article on shadow weaves [1] to make it fit. In our haste, we introduced some errors relating to the figures.

The corrections are:

Page 6, column 1, lines 12 and 13: "The 10 weaves in Figure 4 represent the visual extremes" should read "The eight weaves in Figure 5 represent the visual extremes".

Page 6: The figure at the bottom of the page should be numbered 5, not 9.

Page 7: The last sentence of column 1 that starts "For example, …" and continues to the references in column 2 should be deleted.

Reference

1. "A Weaving Case Study", Icon Analyst 54, pp. 4-7.

Digit Patterns in Primes

We've had the material for this article since we first started the series of articles on character patterns [1]. Since the topic is largely frivolous, we didn't get around to writing it up at the time. It could well have languished forever in one of the many folders we have that contain partially developed Analyst articles.

We presently have 42 such folders as well as three 3-ring notebooks full of older material. There are several reasons why we have such a large backlog and so much unfinished work. Sometimes an article doesn't seem to pan out but still has potential. Sometimes our interests change and we concentrate on new topics, such as weaving. Sometimes an article never seems to fit into the issue of the Analyst we're currently developing. Another factor is the large amount of time and effort it takes to bring an article to finished form. And, of course, there are the ever-present demons Disorganization, Forgetfulness, and Procrastination.

We'll work on removing one folder with this article (while probably adding several more before we complete it).

Before getting to digit patterns in primes, we need to set the stage — to establish the prime mood.

Prime numbers hold endless fascination. Part of the fascination is intrinsic. Part is the quest for ever larger examples. And part of it is the unusual structure of some primes. Mersenne primes [2], which are of the form $2^p - 1$ where p is a prime, are best known.

Many persons interested in primes work in recreational areas. Prime numbers of a certain type are collected and new examples sought.

Classification of Primes

Primes are classified in a number of ways:

size
mathematical form

mathematical properties
mathematical relationships
digit patterns
typographical properties

Size

The focus on size reflects the continuing search for ever-larger primes. The "official" classifications are <1>:

gigantic: 10,000 (decimal) digits or more
titanic: 1,000 digits to 9,999 digits

These classifications are, of course, quite arbitrary. At present there are 10 known primes with more that 100,000 digits. The next largest prime to be found most likely will have over 1,000,000 digits. The presently established size categories stand to make less and less sense as larger primes are found. *Useless factoids:* As of this writing, there are more than 3,500 known gigantic primes. There are so many known titanic primes that only "interesting" ones are being recorded.

It seems to us that it would make more sense to rank a prime by the number of digits required to express the number of digits it has. For example, 89 (2 digits) and 10067 (5 digits) both have rank 1, while the Mersenne prime $2^{216091} - 1$, which has 65,050 digits, has rank 5.

Mathematically, the rank is given by

$$[\log_{10}(\log_{10}(p))]$$

where $[r]$ denotes the integer part of r.

This can be easily cast in Icon without the problems associated floating-point arithmetic:

```
rank := **p
```

The presently largest known primes have rank 6. There are no known rank 7 primes. There are, of course, primes of all ranks; it's just that primes of higher ranks remain to be discovered. In the absence of a formula for primes, even with the ever-increasing computational power available, the search will go slowly.

Mathematical Form

Many known primes have distinctive mathematical forms, of which the Mersenne primes mentioned earlier are the best known.

The Fermat primes have the form $2^{2^n} + 1$. It was once conjectured that all numbers of this form are prime, but that is false. Only five Fermat primes are known: 3, 5, 17, 256, 65537, resulting from $n = 0$, 1, 2, 3, and 4. There also are generalized Fermat primes of the from $d^{2^n} + 1$. The Cullen primes have the form $n \times 2^n \pm 1$. There are factorial primes, which are of the form $n! \pm 1$, the largest known being $3610! - 1$.

The repeated digit ("repdigit") primes have the form $d \times (10^n - 1) / 9$. There are five known "repunit" primes for $d = 1$, which in base-10 notation consist entirely of 1s. The largest known is for $n = 1031$. It is known that any larger repunit primes would require $n > 30{,}000$.

There are many other mathematical forms for primes. See Reference 3 and Links 2 and 3.

Mathematical Properties

A regular prime p is one that does not divide the class number $h(p)$ of the cyclotomic field obtained by adjoining a primitive root of unity to the rational field (Got that?). Irregular primes are, as you'd expect, those that are not regular. There are an infinite number of irregular primes, but it is not known if there are an infinite number of regular primes.

Other primes with particular mathematical properties are the Wieferich primes, for which

$$2^{p-1} = 1 \bmod p^2$$

and the Wilson primes, for which

$$(p - 1)! = -1 \bmod p^2$$

Only three Wilson primes are known: 5, 13, and 563.

Certain kinds of numbers might be included in this category, such as Fibonacci numbers that are primes. 25 Fibonacci primes are known, of which the largest is term 93311 in the Fibonacci sequence.

Mathematical Relationships

Some primes are related in special ways, such as the Sophie Germain primes, p, $2p + 1$ and the twin primes, p, $p + 2$. There also are k-tuplet primes <4>. For example, 11, 13, 17, and 19 comprise a prime quadruplet.

Digit Patterns

We've already mentioned repdigit primes. Other much-studied primes with distinct digit patterns are palindromic primes of various types.

Since main subject of this article is digit patterns, we'll defer the bulk of the discussion until later.

Typographical Properties

Typographical properties have to do with the shapes of digits. Among these are the strobogrammatic primes, which are unchanged if turned upside down. An example is 619. A tetradic (or 4-way) prime is a palindromic strobogrammatic prime that is the same in four ways — from right to left, left to right, top to bottom, or upside down. An example is 18181. There even are "holey" primes that have a preponderance of digits with "holes" in them: 0, 4, 6, 8, and 9.

It seems to us that classifying primes by their typographical properties goes beyond the realm of recreational mathematics and into plain silliness. Note also that some typographical properties depend on the way digits are written. Notably, 4 is sometimes written as 4 without a "hole".

Miscellaneous

In classifications of things, there almost always is a category for things that don't fit anywhere else but don't deserve a category of their own.

For primes, an example is the "prime digit primes", all of whose digits are prime — composed only of the digits 2, 3, 5, and 7. The largest known prime digit prime is

$$(72323252323272325252) \times (10^{3120} - 1) \,/\, (10^{20} - 1) + 1$$

This number can be represented more compactly using pattern forms [4], which we'll explore later.

Another example is the "absolute primes" for which all rearrangements of digits result in primes. 18 of these are known, the largest of which is the current largest known repdigit prime. The largest known absolute prime that is not a repdigit prime is 991.

A Comment on Classifications

In digit patterns, typographical properties, and most miscellaneous primes, most specific types are well-defined for representation in various bases, although only base 10 is usually considered.

Of course, almost all such properties are dependent on the base. For example, a prime that is palindromic in one base usually is not palindromic in another.

Unlike mathematical properties, properties dependent on the base are not intrinsic or fundamental. The prominent mathematician G. H. Hardy

How Many Primes Are There?

In order to put primes with special properties in perspective, it's useful to realize how *many* primes there are.

Surprisingly, it's possible to compute exactly the number of primes ≤ k without having to find them all.

The prime counting function, $\pi(k)$, gives this number <1,2>. The computation is cumbersome and complicated, but it is tractable. As far as we know, the largest value of k for which $\pi(k)$ has been computed is $k = 10^{20}$, although we can only find the values to 10^{19}. From $\pi(k)$, it's easy to produce a list of the number of n-digit primes, which we find more interesting than the number for a specific value. We'll designate the number of n-digit primes by $\pi'(n)$:

n	$\pi'(n)$
1	4
2	21
3	143
4	1061
5	8363
6	68906
7	586081
8	5096876
9	45086079
10	404204977
11	3663002302
12	33489857205
13	308457624821
14	2858876213963
15	26639628671757
16	249393770611366
17	2344318816620308
18	22116397130086627
19	209317712988603747

There are formulas that approximate $\pi(k)$. The simplest originated with Gauss and Legendre:

$$\pi(k) \approx k \,/\, \log(k)$$

It has been proved that this relationship is asymptotic and is called the *prime number theorem.*

This approximation is somewhat low,

continued on next page

but it's close enough to give a feeling for the magnitudes involved. This formula is trivial to implement:

```
procedure pi(k)
   return integer(k / log(k))
end
```

This procedure allows us to compute the approximate number of n-digit primes for large n. For example,

$$\pi'(10^{100}) \approx 3.9 \times 10^{97}$$

How close the approximation is to the exact value can be seen in the ratios of the exact to the approximate values, which we'll designate by $\underline{\pi}'(n)$:

n	$\pi'(n)/\underline{\pi}'(n)$
1	2.00
2	1.24
3	1.16
4	1.13
5	1.10
6	1.08
7	1.07
8	1.06
9	1.05
10	1.05
11	1.04
12	1.04
13	1.04
14	1.03
15	1.03
16	1.03
17	1.03
18	1.03
19	1.02

The point of all this is that there are an enormous number of prime numbers and, except for small values of n, only a few are known. The reason that a large percentage of known large primes have "special" characteristics is that persons working in the field look for them.

Links

1. http://www.treasure-troves.com/math/PrimeCountingFunction.html
2. http://www.utm.edu/research/primes/howmany.shtml

was particularly critical of work that depended on the base, considering it not to be "serious mathematics" or useful in any way [5]. For us, that does not matter: We make no pretense of doing serious mathematics.

Getting the Digits of Primes

In order to investigate digit patterns in primes, it is, of course, necessary to get the digits. We dealt with this subject to some extent in an article on versum primes [2]. We'll include that material here to provide a self-contained discussion.

Except for the smallest primes, primes are recorded by formulas. This is possible if for no other reason than because if p is a prime, $q_1 = p + 1$ and $q_2 = p - 1$ are composite. Since q_1 and q_2 are composite, they have at least two prime factors, so that, in general, p can be written as $p_1 \times p_2 \times \ldots \pm 1$, and the process continued for the prime factors. In most cases, the expressions for large primes are much shorter than the primes themselves.

Often the number of prime factors is large, the most being the Mersenne primes, $2^p - 1$. Many known large primes have the form $p \times 10^n \pm 1$ or $p \times 2^n \pm 1$ (which includes the Mersenne primes). No doubt the reason for this is the comparative ease of finding primes of these mathematical forms. That definitely is the case for Mersenne primes.

In the listing of the largest known primes <5>, the longest expressions have only 48 characters. There are 19 of these, which have the same form up to constant values. An example is:

```
34344713643960928*(2^(3*1189)–2^1189)–6*2^1189–1
```

Incidentally, the large integer in this expression cannot be written in fewer symbols using only standard arithmetic operators.

Trivia quiz: What's the smallest positive integer whose representation as an expression using only digits and standard arithmetic operators is shorter than the integer itself? An example of a small integer that has a shorter representation is 16384 for which an expression is `2^14`. This is not the smallest one, however.

Fortunately for what we want to do, most of the formulas given in the on-line list of large primes are syntactically correct Icon expressions as they stand and have the expected interpretations.

Exceptions to this are suffix and bracketing operators. The suffix operators are *n!*, for factorial,

n!!...!, the "multi-factorial" (which is not the repeated application of the factorial but something different that we'll explain later), and *n*#, the "primorial", which stands for the product of primes ≤ *n*.

The only bracketing operator is [*r*], the "floor" of *r*, which in the case of prime formulas is just integer(r).

Here's a procedure to convert bracketing and suffix operators to legal Icon syntax:

```
link strings

procedure fixup(exp)
   local result, term, op, i

   exp := replacem(exp, "[", "integer(", "]", ")")

   exp ? {
        result := ""
        while result ||:= tab(upto(&digits)) do {
          term := tab(many(&digits))
          if pos(0) then {
            result ||:= term
            return result
            }
          case op := move(1) of {
            "#":   result ||:= "primorial(" || term || ")"
            "!":   {
              i := 1
              i +:= *tab(many('!'))
              result ||:= "mfactorial(" || term || "," || i || ")"
              }
            default:  result ||:= term || op
            }
          }
        return result || tab(0)
        }

end
```

The procedure replacem() is from the strings module in the Icon program library. It operates on its first argument. For subsequent pairs of arguments, all instances of the first string in the pair are replaced by the second.

Procedures are used to implement the primorial and "multi-factorial". Here's primorial():

```
procedure primorial(n)
   local k, m

   m := 1

   every k := primeseq() do {
     if k <= n then m *:= k
     else return m
     }

end
```

The "multi factorial" is defined as follows:

$$n! = n \times (n-1) \times (n-2) \ldots 1$$
$$n!! = n \times (n-2) \times (n-4) \ldots$$
$$n!!! = n \times (n-3) \times (n-6) \ldots$$

and so on. Here's a procedure:

```
procedure mfactorial(n, i)
   local j

   if n < 0 then fail
   if i < 1 then fail

   j := n

   while n > i do {
     n -:= i
     j *:= n
     }

   return j

end
```

Most of the functions that appear in the expressions for primes also are implemented by Icon procedures. A few functions are too complicated for it to be worthwhile to create corresponding Icon procedures. An example is the cyclotomic polynomial of order *n* in *x*, which is given by

$$C_n(x) = \prod_k (x - e^{2\pi i k/n})$$

where *k* runs over all positive integers less than *n* that are relatively prime to *n*

For the few prime expressions that can't be evaluated in Icon, we use Mathematica [6]. There are only 28 expressions among the 5,200 in the current list of largest known primes that require going to Mathematica.

Next Time

We're out of room, and we've only touched on the subject of character patterns in primes. The topic may be frivolous, but it's nonetheless bulky. There's more to come.

References

1. "Character Patterns" Icon Analyst 49, pp. 1-6.
2. "Versum Primes" Icon Analyst 46, pp. 12-16.

3. *The New Book of Prime Number Records*, Paulo Ribenboim, Springer-Verlag, 1995.

4. "Pattern Forms Revisited" Icon Analyst 52, pp. 4-6.

5. *A Mathematician's Apology*, G. H. Hardy, Cambridge University Press, 1992.

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

Links

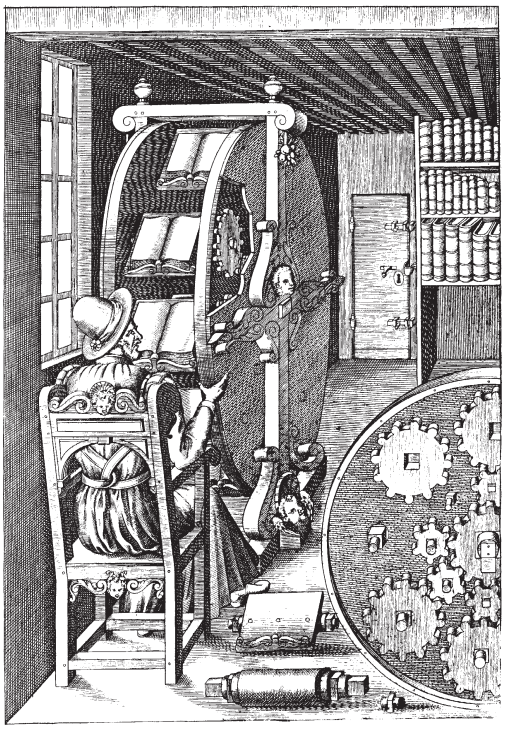
1. http://www.utm.edu/research/primes/index.html

2. http://www.utm.edu/research/primes/lists/top_ten/

3. http://www.utm.edu/research/primes/types.html

4. http://www.ltkz.demon.co.uk/ktuples.htm

5. http://www.utm.edu/research/primes/largest.html



From the Library — Generators

The Icon program library contains many generative procedures — procedures that can generate sequences of values. Most of them generate numbers (primarily integers). The other large group generates strings. A few generate lists, and in the graphics portion of the library, there are many procedures that generate records containing the coordinates of points for various kinds of paths and geometrical figures.

Unfortunately, the generative procedures are scattered among many modules — 103 in all. Most of the generative procedures for numbers and strings, however, are in four modules: genrfncs, numbers, strings, and pdco. Except for generators related to graphics, most of the generative procedures that you might want are linked in the module seqfncs. If you include

```
link seqfncs
```

in programs that need generative procedures, you'll most likely get what your need, and a lot more. (The Icon linker removes code that is not referenced, so any extra baggage does not show up in the resulting executable file.)

Here's a sampling of generative procedures that are not tied to any particular application. See the documentation for the modules for details.

Integer Generators

genrfncs

chasosseq()	Hofstadter's chaotic sequence [1]
factseq()	the factorials
fibseq()	the Fibonacci numbers, including generalizations (Lucas numbers)
figureseq()	the figurate numbers
geomseq()	the geometric numbers
ngonalseq()	the polygonal numbers
partitseq()	integer partitions
powerseq()	the exponential numbers
primeseq()	the prime numbers
versumseq()	the versum numbers [2]

This module also contains procedures for many other less well-known sequences.

random

randseq()	successive values of &random

String Generators

strings

comb()	the combinations of characters of

	a string of a specified length
palins()	palindromes
permute()	permutations
substrings()	substrings

Point Generators

Points are represented by records with three fields, x, y, and z, declared in the module gobjects. Most procedures that generate points use only the x and y fields.

curves

ellipse()	ellipses (including circles)
lissajous()	Lissajous figures
parabola()	parabolas

This module contains 21 other plane curves, including some with esoteric names, like the Lemniscate of Gerono and the Trisectrix of MacLaurin.

fstars

fstar()	fractal stars [3]

rpolys

rpoly()	regular polygons

rstars

rstar()	regular stars

More to Come

In coming articles, we'll mention some additional sequences that are useful in puzzles, quizzes, and specific topics, such as weaving and versum numbers.

In the next article on the Icon program library, we'll continue with generative procedures, covering programmer-defined control operations, most of which are generators.

References

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

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

3. *Graphics Programming in Icon*, Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend , Peer-to-Peer Communications, Inc., 1998, 91-92, 116.

Solutions to Exercises

See Icon Analyst 54, page 8, for statements of the exercises.

1. The code

```
patterns := [seq_old]

  repeat {
    seq_new := [ ]
    every put(seq_new, seq_old[!seq_old])
    if lequiv(seq_new, !patterns) then
      break
    put(patterns, seq_new)
    seq_old := seq_new
    }
```

"self organizes" the list of integers given in seq_old. It creates a new list of integers, seq_new from the positive integers in seq_old, possibly reordering them. The method is to go through the elements of seq_old from beginning to end, appending to seq_new the element of seq_old in position *i* of seq_old. For example, if the first element of seq_old is 3, the 3rd element of seq_old is appended to seq_new. If an element of seq_old is greater than the size of seq_old, it is discarded. An example may help:

[3, 1, 2, 4] ➔ [2, 3, 1, 4]

The process goes like this:

1. The 1st element of seq_old is 3; the 3rd element of seq_old is 2, so it becomes the 1st element of seq_new.

2. The 2nd element of seq_old is 1, so its 1st element, 3, is appended to seq_new.

3. The 3rd element of seq_old is 2, so its 2nd element, 1, is appended to seq_new.

4. The 4th element of seq_old is 4, so it is appended to seq_new.

Successive results are placed in patterns. The loop terminates when there is a repetition of a previous sequence. The loop always terminates because there are only a finite number of possibilities.

For many sequences, the process terminates after only one or two passes. For example,

[3, 1, 2, 4] ➔ [2, 3, 1, 4] ➔ [3, 1, 2, 4]

2. Creating images for modular circles is a problem with many parameters that a user might want to

specify. We've cast the solution as a procedure, so that it can be called from a program that takes parameters as command line options or as a program with a visual interface that takes user specifications interactively.

We've added optional spokes, which sometimes help make the relationships clearer, and an optional offset, which is useful in situations such as counting starting at 1, as for the shafts and treadles of a floor loom.

The procedure uses rpoly() from the Icon program library to obtain a list of points for the appropriate polygon.

The procedure returns a hidden window containing the image, which the caller can display by setting "canvas=normal", write an image file, and so on. It's the responsibility of the caller to close the window when it no longer is needed.

The procedure itself creates a PostScript file for the image; the caller must know what its name will be to use it.

```
link graphics
link psrecord
link rpolys

procedure wheel(mod, spokes, width, radius,
  revs, offset, name, font, gap, Pradius)
    local half, i, win, p

  #  defaults for omitted arguments

  /mod := 8                 # modulus
  /width := 400             # window width
  /radius := 60             # radius of inner circle
  /revs := 2                # revolutions around circle
  /offset := 0              # offset of first value
  /name := "wheel"          # file name
  /font := "times,12"       # font
  /gap := width / 20        # gap between circles
  /pradius := 3             # radius of points

  half := width / 2

  win := WOpen("size=" || width || "," || width,
    "font=" || font, "canvas=hidden") |
      stop("*** cannot open window")

  PSEnable(win, name || ".eps")

  DrawCircle(win, half, half, radius)

  every p :=
    rpoly(half, half, radius, mod, –&pi / 2) do {
      FillCircle(win, p.x, p.y, pradius)
      if \spokes then DrawLine(win, half, half, p.x, p.y)
      }

  i := offset

  every 1 to revs do
    every p := rpoly(half, half, radius +:= gap,
      mod, –&pi / 2) do {
        CenterString(win, p.x, p.y, i)
        i +:= 1
        }

  PSDone()

  return win

end
```

Note that CenterString() is used to position the labels.

Animation — Image Replacement

> *There is a dual aspect to the animation form: part is right-brained, relying heavily on the talent and inspiration of the artist, and part is left-brained, involving the discipline and organization needed to deal with the myriad details associated with production.*
>
> — Gary Chapman [1]

In the case of simple animations, like the kaleidoscope [2, 3], a program may be able to produce acceptable results by just drawing. Reversible drawing and the use of mutable colors for the purpose of animation are tricks that can be useful for some kinds of special effects.

For complex animations, such as realistic action scenes, the only feasible way of producing an animation is by image replacement — replacing the canvas (or a portion of it) by a succession of previously prepared images.

The main problems with using previously prepared images are the speed with which they can be displayed and the space required for the images.

To produce acceptable animations, a "frame rate" of at least 15 frames per second (fps) is necessary, except for some kinds of cartoon animations, for which 10 fps will do. 24 fps is desirable. 15 fps translates into ~67 msec. per image. On the other hand, if the frame rate is too fast, apparent motion will be distorted or there will be visual artifacts.

The achievable frame rate depends heavily on the computer used — its basic processing speed and any special graphics hardware.

Frame rate generally is proportional to the

number of pixels in a frame. For example, doubling image dimensions reduces the achievable frame rate by a factor of four.

Similarly, the amount of space required for an image increases with its size roughly in the same way. Space isn't just a problem of storage; it's often the bottleneck in network communication. For these reasons, animations tend to be small and short.

In Icon, there are three ways of doing image replacement: DrawImage(), ReadImage(), and CopyArea(). Each method has advantages and disadvantages.

DrawImage() allows the manipulation of image strings during animation. It also supports transparency, which, among other things, allows the animation of non-rectangular areas. DrawImage() is, however, too slow for anything but small animations. Image strings also are large in comparison with GIF image files.

ReadImage() supports transparency also, and allows the use of compact file formats, but it is too slow for anything but small animations.

CopyArea() is very fast but images must be loaded into windows prior to animation to take advantage of its speed. Preloading images delays the start of animation, and available memory limits the length of animations.

We used 200x200 pixel image shown in Figure 1 for timing by repeatedly displaying it.

Figure 1. An Image for Animation Tests

The GIF file for this image is about 25,000 bytes, while the corresponding image string is about 40,000 bytes.

Here are the times in milliseconds required to display a frame on a 233 Mhz DEC-Alpha. Asterisks indicate animations with images preloaded.

DrawImage()	44.7
DrawImage()	28.9*
ReadImage()	110.1
CopyArea()	1.4*

Here's how an animation might done using a CopyArea() and preloading images:

```
link graphics

$define Pause 300

procedure main(args)
   local windows, image, win

   windows := []

   every image := !args do
      put(windows, WOpen("image=" || image,
         "canvas=hidden")) |
            stop("*** cannot open ", image)

   if *windows = 0 then stop("*** no images")

   win := WOpen("width=" ||
      WAttrib(windows[1], "width") || "," ||
         WAttrib(windows[1], "height")) |
            stop("*** cannot open animation window")

   every CopyArea(!windows, win) do
      WDelay(Pause)

   WDone()

end
```

The names of the image files are given on the command line. The program assumes the frames are all of the same size.

For "long" animations, it's not feasible to preload images—for an animation only one minute long, 900 frames are needed at 15 fps, which requires more memory for windows than most com-

Supplementary Material

Supplementary material for 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/ia55/

puters have. Even if enough memory were available, the time to preload 900 200×200 pixel images would be about 100 seconds on the platform for which the timing figures were measured.

Note also that for a 1-minute animation, the GIF files amount to approximately 23 MB.

Next Time

We'll conclude the series of articles on animation with one on how to make "movies" — packaged animations.

Reference

1. *Macromedia Animation Studio*, Gary Chapman, Random House/New Media, 1995.

2. "The Kaleidoscope" Icon Analyst 38, pp. 9-13.

3. "The Kaleidoscope" Icon Analyst 39, p. 5-10.

Operations on Sequences

There are many ways of creating sequences by performing operations on other sequences. In the last article on sequences [1], we used several of these operations, including filtering and applying operations to successive values.

Some operations, such as interleaving the values of two or more sequences, cannot be done in Icon without using co-expressions. An expression to interleave the values produced by two expressions, *expr1* and *expr2*, is

```
(e1 := create expr1, e2 := create expr2, |@(e1 | e2))
```

(C usually is used to designate co-expressions; we'll use e here to avoid uppercase.) Since activation of a co-expression produces at most one value, results from e1 and e2 are produced in alternation. Repeated alternation keeps the process going. In this formulation, if one of the expressions runs out of values before the other, the remaining values of the other are produced from this point on.

This expression can, like all such expressions, be cast as a procedure:

```
procedure interleave(e1, e2)
   suspend |@(e1 | e2)
end
```

The caller must supply the co-expressions, as in

```
interleave(create seq() create, –seq())
```

Programmer-defined control operations (PDCOs) make co-expressions easier to use for such purposes [2,3]. When a procedure is invoked with surrounding braces instead of parentheses, a list of co-expressions for the arguments is supplied to the calling procedure. The procedure above then could be cast as

```
procedure interleave(L)
   suspend |@(L[1] | L[2])
end
```

and called as

```
interleave{seq(), –seq()}
```

which is equivalent to

```
interleave([create seq(), create –seq()])
```

A latent naming problem was flushed by the introduction of weaving and pattern-form procedures [4], in which initial uppercase is used to distinguish such procedures from procedures that operate on strings, as in Reverse() and reverse(). The same convention had been used for PDCOs, and Reverse() already existed as a PDCO. We wound up with two (and in one case, three) procedures in the Icon library with the same name. This is "okay" as long as you don't link modules containing more than one, but it is confusing and untenable in the long run (or the short run, as it turned out for us).

To solve this problem, we decided to append PDCO to the names of all PDCOs in the library. The resulting names are ugly and cumbersome, but at least they make it clear when PDCOs are used. You might argue that the pattern-form procedures, a late arrival in the library, should have borne the embarrassment of name defacement. We chose to do it for PDCOs because it serves as a visual reminder of the need for braces for invocation. Consequently, a PDCO for interleaving the values of two sequences now is InterleavePDCO{}.

This PDCO, also known as ParallelPDCO{}, can be generalized to interleave values from an arbitrary number of sequences:

```
procedure InterleavePDCO(L)
   suspend |@!L
end
```

For example,

```
InterleavePDCO{seq(), seq() ^ 2, seq() ^ 3}
```

produces a sequence that interleaves the integers, their squares, and their cubes: 1, 1, 1, 2, 4, 8, 3, 9, 27, … .

Writing PDCOs

Many PDCOs are straightforward once you understand how co-expressions work. Some of the ones in the Icon program library appear arcane as a result of compact or clever coding. For example

```
suspend |@!L
```

can be cast more clearly, at the expense of brevity, as

```
sw := 1                  # start assuming at least 1
while sw > 0 do {
  sw := 0                # none yet
  every e := !L[1] do
    if x := @e then {
      sw +:= 1           # note result
      suspend e
      }
  }
```

What we typically do is write a long, clear form that works and then recast it in a more concise form.

The important points to remember about co-expressions are that

- @e produces one value from the generator for e except when there are no more values, in which case it fails.
- ^e produces a "refreshed" *copy* of e. In the absence of side effects, this copy starts with the first value for the generator. Since it produces a copy of e and does not change e, the usual usage is e := ^e.
- |@e produces all the values from the generator for e, stopping only when there are no more.

Precautions

When dealing with "infinite" generators, like seq(), whose sequences are endless, there are hazards. Some of the hazards are obvious, others subtle. The hazards are particularly serious when using PDCOs, which are designed to operate on generators provided by the caller.

An obvious example is ReversePDCO{}, which reverses the order of a sequence:

```
procedure ReversePDCO(L)
  local result
  result := []
  while push(result, @L[1])
  suspend !result
end
```

It's clear that if *expr* is an infinite generator, ReversePDCO{*expr*} never produces a result and eventually terminates with an error for lack of storage. A more subtle point is that if *expr* is a finite generator but produces a large number of values, the same thing may happen. Even when ReversePDCO{} successfully produces the reversal of a sequence, there may be a considerable time lag before the first value is produced. This may be disconcerting to the user and mistakenly interpreted as an endless loop — there's no way to tell.

Nevertheless, PDCOs with this kind of problem are useful in many situations. One problem is that the concept of reversing an infinite sequence is little more troublesome than the concept of infinity as a number. It's this kind of problem that has led mathematics into ongoing problems with infinite quantities, especially in set theory [5].

There are other pitfalls in operations on sequences that are less obvious. For example,

```
seq() < 5
```

generates 1, 2, 3, 4 and then continues running without ever terminating. The problem here is not that the concept of "values less than 5" is questionable; it's the consequence of its application to an infinite generator for which only a few values satisfy the condition. Note that there is no run-time error in such a case; just endless computation with no value produced.

Of course, there are many other forms of this problem. An example is IncreasingPDCO{}, which filters out values of an integer sequence that are less than or equal to preceding values:

```
procedure IncreasingPDCO(L)
  local last, current
  last := @L[1] | fail
  suspend last
  while current := @L[1] do {
    if current <= last then next
```

```
      else {
        suspend current
        last := current
        }
      }
end
```

This PDCO works fine for many integer sequences, like the Fibonacci numbers, but is a "black hole" for, say, –seq().

Another example is RotatePDCO{} which rotates a sequence a specified number of values:

```
link lists
procedure RotatePDCO(L)
  local result, i
  i := @L[2]) | fail
  result := []
  while put(result, @L[1])
  suspend !|rotate(result, i)
end
```

As written, this procedure has the same problem that ReversePDCO{} has. However, for left rotation, indicated by a negative rotation value, it can be written so that it works for infinite sequences, since, in that case the rotated values are never reached:

```
link lists
procedure RotatePDCO(L)
  local result, i
  i := @L[2] | fail
  result := []
  if i <= 0 then {
    every 1 to i do                 # save first values
      put(result, @L[1])
    suspend |@L[1]                  # generate the rest
    suspend !result                 # ones rotated out
    }
  else {
    while put(result, @L[1])
    suspend !|rotate(result, i)     # rotate whole list
    }
end
```

Of course, for right rotation, the procedure never produces a value. The subtlety here is that if the rotation is computed, a user may not sense the hazard.

The usual way to prevent problems that can arise with infinite generators is to use limitation, as in

```
(expr \ 1000) < 5
```

Note that

```
(expr < 5 ) \ 1000
```

does not solve the problem. Note that it is not appropriate to put a limit in the code for PDCOs that have potential termination problems: That would modify the sequence being operated on and potentially produce invalid results.

Generality

Some PDCOs have arguments that seem most naturally cast as constants, not generators. UnopPDCO{}, which applies a unary operator to a to a sequence, is an example:

```
procedure UnopPDCO(L)
  local op, x
  op := proc(@L[1], 1)              # unary interpretation
  while x := @L[2] do
    suspend op(x)
end
```

For example,

```
UnopPDCO{"*", seq() ^ 3}
```

generates the sizes of the cubes: 1, 1, 2, 2, 3, 3, 3, 3, … .

The expression

```
while x := @L[2] do
    suspend op(x)
```

is used instead of

```
suspend op(|@L[2])
```

to distinguish between the end of a sequence and an operation that fails. This procedure does not check for possible failure of

```
proc(@L[1], 1)
```

What would be appropriate in such a case?

As we've said in previous Analyst articles, when programming in Icon, *think generators*. This suggests that the operator argument could itself be a sequence. If this were done, things like this would be possible:

```
UnopPDCO{|("+" | "–"), seq()}
```

which produces the integers with alternating signs:

1, –2, 3, –4, 5, –6, … .

In order to accomplish this, the generated values for the first argument need to be used:

```
procedure UnopPDCO(L)
   local op, x

   repeat {
      op := @L[1]          # may fail, not changing op
      op := proc(op, 1)
      x := @L[2]  | fail
      suspend op(x)
      }

end
```

The way this procedure is written, if the first argument runs out of values before the second, the last value for the first argument is used for the remainder of the operation. This approach has the nice property that the first argument need not be a generator at all, and UnopPDCO{"∗", *expr*} works as before.

Examples

One frequently used operation on integer sequences is taking the differences of successive terms. That's easy enough to do. Here's a PDCO:

```
procedure DeltaPDCO(L)
   local i

   repeat {
      i := @L[1] – @L[1] | fail
      suspend i
      }

end
```

In some kinds of weaving, the shaft and treadling numbers must alternate between odd and even. When such drafts are derived from other drafts, it may be necessary to insert "incidentals" to get the necessary even-odd alternation [6].

Here's a PDCO that does that. The inserted number is one greater than the first member of a pair that must be separated:

```
procedure OddEvenPDCO(L)
   local val, val_old

   while val := @L[1] do {
      if (val % 2) = (\val_old % 2) then
         suspend val_old + 1
      suspend val
      val_old := val
      }
```

The nonnull test bypasses the first number.

We earlier showed a PDCO to apply unary operations to sequences. Here's the equivalent for binary operations:

```
procedure BinopPDCO(L)
   local op, x, y

   repeat {
      op := @L[1]
      op := proc(op, 2)
      (x := @L[2] & y := @L[3]) | fail
      suspend op(x, y)
      }

end
```

Here's a PDCO that replicates each value from the first sequence by the values produced by the second sequence:

```
procedure ReplPDCO(L)
   local x, i

   while x := @L[1] do {
      i := @L[2] | | fail
      suspend (1 to i) & x
      }

end
```

The expression

```
(1 to i) & x
```

is an Icon idiom for generating x i times [7].

References

1. "Generating Sequences", Icon Analyst 54, pp. 13-15.

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

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

4. "A Weaving Language", Icon Analyst 52, pp. 1-3.

5. *Mathematics: The Loss of Certainty*, Morris Kline, Oxford University Press, 1980.

6. "Commemorate with a Name Draft", Norma Smayda, *Shuttle, Spindle and Dyepot*, Vol. 91, Summer 1992, pp. 42-45.

7. "Idiomatic Programming", Icon Analyst 25, pp. 1-5.

Weave Structure

Everything in weaving is so simple after you understand it, but before you do, it seems so hard. — unknown weaving teacher

Technically a loom is any mechanism that creates a shed in which some warp threads are raised above others to allow a weft thread to pass through [1].

A loom facilitates the process of weaving. There are many kinds of looms, some dating from antiquity, and there are seemingly endless variations on each. In our first article on weaving [1], we used a typical floor loom as a model.

It is easy to focus on a particular kind of loom, the way it works, how weaves for it are designed and specified, but lose track of the larger picture.

Weave structure is defined as the particular interlacement of warp and weft threads. This concept is independent of any loom.

The structure of a weave is expressed in a drawdown [1], which shows the interlacing as a grid. In the interlacing, a black square indicates the warp thread is on top of the weft thread at that point, while a white square indicates the weft thread is on top. See Figure 1.

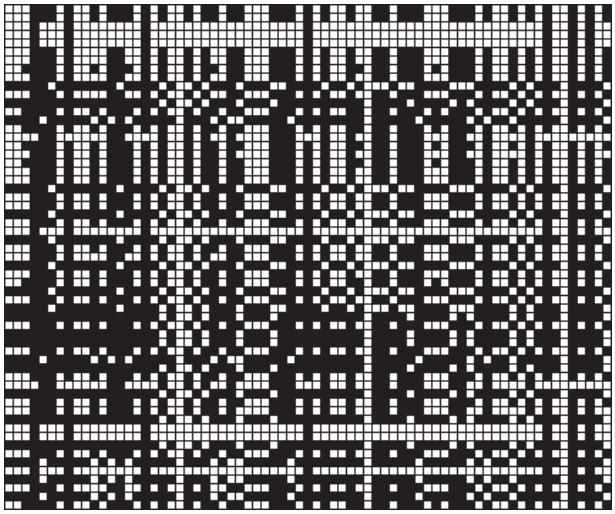


Figure 1. A Drawdown

How this structure might be woven depends on the loom used. In fact, it may or may not be possible, depending on the capabilities of the loom.

Looms both enable and constrain. For example, a floor loom allows a weave structure to be set up in a systematic manner using shafts and treadles. The weave structures a floor loom can weave are limited by the number of shafts and treadles the loom has.

Any structure can be woven on a floor loom that has enough shafts and treadles. The problem is that a particular structure may require more shafts and treadles than the loom has. There are (large and expensive) looms with many shafts and treadles, but there is a practical limit. In the worst case, a weave structure may require as many shafts as there are warp threads and as many treadles as there are weft threads. So a weave structure with 100 warp and 100 weft threads in the worst case would require a floor loom with 100 shafts and 100 treadles. No such animal exists, as far as we know.

There are other kinds of looms, such as drawlooms [2] and Jacquard looms [3] <1>, that do not have these limitations. The illustration on the last page of the last issue of the Analyst is of an old Chinese drawloom. The image on the last page of this issue is of an industrial Jacquard loom. We will have more to say about such looms in a subsequent article.

The article **Dobby Looms and Liftplans** that starts on page 17 describes a method of extending the capabilities of floor looms.

Next Time

In a subsequent article on weaving, we will show how to produce floor-loom drafts from drawdowns in a way that minimizes the number of shafts and treadles needed. At this point, we will be able to show algorithms and programs.

Also on the agenda is the more difficult issue of color: Creating drafts from color interlacement diagrams and determining if it is even possible to find an interlacement for a particular colored pattern.

References

1. "A Weaving Language" Icon Analyst 51, pp. 5-11.

2. *The Book of Looms*, Eric Broudy, Brown University Press, 1979, pp. 124-134.

3. *The Book of Looms*, pp. 134-137.

Link

1. http://www.cs.arizona.edu/patterns/weaving/weavedocs.html

Answers to Quiz on Sequences

See Icon Analyst 54 (page 16) for the quiz.

In the answers for problems 1 though 7, we've put some of complicated expressions on separate lines to improve readability. Note that they all are mutual evaluation expressions.

1. (i := seq(), if i == reverse(i) then i)

2.
```
(
  k:= &null, m := 0, i := seq(),
  if /k then k := i else 1,
  if k ~= i then
    (
    m +:= 1,
    if m % 2 = 1 then i else 1(.k, k := i)
    )
  )
```

Comment: This solution seems unnecessarily involuted and complicated, but we couldn't find a better one. See the quiz on the next page, where we pose this problem again for solution using a PDCO.

3. (i := seq(), 1 to i)

4. (i := seq(), (1 to i) | (i – 1 to 1 by –1))

5. (i := seq(), j := 0, (every j +:= !i) | j)

6.
```
(
    i := seq(),
    repeat {
      j := 0
      every j +:= !i
      if *j = 1 then break j
      i := j
      }
    )
```

7.

(a) 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6 …

(b) 1, 1, 4, 1, 4, 9, 1, 4, 9, 16, 1, 4, 9, 16, 25, 1, 4, 9, 16, 25, 36, 1, 4, 9, 16, 25, 36, 49, 1, 4, …

(c) 1, 1, 2, 3, 4, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, …

(d) 1, 1, 4, 9, 16, 1, 4, 9, 16, 25, 36, 49, 64, 81, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, …

(e) 1, 1, 1, 1, 1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 1, 2, 1, 2, 3, 1, 1, 2, 1, 2, 3, 1, 1, 1, 2, 1, 2, 3, …

(f) 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, …

(g) 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, …

(h) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, …

Note: The second generator in the alternation posed is a red herring; it never gets evaluated because the first generator produces an endless sequence.

(i) –1, 1, –1, 1, –1, 1, –1, 1, –1, 1, –1, 1, –1, 1, …

(j) There was an extra parenthesis in the expression given. It should have been (5 * seq()) % 9, for which the sequence is

5, 1, 6, 2, 7, 3, 8, 4, 0, 5, 1, 6, 2, 7, 3, 8, 4, 0, 5, 1, 6, 2, 7, 3, 8, 4, 0, 5, 1, 6, 2, 7, 3, 8, 4, 0, …

8.

(a) !seq()

(b) !seq() % 7

(c) seq() \ seq(1,2)

(d) (i := seq(), i ^ i)

(e) We accidentally deleted a digit in the fourth term for the sequence given. The sequence should have been

3, 81, 19683, 43046721, 847288609443, 150094635296999121, …

for which a solution is 3 ^ (seq() ^ 2).

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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

Quiz — Programmer-Defined Control Operations

1. Write PDCOs as follows:

(a) ExchangePDCO{}, which exchanges the order of successive terms in a sequence. For example, ExhangePDCO{seq()} should produce

2, 1, 4, 3, 6, 5, …

(b) CumulativePDCO{}, which produces the cumulative sum of a sequence of integers. For example, CumulativePDCO{seq()} should produce

1, 3, 6, 10, 15, …

(c) NonintegerPDCO{}, which filters out non-integer values in a sequence.

(d) ModnPDCO{}, which produces the remainder of each term in an integer sequence divided by its position. For example,

```
ModnPDCO{
  InterleavePDCO{seq(), seq() ^ 2, seq() ^ 3}
  }
```

should produce

0, 1, 1, 2, 4, 2, 3, 1, 0, …

(See page 11.)

Which of the PDCOs above have potential problems with infinite sequences?

2. Show the sequences these expressions produce and describe them in words (see pages 6 and 10-13 for a description of the procedures used).

(a) BinopPDCO{"+", seq(), fibseq()}

(b) BinopPDCO{|!"+–", primeseq(), fibseq()}

(c) ReplPDCO{seq(), seq()}

(d) DeltaPDCO{primeseq()}

(e) OddEvenPDCO{fibseq()}

(f) InterleavePDCO{fibseq(), primeseq()} % 8

(g) InterleavePDCO{fibseq(), primeseq() % 8}

(h) UnopPDCO{"*", fibseq()}

(i) UnopPDCO{"!", seq()}

3. Figure out these sequences, describe them in words, and write expressions that produce them. All can be produced by the procedures described on pages 6 and 10-13.

(a) 1, 1, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, …

(b) 1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 9, 9, 9, 9, 10, 10, 10, 10, 11, 11, …

(c) 2, 5, 3, 6, 5, 8, 7, 10, 11, 14, 13, 16, 17, 20, 19, 22, 23, 26, 29, 32, 31, 34, 37, 40, 41, 44, …

(d) 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 17, 18, 19, 20, 23, 24, 29, 30, 31, 32, 37, 38, 41, 42, 43, 44, …

4. Describe in words what these PDCOs do.

(a)

```
procedure Puzzle1PDCO(L)
  local x

  while x := @?L do suspend x

end
```

(b)

```
procedure Puzzle2PDCO(L)
  local i

  suspend @L[1]

  repeat {
    i := @L[2] | fail
    every 1 to i do
      @L[1] | fail
    suspend @L[1]
    }

end
```

(b)

```
procedure Puzzle3PDCO(L)
  local i, j

  j := @L[1] | fail

  while i := @L[1] do {
    if i > j then suspend j to i – 1
    else if i < j then suspend j to i + 1 by –1
    else suspend j
    j := i
    }

  suspend j

end
```

Dobby Looms and Liftplans

Keep in mind that a loom like a large dog is more afraid of you than you are of it. — Allen Fannin

The kind of loom we've used as a model in previous articles on weaving [1, 2] has foot powered treadles that raise shafts to make sheds through which successive weft threads pass (*picks*). A treadle can be tied up to several shafts, but only one treadle is pressed for each pick. Some looms prevent more than one treadle from being pressed, but even for looms that do not, since human beings only have two feet, it's not practical to press more than two treadles at a time. As far as we know, it's not done.

The reason that multiple treadling is useful is that it allows more complex weaves to be produced by providing a greater variety of sheds with a given number of shafts and treadles.

Dobbies

The solution to the multiple-treadling problem is a *dobby*, which allows any combination of shafts to be raised using only a single treadle. Dobbies are mounted atop conventional floor looms. They are controlled by a mechanism that originally rotated a belt containing rows of pegs that determined which shafts were lifted. The "peg plan" was set up in advance for a particular weave. One loop around the belt produced one weft repeat.

Modern dobbies have electrical controllers to determine what shafts to raise (although the weaver provides the foot power to raise the shafts). Most dobby devices now are driven by weaving programs running on personal computers.

Figure 1 shows a loom equipped with a simple dobby but without the related paraphernalia. The wires from the dobby are connected to the shafts, and different patterns of wires are raised to produce different sheds.

Figures 2 and 3 show a loom with a more sophisticated dobby and its controlling devices.

Back Issues

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

Figure 1. Schact Floor Loom with Dobby

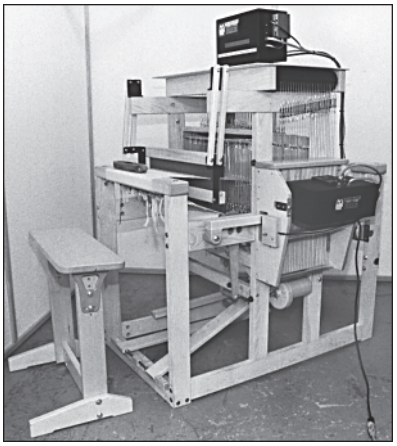


Figure 2. AVL Compu-Dobby Loom

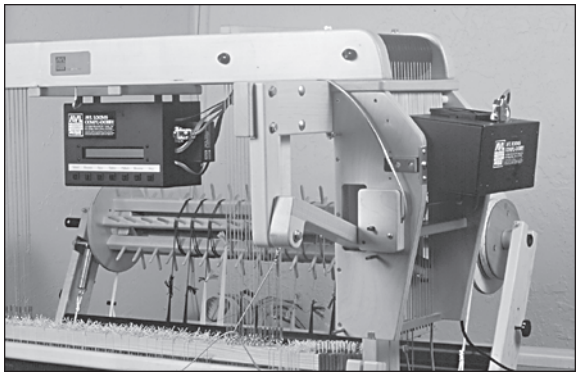


Figure 3. Close-Up of Compu-Dobby

Liftplans

The term *liftplan* now is more commonly used

than peg plan, although both terms still are used in current weaving literature. The term liftplan has the advantage of abstracting the concept from its original mechanical implementation.

In weaving drafts, a liftplan replaces the treadling, the difference being that a row of a liftplan (which corresponds to a pick) may specify several treadles as opposed to treadling plan, in which only one treadle can be specified for a row.

The tie-up still exists as a remanent of the single-treadle loom, but it always is a *direct tie-up* in which treadle 1 is tied to shaft 1, treadle 2 to shaft 2, and so on. Figure 4 shows a draft with a liftplan.

Representing Liftplans in Programs

Treadling plans can be represented by a sequence of numbers that label the treadles. Liftplans require a more capable representation.

In the widely used WIF format [2] for exchanging drafts among weavers with different weaving programs, a liftplan section consists of a line for each pick. Each line begins with an integer that identifies the corresponding weft thread. Following the number and an equal sign, there is a comma-separated list of the shafts that are to be raised for the pick. Although the WIF format allows weft threads to be given in any order, all the ones we've seen give them in the order the picks are actually made, and we can see no good reason for them to be given in any other order. Here is a typical WIF liftplan section.

```
[LIFTPLAN]
1=1,3,5,7
2=1,2,5
```

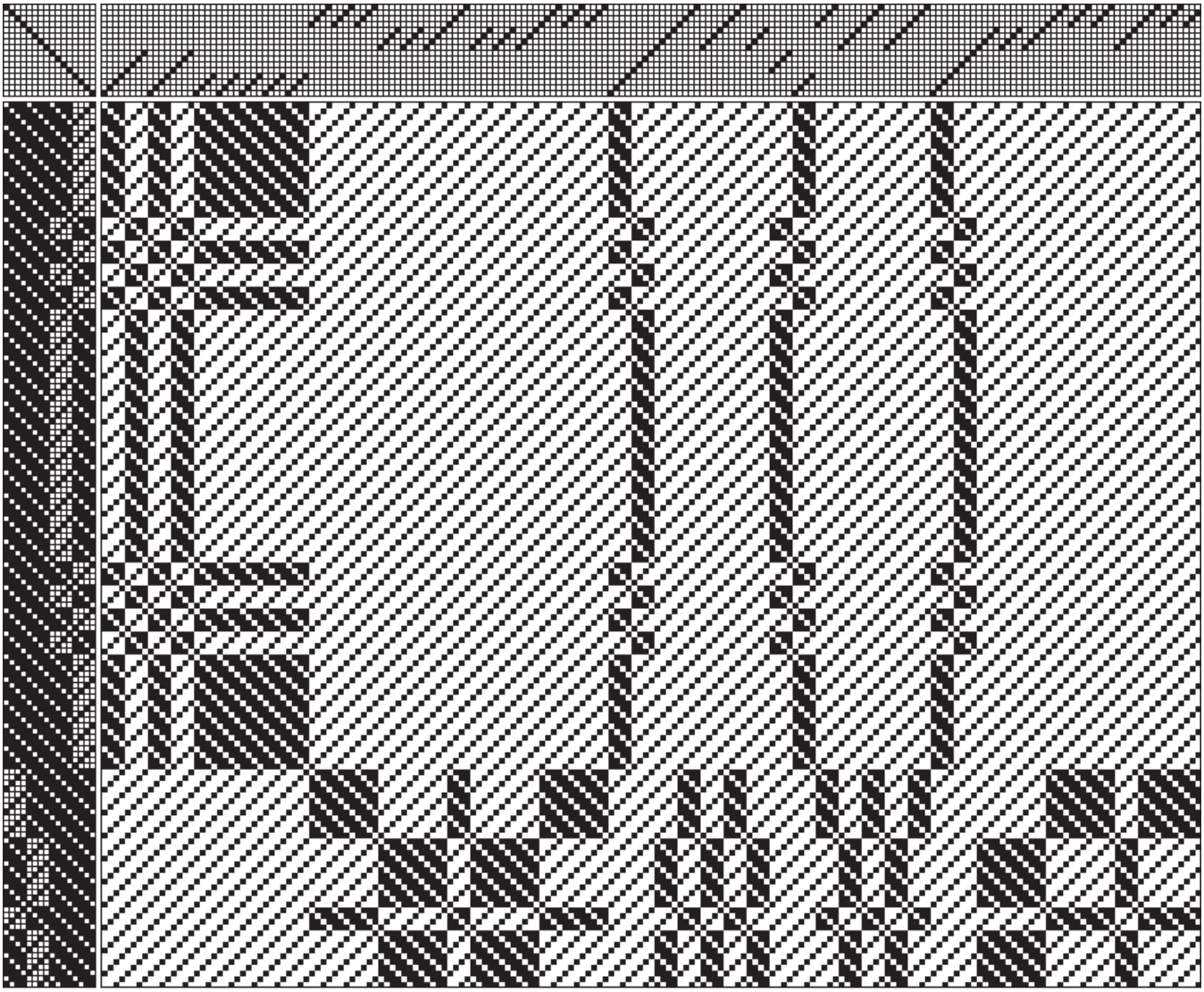


Figure 4. Liftplan Draft and Drawdown

```
3=2,4,6,8
4=3,4,7
5=1,3,5,7
6=3,4,7
7=2,4,6,8
8=1,5,6
9=1,3,5,7
10=1,5,6
   …
291=2,4,6,8
292=3,4,7
293=1,3,5,7
294=3,4,7
295=2,4,6,8
296=3,4,7
297=1,3,5,7
298=3,4,7
299=2,4,6,8
300=3,4,7
```

This notation is egregiously cumbersome and verbose. (The source of this notation is the Windows INI file format. Many have lamented the fact that file formats often are created by persons who have no experience with the art and seem to know none of the principles for creating good formats.)

A liftplan is, in effect, a binary matrix, as is the tie-up for single treadling. Since tie-ups are comparatively small, we settled for representing them in pattern-form drafts [2] as bit strings obtained by concatenating the bit patterns for successive rows. This representation is impractical for liftplans, which may be quite long.

The number of possible row patterns in a liftplan is 2^n, where n is the number of treadles. In practice many row patterns cannot be used because they would produce structurally unsound fabrics. More significant is that, in practice, the number of different row patterns in a liftplan for a specific weave usually is quite small — the sequence of the row patterns is fundamental to patterns in the resulting weave.

One possibility is to assign a label to each different row and represent the liftplan as a character pattern composed of these labels [3]. Another possibility is to think of the liftplan as a black-and-white image and represent it by a bi-level image pattern [4].

We'll explore these possibilities in a subsequent article, but before ending this article, we want to mention a (theoretical) possibility in the spirit of liftplans.

Shaftplans

If multiple treadles can be used in the treadling, why not allow warp threads to be passed through more than one shaft, so that the same warp thread can be raised by more than one shaft — a "shaftplan"?

As far as we can determine, no loom provides for this. Warping, which is done as a preparatory step before actually weaving, is accomplished by raising, for each warp thread, a single shaft through which the thread passes. It's not mechanically possible, as far as we know, to raise more than one shaft during warping. Weavers also say that warping a loom before the actual weaving is the hardest and most onerous part of producing woven fabric. Making warping any more complicated probably would not find a receptive audience, whatever its advantages might be.

For a computer program, however, handling a shaftplan is no more difficult than handling a liftplan. In fact, SwiftWeave <1> provides such a facility, commenting that although no loom known to them supports such a feature, the ability to have multiple warp threadings may help in design.

We expect to come back to this idea in a future article.

Acknowledgments

Figure 1 is from a catalog issued by the Schacht Spindle Company <2> and is used by permission. Figures 3 and 4 are from the AVL Looms Web site

<3> and are used by permission.

References

1. "A Weaving Language" Icon Analyst 51, pp. 5-11.

2. "Weaving Drafts" Icon Analyst 53, pp. 1-4.

3. "Character Patterns" Icon Analyst 48, pp. 5-7.

4. *Graphics Programming in Icon*, Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend , Peer-to-Peer Communications, Inc., 1998, pp. 157-160.

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

© 1999 by Ralph E. Griswold, Madge T. Griswold, and Gregg M. Townsend

All rights reserved.

Links

1. http://www.swiftweave.com/

2. http://www.schachtspindle.com

3. http://www.avlusa.com/Looms/AVLLooms.html

What's Coming Up

> In the long run every program becomes rococo — then rubble.
>
> — Alan Perlis

We don't have a particularly good record at predicting what will appear in the next issue of the Analyst, so please take comments made in this section as expectations, not promises.

We had planned an article on shadow-weave wallpaper for this issue, but we had to postpone it because we've changed the form of pattern form drafts. Articles on both of these topics should appear in the next issue.

We didn't have enough space in this issue to describe the implementation of tile exploration. Even the most essential parts of the program run many pages. You can find the complete program on the Web site for the last issue of the Analyst.

In the series on weaving, in addition to the articles mentioned above, we expect to have articles on creating drafts from drawdowns and how to create woven images from drafts.

We'll continue the series on sequences with an application with a visual interface that creates sequences from Icon expressions. And we'll review available PDCO resources in **From the Library**, and continue with the quizzes.