# The Icon Analyst

*In-Depth Coverage of the Icon Programming Language*

**October 1999**
**Number 56**

### In this issue …



## Weave Draft Representation

### Pattern-Form Drafts Revisited

We designed pattern-form drafts (PFDs) so that patterns in threading and treadling sequences could be preserved [1]. The file format we chose was designed to be compact and to be processed by programs. Each line contains one component of the draft as shown in Figure 1. Notice that the number of shafts and number of treadles are encoded in the tie-up.

Since information is positional and not self-identifying, it is not well suited for manual editing, although that's possible: It's ASCII text with few enough lines that individual components can be identified.

Pattern-form drafts have been central to our work on weave structure. It's important that PFDs can represent all the information we need and do that in a convenient way.

The original format did not handle all aspects of weave structures that we subsequently found to be important, such as liftplans.

As we mentioned in the article about Dobby looms and liftplans [2], liftplans often are large compared to tie-ups. Although using a bit-string representation for liftplans is possible, it's awkward and impractical. And while we once hoped to deal with patterns in tie-ups in a way similar to the what we did with threading and treadling sequences, the kinds of patterns they have require a different approach. This freed us from an immediate need to represent them in drafts in a manner that made their structure evident.

Both liftplans and tie-ups are binary matrices. We therefore decided to us Icon's bi-level pattern format for conciseness (see the **Graphics Corner** article that starts on page 4). An important consideration in making this choice was the existence of several programs and procedures in the Icon program library for creating and manipulation bi-level patterns. The interactive pattern manipulator is described in *Graphics Programming in Icon* [3] is particularly useful.

```
aquadesign                                                                  name
[[1>8]*5][[7<1]8*4]765432[[1>8]*5][[7<1]8*4][7<1]                           threading
[[1>8]*10]                                                                  treadling
[G*20][H*39][G*20][G*19][H*39][G*20]                                        warp colors
[0–>80]                                                                     weft colors
c1                                                                          palette
8;8;1001001111000001111000000111000000111001100101000100101000100101      tie-up
```

**Figure 1. A Pattern-Form Draft**

It's worth noting that bi-level pattern strings, while compact, are not easy to decipher without the aid of a program. For example, consider the tie-up grid diagram in Figure 2.

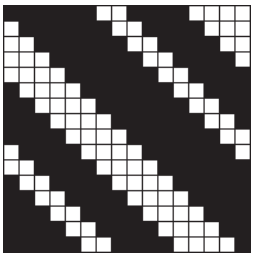


**Figure 2. A Tie-Up Grid**

The corresponding bi-level pattern is

```
16,#0f3f1e7e3cfc79f8f3f0e7e1cfc39f873f0f7
e1efc3cf879f0f3e1e7c3cf879f
```

This string is 68 characters long compared to the 262 characters the bit-string representation would require.

While we were revamping the PFD format, we decided that we needed more generality in dealing with warp and weft color sequences. Originally they were strings of palette keys. In the new PFD format, they are strings of characters that index a string of palette keys (the character encoding for indexes that are greater than 9 is the same as used in the threading and treadling sequences).

The new PFD format has 11 lines:

| | |
|---|---|
| *name* | text |
| *threading* | pattern form |
| *treadling* | pattern form |
| *warp colors* | pattern form |
| *weft colors* | pattern form |
| *palette* | palette name |
| *keys* | palette keys |
| *tie-up* | bi-level pattern |
| *shafts* | integer |
| *treadles* | integer |
| *liftplan* | bi-level pattern |

The lines for the number of shafts and treadles are included so that it's not necessary to extract them from the tie-up.

The liftplan may be empty. If it is present, the tie-up and treadling may be empty, although in WIFs [1] they usually are included in addition to a liftplan so the draft can be used on a loom without a dobby device.

## Problems with Pattern-Form Drafts

Pattern-form drafts have several limitations. The number of shafts and treadles is limited to the number of characters that are available for encoding integers. Although this is not a problem for real looms, some computer weaving programs are capable of dealing with 256 shafts and/or treadles.

The use of palettes instead of actual color values is more limiting than it might seem. Most drafts do not have a large number of colors (although some do). But built-in palettes provide no way for representing, say, 32 equally spaced shades of blue. A more serious practical problem is that some drafts specify combinations of subtly different hues. For these, no Icon color palette may be able to separate them and they may come out to be the same using PaletteKey().

## Internal Representation of Drafts

PFD is a file format. In order to manipulate a draft in a program in a reasonable way, it's necessary to convert it to an internal format that typically involves lists and arrays (lists of lists) [4].

It would be useful to be able to save an internal draft as-is and to be able to use it later, perhaps in a different program. The procedures xencode() and xdecode() in the Icon program library module xcode [5], make this easy to do.

In order to encode an entire internal draft structure as a single file, it's necessary to have all the components in one structure — a top-level structure that includes the rest. A record is the natural choice for this.

A record declaration for an internal structure draft (ISD) has 12 fields:

```
record isd(
   name,
   threading,
   treadling,
   warp_colors,
   weft_colors,
```

```
    color_list,
    shafts,
    treadles,
    width,
    height,
    tieup,
    liftplan
    )
```

The name field contains a string. The shafts, treadles, width, and height fields contain integers. The width and height fields are included so that the dimensions of the weave can be specified independently of the lengths of the threading and treadling lists. The sequences can be truncated or extended as needed [6].

The tieup and liftplan fields contain binary matrices. The other fields contain lists. The color list is a list of color values, which can be in any form that Icon supports (except mutable colors). The remaining lists are composed of numbers (not character codes representing numbers).

Given an ISD, it can be saved to a file by

```
xencode(draft, file)
```

and restored from a file by

```
xdecode(file)
```

The big disadvantage of ISDs is that they have no way of representing patterns (but we're working on that …). A minor disadvantage compared to PFDs is that they are larger — typically by a factor of 3 or 4, but ISDs are smaller than corresponding WIFs. In return for the increased size, ISDs provide ready-made internal structures, the capability for representing any number of shafts and treadles, and the capability for handling any color value that Icon can handle.

### References


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

2. "Dobby Looms and Liftplans", Icon Analyst 55, pp. 17-20.

3. *Graphics Programming in Icon*, Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend , Peer-to-Peer Communications, Inc., 1998, pp. 299-326.

4. "Arrays", Icon Analyst 14, pp. 2-4.

5. "From the Library", Icon Analyst 34, pp. 9-12.

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


## From the Library — Programmer-Defined Control Operations

> *A book ought to be like a man or a woman, with some individual character in it, though eccentric, yet its own; with some blood in its veins and speculation in its eyes and a way and a will of its own.* – John Mitchel

The collection of programmer-defined control operations in the Icon program library has grown quite large. Since the on-line version of the library is updated only infrequently, we've put the current version of this module, pdco.icn, on the Web site for this issue of the Analyst.

Some of the PDCOs in this module illustrate how various control structures can be modeled. Examples are:

| | |
|---|---|
| AltPDCO{*e1*, *e2*} | *e1* \| *e2* |
| EveryPDCO{*e1*, *e2*} | every *e1* do *e2* |
| GaltPDCO{*e1*, *e2*, … } | *e1* \| *e2* \| ... |
| GconjPDCO{*e1*, *e2*, … } | *e1* & *e2* & … |
| LimitPDCO{*e1*, *e2*} | *e1* \ *e2* |
| RepaltPDCO{*e*} | \|*e* |
| ResumePDCO{*e1*, *e2*, *e3*} | every *e2* \ *e2* do *e3* |

The main value of these PDCOs is pedagogical. By studying them, you can learn the details of Icon's control structures.

Other PDCOs of main interest in the library follow. The code for some is given in Reference 1.

BinopPDCO{*e1*, *e2*, *e3*} applies the binary operations from *e1* to values from *e2* and *e3*.

ComparePDCO{*e1*, *e2*} compares the sequences *e1* and *e2*. It succeeds if the sequences are the same but fails otherwise.

ComplintPDCO{*e*} produces the integers starting at 0 that are not in *e*. The sequence produced by *e* must be non-decreasing.

DeltaPDCO{*e*} produces the differences of successive integer values from *e*.

IncreasingPDCO{*e*} removes values from *e* as necessary to produce an increasing sequence.

IndexPDCO{*e1*, *e2*} selects values of *e1* in the positions produced by *e2*. The sequence produced by *e2* must be non-decreasing.

InterPDCO{*e1*, *e2*, …} interleaves values from *e1*, *e1*, … . *Note:* This procedure was named InterleavePDCO{} in Reference 1.

LengthPDCO{*e*} produces the length of (number of terms) in *e*.

OddEvenPDCO{*e*} inserts values into *e* to make odd-even sequence.

PalinPDCO{*e*} produces a palindrome sequence.

PatternPalinPDCO{*e*} produces a pattern palindrome sequence [2].

RandomPDCO{*e1*, *e2*, …} produces values from *e1*, *e2*, … selected at random.

ReducePDCO{*e1*, *e2*} "reduces" *e2* by applying the binary operation given by *e1* to the values from *e2*.

ReplPDCO{*e1*, *e2*} replicates each value from *e1* *e2* times.

ReversePDCO{*e*} produces the reversal of *e*.

RotatePDCO{*e*, *i*} rotates *e* by *i* terms. Positive *i* rotates to the left, negative *i* to the right.

SeqlistPDCO{*e*, *i*} returns the first *i* values of *e* in a list.

SkipPDCO{*e1*, *e2*} produces *e1*, skipping the number of terms given by *e2*.

TrinopPDCO{*e1*, *e2*, *e3*, *e4*} applies the trinary operations from *e1* to the values produced by *e2*, *e3*, and *e4*.

UniquePDCO{*e*} filters out duplicate values from *e*.

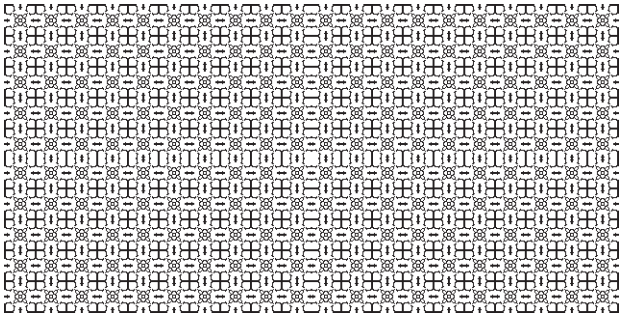
UnopPDCO{*e1*, *e2*} applies the unary operations from *e1* to *e2*.

See also the answers to the quiz on PDCOs on page 11.

### References

1. "Operations on Sequences", Icon Analyst 55, pp. 10-13.

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



## Graphics Corner—Bi-Level Patterns

We have discussed image strings in previous articles [1,2]. Such image strings are based on palettes and have a palette character (key) for every pixel in the image.

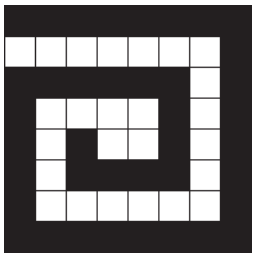
For bi-level (two-color) images, Icon supports a more compact representation. A bi-level image string is drawn in the current foreground and background colors. By default, these are black and white, respectively, but they can be any colors. Hence bi-level image strings are not equivalent to palette-based image strings with the g2 palette.

### Data Format

A bi-level image string, also called a pattern, has the form *width*,#*data*. Note that the # distinguishes bi-level image strings from palette-based image strings. The *data* portion contains a sequence of hexadecimal digits that specify rows from top to bottom. Each row is specified by *width* / 4 digits with fractional values rounded up.

The digits of each row are interpreted as hexa-

decimal numbers. Each bit of a hexadecimal digit corresponds to a pixel: 0 for background, 1 for foreground. The bits that form a hexadecimal digit are read from right to left. Figure 1 shows an example:



```
1111 1111   f f
0000 0001   8 0
1111 1101   b f
1000 0101   a 1
1010 0101   a 5
1011 1101   b d
1000 0001   8 1
1111 1111   f f
```

8,#ff80bfa1a5bd81ff

**Figure 1. A Bi-Level Pattern**

This ordering is confusing, but it's rarely necessary to construct or interpret a bi-level image string by hand: There are library programs for this [3].

## Built-In Patterns

A few patterns of a general nature are built into the Icon repertoire. These are shown in Figure 2.

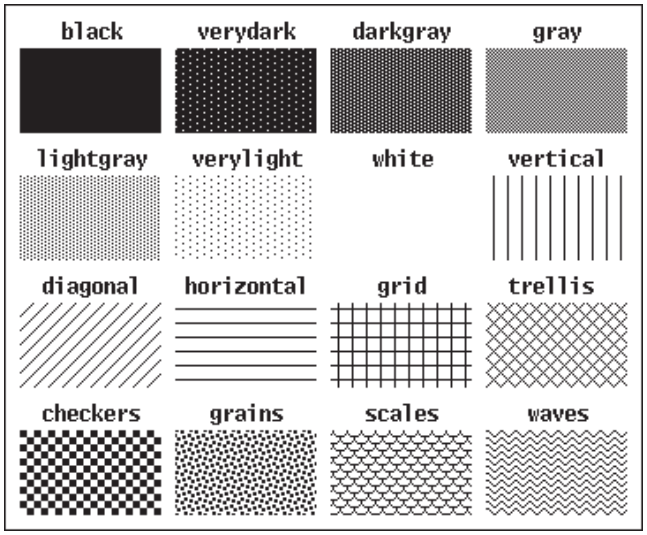



**Figure 2. Built-In Patterns**

## Using Patterns

A pattern is specified by the pattern attribute. This attribute can be set in several ways. For example, both

```
Pattern("8,#ff80bfa1a5bd81ff")
```

and

```
WAttrib("pattern=8,#ff80bfa1a5bd81ff")
```

set the pattern to the example given earlier. The pattern attribute also can be given in WOpen(), as in

```
WOpen("pattern=8,#ff80bfa1a5bd81ff", …)
```

The built-in patterns are specified by their string names, as in

```
Pattern("checkers")
```

The pattern is used for all drawing operations, with the details depending on the fill style. With "fillstyle=solid", the default, the pattern has no effect on drawing. With "fillstyle=textured", drawing is done with the foreground and background as specified by the pattern. With "fillstyle=masked", drawing is done with the foreground as specified by the pattern, but background pixels are left unchanged.

Patterns are aligned with the upper-left corner of the window and tile across it. You can imagine drawing with a fill style of "textured" or "masked" as exposing an underlying pattern.

The following program illustrates these features of patterns:

```
link graphics

procedure main()

   WOpen("size=600,300", "pattern=trellis") |
      stop("*** cannot open window")

   WAttrib("fillstyle=solid")        # (the default)

   FillRectangle()

   WriteImage("figure_3.gif")

   Pattern("trellis")

   WAttrib("fillstyle=textured")

   FillCircle(210, 150, 100)
   FillCircle(390, 150, 100)

   WriteImage("figure_4.gif")

   WAttrib("fillstyle=masked")

   Pattern("vertical")

   FillCircle(210, 150, 100)
   FillCircle(390, 150, 100)

   WriteImage("figure_5.gif")
```

```
    WAttrib("fillstyle=textured")

    FillCircle(210, 150, 100)
    FillCircle(390, 150, 100)

    WriteImage("figure_5.gif")

end
```

Here are the images produced by this program:

**Figure 3. Filled Rectangle with Solid Fill Style**

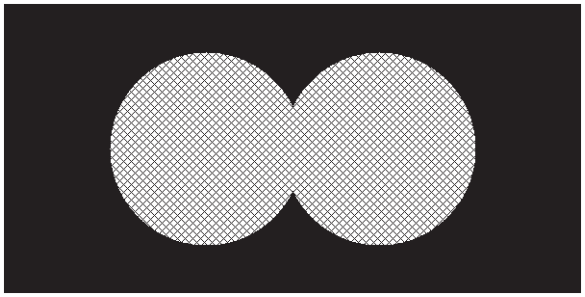


**Figure 4. Filled Circles with Textured Fill Style**

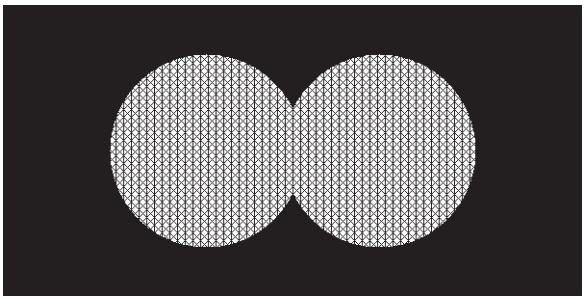


**Figure 5. Filled Circles with Masked Fill Style**

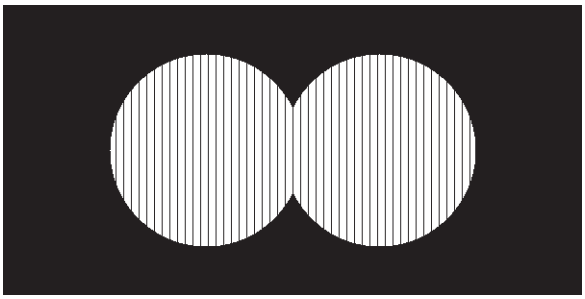


**Figure 6. Filled Circles with Textured Fill Style**

Figure 3 is solid black because the pattern has no effect with the solid fill style. Figure 4 shows the results of "punching out" two filled circles with the trellis pattern and textured fill style. Notice that the overlapping area tiles seamlessly.

In Figure 5, the filled circles are drawn again with the vertical pattern and masked fill style. Note that the background pixels left over from the trellis pattern are unchanged.

Finally, in Figure 6 the circles are filled with the vertical pattern again, but with the textured fill style. This wipes out the remains of the trellis pattern.

## Conclusion

Patterns, like many aspects of computer graphics invite unusual and creative uses. You'll find some examples in Reference 4.

## References


1. "Graphics Corner — Fun with Image Strings", Icon Analyst 50, pp. 11-13.

2. "Graphics Corner — More Fun with Image Strings", Icon Analyst 51, pp. 14-16.

3. *Graphics Programming in Icon*, Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend , Peer-to-Peer Communications, Inc., 1998, pp. 229-336.

4. *Graphics Programming in Icon*, pp. 158-160.


## Supplementary Material

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

http://www.cs.arizona.edu/icon/analyst/iasub/ia56/

## Exploring Sequences Interactively

Using Icon's built-in repertoire of generators and the procedures in the Icon program library, it's possible to produce an endless number of sequences of great variety.

These can be explored by writing individual programs, but that is tedious and time consuming.

The article describes an application that allows the user to enter and edit expressions that produce sequences and see the result quickly (or at least as quickly as the sequences can be computed).

### The Application

The interface for this application is shown in Figure 1.

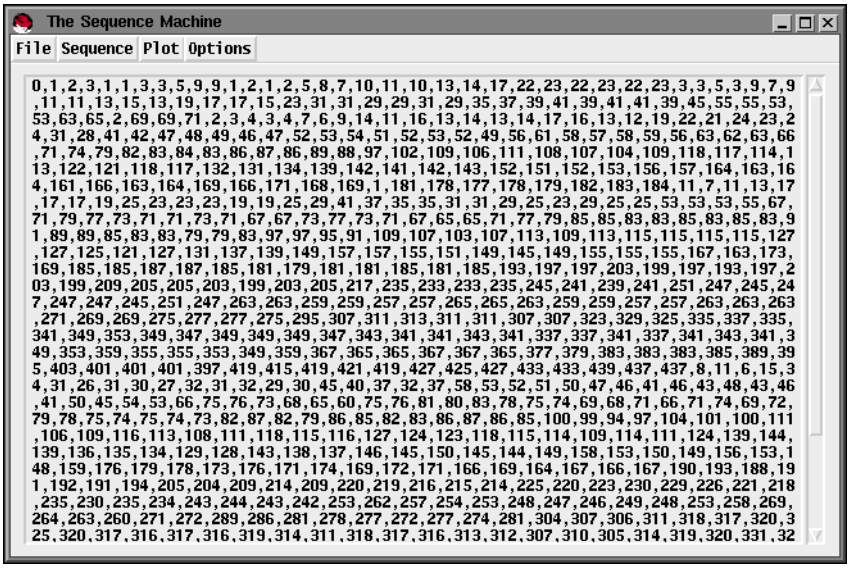



**Figure 1. The Application Interface**

The current sequence is shown in a scrolling text list that occupies most of the window.

Expressions are entered and edited in a dialog, which is shown in Figure 2.


**Figure 2. The Edit Dialog**

The File menu provides the usual items for saving the current expression and sequence, as well as for quitting the application.

The Sequence menu provides items for calling up the edit dialog and generating the sequence for the current expression.

The Options menu has items for limiting the number of terms produced and for specifying the separator between them.

The Plot menu provides items for presenting the current sequence visually. At present, only grid plots and point plots are supported. A grid plot is shown in Figure 3.

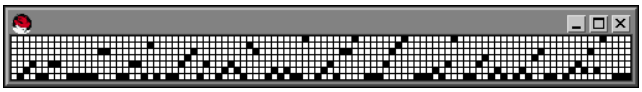


**Figure 3. A Grid Plot**

Since sequences may have very large values and many terms, grid plots and point plots have limited usefulness, and it's often not possible to show them visually in conventional ways.

We'll consider this problem and explore possible ways of visualizing sequences using unconventional techniques in a subsequent article.

### The Implementation

The implementation of this application is largely straightforward and uses techniques described in previous Analyst articles. We'll only describe a few procedures here.

The global variables used by these procedures are:

```
global current_exp      # current expression
global display          # text–list vidget
global limit            # limit on number of terms
global results          # list of current results
global separator        # separator for terms
```

The expression given in the edit dialog is incorporated in a program, which is written to a file and compiled using the system() function. The program is then run as a pipe so that the results can be read into a list. Note that error output is redirected to a file in the /tmp directory. This allows the cause of a problem to be displayed in case there is an error in compilation or execution.

```
procedure run()
   local input, output, k, signal, result
```

```
    static call

    initial
      call := "icont –s –u expr_.icn 2>/tmp/sequent.err"

    output := open("expr_.icn", "w") | {
      Notice("Cannot open file for expression.")
      fail
      }

    write(output, "link seqfncs")
    write(output)
    write(output, "procedure main()")
    write(output)
    write(output, "every write(", current_exp, ") \\", limit)
    write(output)
    write(output, "end")

    close(output)

    WAttrib("pointer=watch")

    if system(call) ~= 0 then {     # didn't compile
      remove("expr_.icn")
      WAttrib("pointer=arrow")
      show_error()
      fail
      }

    input := open("expr_ 2>/tmp/sequent.err", "p")

    results := []

    while result := read(input) do {
      result := numeric(result)
      put(results, result)
      }

    signal :=  close(input)

    remove("expr_.icn")             # remove debris
    remove("expr_")

    WAttrib("pointer=arrow")

    if signal ~= 0 then {           # run-time error
      show_error()
      fail
      }

    display_results()               # display results

    return

  end
```

Displaying the results takes a little work in order to ensure that lines are broken so that they will fit in the width of the text list.

```
  procedure display_results()
    local result_list, term, disp_list, line
    static line_width

    initial line_width := (display.aw – Fudge) /
      WAttrib("fwidth")             # save a little room

    result_list := []

    every put(result_list, image(!results) || separator)

    line := ""
    disp_list := []

    while term := get(result_list) do {
      if *line + *term > line_width then {
        if *line = 0 then {
          put(disp_list, term[1+:line_width])
```

## The Encyclopedia of Integer Sequences

The encyclopedia of integer sequences [1] contains a vast collection of integer sequences from a wide range of disciplines.

In it you can find all kinds of things, including sequences related to primes, Mersenne numbers, versum sequences, "self-organizing" sequences, sequences related to chess problems, continued fractions, and strange (to us) sequences like "Remoteness Numbers for Tribulations", and specialized mathematical sequences like "Unique Attractors for the Sliding Möbius Transform".

The book is well worth owning if you are interested in recreational mathematics, but there's a more accessible and extensive source on the Web <1>. With it you can look up sequences, give the terms of a sequence and find out if it's in the database, and submit new sequences.

You also can download the entire database <2>, which at this writing has 49 sections containing nearly 50,000 sequences.

To get an idea of the developing database, at the present time about 10,000 new sequences are being added each year.

### Reference

1. *The Encyclopedia of Integer Sequences*, N. J. A. Sloane and Simon Plouffe, Academic Press, 1995.

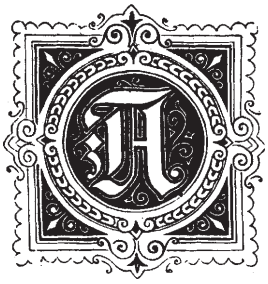
### Links

1. http://www.research.att.com/~njas/sequences/index.html

2. http://www.research.att.com/~njas/sequences/Seis.html

```
            line := term[line_width:0]
            }
          else {
            put(disp_list, line)
            line := term
            }
          }
        else line ||:= term
        }
    if *line > 0 then put(disp_list, line)
    VSetItems(display, disp_list)   # display sequence
    return
end
```

The complete program is on the Web site for this issue of the Analyst. Be aware, though, that the program still is under development and probably will have more features than shown here.



## Answers to Quiz on Programmer-Defined Control Operations

See Icon Analyst 55, page 16, for the questions.

1.

(a)

```
procedure ExchangePDCO(L)
   local i
   while i := @L[1] do
      suspend @L[1] | i
end
```

(b)

```
procedure CumulativePDCO(L)
   local i
   i := 0
   while i +:= @L[1] do
      suspend i
end
```

*Note:* This operation can be done by

```
ReducePDCO{"+", expr}
```

See page 4.

(c)

```
procedure IntegerPDCO(L)
   local x
   while x := @L[1] do
      if type(x) == "integer" then suspend x
end
```

*Note:* The problem was poorly phrased. A procedure that filters *out* non-integer values should be named IntegerPDCO{} as above, not NonintegerPDCO{}. Notice that the version given above only passes through values of type integer and does not attempt to convert values of other types. The code for the latter interpretation is

```
   while x := @L[1] do
      suspend integer(x)
```

(d)

```
procedure ModnPDCO(L)
   local i, j
   every i := seq() do {
      j :=  @L[1]  | fail
      suspend j % i
      }
end
```

*Note:* This operation can be done by

```
BinopPDCO{"%", e, seq()}
```

See page 3.

None of these PDCOs has a problem with infinite sequences, *per se*. However, IntegerPDCO{} stops producing values but doesn't terminate if an infinite sequence stops producing integer values.

2.

(a) Term-wise sum of the integers and the Fibonacci numbers: 2, 3, 5, 7, 10, 14, 20, 29, 43, 65, 100, 156, 246, 391, 625, 1003, …

(b) Alternating sums and differences of the primes and Fibonacci numbers: 3, 2, 7, 4, 16, 5, 30, -2, 57, -26, 120, -107, 274, -334, 657, -934, 1656, -2523, 4248, -6694, 11019, -17632, …

(c) The integers *i* repeated *i* times: 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, …

(d) Differences of successive primes: 1, 2, 2, 4, 2, 4, 2, 4, 6, 2, 6, 4, 2, 4, 6, 6, 2, 6, 4, 2, 6, 4, 6, 8, 4,

2, 4, 2, 4, 14, 4, 6, 2, 10, …

(e) The Fibonacci numbers with insertions, where necessary, to make terms alternate between odd and even. The inserted terms (every fourth term in this case) are underscored: 1, 2, 1, 2, 3, 4, 5, 8, 13, 14, 21, 34, 55, 56, 89, 144, 233, 234, 377, 610, 987, 988, 1597, 2584, 4181, 4182, 6765, 10946, 17711, 17712, …

(f) The Fibonacci numbers and the primes interleaved and then reduced modulo 8: 1, 2, 1, 3, 2, 5, 3, 7, 5, 3, 0, 5, 5, 1, 5, 3, 2, 7, 7, 5, 1, 7, 0, 5, 1, 1, 1, 3, 2, 7, …

(g) The Fibonacci numbers interleaved with the primes taken mod 8: 1, 2, 1, 3, 2, 5, 3, 7, 5, 3, 8, 5, 13, 1, 21, 3, 34, 7, 55, 5, 89, 7, 144, 5, 233, 1, 377, 3, 610, 7, …

(h) The sizes (numbers of digits) of the Fibonacci numbers: 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 8, 8, 8, 8, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 13, 13, 13, 13, …

(i) The individual digits of the Fibonacci numbers (since ! is a generator, each application runs through all the characters (digits) of the term to which it is applied): 1, 1, 2, 3, 5, 8, 1, 3, 2, 1, 3, 4, 5, 5, 8, 9, 1, 4, 4, 2, 3, 3, 3, 7, 7, 6, 1, 0, 9, 8, 7, 1, 5, 9, 7, 2, 5, 8, 4, 4, 1, 8, 1, 6, 7, 6, 5, 1, 0, 9, 4, 6, 1, 7, 7, 1, 1, 2, 8, 6, 5, 7, 4, 6, 3, 6, …

3.

(a) The integers *i* repeated *p* times by the corresponding primes *p*:

```
ReplPDCO{seq(), primeseq()}
```

(b) The integers *i* repeated by *i* repeated *i* times (as in solution 2(c)):

```
ReplPDCO{seq(), ReplPDCO{seq(), seq()}}
```

(c) The primes interleaved with the primes plus 3:

```
InterPDCO{primeseq(), primeseq() + 3}
```

(d) The primes made into an odd-even sequence:

```
OddEven{primeseq()}
```

4.

(a) Puzzle1PDCO{*e1*, *e2*, …, *en*} produces results from its argument expressions selected at random.

(b) Puzzle2PDCO{*e1*, *e2*} skips the number of terms in *e1* given by *e2*. For example,

```
Puzzle2PDCO{seq(), primeseq()}
```

produces

1, 4, 8, 14, 22, 34, 48, 66, 86, 110, 140, 172, 210, 252, 296, 344, 398, 458, 520, 588, 660, 734, 814, 898, 988, 1086, 1188, 1292, 1400, 1510, 1624, 1752, 1884, 2022, 2162, …

(c) Puzzle3PDCO{*e*} fills in *e* with runs of consecutive integers as necessary. For example,

```
Puzzle3PDCO{
  InterleavePDCO{primeseq(), seq()}
  }
```

produces

2, 1, 2, 3, 2, 3, 4, 5, 4, 3, 4, 5, 6, 7, 6, 5, 4, 5, 6, 7, 8, 9, 10, 11, 10, 9, 8, 7, 6, 5, 6, 7, 8, 9, 10, 11, 12, 13, 12, 11, 10, 9, 8, 7, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, …

## Woven Images

Producing images from weaving drafts is not particularly difficult, but it's well worth thinking about how to do it efficiently.

The naive approach is to examine each point of interlacing to determine whether the warp thread or the weft thread is on top and drawing the intersection in the appropriate color. We'll just draw a point, so that the threads are one-pixel in width.

### The Naive Approach

Using ISDs (see pages 2 and 3), the naive approach is:

```
every x := 1 to *draft.threading do {
  every y := 1 to *draft.treadling do {
    if draft.tieup[x, y] = 1 then
      Fg(draft.color_list[draft.warp_colors[x]])
    else
      Fg(draft.color_list[draft.weft_colors[y]])
    DrawPoint(x – 1, y – 1)
    }
  }
```

This method, requiring a separate computation for every pixel, is painfully slow because the complexity is $n \times m$ for an $n \times m$ image. For example, for a small image of $100 \times 100$ threads, there are $10^4$ iterations of the inner loop.

## Insight

One way to speed the process is to draw in all the warp threads as vertical stripes and then overlay the weft threads in those places where they are on top:

```
every x := 1 to *draft.threading do {
    Fg(draft.color_list[draft.warp_colors[x]])
    DrawLine(x – 1, 0, x – 1, *draft.treadling – 1)
    }

every x := 1 to *draft.threading do {
  every y := 1 to *draft.treadling do {
    if draft.tieup[x, y] = 0 then
      Fg(win, draft.color_list[draft.weft_colors[y]])
    DrawPoint(x – 1, y – 1)
    }
  }
```

Figure 1 shows what a typical woven image looks like after the warp background is drawn.

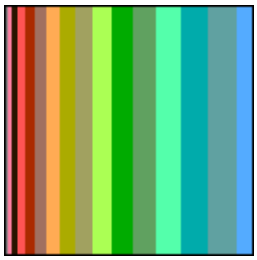


**Figure 1. Warp Background**

Drawing the warp background first saves a DrawPoint() for those intersections where the warp thread is on top, but the number of loop iterations is slightly larger — $10^4 + 100$ for a $100 \times 100$ image. There is a noticeable gain in speed, but the basic problem remains.

Incidentally, we could draw a weft background of horizontal stripes and overlay the warp. We chose to follow the order used in actual weaving, where the warp is set up in advance and the weft threads added during the weaving process. In addition, the performance wouldn't change much for narrow weaves — the warp interlacements would be longer in that case.

## More Insight

A significant improvement can be made by noting that there can be only as many different weft overlay patterns as there are treadles. These can be pre-computed and put in a list indexed by the treadle number. By representing the patterns as lists of points, all the weft pixels can be drawn in a single call of DrawPoint(). Here's code for the pre-computation of the treadle lists:

```
treadle_list := list(draft.treadles)

every !treadle_list := []

every i := 1 to draft.treadles do {
  every j := 1 to draft.shafts do {
    if draft.tieup[i, j] = 0 then {
      every k := 1 to *draft.threading do {
        if draft.threading[k] = j then
          put(treadle_list[i], k – 1, 0)
        }
      }
    }
  }
```

Note that the y coordinates are all set to 0; their actual values aren't known until the weft overlay is drawn. It's not necessary, however, to change them; the y coordinate can be set by using translation:

```
every y := 1 to *draft.treadling do {
  treadle := draft.treadling[y]
  Fg(draft.color_list[draft.weft_colors[y]])
  WAttrib("dy=" || (y – 1))
  DrawPoint ! treadle_list[treadle]
  }
```

This greatly improves the speed of drawing. Ignoring the pre-computation costs, which amount to an insignificant percentage of the total cost even for small images, the complexity drops from $n \times m$ to $m$ — from $10^4$ to 100 for our example.

We can further improve the performance by keeping track of the picks that use the same color. For each color, we can then set the foreground accordingly draw all the weft overlays for that color. Again, this adds to the complexity of the code:

```
                    …
treadle_colors := list(*draft.color_list)
every !treadle_colors := []

every i := 1 to *draft.threading do {
```

```
   j := draft.weft_colors[i]
   put(treadle_colors[j], i)
   }
every i := 1 to *treadle_colors do {
   Fg(win, draft.color_list[i]) | stop("bogob")
   every y := !treadle_colors[i] do {
      WAttrib(win, "dy=" || (y – 1))
      DrawPoint ! treadle_list[draft.threading[y]]
      }
   }
```

## Special Cases

It's worth adding code for special cases that arise frequently: when the warp and/or weft threads are the same color. The notable example of this is in drawdowns in which the warp threads are all black and the weft threads are all white. If the warp threads are all the same color, the background can be filled in with FillRectangle(). If the weft threads are all the same color, the foreground need be set only once. Here's the code for handling the case where all the warp threads are the same color:

```
if *set(draft.warp_colors) = 1 then {
      Fg(draft.color_list[draft.warp_colors[1]])
      FillRectangle()
      }
else …                    # general case
```

Note how easy it is to check for this case.

Here's what's needed for the case the weft threads are all the same color:

```
if *set(draft.weft_colors) = 1 then {
   Fg(draft.color_list[draft.weft_colors[1]])
   every y := 1 to *draft.treadling do {
      treadle := draft.treadling[y]
      WAttrib("dy=" || (y – 1))
      DrawPoint ! treadle_list[treadle]
      }
else …                    # general case
```

## Perhaps Too Much Cleverness

We toyed with the idea of drawing line segments for weft overlays so that several weft threads are that on top in succession could be done with one drawing operation. We decided the potential advantages were outweighed by the additional complexity that would be involved. In addition, most weaves have relatively few "floats" where a weft thread is on top of several warp threads in a row. We'll have more to say about floats in a future article.

There is another possibility for improving performance: Keep track of where the first weft overlay pattern is drawn in each weft color and when that pattern occurs again in the same color, use CopyArea() instead of DrawPoint() for that line.

Although CopyArea() is very fast, it's not clear that the gain would be enough to justify — or even offset — the extra testing that would be required (not to mention the more intricate code needed).

## Drawdowns

As mentioned earlier, a drawdown is obtained by using black for all warp threads and white for all weft threads. Consequently, the same code can be used for drawdowns as for regular woven images.

Drawdowns, however usually are shown on grids with squares several pixels on a side for each intersection rather than the single pixel we've used here. This can be handled easily enough using the methods given here. It's probably best to use a separate procedure for drawdowns rather than to further complicate the code used for ordinary woven images.

Incidentally, magnified images are easily obtained by using Zoom() from the graphics module of the Icon program library.

## Shadow-Weave Wallpaper

In an article on shadow weaves [1], we explored one of Painter's built-in drafts and showed the fascinating structure of its threading sequence, which is composed of a sequence of anchor points and palindromes connected by runs.

From there, we explored variations on the weave by making systematic modifications to the way the sequence was put together. We did not begin to explore all possible variations — the number of them is incomprehensibly vast. Yet some variations of a more radical nature produce interesting results. See Figure 1.

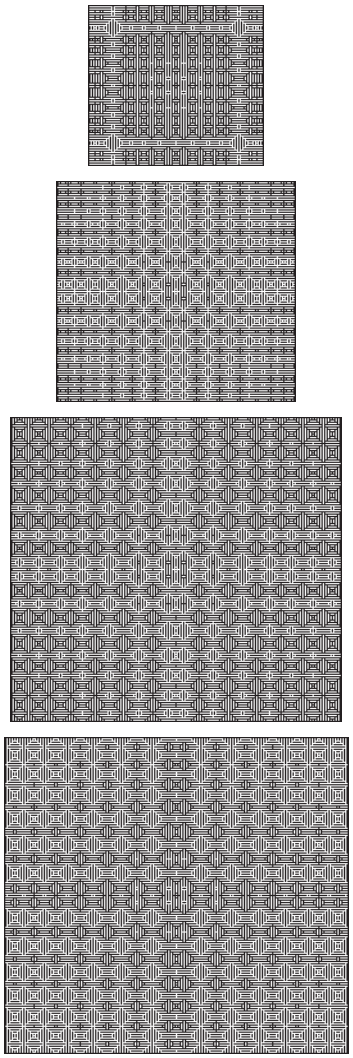


**Figure 1. Variations on a Shadow Weave**

One approach to further explorations would be to try to deduce the kinds of variations that might prove interesting. Another approach is to produce random changes in hopes of stumbling on interesting specimens.

We do not know enough to deduce interesting variations in a controlled fashion. And random variations are a lot easier to do.

Coincidental with our pondering this problem, a weaver who was interested in our work on shadow weaves asked if we could put up some shadow-weave "wallpaper" on the Web — a page with a shadow-weave background that changes periodically to show variations.

This was relatively easy to do. We created one Web page with two images, one of the original shadow weave and another with a variation that changes periodically. The image that changes periodically is linked to another page that is featureless except the image is used as a background.

Miniature versions of these pages are shown in Figures 2 and 3.

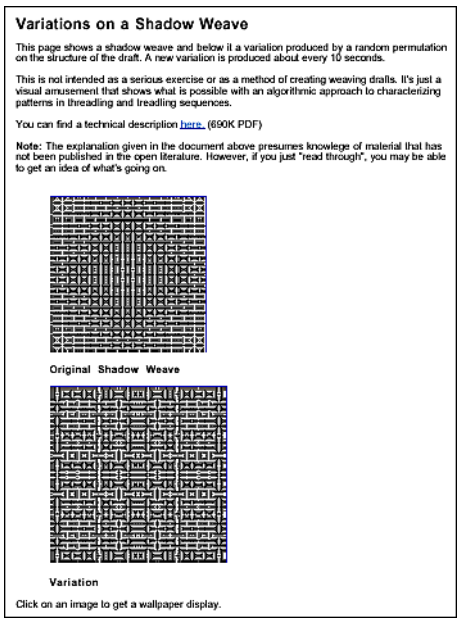



**Figure 2. Shadow-Weave Page**

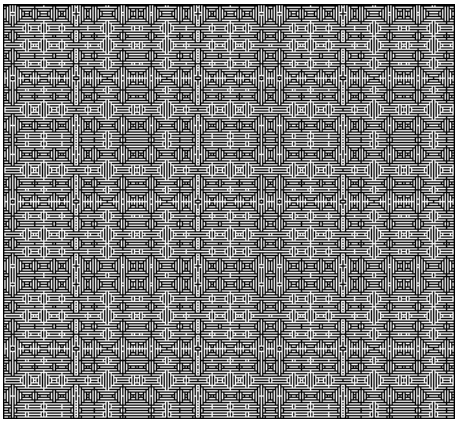


**Figure 3. Shadow-Weave Wallpaper Page**

The HTML for the wallpaper page is simplicity itself:

```
<HTML>
<HEAD>
<TITLE>Shadow Weave</TITLE>
</HEAD>
<BODY BACKGROUND="bandw.gif">
</BODY>
</HTML>
```

The program that produces the images uses ISDs instead of PFDs (see pages 1 through 3) and constructs the sequences explicitly rather than creating pattern forms that are expanded, as was done in the earlier version.

```
link lists
link patutils
link random
link strings
link weavegif
link weavutil

global anchors
global palpat
global palindromes

procedure main(args)
   local tieup, palette, mutant, win1, win, colorways, i

   randomize( )

   anchors := [ ]
   every put(anchors, 1 to 7)

   palpat := [ ]
   every put(palpat, integer(!"8214365"))

   palindromes := list(*palpat)

   every i := 1 to *palpat do
      palindromes[i] := lreflect(palpat[1:i + 1], 2)

   mutant := isd()
   mutant.name := "shadowweave"
   mutant.shafts := 8
   mutant.treadles := 8
   mutant.color_list := ["black", "white"]
   mutant.tieup := pat2rows("8,#55aa956aa55aa956")

   repeat {
      palindromes := shuffle(copy(palindromes))
      anchors := shuffle(copy(anchors))
      mutant.threading := mutant.treadling :=
         sequence(anchors, palindromes)
      mutant.warp_colors :=
         lextend([1, 2], *mutant.threading)
      mutant.weft_colors :=
         lextend([2, 1], *mutant.treadling)
      win := weavegif(mutant)
      WriteImage(win, "bandw.gif")
      WDelay(win, 10000)
      WClose(win)
      }
end

procedure sequence(anchors, palindromes)
   local i, j, k, p, threading

   anchors := copy(anchors)
   palindromes := copy(palindromes)

   threading := []

   i := put(threading, get(anchors)) |
      stop("program malfunction")

   while p := copy(get(palindromes)) do {
      every put(threading, run(threading[–1], get(p)))
      every put(threading, !p)
      i := get(anchors) | break
      every put(threading, run(threading[–1], i))
      }

   threading := lreflect(threading, 2)

   return threading

end

procedure run(i, j)

   if i < j then suspend i + 1 to j
   else if i > j then suspend i – 1 to j by –1
   else fail

end
```

We ran into an unexpected problem. After some number of images were created, the program crashed for lack of memory, even though every window was closed before a new one was created: There was a memory leak. The leak probably is in Icon's storage management for window resources, although it conceivably could be in X.

The solution was to terminate the program after a safe number of images had been processed, while launching another copy of it before terminating — a kind of suicidal self-cloning.

The changed code is:

```
   every 1 to 100 do {
      palindromes := shuffle(copy(palindromes))
      anchors := shuffle(copy(anchors))
         …
      }

   system("wallpapr &")

   exit()
```

If you investigate the shadow-weave Web pages <1, 2>, you can see variations by reloading the pages at intervals.

Of course, there's the ever present danger that the server on which the program runs will crash. The server is quite stable and the program has been known to run for weeks at a time. There is, however, nothing to be done about a power outage, which happens on occasion, especially during our "monsoon" season when there is a lot of electrical activity. And then there was the raccoon who passed on brilliantly, most literally, by chewing through the insulation on a cable at a nearby power substation.

### What's Left?

The program shown in this article makes only minor variations on the original shadow weave. There are all kinds of other, more radical variations.

In thinking about these, we've become interested in other kinds of sequences produced by patterns connected by runs. We're not quite ready to write an article about this yet, but expect to see something, perhaps in disguise, in an article a few issues down the line.

### Reference

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

### Links

1. http://www.cs.arizona.edu/patterns/weaving/shadow.html
2. http://www.cs.arizona.edu/patterns/weaving/bandw.html

## Animation — Making Movies

In the context of computer presentation, a "movie" is a packaged animation. Sound may be included, but that is beyond the scope of this article.

Movies are a very hot topic in computing at the present time and there are several commercial applications that provide a variety of facilities.

Several formats are in widespread use. The main ones are MPEG, QuickTime, and AVI (Windows only).

### Animated GIFs

The simplest and most widely used format for packaged animations, especially for the Web, is GIF89a ("animated GIFs"). GIF89a allows a sequence of images to be stored in one file. Application software then can produce an animation by displaying successive images (frames).

Most programs that create animated GIFs do so from a collection of previously prepared single–image GIFs. These can be in GIF87a or GIF89a format. The GIF89a file format allows control information to be included so that the application that displays the images can determine how they are to be presented.

The following options are supported for controlling the display. They are specified in the application that builds the animated GIF.

interlaced
interframe delay
loop
transparent background
frame position
disposal method

When interlacing is specified, each frame is displayed progressively and gradually filled in to the final detail. Interlaced images do not display any faster than non–interlaced ones, but they give the user something to look at while a large image is being downloaded. Animated GIFs usually are not interlaced, because this interferes with the visual transition between successive frames.

The interframe delay is the amount of time between drawing frames. It can be set to 0, but that may cause the animation to run too fast on some platforms.

If the looping value is greater than 0, the animation repeats the specified number of times and then stops. Not all programs support specific values — if you want an animation to display more than once, it's safer to use the "forever" option, which causes the animation to loop until it is interrupted.

Transparent backgrounds serve the same purpose that they do in GIFs that are not animated [1].

Frames can be shifted from the origin by arbitrary amounts. This can be useful for specialized animations.

Frame disposal refers to what is done with the currently displayed frame when the next frame is

drawn. "Do not dispose" is recommended for opaque animations and "Revert to Background" for transparent animations.

Some applications that create animated GIFs allow the frame–related options to be set separately for each frame. Others only apply the specified options to all frames.

Some also provide optional optimization, which crops all frames but the first to the part that is different from the preceding frame. In some kind of animations, this can considerably reduce the file size and increase the speed with which animations can be downloaded and displayed.

#### Creating Individual GIFs

Many applications can create the individual GIFs that go into an animation. For example, to create an animation of the Icon kaleidoscope program, a series of GIF images can be written as a program executes and can be packaged later.

All this requires is placing calls to WriteImage() at appropriate places — wherever the display is changed.

For the kaleidoscope application [2, 3], there is only one place that images need to be written:

```
procedure outcircle(off1, off2, radius, color)
   Fg(pane, color)
   draw_proc(pane, off1, off2, radius)
   draw_proc(pane, off1, –off2, radius)
   draw_proc(pane, –off1, off2, radius)
   draw_proc(pane, –off1,–off2, radius)
   draw_proc(pane, off2, off1, radius)
   draw_proc(pane, off2, –off1, radius)
   draw_proc(pane, –off2, off1, radius)
   draw_proc(pane, –off2,–off1, radius)
   WriteFrame()                    # write frame
   return
end
procedure WriteFrame()
   static count
   initial count := 1
   WriteImage(pane, "kaleido" ||
      right(count, 4, "0") || ".gif", –half, –half, size, size)
   write(&errout, count)
   count +:= 1
   return
end
```

The procedure WriteFrame() is used to isolate the necessary code, and it is particularly useful if images need to be written at several places in a program.

The images are numbered serially. This makes creating an animation from them easier, since many applications for composing animated GIFs order the individual images by the sorting order of their names. Writing the count to standard error output helps the person creating the frames keep track of how many have been written.

For the kaleidoscope, it is not necessary or desirable to write a frame for each of the eight symmetric drawings. Unless the animation is very fast, it would look peculiar, and it would increase the size of the animation by a factor of about eight.

Another place a frame might be written in the kaleidoscope program is when the diplay is cleared. This would clearly show the transitions between different parameter sets.

## Creating Animated GIFs

There are freeware, shareware, and commercial applications that can package existing GIF images.

For UNIX, there is a freeware application, gifmerge <1>, that runs from the command line. For the Macintosh, there is a very capable freeware program, GifBuilder <2>, that runs interactively and supports "drag and drop". GifBuilder also can extract individual images from a packaged animation as well as convert between other movie formats. For Windows, there is a shareware program, GIF Construction Set <3>. As far as we know, there is no freeware GIF animation builder for Windows at the present time.

## References


1. "Animation — Image Replacement", Icon Analyst 55, pp. 8-10.

2. "The Kaleidoscope", Icon Analyst 38, pp. 8-13.

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


## Links

1. http://www.tu-chemnitz.de/~sos/GIFMERGE/index.html

2. http://iawww.epfl.ch/Staff/Yves.Piguet/clip2gif-home/GifBuilder.html

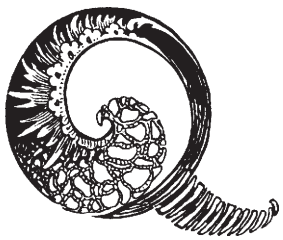
3. http://www.mindworkshop.com/alchemy/gifcon.html

## Sending E-Mail About the Analyst

If you have questions, comments, corrections, or any other concerns related to the Analyst, send e-mail to

icon-analyst@cs.arizona.edu

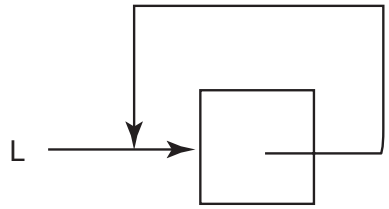
Mail to this address goes only to the editors of the Analyst.



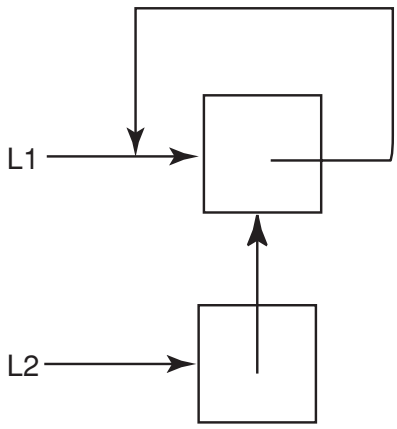
## Quiz — Pointer Semantics

1. Write code segments that produce the following list structures. Each box represents a list element.

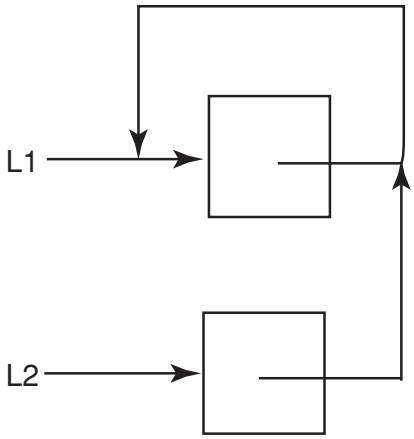
(a)



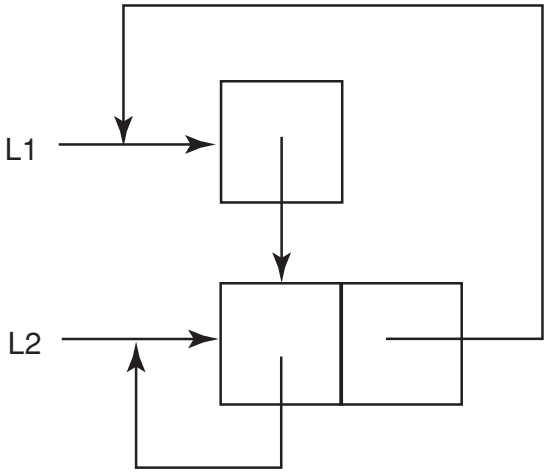

(b)




(c)




(d)




2. Diagram the list structures produced by the following code segments.

(a)

```
L := []
push(L, L, L, L)
```

(b)

```
L1 := []
L2 := copy(L1)
put(L1, L2)
```

(c)

```
L1 := []
L2 := copy(L1)
push(L2, L1)
```

(d)

```
L1 := list(5, 1)
push(L1, [], L1)
L1[1] := 0
pull(L1)
```

## Drawups

*The language of weaving is not easy to understand nor to write. Most of the weaving words we use are part of our non-weaving vocabulary: pattern, unit, block, simple, shadow, fancy, satin, plain, tie, profile, halftone, turned. You may not recognize the very specific ways these words are used in a sentence about weaving.* — Madelyn van der Hoogt [1]

### The Problem

A drawup is, in a sense, the opposite of a drawdown — a draft created from a drawdown, which is a representation of the interlacement of a weave [2].

Early in our explorations of weaving we recall encountering a well-known book that shows only drawdowns with no corresponding drafts that would show how to weave them [3]. Figure 1 is an example scanned from the book and Figure 2 is a drawdown obtained from this image by a program we'll describe in a later article.

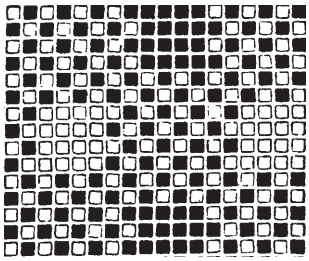


**Figure 1. A Scanned Drawdown**

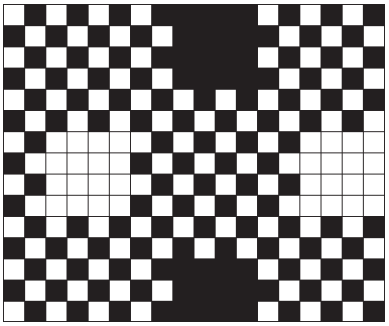


**Figure 2. A Drawdown Grid**

We were puzzled how a weaver could use drawdowns as a basis for weaving. We later were told by an experienced weaver "that's left as an exercise".

It wasn't at all obvious to us how to create a draft from a drawdown (and most weavers don't know how), so we set out to (what else?) write a program to do it. A primary objective was to produce a drawup with the fewest number of shafts and treadles. (The problem is trivial if a different treadle is used for every row and a different shaft is used for every column — but that's not helpful for actual weaving.)

The key observations are that if a drawdown contains duplicate rows, these rows can be produced by the same treadle, and if there are duplicate columns, they can be produced by the same shaft. Conversely, the draft must have at least as many shafts as there are different columns, and similarly for the treadles and rows. If there are no duplicates, then the number of treadles required is the number of rows in the drawup and the number of shafts required is the number of columns in the drawup.

It's then just a matter of identifying the duplicate rows and columns and creating a tie-up that connects them in a way that produces the desired result.

### The Program

The following program works with a drawdown represented by a bi-level pattern (see pages 4 through 6) and produces an ISD (see pages 2 and 3).

```
link options
link patutils          # for pat2rows()
link patxform          # for protate()
link weavutil          # for isd declaration
link xcode

record analysis(rows, sequence, patterns)

procedure main(args)
   local threading, treadling, tie, pattern, i
   local symbols, symbol, drawdown, draft, opts

   opts := options(args, "n:")

   drawdown := pat2rows(read()) |
      stop("*** invalid input")

   treadling := analyze(drawdown)
   drawdown := protate(drawdown, "cw")
   threading := analyze(drawdown)

   symbols := table("")

   every pattern := !treadling.patterns do {
      symbol := treadling.rows[pattern]
      symbols[symbol] := repl("0", *threading.rows)
      pattern ? {
```

```
      every i := upto('1') do
       symbols[symbol][threading.sequence[i]] := "1"
      }
   }

symbols := sort(symbols, 3)
tie := ""

while get(symbols) do
   tie ||:= get(symbols)

draft := isd()
```


# 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*




```
   draft.name := \opts["n"] | "drawup"
   draft.threading := threading.sequence
   draft.treadling := treadling.sequence
   draft.warp_colors := list(*threading.sequence, 1)
   draft.weft_colors := list(*treadling.sequence, 2)
   draft.color_list := ["black", "white"]
   draft.shafts := *threading.rows
   draft.treadles := *treadling.rows
   draft.tieup := tie2matrix(*threading.rows,
     *treadling.rows, tie)

   xencode(draft, &output)

end

procedure analyze(drawdown)
   local sequence, rows, row, count, patterns

   sequence := []
   patterns := []

   rows := table()

   count := 0

   every row := !drawdown do {
     if /rows[row] then {
       rows[row] := count +:= 1
       put(patterns, row)
       }
     put(sequence, rows[row])
     }

   return analysis(rows, sequence, patterns)

end

procedure tie2matrix(shafts, treadles, tieup)
   local matrix

   matrix := []

   tieup ? {
     every 1 to treadles do
       put(matrix, move(shafts))
         }

   return matrix

end
```

In order to manipulate the drawdown, it is converted from a bi-level pattern to a binary matrix: a list of strings composed of 0s and 1s.

The procedure analyze() goes through the rows of that matrix, using the table rows to hold the distinct rows, to which identifying numbers are assigned. At the same time, the list patterns is built to record the order in which the rows appear.

The procedure analyze() first is used on the rows of the drawdown and then, by rotating the

matrix using protate(), the columns.

All that remains is to create the tie-up. For every treadling pattern, its row is initialized with 0s, which indicate no tie. Then for every position in the row that is 1, the corresponding value is set to 1, indicating a tie.

Finally an ISD is assembled and output using xencode().

Figure 3 shows the drawup draft for the drawdown shown in Figure 2. Notice that it only requires six shafts and six treadles.

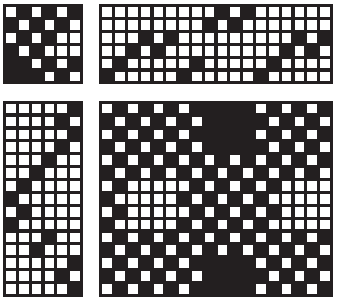


**Figure 3. A Drawup Draft**

### Observation

The method described above can be used to create a draft for any two-color image. Although drawdowns usually are shown as grid diagrams with the squares large enough to see the interlacing easily, an image in which the interlacement is represented by single pixels contains the same information.

Be aware, though, that unless there are many duplicate rows and columns — or the image is tiny — the resulting draft will require more shafts and treadles than are available on treadle looms. There also is the important question of whether the fabric would hold together, a topic we'll cover in a subsequent article.

### To Come

The next step beyond creating drafts from drawdowns and two-color images is to create them for multicolored "drawdowns" or, what is equivalent, multicolored images.

This is a much more difficult problem and, in general, there may be no solution. Figure 5 shows a three-color pattern for which there is no draft.

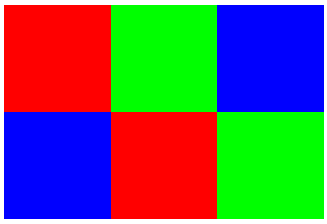


**Figure 5. An Undraftable Color Pattern**

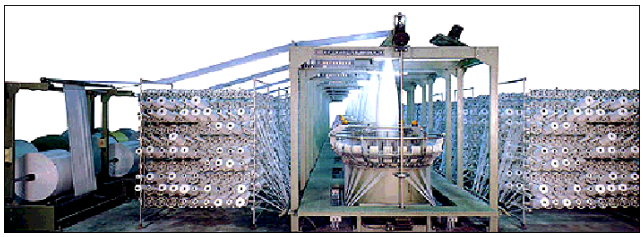
Try to assign colors to the rows and columns of this pattern and you'll see the problem.

## References


1. *The Complete Book of Drafting for Handweavers*, Madelyn van der Hoogt, Shuttle Craft Books, 1993.

2. "Weave Structure", Icon Analyst 55, p. 14.

3. *A Handbook of Weaves*, G. H. Oelsner, 1915, Macmillan, reprinted by Dover.

## What's Coming Up

*Everything should be built top-down, except the first time.* — Alan Perlis

In the next issue of the Analyst, we plan to have an article on determining whether a color pattern can be drafted and woven, and, if so, how to create a draft. Continuing with weaving, we'll have an article on a weave design technique known as name drafting as well as an article on modular arithmetic as it applies to the numbering of shafts and treadles.

Versum sequences will reappear in a generalized form, and we may show a few versum weaves.

Our series on sequences will continue with an article on periodic sequences in which the same pattern of values repeats endlessly.

In **From the Library**, we plan to cover modules that support rational and complex arithmetic.

### Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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