The Icon Analyst

In-Depth Coverage of the Icon Programming Language

December 1998
Number 51

In this issue …

Pattern-Form Metrics

Patterns possess utility as well as beauty. Once we have learned to recognize a background pattern, exceptions suddenly stand out.

— Ian Stewart [1]

Character patterns, pattern forms, and pattern grammars provide ways of analyzing and expressing structure in sequences of values [2-4].

When a string of interest is identified, it can be represented by a pattern form, and added to the grammar as a new definition. All instances of the string then are replaced by the variable for the new definition. In most cases, this results in a decrease in the size of the grammar, while revealing structure in the character pattern.

In using grammars to compress image strings [5], the goal is to find a small grammar (possibly consistent with keeping the grammar depth small). In some situations, identifying significant elements in a character pattern may be more important than reducing the size of the grammar. In other situations, short definitions may be important even if creating them increases the size of the grammar. In general, however, the motivations for using character grammars are consistent with small size.

There are several ways the size of a grammar might be measured. For the purposes of this article, we'll define the size of a grammar to be the number of characters the grammar takes when saved as a file, not counting L-System directives like the axiom.

Changes in Grammar Size

Although charpatt [4] computes the savings for pattern forms, it is useful to understand how savings result from using different pattern forms.

When n occurrences of a string of length i are replaced by a pattern form, each occurrence is replaced by a single character for the new variable, resulting in a reduction in the size of the grammar of $n \times (i-1)$. The new definition, on the other hand, adds to the size of the grammar: one character for the variable, two characters for –>, m characters for the pattern form, and one or two characters for the line terminator, depending on the platform [6]. For simplicity, we'll assume one-character line terminators, so the increase in the size of the grammar because of the new definition is $i + 4$. The net change in the size of the grammar is $n \times (i-1) - m - 4$, where m depends on the pattern form.

For a constant, the pattern form is just the string, and $m = i$. So the change in the size of the grammar is

$$\Delta_c = n \times (i-1) - i - 4 \qquad \textit{constant}$$

For a repetition $[s, k]$, where s is of length j and k is a d-digit number, $m = j + d + 3$, (the 3 is for the pattern-form meta-characters). Note that $i = j \times k$. The change in the size of the grammar is

$$\Delta_r = n \times (i-1) - j - d - 7 \qquad \textit{repetition}$$

Since d usually is 1 or 2 and rarely much larger, its effect is minor. The change in the grammar size also can be written as

$$\Delta_r = n \times (i-1) - i/k - d - 7$$

where, of course, k divides i evenly.

Note: The lengths of the strings can, of course, be obtained from the strings themselves. If we let $\lambda(s)$ be the length of s, then

$$\Delta_c = n \times (\lambda(s) - 1) - \lambda(s) - 4$$

$$\Delta_r = n \times (\lambda(s) - 1) - j - \lambda(k) - 7$$

We prefer implicitly derived values *i* and *d*, especially for the purposes of plotting.

The reversal pattern form, <*s*>, simply adds 2 to the change in size for a constant, so

$$\Delta_v = \Delta_c - 2 = n \times (i - 1) - i - 6 \qquad \textit{reversal}$$

The decollation pattern form $\{s_1, s_2, \ldots s_k\}$ requires $i + k + 2$ characters, since the sum of the lengths of $s_1, s_2, \ldots s_k$ is *i* and there are *k* commas and 2 braces. Therefore,

$$\Delta_d = n \times (i - 1) - i - k - 6 \qquad \textit{decollation}$$

Notice that

$$\Delta_d = \Delta_c - k - 2$$

The "worst" case for constants occurs for $n = 1$ (we disallow $n = 0$), so

$$\Delta_c \geq -5$$

In other words, defining a constant that only occurs once results in an increase of 5 in the size of the grammar. Of course

$$\Delta_v \geq -7$$

For repetitions, the worst case occurs for $n = 1$ and $i = 1$, so

$$\Delta_r \geq -7$$

The situation is different for decollations, for which the worst case occurs for $n = 1$ and $k = i$ (in other words splitting an *i*-character string into *i* one-character parts). The result is

$$\Delta_d \geq i - 7$$

That is, there is no limit to how much a decollation can add to the size of a grammar.

Plots

Over the years, we've been confronted with many complicated plots and graphs that were baffling if not incomprehensible. Now it's our turn to produce some.

Figure 1 shows the metrics for constants as pattern forms. Figure 2 on the next page shows the metrics for repetitions with different symbols for different values of *k*. Finally, Figure 3 on page 4 shows the metrics for decollation.

We were tempted to overlay the three plots as a joke, but that would take too much space and it's not really all that amusing anyway.

We also could show the crossover points where one pattern form saves more than another, but that information isn't particularly useful.

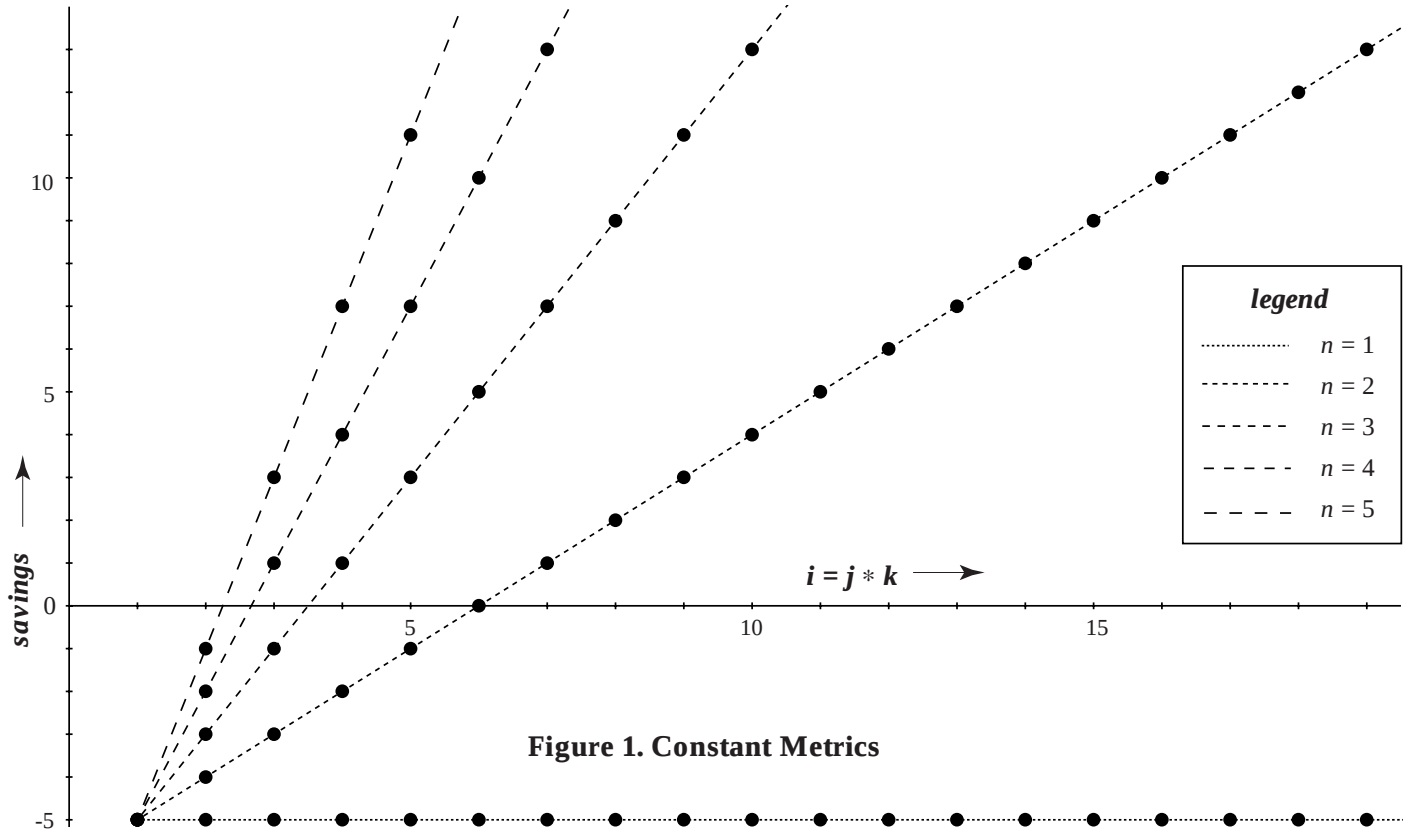


Figure 1. Constant Metrics

Choosing Pattern Forms

In most situations, there isn't a choice among pattern forms for a given string. When there is, however, the choice may be significant not only in reducing the size of the grammar but also in subsequent development of the grammar.

Consider, for example, this grammar with only a portion of interest shown:

```
*–>...12341234123412341234123412341234123412341234...
```

There are three different pattern forms that can be used to encode the string of interest: constant, repetition, and collation. If we assume no other occurrences of this string or its primary components elsewhere in the definition, then we have three plausible possibilities:

```
* –> ...AAAAAAAAAA...
A –> 1234
```

```
* –> ...B...
B –> [1234,10]
```

```
* –> ...C...
C –> {1111111111,222222222,3333333333,4444444444}
```

The net savings are 22, 26, and –10, respectively, and all increase the grammar depth by 1.

For the constant, we can continue:

```
* –> ...D...
A –> 1234
D –> [A,10]
```

which increases both the size of the grammar and the depth by 1.

In the case of the decollation, we also can continue:

```
*–>...{E,F,G,H}...
E–> [1,10]
```

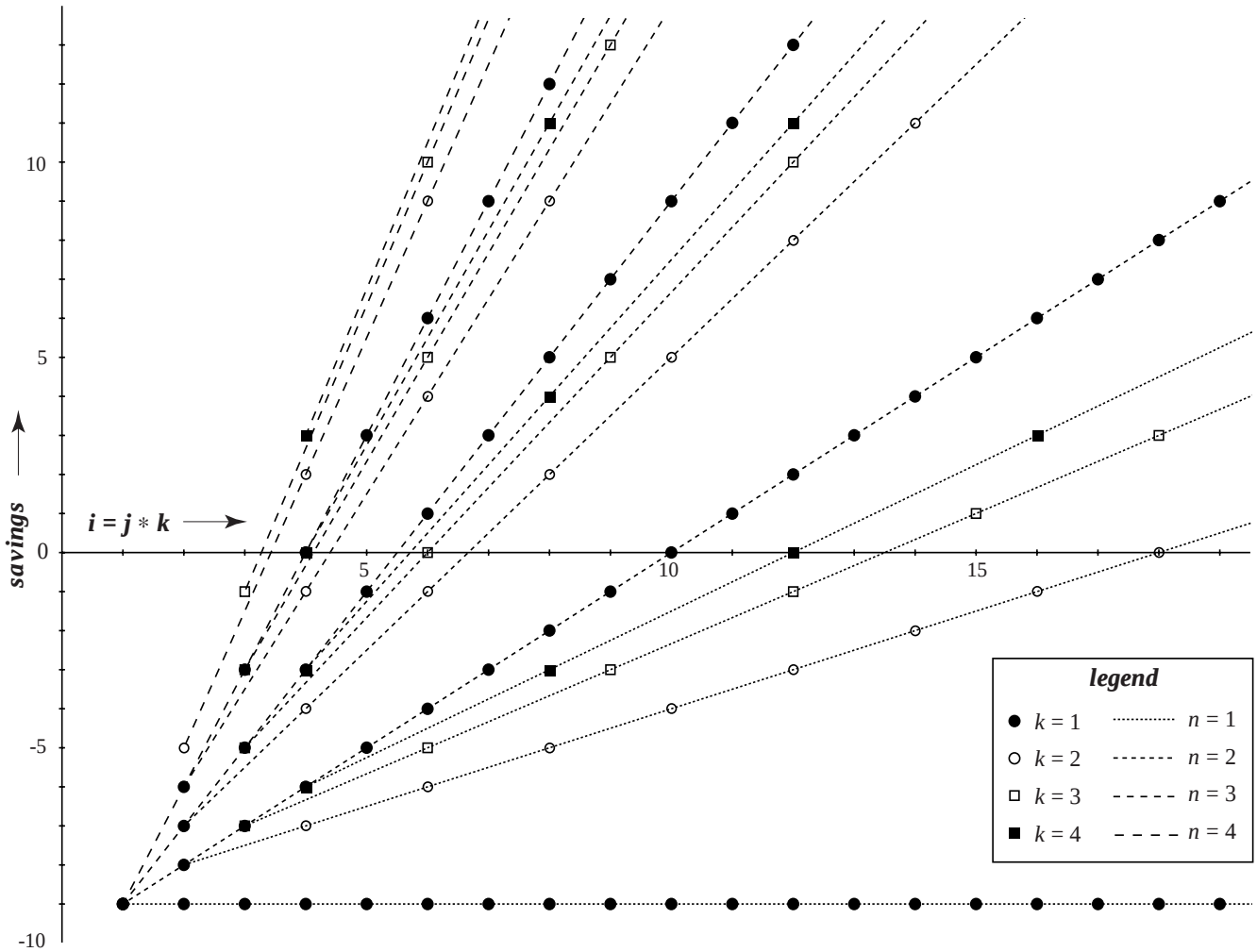


Figure 2. Repetition Metrics

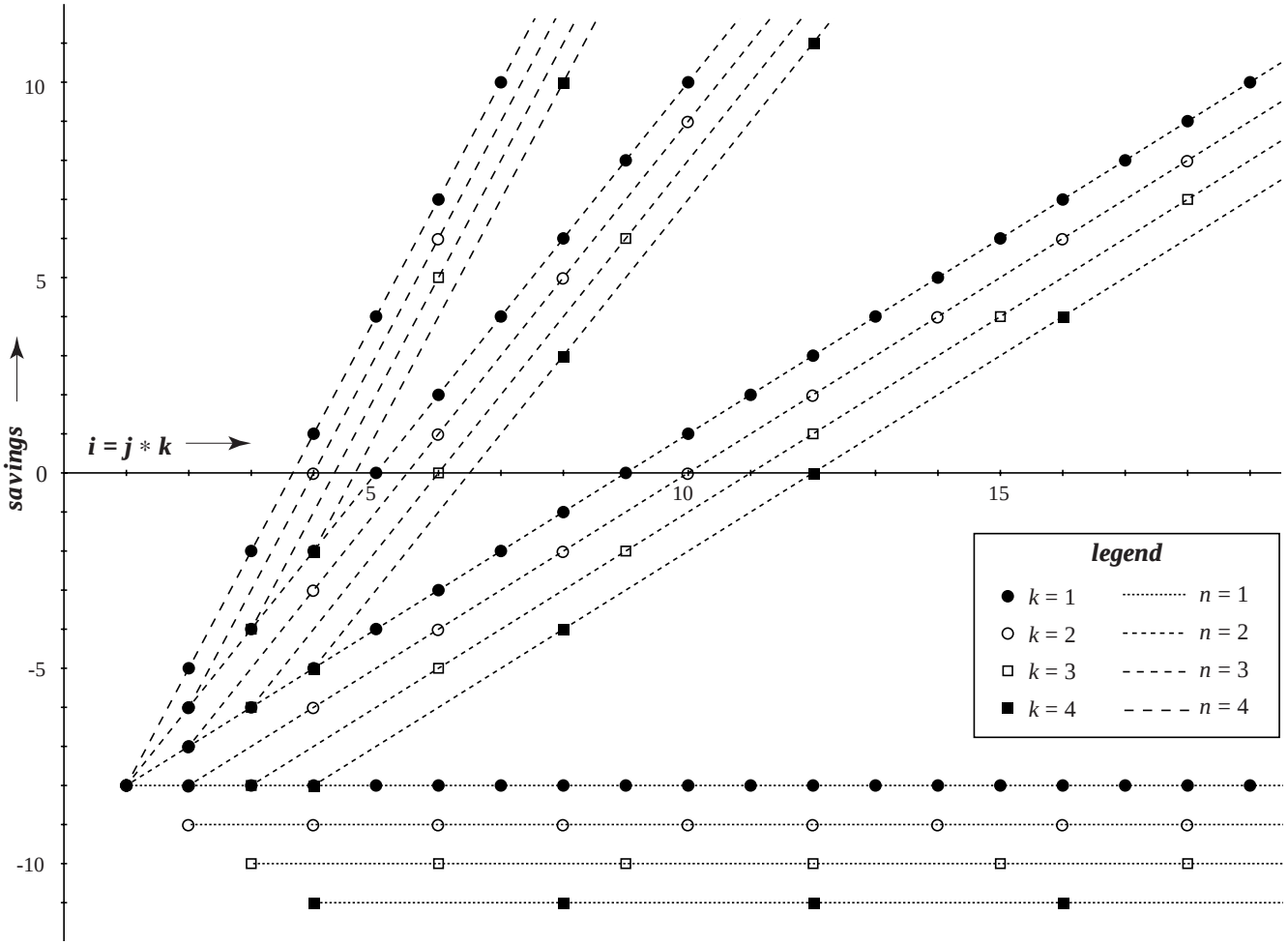


Figure 3. Decollation Metrics

```
F–> [2,10]
G–> [3,10]
H–> [4,10]
```

which increases the grammar size by 4 and the depth by 1.

You may wonder why decollation would be picked over repetition in a case like this. It might be because decollation was such an important structural consideration that the increase in grammar size was worth it. It might be just seeing the pattern as a decollation and not also seeing it as a repetition.

The point is that once such a direction has been taken, the mistake, if it is one, is likely to be lost in the maze of subsequent grammar development. Things of this kind happen often in developing grammars — that's what makes the process so challenging and interesting.

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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

References

1. *Nature's Numbers: The Unreal Reality of Mathematics*, Ian Stewart, Basic Books, 1995.

2."Character Patterns", Icon Analyst 48, pp. 1-7.

3."Character Patterns", Icon Analyst 49, pp. 1-6.

4."Analyzing Character Patterns", Icon Analyst 50, pp. 1-7.

5."Tricky Business — Image Grammars", Icon Analyst 50, pp. 14-18.

6."Line Termination", Icon Analyst 48, pp. 1-3.

A Weaving Language

Weaving is such an astonishingly complex field; a lifetime is not long enough to learn all one needs to know, let alone enough to weave all one wants to weave.

— Madelyn van der Hoogt [1]

In previous articles [2-4], we used pattern forms that proved useful in a variety of contexts: constants, repetitions, reversals (and hence palindroids), and decollations. In this article, we'll describe pattern forms that are used in weaving. Some of these are ones we've already used. Some are idiosyncratic to weaving. Others have broader applicability.

About Weaving

Editors' Note: We are not weavers; ours is "book learning". We have attempted to describe matters related to weaving in an accurate manner. If we have made mistakes, please let us know.

Before going on, we need to say something about the concepts, techniques, and terminology used in weaving. If you are a weaver, you may wish to skip this section.

The description that follows leaves out many aspects of weaving that are not relevant to pattern forms. Details are omitted also, and some simplifications have been made. If you are new to weaving and are interested in learning more, we recommend Reference 5 as a place to start.

There are many ways to make fabric. Weaving arguably is the oldest, possibly dating back 10,000 years. Ancient artifacts in widely separated places suggest that weaving developed independently in many cultures.

The distinguishing characteristic of woven fabric is the intertwining of crossed threads. Figure 1 shows an example of a crudely woven fabric.

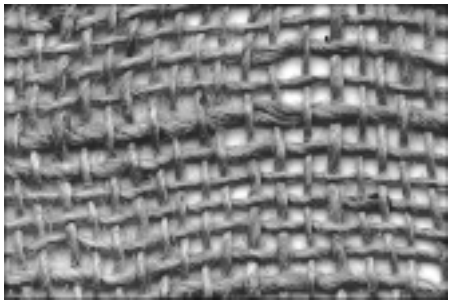


Figure 1. Burlap

The intertwining of threads gives woven fabric its strength and texture — the tactile and visual nature of its surface. Elaborate patterns can be produced by using colored threads.

Present industrial methods for producing woven fabric are very efficient and involve complicated, sophisticated machinery with computer control. It is easier to understand weaving as it is practiced as a hand craft on manually operated looms.

There are many kinds of looms and seemingly endless variations on each. We'll use a typical floor loom, such as the one shown in Figure 2, which has all the capabilities we need for our purposes. The terms used in this figure are explained in what follows.

Figure 2. A Floor Loom

On a floor loom, one set of threads, called the *warp*, passes from the back of the loom to the font. *Weft* threads (also called *filler* and *woof*) are passed successively through the warp as it advances toward the front. The intertwining is accomplished by raising some of the warp threads to form a *shed* through which a weft thread passes. When these warp threads are lowered, they are above the weft thread and the remaining warp threads are below it — thus the warp and weft threads are interlaced.

For a plain weave, the simplest of all, odd-numbered warp threads are raised for one shed and even-numbered for the next, repeating. Figure 3 on the next page shows the shed schematically. The interlacing is shown in the simulated weave in Figure 4 and also in the real weaving pictured in Figure 1.

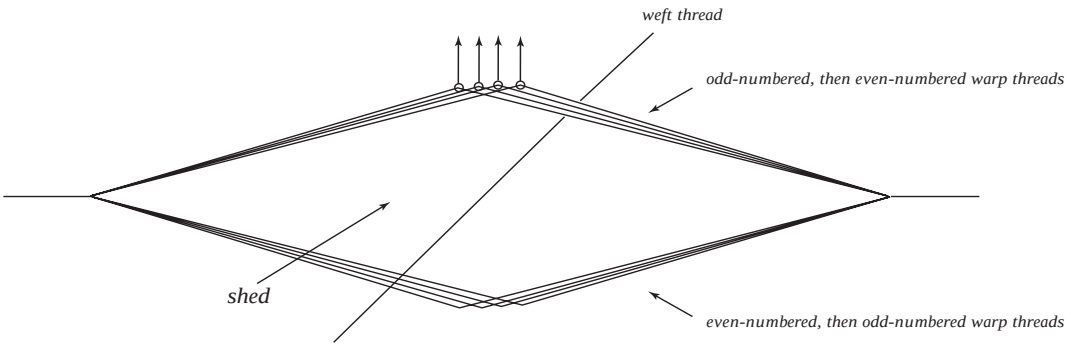


Figure 3. Shed for a Plain Weave

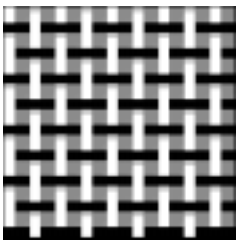


Figure 4. A Plain Weave

If the sets of warp threads used to form sheds are chosen differently, different interlacings occur. See Figure 5.

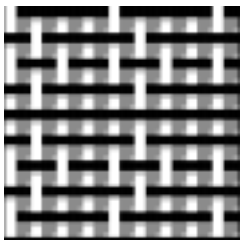


Figure 5. A More Complicated Weave

The question of how different sheds are created gets us into the mechanics of looms.

As mentioned earlier, warp threads are strung from front to back. In their paths are horizontal *shafts* (also called *harnesses* and *heddle frames*) that extend the width of the loom. On these shafts are *heddles* through which warp threads can be strung. Each warp thread is threaded through a heddle on exactly one shaft. See Figure 6.

When a shaft is raised, the warp threads strung through it are raised above the others to make a shed.

Shafts are connected (*tied up*) to *treadles* via *lamms*. More than one shaft can be connected to a treadle. When a treadle is pressed, the shafts connected to it are raised to form the shed.

Note that in general there is not a symmetric relationship between the warp and the weft. The warp is threaded and the tie-ups made before the actual process of weaving begins.

There is a pattern to the threading of warp threads through the shafts, a pattern in the order treadles are pressed during weaving, and, if different colors threads are used, there are patterns in their orders. There even are pattern in the tie-ups. Have we mentioned the ubiquity of patterns recently?

Weaving Drafts

Weaves are described by *drafts*, which typically consist of a representation of the threading, treadling, and tie-up, along with a *drawdown* that shows which threads will be in front on the woven

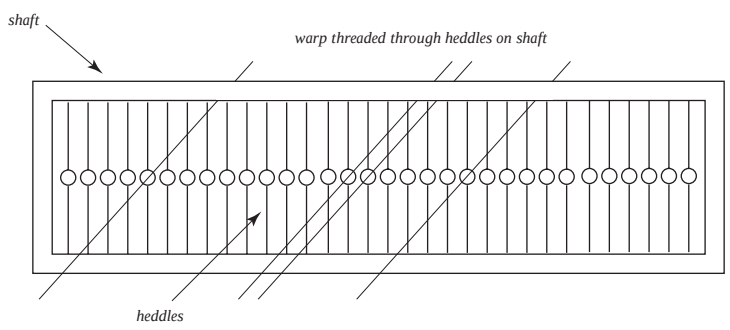


Figure 6. A Shaft

fabric.

Drafts are written using many different notational systems and often contain comments and notations specific to the author. A typical draft shows one full repeat, after which the weaver continues from the beginning depending on the width and length of the weave. In this sense, repeats are fundamental to most weaves.

Weaving drafts traditionally have been done by hand. Even though there are now computer programs that help with this task, most weavers still use the old method. Here's a quote from a book published in 1961 [6], which still is apt for most weavers:

> Drafts and development diagrams are made on cross-section paper. The best type is engineering paper with ten squares per inch and with each full inch indicated by a heavy line. Equipment for draft-writing and developing includes India ink, a ruling pen and a crowquill or a fine-line India ink fountain pen, a straight lettering pen with a width of one-tenth of an inch (the Estabrook #13, for instance), and a transparent 45-degree triangle.

This reminds us of the golden days of slide rules …

We couldn't find a copyright-free hand-done draft, so we used one from a computer program, as shown in Figure 7. In the drawdown, black squares indicate warp threads that are in front, while white squares show weft threads that are in front.

Color in hand-drawn drafts is indicated in a variety of ways, sometimes apart from the draft itself. Most computer weaving programs work in color and can provide color drawdowns, but that obscures the warp / weft pattern. Such programs can, of course, produce images of weaves. These may simply consists of pixels of the appropriate colors, or they may simulate the three-dimensional appearance of the weave.

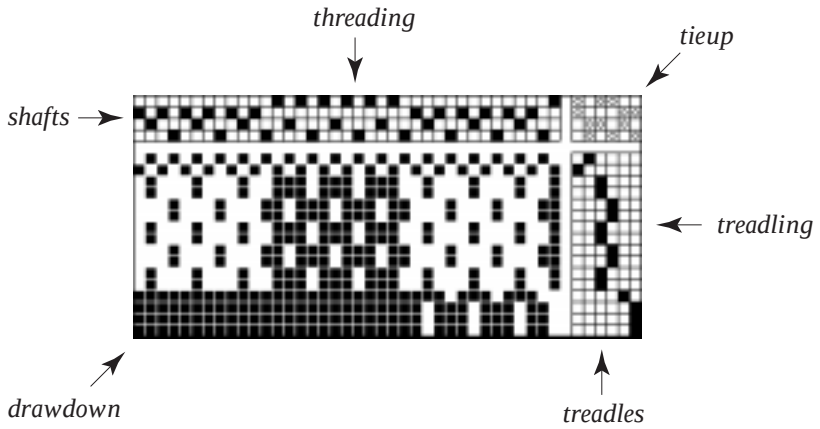


Figure 7. Weaving Draft

Weaving Programs

Weaving programs vary from freeware <1> and commercial programs for personal computers <2-9> to packages for industrial weaving that cost more than $100,000. Most commercial weaving applications can control *dobby* devices that can be attached to hand looms. Dobby devices can raise shafts in any combination and allow the creation of sheds that are not possible with fixed tie-ups. Except for high-end applications that control sophisticated powered looms, the weaving itself is done by hand.

Most weaving programs mimic manual techniques. An exception is the weaving engine in Painter <10>, the most popular personal computer application for producing artistic effects.

The Painter Weaving Language

Among its many facilities, Painter has the ability to produce images of weaves. Most Painter users probably don't get beyond the built-in weaves and modifications that can be made to them. Hidden away, and only documented in a PDF file on the distribution CD-ROM, is a weaving language [7] that allows threadings and treadlings, as well as warp and weft color sequences, to be described in terms of expressions. The expressions can characterize most of the patterns commonly found in weaving drafts.

The Painter weaving language is not a programming language. It only has expressions that describe patterns of characters, much in the way pattern forms do. The weaving language consists of labeling characters, integers, and operators.

There are two kinds of labeling characters: (1) digits that label shafts and treadles and (2) letters that stand for colors.

The Painter loom has eight shafts and eight treadles (most real looms have more treadles than shafts). The digits 1 through 8 are used to label both the shafts and the treadles.

Sequences of digits correspond

to sequences of shafts or treadles, depending on context. For example, in threading the shafts, 1346 means the first warp thread goes through shaft 1, the second through shaft 3, the third through shaft 4 and the fourth through shaft 6. In treadling, 1346 means treadle 1 is pressed for the first weft thread, treadle 3 for the second weft thread, 4 for the third, and 6 for the fourth. Full sequences are, of course, much longer than this. The length of the threading sequence determines the width of the fabric (in threads), and the length of the treadling sequence, the length of the fabric.

Figure 8 shows a dialog in which the weaving specifications can be entered and edited.

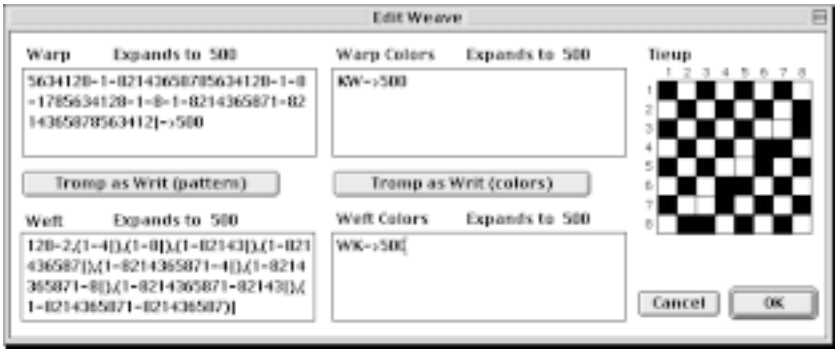


Figure 8. A Painter Weaving Specification

In case you're interested, Figure 9 shows a portion of the resulting weaving. The three-dimensional effect comes from the pattern of the warp and weft threads, not from any simulated thickness in threads — the threads are one pixel wide and packed solidly.

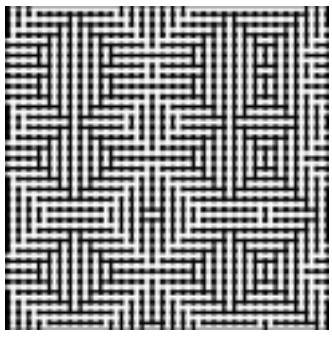


Figure 9. Result from a Weaving Specification

In the dialog, the two text-entry fields at the left contain weaving expressions for the threading (warp) and treadling (weft). The two text-entry fields in the center contain expressions for the colors of the warp and weft. The array in the upper-right corner is the tie-up, with black squares indicating which shafts (horizontal) and tied to which treadles (vertical). Incidentally, "tromp as writ" is weaving jargon for using the same sequence for the weft as for the warp. Pressing a tromp as writ button merely copies the expression above it to the field below.

You may see why the users of Painter are not encouraged to get into the weaving language. Even without an explanation of the operators (which is coming up), it's clearly not the kind of thing most artists would find attractive.

Take deep breath. We're about to plunge into the weaving language itself. The operators essentially are the same for colors as for the threading and treadling, but some operators do not apply to colors.

In our examples, we'll concentrate on the threading and treadling expressions.

There are 15 operators in all, which range from simple to complex and arcane. We will make the description concrete by showing Icon procedures to implement the operators. This provides a good example of the value of high-level string-manipulation facilities in a programming language.

Basic Operators

A typical operator is replication, indicated by *. For example, 1346*7 "expands" to

```
1346134613461346134613461346
```

Icon' built-in repl() implements this directly.

Another operation that is implemented by an Icon function is reversal (`). In addition, rotation (#) is implemented by the procedure rotate() in the strings module of the Icon program library. Beyond these, we need new procedures.

The concept of *extension* permeates the weaving language, as it does weaving itself. Extension, indicated by the operator –>, replicates its left operand to produce a result whose length is given by its right operand. For example, 1345–>16 expands to

```
1346134613461346
```

Note that the right operand is an integer, not a sequence of labels.

If the replication does not come out even, it is truncated on the right. For example, 1346–>15

expands to

```
134613461346134
```

Here's a procedure that implements extension:

```
procedure Extend(p, i)
   i := integer(i)
   return case i of {
      *p > i   : left(p, i)
      *p < i   : left(repl(p, (i / *p) + 1), i)
      default  : p
      }
end
```

Note that left() truncates its argument at the right if it is longer than the specified width. The first two case clause expressions are comparisons. If one succeeds, its value is the value of its right operand, namely the value of i, which compares successfully with the value of i in the case expression. A handy trick — uhh ... idiom. Note that it is necessary to convert i to an integer, since it might be passed as a numeral string, in which case the values would fail to match, since arithmetic comparison operations convert their results to numbers. Why is i converted to an integer before the case expression instead of using case integer(i)? What happens if i is not convertible to an integer?

Incidentally, the code for *p < i works for all cases, at the expense of extra computation. For this, the procedure is simplicity itself:

```
procedure Extend(p, i)
   return left(repl(p, (i / *p) + 1), i)
end
```

Extension can be specified explicitly, but it also is implicit in operations on sequences that may be of different length. If that's the case, the shorter sequence is extended to the length of the longer, except as noted in the descriptions that follow.

The interleaving operation, ~, which is collation but only for two strings, provides an example:

```
procedure Interleave(p1, p2)
   local i, p3
   if *p1 < *p2 then p1 := Extend(p1, *p2)
   else if *p2 <  *p1 then p2 := Extend(p2, *p1)
   p3 := ""
   every i := 1 to *p1 do
      p3 ||:= p1[i] || p2[i]
   return p3
end
```

It is necessary to check the sizes of the arguments before attempting to extend them — for example, if *p1 > *p2, Extend(p1, *p2) without testing would incorrectly shorten p2.

Palindromes are common in weaving and also in other designs. In the weaving language the palindrome operator is | in suffix position, as in 1346|. This does not produce what you might expect. In fact, the result is not a real palindrome, so we'll call it a pattern palindrome to avoid confusion with the real thing.

The reason pattern palindromes are not true palindromes, which read the same way forward and backward, has to do with their use in repeats. Consider, for example, the (true) palindrome 1346431, derived from 1346. If this is repeated, the result is

```
1346431134643113464311346431 ...
```

Note the duplication of 1s at the boundaries of the repeats. This produces an unavoidable artifact. So, for weaving palindromes, 1346 produces 134643, the first character not being included in the reversal. When it is repeated, the result is

```
1346431346431346431346431 ...
```

You might wonder why we made the true palindrome 1346431 instead of 13466431. For the same reason, this time with the "artifact" in the middle.

In any event, producing a pattern palindrome is easy.

```
procedure PatternPalindrome(p)
   p ||:= reverse(p[2:–1])
   return p
end
```

If the length of p is less than two, p[2:–1] fails and p is returned unmodified. Painter does this for strings of length 1, but in general a zero-length argument is syntactically erroneous in Painter. We decided to punt on this, although it would be easy enough to have the procedure fail in this case.

We're not entirely out of the woods on palindromes. We'll come back to the subject in a later article.

Concatenation is not explicit in the weaving language, but instead is indicated by the comma operator, as illustrated by

```
1346,1346|
```

which expands to

```
1346134643
```

Now we'll go on to some of more esoteric operators in the weaving language.

The Domain

In the weaving language, the *domain* is the ordered sequence of labels for the shafts / treadles: 12345678. (If the Painter loom had more shafts / treadles, this could be extended using letters, as in hexadecimal notation. For real looms, this is fine, but some weaving programs allow as many as 256 shaft / treadles.)

How the domain figures into weaving expressions is illustrated by the "upto" operator <, which concatenates its left and right operands but inserts between them the portion of the domain between the last character of the left operand the first character of the right operand. For example,

```
1346<8
```

expands to

```
134678
```

If, however, the last character in the left operand is greater than the first character in the right operand, the intervening portion "cycles" through the domain, so that

```
1346<3
```

expands to

```
134678123
```

And, as if that was not enough, "tick marks", indicated by a single quotes, can be placed in front of the right operand. When this is done, the number of tick marks specifies the number of complete domain runs to be added in. To illustrate,

```
1346<"8
```

adds two domain runs and expands to

```
134678123456781234567 8
```

Can you grok the mind of a weaver?

To implement this as a procedure, we treated the tick marks as part of the right argument, not as operator symbols. Here's the result:

```
procedure Upto(p1, p2)
   local cycles

   cycles := 0

   p2 ?:= {
      cycles +:= *tab(many('\''))
      tab(0)
      }

   return p1 || UpRun(p1, cycles) ||
      UpBetween(p1, p2) || p2

end
```

UpRun() and UpBetween() are helper procedures, which also are needed elsewhere. They extract their results from a duplicated domain:

```
$define Domain "12345678"
$define DomainUp "1234567812345678"

procedure UpBetween(p1, p2)

   DomainUp ? {
      tab(upto(p1[–1]) + 1)
      return tab(upto(p2[1]))
      }

end

procedure UpRun(p, cycles)

   DomainUp ? {
      tab(upto(p[–1]) + 1)
      return repl(move(*Domain), cycles)
      }

end
```

As you might expect, there is a corresponding "downto" operator, >, for which there are similar procedures. And, as an abbreviation, the operator –– can be used in place of both < and >, provided the last character of the left operand is strictly less than or greater than the first character of the right operand, respectively. For example, 21––82 can be used in place of 21>82, and 28––12 can be used for 28<12.

The following procedure, which implements ––, selects the appropriate "direction":

```
procedure UpDownto(p1, p2)
   local c

   p2 ? {
      tab(many('\"))
      c := move(1)          # first non–tick character
      }

   if p1[–1] << c then return Upto(p1, p2)
   else return Downto(p1, p2)

end
```

Strictly speaking, identical end characters should be an error. We didn't bother with that, but just let Downto() handle that case, which should never arise.

We have run out of room. (Thank heavens, you say.) We'll finish up our description of the Painter weaving language in the next issue of the Analyst.

What's in Store

One more article will take care of Painter's weaving language, but as usual, this investigation has led to other things.

To begin with, our pattern-form language needs revisiting in light of weaving expressions that have broad applicability.

We also plan to explore weaving grammars and produce an application for creating weaving specifications from numerical sequences.

Acknowledgment

The loom shown in Figure 2 is an 8-shaft "Mighty Wolf" manufactured by Schacht Spindle Company <11> and is used by permission.

We scanned the image from their catalog and added the labels.

A color version is on the Web site for this issue of the Analyst.

References

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

2."Character Patterns", Icon Analyst 48, pp. 1-7.

3."Character Patterns", Icon Analyst 49, pp. 1-6.

4."Analyzing Character Patterns", Icon Analyst 50, pp. 1-7.

5. *Learning to Weave*, Deborah Chandler, Interweave Press, 1995.

6. *The Weaver's Book: Fundamentals of Handweaving*, Harriet Tidball, Macmillan, 1961.

7. *Advanced Weaving*, PDF on Painter 5 CD-ROM, 1998.

Links

1.WinWeave (for Windows):
http://www.contrib.andrew.cmu.edu/~keister/winweave.html

2.WeaveIt (for Windows):
http://www.weaveit.com/support.htm

3. FiberWorks PCW (for Windows and Macintosh):
http://www3.sympatico.ca/fiberworks.pcw/

4.Patternland Weave Simulator (for Windows):
http://www.mhsoft.com/

5. ProWeave (for Windows and Macintosh):
http://www.proweave.com/

6.WEAVE for Windows:
http://www.ghgcorp.com/stilgar/shuttleworks/WFW.html

7.WeavePoint (for Windows):
http://www.avlusa.com/TDS/SOFTWARE/WeavePoint.html

8. SwiftWeave (for Macintosh):
http://www.swiftweave.com/

9.WeaveMaker One (for Windows and Macintosh):
http://www.weavemaker.com/

10. Painter (for Windows and Macintosh):
http://www.metacreations.com

11.Schacht Spindle Company:
http://www.schachtspindle.com

Animation — Reversible Drawing

Motion is the first seduction of the eye.

— R. Shamms Mortier [1]

To *animate* literally means to bring alive. In terms of computer-generated images, the word has come to refer to any image that changes over time — whether it be fluid motion in a scene or just stationary stars that twinkle.

In this article and ones to follow, we'll concentrate on apparent motion. The word apparent is important — perceived motion in an image is an illusion created by the human visual system.

There are basically three mechanisms for producing animated images using Icon: reversible drawing, mutable colors, and image replacement. There are, of course, hybrids and various *ad hoc* ways of creating animations. We'll concentrate on the basic methods, starting with reversible drawing in this article and then going on to the others in subsequent articles.

Reversible Drawing

With the graphics context attribute drawop set to "reverse", as in

```
WAttrib("drawop=reverse")
```

a drawing operation that is done a second time "erases" the first drawing, restoring the canvas to its state before the first drawing. By performing the drawing operation again, but at a slightly different place, the drawing appears to move there. Continuing this process produces the illusion of motion. The steps in the animation are called frames.

This is illustrated by the following program, which produces an animation of a small body orbiting a larger one in an elliptical orbit.

```
link graphics

record Point(x, y)

$define  A      150      # major semi-axis
$define  B      100      # minor semi-axis
$define  X      200      # center
$define  Y      200
$define  Orad     5      # radius of orbiting body
$define  Srad    13      # radius of orbited body
$define  Delay   10      # delay between frames

procedure main()
   local focus_x, distance, point, new_point, old_point

   WOpen("size=" || (2 * X) || "," || (2 * Y))

   #  Trace the orbit.

   every point := ellipse(X, Y, A, B) do
      DrawPoint(point.x, point.y)

   #  Draw the orbited body.

   focus_x := X + sqrt(A ^ 2 – B ^ 2)

   FillCircle(focus_x, Y, Srad)

   WAttrib("drawop=reverse")

   #  Animate the orbit.

   until WQuit() do {                  # new frame
      every new_point := ellipse(X, Y, A, B) do {
         FillCircle((\old_point).x, old_point.y, Orad)
         FillCircle(new_point.x, new_point.y, Orad)
         distance := sqrt((focus_x – new_point.x) ^ 2 +
            (Y – new_point.y) ^ 2)
         WDelay(Delay)
         old_point := new_point
         }
      }

end

procedure ellipse(x, y, a, b)
   local incr, theta

   every theta := 1 to 360 do
      suspend Point(
         x + a * cos(dtor(theta)),
         y + b * sin(dtor(theta))
         )

end
```

Figure 1 shows one frame from the animation and Figure 2 shows some reduced snapshots taken every 10 frames.

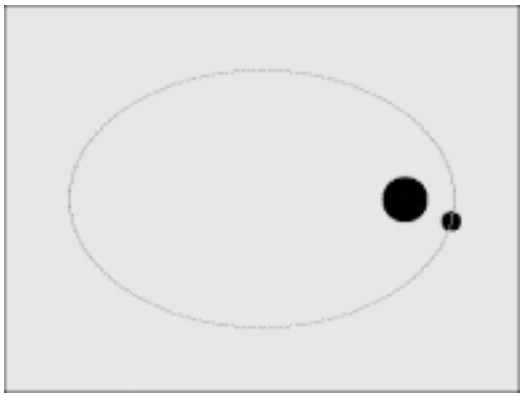


Figure 1. Orbital Animation Frame

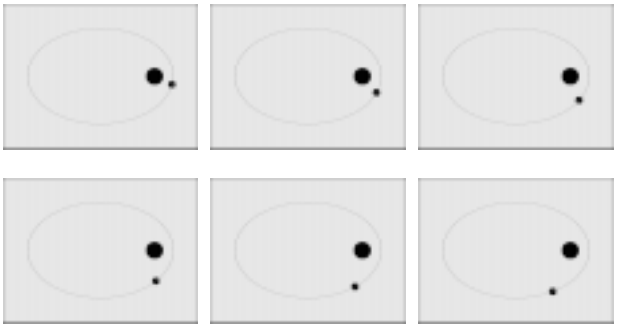


Figure 2. Orbital Animation Snapshots

To see the actual animation, visit the Web page for this issue of the Analyst.

The delay between the frames of the animation is necessary to avoid a meaningless blur with artifacts that result because the graphics system can't keep up with the computation.

The constant delay produces an animation that does not obey the laws of physics for orbiting bodies. You might try your hand at fixing this. *Hint:* To a first approximation, the velocity *v* of a body orbiting another is proportional to *sqrt*(1 / *r*), where *r* is the distance between the centers of the two bodies.

Reversible drawing works for DrawString() (but not for WWrite()). DrawString() therefore can be used to animate text. Here's a program in the same vein as the previous one, but with text circling a blinking "light". Snapshots are shown in Figure 3.

```
link graphics

record Point(x, y)

$define X              200     #  center
$define Y              200
$define Delay           30     # frame delay
$define Message    "Awake!"
$define Radius          40     # radius of blinker
$define Ring            44     # radius of ring
$define Modulus         12     # blinker delta
$define Orad           120     # orbit radius

procedure main()
  local new_point, old_point, count

  WOpen("size=" || (2 * X) || "," || (2 * Y),
    "bg=light gray", "drawop=reverse",
      "font=Helvetica,42") |
        stop("*** cannot open window")

  # Create background for blinker.

  FillCircle(X, Y, Ring)

  #  Animate the display.

  count := 0

  until WQuit() do {
    every new_point := circle(X, Y, Orad) do {
      CenterString((\old_point).x,
        old_point.y, Message)
      CenterString(new_point.x, new_point.y,
        Message)
      count +:= 1
      if count % Modulus = 0 then
        FillCircle(X, Y, Radius)
      WDelay(Delay)
      old_point := new_point
      }
    }
end

procedure circle(x, y, r)
  local theta

  every theta := 1 to 360 do
    suspend Point(
      x + r * cos(dtor(theta)),
      y + r * sin(dtor(theta))
      )
end
```

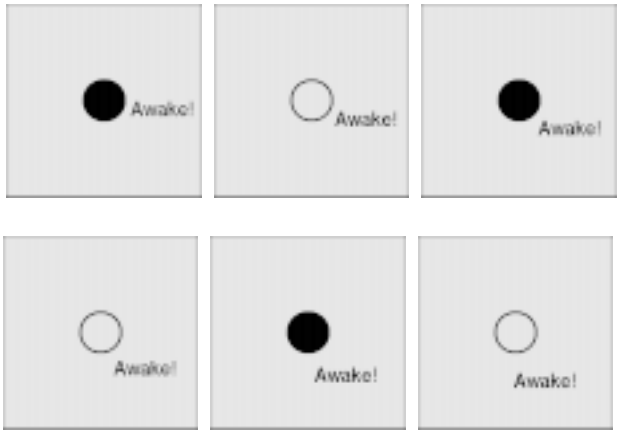


Figure 3. Wake Up Animation Snapshots

Many other animation effects can be accomplished using reversible drawing. See Reference 2 for a "bouncing ball" example. Here are some things you might want to try yourself:

- a spinning propeller
- a walking stick figure
- an erratically moving target for which points are scored when the user manages to click on it.

Problems with Reversible Drawing

Unfortunately, reversible drawing is not supported (or does not work properly) on all platforms. It works on most UNIX platforms running the X Window System, but it does not work correctly on Microsoft Windows platforms.

There is a subtle, underlying problem on platforms on which reversible drawing does work. While it does what you'd expect with fixed foreground and background colors, reversible drawing produces unpredictable colors when drawn over colors that are neither the current foreground or background ones. The colors that result may depend on the history of program execution or even on other applications that share the screen. The good news is that drawing a second time restores the canvas to whatever it was before the first one. In other words, it erases the drawing, but the colors between the two drawings can't be predicted.

We used reversible drawing in the kaleidoscope application [3], and, in fact, the colors that this application produces are unpredictable. That doesn't matter, since the application is just a visual amusement and the specific colors aren't important.

For the examples given in this article, bizarre effects may occur if other colors are present. For example, if the background color is changed after drawing the first body, as in

```
    ...
FillCircle(focus_x, Y, Srad)
Bg("red")
WAttrib("drawop=reverse")
    ...
```

the color of the orbiting body may change to something unrelated, such as white as shown in Figure 4. Notice that the orbit shows through the orbiting body. Although you can't see it here, the portion of the orbit "behind" the orbiting body is red.

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/ia51/

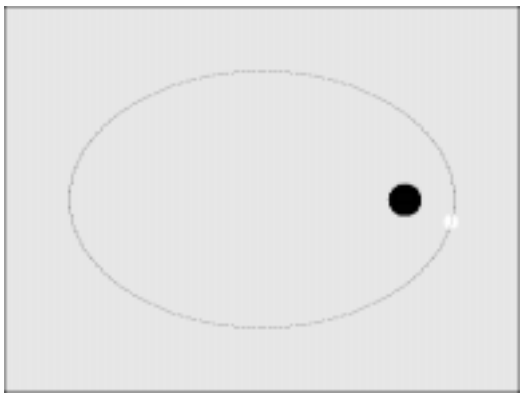


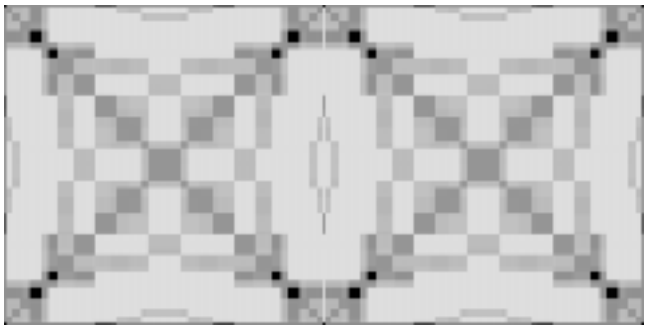
Figure 4. Color Artifact in Reversible Drawing

References

1. *The Bryce 3D Handbook*, R. Shamms Mortier, Charles River Media, 1998.

2. *Graphics Programming in Icon*, Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend , Peer-to-Peer Communications, Inc., 1998, pp. 96-97.

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



Graphics Corner — More Fun with Image Strings

In the last article on image strings [1], we showed how the pixels in an image could be rearranged to produce transformations such as rotations and flips. The other main category of operations on image strings does not rearrange pixels but rather changes their colors by mapping.

A simple example of this is creating a "negative" of a grayscale image. Because grayscale palettes have uniformly spaced colors from black to white, a negative can be produced by a mapping the palette characters to their reverse:

```
procedure negative(imr)
   local chars
```

```
   chars := PaletteChars(imr.palette)
   imr.pixels := map(imr.pixels, chars, reverse(chars))
   return imr
end
```

Figures 1 and 2 show examples for the g7 and g256 palettes, respectively.

Figure 1. Black and White Queens

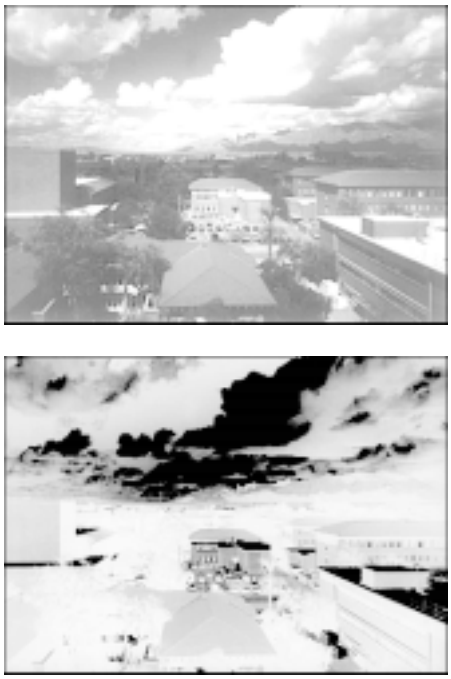


Figure 2.View, Positive and Negative

Another thing that the structure of grayscale palettes makes easy is transformations such as the following, which gradually replaces grayscale extremes by mid-range grays so that an image fades out over time:

```
link imrutils

procedure main(args)
   local imr, first, second, middle, nextolast
   local last, chars, imap, omap
   imr := imstoimr(
      $include "view.ims"
      )

   openimr(imr)

   chars := PaletteChars(imr.palette)

   while *chars > 2 do {
      drawimr(0, 0, imr)
      WriteImage("fade" || right(count, 3, "0") || ".gif")
      chars ? {
         first := move(1)
         second := move(1)
         middle := tab(–2)
         nextolast := move(1)
         last := move(1)
         chars := second || middle || nextolast
         imap := first || last
         omap := second || nextolast
         imr.pixels := map(imr.pixels, imap, omap)
         drawimr(0, 0, imr)
         }
      imr.pixels := map(imr.pixels, chars[1], chars[2])
      }

   WriteImage("fade" || right(count, 3, "0") || ".gif")

   WDone()

end
```

Figure 3 shows an original image, two intermediate stages, and the final result.

Figure 3. Fade Out

Color palettes do not lend themselves to the kinds of manipulations that grayscale ones do. Mappings that produce negatives and fadeouts are possible, but they're not easy to compute. If, for example, you map color palette characters into their reversal, you may get something interesting, but most likely it just will be bizarre.

Mapping, however, can be used to convert from one palette to another. Here's how it's done:

```
procedure imrpcvt(imr, new_palette)
   local ochars, nchars

   ochars := string(cset(imr.pixels))       # those used
   nchars := ""

   every nchars ||:= PaletteKey(new_palette,
      PaletteColor(imr.palette, !ochars))
```

```
    imr.pixels := map(imr.pixels, ochars, nchars)
    imr.palette := new_palette

    return imr

end
```

PaletteColor() produces color specifications for the characters in the pixel data. PaletteKey(), working with the new palette, gives characters for the colors that are close to the given ones. Then it's just a matter of mapping.

Of course, you cannot improve the color quality of an image by changing palettes, and you may lose a lot. For example, converting from a color palette to g2 gives you a black-and-white image according to whether a given color is closer to black or white. On the other hand, converting from a color palette to g256 gives a credible grayscale version of the image. And, in many cases, it's possible to reduce the number of colors in an image without degrading its quality noticeably.

Next Time

We're not through with image strings. We've kept their most powerful capability, transparency, for last. Although we've discussed transparency before [2], there's much more that can be done with it.

References

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

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

Generating Versum Numbers

To be a good mathematician, or a good gambler, or good at anything, you must be a good guesser.

— George Pólya [1]

Almost from the time we started working on versum numbers, we needed *n*-digit versum numbers to test hypotheses. For a while, we were able to get along with small values of *n*, typically up to $n = 6$ (it's necessary to get to 5 or 6 to get past anomalies that occur for smaller values). As things went along, we found we needed versum numbers for increasingly larger values of *n*.

For the most part we've put aside 1:… versums — versum numbers whose initial digit is 1 — because they are more complex and difficult to deal with than 2:… versums. That's still the case; this article is about 2:… versums.

We tried various ways of generating *n*:2… versums, including to trying to derive them directly from (*n*–1)- and (*n*–2)-versums. It seemed like this should work, but it didn't.

The brute-force method, which relies on testing candidates [2], helped us along, but it is impracticably slow for $n > 10$. The primary reason for this is the number of *n*:… versums. The number is given by this recurrence, which we showed earlier [3]:

$$V^2(1) = 1$$
$$V^2(2) = 1$$
$$V^2(3) = 10$$
$$V^2(n) = 19 \times V^2(n–2) \qquad n > 3$$

This recurrence can be cast as a procedure:

```
procedure v2count(n)

  return case n of {
    1 | 2     : 1
    3         : 10
    default   : 19 * v(n – 2)
    }

end
```

The actual numbers make the problem clearer:

1	1
2	1
3	10
4	19
5	190
6	361
7	3610
8	6859
9	68590
10	130321
11	1303210
12	2476099
13	24760990
14	47045881
15	470458810
16	893871739

As noted earlier [2], the brute force method requires that every candidate be tested using vpred(), which is slow. No matter how cleverly the candidates are chosen, vpred() is called more times than there are *n*-digit versums. Even for $n = 11$, the

number of calls is well in excess of 1,303,210. In fact, with the best candidate generator we have, it's many times that.

It was our hope when we studied versum deltas [2,4] that they would give us an algorithmic way of generating *n*:2… versums. For n_e, we were able to generate the *set* of deltas algorithmically and hence reduce the number of candidates, but we were not able to generate the *sequence* of deltas algorithmically, leaving us with the vpred() problem.

It took us a while (a.k.a. way too long) to realize that the way to avoid vpred() is to generate *n*:2… versums from *predecessor* candidates.

Again, in the absence of finding a direct method of generating predecessors, we resorted to the brute-force candidate approach. Here, however, the test is much simpler. The reverse sum of any positive number is versum by definition. The only question is whether the first digit of the reverse sum is 2.

Every *n*:2… versum has a predecessor that begins with a 1 (some *n*-digit numbers that begin with a 2 and end with a 0 have reverse sums that are *n*:2…, but these numbers all have equivalents that begin with a 1). Not only that, but all the predecessors have one of two forms: 10…1 or 1…09. So to generate the predecessors of the *n*:2… versums, it is necessary to test only $2\times10^{n-3}$ candidates.

A program to generate the *n*:2… versums using the predecessor method is:

```
procedure main(args)
   local i, j, n, m, lo, hi, limit

   n := args[1] | 6                  # 6 is testing default

   lo := 2 * (10 ^ (n – 1)) – 1      # 1999...
   hi := 3 * (10 ^ (n – 1))          # 3000...

   limit := 10 ^ (n – 3)) – 1

   every m := right(0 to limit, n – 3, "0") do {
      every i := ("1" || m || "90") | ("10" || m || "1") do {
         j := i + reverse(i)
         if lo < j < hi then write(j)
         }
      }

end
```

There is a problem, however. This method produces duplicates — lots of them, as we'll show later.

It might seem easy to get rid of duplicates: Just store the versums in a set and output the results at the end of the program:

```
                  …
   results := set()
                  …
   every m := right(0 to limit, n – 3, "0") do {
      every i := ("1" || m || "90") | ("10" || m || "1") do {
         j := i + reverse(i)
            if lo < j < hi then insert(results, j)
            }
         }

   every write(!sort(results))
                  …
```

That's fine for small *n*, but for values of interest, it requires far too much memory. Refer to the figures given earlier.

An alternative is just to generate all the results and use a sorting utility to remove the duplicates. On a UNIX platform,

```
sort –u
```

does this.

This sounds good until you try to do it — or better, think about it. Not only does the number of *n*:2… versums become large as *n* increases, but the amount of duplication increases much faster: The total number of values produced is $19\times10^{n-4}$.

The increase in the numbers of *n*:2… versums as *n* increases is 10.0 for n_o and 1.9 for n_e, with an average of 5.26. So, on average, the number of *n*:2… versums is 5.26^n. The ratio of $19\times10^{n-4}$ and 5.26^n becomes very large. For $n = 10$, it is 145.79 and for $n = 20$, it is 5.89×10^5. Put the other way around, not only does the number of values produced become impossibly large, but the yield of versums becomes vanishingly small.

At this point we envisioned several possible ways of dealing with the problem, as shown in Figure 1.

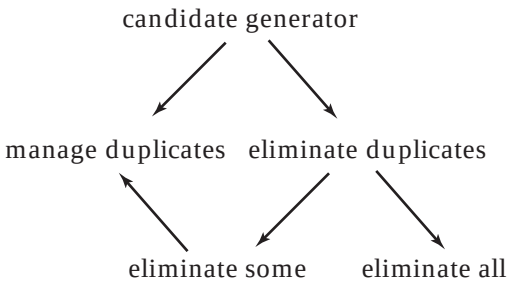


Figure 1. The Problem Graph

It might seem as if managing the duplicates is a simple if tedious process — just break up the generation into pieces, remove the duplicates in the pieces, merge the results, and repeat until it's all done. In fact, you either run out of disk space for the intermediate results and sorting or you plunge into bookkeeping hell. So, what about eliminating duplicates?

One way to characterize duplicates is by the "middles", *m*, that produce the same result. If we divide the problem into two parts — 10…1 and 1…09 predecessors — then each *m* produces at most one 2… versum. For each case, this puts the values of *m* in equivalence classes. The sizes of these equivalence classes plotted against the sorted versums they produce have interesting patterns, which are distinctly different for the two cases. See Figures 2 and 3.

Figure 2. The Number of *m*s for 10…1

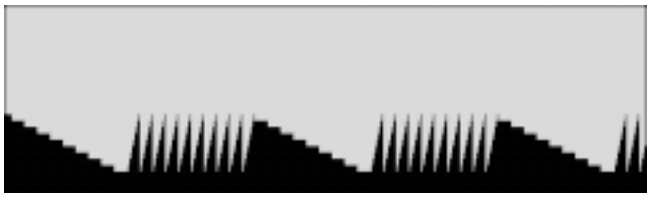


Figure 3. The Number of *m*s for 1…09

Even more interesting are the actual values of the *m*s. Here is a typical portion for the 10…1 part of the 6:2…versum generation, with the versum in the left column followed by the *m*s that produce it:

```
200002 000
210012 001
220022 002
230032 003
240042 004
250052 005
260062 006
270072 007
280082 008
290092 009
201102 010 100
211112 011 101
221122 012 102
231132 013 103
241142 014 104
251152 015 105
261162 016 106
271172 017 107
281182 018 108
291192 019 109
202202 020 110 200
212212 021 111 201
222222 022 112 202
232232 023 113 203
242242 024 114 204
252252 025 115 205
262262 026 116 206
272272 027 117 207
282282 028 118 208
292292 029 119 209
203302 030 120 210 300
213312 031 121 211 301
```

Here's a portion for the 1…09 part:

```
201091 019 028 037 046 055 064 073 082 091
202191 029 038 047 056 065 074 083 092
203291 039 048 057 066 075 084 093
204391 049 058 067 076 085 094
205491 059 068 077 086 095
206591 069 078 087 096
207691 079 088 097
208791 089 098
209891 099
200101 100
201201 101 110
202301 102 111 120
203401 103 112 121 130
204501 104 113 122 131 140
205601 105 114 123 132 141 150
206701 106 115 124 133 142 151 160
207801 107 116 125 134 143 152 161 170
208901 108 117 126 135 144 153 162 171 180
210001 109 118 127 136 145 154 163 172 181 190
211101 119 128 137 146 155 164 173 182 191
212201 129 138 147 156 165 174 183 192
213301 139 148 157 166 175 184 193
214401 149 158 167 176 185 194
215501 159 168 177 186 195
216601 169 178 187 196
217701 179 188 197
218801 189 198
219901 199
210111 200
211211 201 210
212311 202 211 220
213411 203 212 221 230
```

If you look at these closely, you'll see that the differences (Δs) for equivalent *m*s all are 90 for the 10…1 part and 9 for the 1…09 part. If this were to hold true for other values of *n*, we could do the following:

If an *m* produces a 2… versum, see if $m + \Delta$ does. If it does, don't produce the versum but wait for $m + \Delta$ (or its "successor") to come around. If not, produce the versum.

Unfortunately, there is more than one Δ for equivalent *m*s as *n* increases. Here are the first few:

form	Δs	*number*
6:10...1	9	774
6:1…09	90	765
7:10…1	990	7695

7:1…09	99	7695
8:10…1	990	76995
	1890	774
	2790	1539
	3690	1539
	4590	1539
	5490	1539
	6390	1539
	7290	1539
	8190	1539
	9090	1539
	9990	1539
8:1…09	90	76905
	189	765
	279	1539
	369	1539
	459	1539
	549	1539
	639	1539
	729	1539
	819	1539
	909	1539
	999	1539

If you count, you'll see there is one Δ for 6 and 7, and 11 for 8. Can you guess the number for 9? Right, it's 11. What about 10 and 11? It's 111; there are patterns everywhere.

There are patterns in the values of the Δs, and their numbers, and the Δs of successive Δs, but they become more complex as *n* increases. In terms of eliminating most of the duplicates, the smallest Δ does nicely, occurring much more often than the others. And we have formulas for the smallest Δs:

10…1, n_e: $9\times10^{(n/2)-2}$

10…1, n_o: $99\times10^{((n-1)/2)-2}$

1…09, n_e: $9\times10^{(n/2)-3}$

1…09, n_o: $99\times10^{((n-1)/2)-3}$

(We guessed these.)

Here's how the Δs are used:

```
                …
   every m := right(0 to limit, n – 3, "0") do {
      every i := ("1" || m || "90") | ("10" || m || "1") do {
         j := i + reverse(i)
         k := "10" || (m + delta) || "1"
         if k + reverse(k) = j then next       # wait
         if lo < j < hi then write(j)
         }
                …
```

This drastically reduces the number of duplicates and we might have stopped there but for an interesting property of equivalent *m*s: They all have the same digit sums. Of course, it's not that simple: Some inequivalent *m*s also have the same digit sums.

Equivalent digit sums reminded us of equivalent versum numbers, which have equivalent digit sums [5]. The *m*s are not, of course, necessarily versums, but early in the study of versum numbers, we developed a procedure, vprimary(), for finding the smallest of a set of equivalent versum numbers:

```
procedure vprimary(s)

   return vprimary_(s, 1)

end

procedure vprimary_(s, low)
   local h, mpart, lpart, rpart

   if (*s < 2) | (s = 0) then return s
   else {
      s ? {
         lpart := tab(2)
         mpart := tab(–1)
         rpart := tab(0)
         until (lpart = low) | (rpart = 9) do {
            lpart –:= 1
            rpart +:= 1
            }
         return lpart || vprimary_(mpart, 0) || rpart
         }
      }

end
```

This procedure works for equivalent *m*s! So now we can eliminate all duplicates. Here's the program for the 10…1 form; there is a similar one for the 1…09 form:

```
link vprimary

procedure main(args)
   local i, j, n, m, lo, hi, limit

   n := args[1] | 6

   lo := 2 * (10 ^ (n – 1)) – 1              # 1999...
   hi := 3 * (10 ^ (n – 1))                  # 3000...
   limit := 10 ^ (n – 3)) – 1

   every m :=
      right(0 to limit, n – 3, "0") do {
         every i :=  ("10" || m || "1") do {
            if i ~= vprimary(i) then next
            j := i + reverse(i)
```

```
            if hi > j > lo then write(j)
            }
         }
end
```

This approach to generating *n*:2... versums is a vast improvement over previous ones. Granted, the calls of vprimary() take some time, but vprimary() is much faster than vpred().

There may be a more efficient way of generating *n*:2... versums — in fact, we think there is an algorithmic method we've just not had the wit to see. If we discover such a method, we probably will kick ourselves, hard, for all the work we put into other methods. That work, however, will not have been a total loss. We learned a lot about guessing and methods of approaching such problems, as well as about programming techniques.

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

References

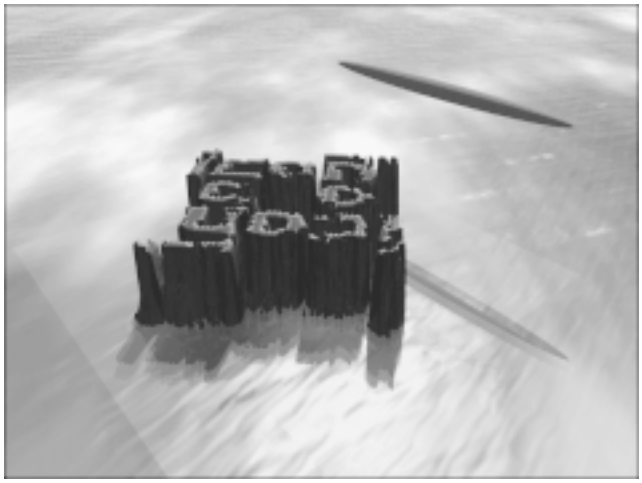
1. *Mathematics and Plausible Reasoning*, George Pólya, Princeton University Press, 1954.

2. "Versum Deltas", Icon Analyst 49, pp. 6-11.

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

4. "Versum Deltas", Icon Analyst 50, pp. 7-10.

5 "Equivalent Versum Sequences", Icon Analyst 32, pp. 1-6.



What's Coming Up

In the next issue of the Analyst, we plan to have another article on weaving and an article on using mutable colors to produce animations.

The weaving language we're describing has led to some second thoughts on pattern forms. They are scheduled for the next issue also.

The plots on pages 2-4 of this issue were done in PostScript using a package from the Icon program library that provides PostScript emulation for Icon's drawing functions. That will be the subject of a **From the Library** article.

We also feel a quiz coming on.