The Icon Analyst

In-Depth Coverage of the Icon Programming Language and Applications

April 2001
Number 65

In this issue

T-Sequence Collation

Generalized Collation

Although most collations are simple, with one term coming from first sequence, the next from second sequence, and so on, this is not always the case. For example, two terms may come from one sequence, three from another, and in all sorts of variations.

We'll handle this by introducing an index sequence whose terms are indexes that determine from which sequences the terms of the collation come.

We'll use the notation

$$\overset{I}{\sim}(S_1, S_2, S_3, \ldots, S_n)$$

for generalized collation, where I is a sequence of values in the range 1 to n and determines in order the sequence from which the next term of the collation is taken. If I runs out before any of S_1, S_2, S_3, …, S_n do, then I is repeated.

For example, if

$$I = 1, 1, 2, 2, 2$$

then

$$\overset{I}{\sim}(1 \to 6, 5 \to 2)$$

produces

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

The default for I is 1, 2, … n, where there are n sequences to be collated.

Layered Collation

In the examples in the previous article, the sequences used in collation were on disjoint sets of shafts. Although this need not be the case, it often is because different sets of shafts frequently serve different purposes in weaving.

Such T-sequences are composed by collation, but the specific shafts used are largely arbitrary, provided the sequences in the collation are on disjoint sets of shafts.

In constructing such *layered* collations, a sequence may be offset much in the manner of a motif along a path. If such a collation is built up from the bottom, for example, shafts 1 to m may be used for one sequence and then the next sequence is offset by m. Since the offset of the second sequence depends on the bound on the first sequence, it is awkward to describe such sequences using the collation operation.

Instead, we will add a *layering* operation,

$$\backslash(S_1, S_2, S_3, \ldots, S_n)$$

in which the sequences are collated but S_2 is offset by $\beta(S_1)$, S_3 is offset by $\beta(S_1) + \beta(S_2)$, and so on.

For example,

$$\backslash(\overline{1 \to 3}^{2}, \overline{1,2}^{3})$$

is equivalent to

$$\sim(\overline{1 \to 3}^{2}, \overline{4,5}^{3})$$

and produces

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

As with collation, an indexing sequence can

be added to specify the order in which terms are taken:

$$\backslash^{l}(S_1, S_2, S_3, \ldots, S_n)$$

Implementation

Here are the procedures for general collation and layering:

```
procedure scollate(indices, args[])
   local lseq, i

   /indices := srun(1, *args)

   args := copyl ! args

   lseq := []

   every i := !|indices do
      put(lseq, get(args[i])) | break

   return lseq

end

procedure slayer(indices, args[])
   local lseq, i, shift

   args := copyl ! args

   shift := sbound ! args[1]

   every i := 2 to *args do {
      bound := sbound ! args[i]
      every !args[i] +:= shift
      shift +:= bound
      }

   push(args, indices)

   return scollate ! args

end
```

"Use Tabby"

For a person not familiar with weaving, one of the most puzzling aspects of weave drafts is the omission of a component that is taken for granted by experienced weavers — using tabby.

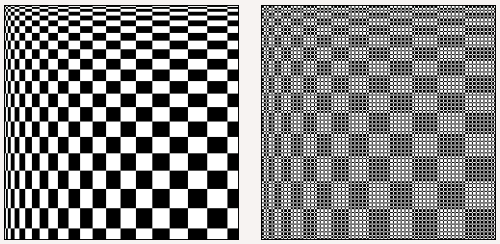
Tabby weave, also called plain weave, is a strict one-over, one-under interlacing. One purpose tabby serves in combination with more complex interlacements is to add strength and integrity to the fabric. If tabby is alternated with another interlacement, there can't be any long floats.

Omitting the tabby part of a weave from a draft (including omitting the necessary component of the tie-up) can considerably reduce the size of the draft. This is an important consideration, since otherwise many drafts would be too large to fit on a printed page.

In drafts where the tabby parts are omitted or greatly abbreviated, you'll often see the phrase "use tabby" by portions of the draft.

In addition to alternating tabby with another interlacement, many weaves have sections of tabby — plain weave between panels of patterns.

The effect of alternating tabby with a pattern can be seen in the patterns below. The pattern at the left is from the "multi" sequence, 1, 2, 2, 3, 3, 3, 4, 4, 4, 4… . A fabric woven with just this sequence would have unacceptably long floats. Alternating tabby, as shown in the pattern at the right, produces a strong fabric with short floats. Notice how the pattern is spread out.



Of course, it's not necessary to use tabby between every interlacement of another kind. The choice of using tabby and if so, how, depends both the need to strengthen the fabric and the visual effect the tabby produces.

Constructing T-Sequences

Most of the T-sequence operations we've described so far can be used for constructing, describing, and analyzing T-sequences. Most of these operations, such as repeats and palindromes, leave recognizable traces — you can detect their work in the final result. Of course, most T-sequences can be constructed in a variety of ways, so determining the tools of construction is problematical.

There are, however, operations that leave no visible trace. One operation that we discussed some time ago is taking residues of T-sequences [1]. There is no way, in general, to tell whether or not a given T-sequence is the residue sequence of another sequence and, if so, from what sequence.

In this article, we'll introduce operations that fall into this category — ones useful for constructing T-sequences but that do not leave visible traces of their origins.

Term Selection

Term selection extracts specified terms from a sequence to form a new sequence.

We'll use the notation

$S\{T\}$

for term selection. It produces a sequence of the terms of S in positions specified by T. Terms are produced in the order they are given in T, and T may contain duplicate values. For example, if

$S = \rightarrow(1, 8, 1, 7, 2, 1)$

then

$S\{(\rightarrow(1, 6, 1), \rightarrow(16, 26))\}$

produces

1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 2, 3, 4, 5, 6, 7,
6, 5, 4, 3, 2, 1

See Figures 1 and 2.

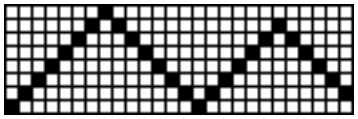


Figure 1. $S = \rightarrow(1, 8, 1, 7, 2, 1)$

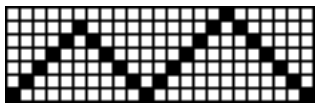


Figure 2. $S\{(\rightarrow(1, 6, 1), \rightarrow(16, 26))\}$

As another example,

$S\{3\}$

produces a unit sequence consisting of the third term of S.

As in Icon, negative position values are taken with respect to the right end of S. For example,

$S\{-1\}$

produces a unit sequence consisting of the last term in S.

If a term in T does not correspond to a position in S (that is, if it is out of bounds), it is ignored.

Modular Reduction

We have discussed residue sequences at some length in an earlier article [2]. In the present context, we'll use the notation

$S \equiv i$

for producing the modular reduction of S, shaft-modulo i. For example,

$\rightarrow(1, 10, 3, 12) \equiv 8$

produces

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

See Figures 3 and 4.

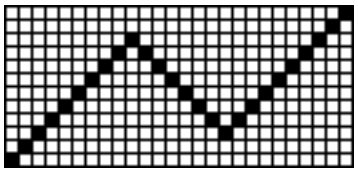


Figure 3. $\rightarrow(1, 10, 3, 12)$

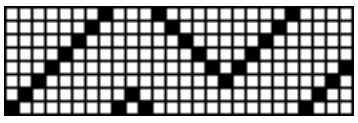


Figure 4. $\rightarrow(1, 10, 3, 12) \equiv 8$

Mapping

Mapping the values in a sequence to other values is the sequence equivalent of map() for strings.

For mapping, we'll use the notation

$\otimes(S, T, U)$

For example,

$$\otimes(\rightarrow(1, 8, 1), \rightarrow(4, 6), \rightarrow(6, 4))$$

produces

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

See Figures 5 and 6.

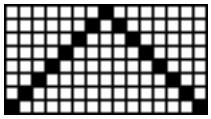


Figure 5. →(1, 8, 1)

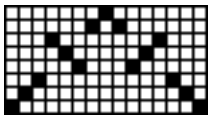


Figure 6. ⊗(→(1, 8, 1), →(4, 6), →(6, 4))

Mutation

In a recent **Graphics Corner** [3], we discussed permutations — the rearrangement of terms in a sequence — and the more general "mutations", which allow also for duplications and deletions.

We'll use the notation

$$S \oplus T$$

for mutation. For example, if

$$S = \rightarrow(1, 7, 3, 8, 1)$$

then

$$S \oplus (\rightarrow(7, 1), \rightarrow(22, 16)))$$

produces

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

See Figures 7 and 8.

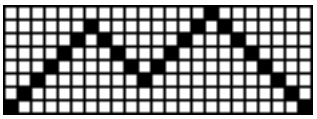


Figure 7. *S* = →(1, 7, 3, 8, 1)

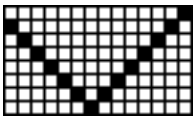


Figure 8. *S* ⊕ (→(7, 1), →(22, 16)))

Parity Adjustment

In an earlier article [4], we mentioned that some kinds of weaving require alternating parity in T-sequences: odd, even, odd, even … or even, odd, even, odd … . We can represent parity by sequences of 2s (even) and 1s (odd), as in $\overline{1,2}$ and $\overline{2,1}$. Although it's speculative, we can consider more general parity sequences, such as

$$\underline{2}_4, \underline{1}_8, \underline{2}_4$$

Since all that matters in parity is the residue modulo 2, any sequence with appropriate residues modulo 2 can be used as a parity sequence. For example,

$$\rightarrow(1, 100)$$

is equivalent to

$$\overline{1,2}^{\,50}$$

If a sequence does not have the desired parity pattern, it can be modified so that it does, which is called *parity adjustment*.

We'll use the notation

$$S \pm T$$

to denote the result of adjusting the parity of *S* according to the parity of *T*.

There are many ways parity might be adjusted. One possibility would be to delete terms that do not have the desired parity. This is not satisfactory for weave design — if nothing else, it might produce an empty sequence. Another possibility is to add or subtract one from terms that do not have the desired parity. The problem with this is that terms in T-sequences usually have the values they do for a reason. Instead, the method of parity adjustment usually used by weavers is to add "incidental" terms to produce the desired parity pattern.

Again, there are many ways incidentals might be added. For the time being, we'll take a simple approach. If a term does not have the desired parity, an incidental term one greater than it will be inserted before it. For example,

$$(\rightarrow(1, 7) \times 2) \pm (\rightarrow(1, 7))$$

produces

1, 4, 3, 5, 8, 7, 9, 12, 11, 13

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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

See Figures 9 and 10.

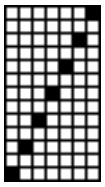


Figure 9. (→(1, 7) ×2)

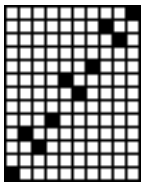


Figure 10. (→(1, 7) ×2)±(→(1, 7))

Compression

Some T-Sequences do not have all the values in the range from 1 to their bounds. This may happen because a draft does not require as many shafts as a loom has. See Figure 11.

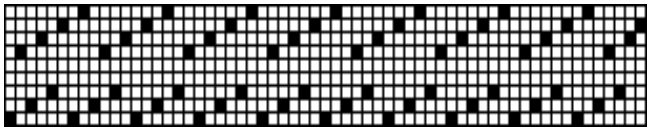


Figure 11. A T-Sequence with Missing Terms

It may be desirable to remove gaps in a T-sequence by compressing it. For example, the result clearly shows how many shafts are required. See Figure 12.

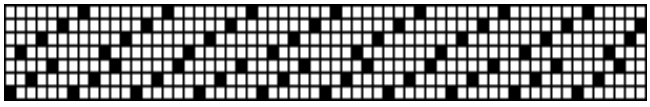


Figure 12. A Compressed T-Sequence

Compression moves shafts down as necessary to fill in gaps, starting with the lowest gap.

We'll use the notation

$$\perp S$$

to indicate the compression of S. Of course, if there are no missing terms in S, S is unchanged by this operation.

Duplicate Removal

Adjacent duplicate terms in T-sequences can cause structural problems in weaving. Duplicates may arise accidentally, as for example, when combining two separately developed sequences or by taking the residues of an integer sequence of a mathematical origin.

We'll use the notation

$$\varnothing S$$

to indicate the removal of adjacent duplicate terms. For example,

$$\varnothing((\rightarrow(8,1,8), \rightarrow(8,3,6)))$$

produces

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

See Figures 13 and 14.

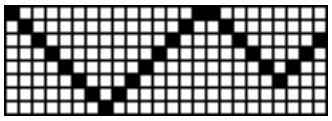


Figure 13. (→(8,1, 8), →(8, 3, 6))

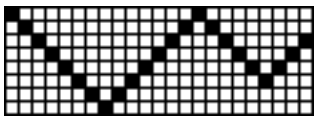


Figure 14. ∅((→(8,1, 8), →(8, 3, 6)))

Summary

In this article we have introduced the following T-sequence operations:

$S\{T\}$	*term selection*
$S \equiv i$	*modular reduction*
$\otimes(S, T, U)$	*term mapping*
$S \oplus T$	*mutation*

Supplementary Material

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

http://www.cs.arizona.edu/icon/analyst/iasub/ia65/

$S \pm T$	*parity adjustment*
$\perp S$	*compression*
$\varnothing S$	*duplicate removal*

Implementation

```
procedure sselect(x1, x2)        # term selection
   local lseq, i

   x1 := spromote(x1)
   x2 := copy(spromote(x2))

   lseq := []

   while i := get(x2) do
      put(lseq, x1[i])           # may fail

   return lseq

end

procedure smod(x, i)             # residue reduction
   local lseq

   x := spromote(x)

   lseq := []

   every put(lseq, residue(!x, i, 1))

   return lseq

end

procedure smap(x1, x2, x3)       # term mapping
   static tdefault
   local i, smaptbl

   initial tdefault := []

   x1 := copy(spromote(x1))
   x2 := spromote(x2)
   x3 := spromote(x3)

   if *x2 ~= *x3 then fail

   smaptbl := table(tdefault)    # mapping table

   every i := 1 to *x2 do        # build the map
      smaptbl[x2[i]] := x3[i]

   every i := 1 to *x1 do        # map the values
      x1[i] := (tdefault ~=== smaptbl[x1[i]])

   return x1

end

procedure smutate(x1, x2)        # mutation
   local lseq

   x1 := spromote(x1)
   x2 := spromote(x2)

   lseq := []

   every put(lseq, x1[!x2])

   return lseq

end

procedure sparity(x1, x2)        # parity adjustment
   local lseq, i, j, k

   x1 := spromote(x1)
   x2 := spromote(x2)

   lseq := []

   every i := 1 to *x1 do {
      j := x1[i]
      k := x2[i]
      if (j % 2) = (k % 2) then put(lseq, j)
      else put(lseq, j + 1, j)
      }

   return lseq

end

procedure scompress(x)           # compacting
   local unique, target

   x := spromote(x)

   unique := set(x)

   target := []

   every put(target, 1 to *unique)

   return smap(x, sort(unique), target)

end

procedure sremdupl(x)            # duplicate removal
   local lseq, i

   x := copy(spromote(x))

   lseq := [get(x)] | return []

   while i := get(x) do
      if lseq[–1] ~= i then
         put(lseq, i)

   return lseq

end
```

References

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

2. "Residue Sequences", Icon Analyst 58, pp. 5-6.

3. "Graphics Corner — Image Permutations", Icon Analyst 64, pp. 13-18.

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

Generalizing T-Sequence Operands

In the motif-along-a-path operation

$$M \mathbin{@} P$$

M is positioned at the first term in P, followed by M at the second term in P, and so on. This is, of course concatenation.

If we had first introduced the concept of changing the vertical origin of a sequence, we might have cast an offsetting operation as

$$M \uparrow i$$

Then we might have cast a motif along a path as a concatenation:

$$M \uparrow i, M \uparrow j, ...$$

where the path is $i, j, \ldots$.

Of course, $M \mathbin{@} P$ is much more compact and captures the essence of the operation in the way that the explicit concatenation does not.

But, had we started with $M \uparrow i$, we might never have thought of the second argument being a sequence instead of an integer.

On the other hand, we have already allowed the interpretation of an integer as a unit sequence, so $M \mathbin{@} i$ perfectly valid.

All this raises the question of operations that have been defined with integer operands. Consider term repetition, $\underline{S}_i$. What might $\underline{S}_T$ mean? If we use motif-along-a-path as a model, it should mean the concatenation of $\underline{S}_i$, $\underline{S}_j$, … where $T = i, j, \ldots$.

For example, if

$$S = 1 \rightarrow 3$$

and

$$T = 4, 2$$

then $\underline{S}_T$ is equivalent to

$$\underline{1 \rightarrow 3}_4, \underline{1 \rightarrow 3}_2$$

and produces

1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 1, 1, 2, 2, 3, 3

We now will adapt the view that promoting an integer argument to a sequence amounts to the concatenation of the results for all the integers in the sequence.

What are the consequences of this? For some operations, it is of no apparent use, even if it is well defined. An example is repetition, which generalizes as

$$\overline{S}^T$$

The result of this operation is

$$\overline{S}^i, \overline{S}^j, \ldots$$

where $T = i, j, \ldots$; in other words

$$\overline{S}^{(i + j + \ldots)}$$

A more interesting case is the generalization of $i \rightarrow j$ to $S \rightarrow T$. What does this mean? Presumably a concatenation, but of what and in what order, now that we have two integer operands promoted to sequences.

We basically have two choices: The concatenation of runs from the arguments of S and T in parallel or in cross-product evaluation as in Icon. That is, if

$$S = i, j, \ldots$$

and

$$T = n, m, \ldots$$

parallel evaluation gives

$$i \rightarrow n, j \rightarrow m, \ldots$$

while cross-product evaluation gives

$$i \rightarrow n, i \rightarrow m, \ldots j \rightarrow n, j \rightarrow m, \ldots$$

We generally tend to favor cross-product evaluation, if only because it is more powerful than parallel evaluation. As you probably know, cross-product evaluation permeates Icon, while parallel evaluation motivated co-expressions, which were an add-on to the original language. It is cross-product evaluation that makes Icon such an interesting and powerful programming language.

On the other hand, parallel evaluation seems appropriate in the context of T-sequences. We'll opt for parallel evaluation. For example,

$$(1 \rightarrow 6) \rightarrow (3, 6, 1, 6, 1, 2)$$

produces the result shown in Figure 1.

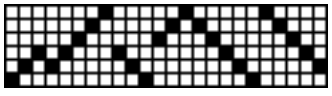


Figure 1. Generalized Runs

Runs have another feature that needs consideration — the optional increment, k:

$$i \xrightarrow{k} j$$

Consistency demands that the increment be allowed to be a sequence also:

$$S \xrightarrow{U} T$$

This operation terminates when *S* or *T* runs out. On the other hand *U* is repeated as necessary and the default for it is 1.

Summary

Here are the operations for which promotion of integer arguments to sequences applies:

$S \xrightarrow{U} T$	*run*
$\overline{S}^{T}$	*repetition*
$S \Rightarrow T$	*extension*
$\underline{S}_{T}$	*term repetition*
$S \times T$	*scaling*
$\overline{\overline{S}^{T}}$	*closed palindrome*
$S \equiv T$	*modular reduction*

Implementation

Here are the corresponding procedures:

```
procedure sruns(x1, x2, x3)     # run
   local lseq, i, j, k, limit

   x1 := copy(spromote(x1))
   x2 := copy(spromote(x2))
   x3 := copy(spromote(x3))

   lseq := []

   repeat {
      i := get(x1) | break
      j := get(x2) | break
      k := get(x3) | break
      put(x3, k)                # recycle
      if j < i then k := –k
      every put(lseq, i to j by k)
      }

   return lseq

end

procedure srepeat(x1, x2)       # repeat
   local lseq, count

   x1 := spromote(x1)

   count := 0

   every count +:= !spromote(x2)

   lseq := copy(x1)

   every 2 to count do
      lseq |||:= x1

   return lseq

end

procedure sextend(x1, x2)       # extend
   local lseq, part, i

   x1 := spromote(x1)

   lseq := []

   every i := !spromote(x2) do {
      part := []
      until *part >= i do
         part |||:= x1
      lseq |||:= part[1+:i]
      }

   return lseq

end

procedure srepl(x1, x2)         # replicate terms
   local lseq, i, j

   x1 := spromote(x1)
   x2 := spromote(x2)

   lseq := []

   every i := !x2 do
      every j := !x1 do
         every 1 to i do
            put(lseq, j)

   return lseq

end

procedure sscale(x1, x2)        # scale
   local lseq, j, i

   x1 := spromote(x1)

   lseq := []

   every i := !spromote(x2) do
      every j := 1 to *x1 do
         put(lseq, (x1[j] – 1) * i + 1)

   return lseq

end


procedure scpal(x1, x2)         # closed palindrome
   local lseq, i

   x1 := spromote(x1)
```

```
   (/x[2] := [1]) | (x2 := spromote(x2))
   i := 0
   every i +:= !x2
   lseq := srepeat(sopal(x1), i)
   put(lseq, lseq[1])
   return lseq
end
procedure smod(x1, x2)          # modular reduction
   local lseq, i
   x1 := spromote(x1)
   x2 := spromote(x2)
   lseq := []
   every i := !x2 do
      every put(lseq, residue(!x1, i, 1))
   return lseq
end
```

Solving Square–Root Palindromes II

In the last article on square-root palindromes, we showed how the solutions of square-root continued fractions can be used to provide empirical results from which more general formulas can be deduced [1]. In that article, however, we limited our investigation to continued fractions with specific numerical coefficients.

In this article, we'll look at coefficients that are variables. Figure 1 shows an example as displayed in *Mathematica*.

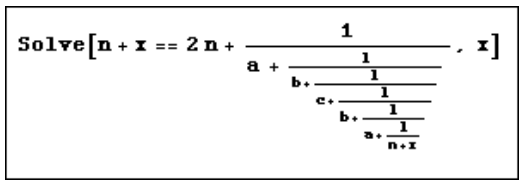


Figure 1. Continued Fraction for Palindrome *a, b, c, b, a*

In what follows, we'll use some conventions to simplify the handling of data. The letter *n* will play the role it has before in $\sqrt{n^2+m}$. Other lowercase letters, starting at the beginning of the alphabet — *a*, *b*, *c*, … — will be used as variables. See Figure 1.

The *Mathematica* solution for this continued fraction is shown in Figure 2.

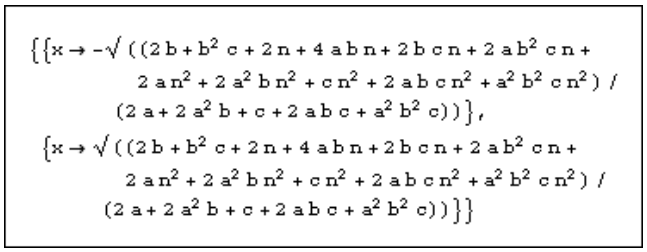


Figure 2. Solution for *a, b, c, b, a*

The first step is to convert one of the (identical) radicands in the *Mathematica* solution to a useful form.

As shown in the previous article, the internal *Mathematica* format for expressions is relatively messy. Here's what it looks like for one of the radicands in Figure 2:

```
\!\(\((\((2\ b + b\^2\ c + 2\ n + 4\ a\ b\ n + 2\ b\ c\ n +
2\ a\ b\^2\ c\ n + 2\ a\ n\^2 + 2\ a\^2\ b\ n\^2 +c\ n\^2 +
2\ a\ b\ c\ n\^2 + a\^2\ b\^2\ c\ n\^2)\)/\((2\ a + 2\ a\^2\
b + c + 2\ a\ b\ c + a\^2\ b\^2\ c)\))\)\)
```

What we want to do is convert *Mathematica*'s syntax to Icon's syntax so that we can evaluate such expressions for different values of the variables. To do this, we'll write a program, transcf.icn, which will take *Mathematica*-style input and output an Icon program evalcf.icn to do the evaluation. See Figure 3.

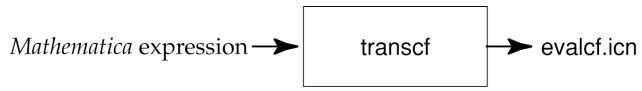


Figure 3. Processing *Mathematica* Expressions

Here, in a somewhat neater layout than actually will be produced, is what evalcf.icn will look like:

```
procedure main()
      …        # heading
  every n := 1 to 50 do {
    every a := 1 to n do {
      every b := 1 to n do {
        every c := 1 to n do {
          N := …        # translated numerator
          D := …        # translated denominator
          if (N % D) ~= 0 then next
          m := N / D    # integer result
          write(a, "      ", b, "    ", c, "    ", m,"    ", n)
          }
        }
      }
    }
end
```

The main program loops over a range for *n*. Within that loop, each of the variables ranges from 1 to *n*. (The limit on *n* for the inner loops is possible because no coefficient in a square-root continued fraction can exceed *n* [2].)

The numerator and denominator are evaluated separately and if the denominator (D) evenly divides the numerator (N), there is an integer result (m), and a solution is written.

With this in mind, we can look at transcf.icn. Most of evalcf.icn is boiler plate and is just put out literally. The more difficult part of transcf.icn lies in converting *Mathematica* syntax to Icon syntax.

Comparing the displayed form in Figure 2 with the text output, it's easy to decipher the *Mathematica* syntax. The *Mathematica* syntax for addition works as it stands. For most other operators, backslashes serve to escape operator symbols and we can just delete the backslashes. An apparent exception is multiplication. A term like 2\ b stands for 2b and the equivalent Icon expression is 2 * b. Actually, the blank as an operator symbol stands for multiplication.

We'll use replacem() from the Icon program library module strings to perform the required mappings. This procedure is called as

```
replacem(s, s1, t1, s2, t2, … sn, tn)
```

It produces the result of replacing in s all instances of s1 by t1, s2 by t2, … sn by tn.

Here's the code to convert *Mathematica* syntax to Icon syntax:

```
exp := replacem(exp,
    "\\ ", " * ",          # multiplication
    "\\^", " ^ ",          # exponentiation
    "\\(", "(",            # left parenthesis
    "\\)", ")",            # right parenthesis
    "\\/", " / ",          # division
    "\\!", ""              # ?
    )
```

The result for the example above is:

```
(((((2 * b + b ^ 2 * c + 2 * n + 4 * a * b * n + 2 * b *
c * n + 2 * a * b ^ 2 * c * n + 2 * a * n ^ 2 + 2 *
a ^ 2 * b * n ^ 2 + c * n ^ 2 + 2 * a * b * c * n ^ 2 +
a ^ 2 * b ^ 2 * c * n ^ 2)) / ((2 * a + 2 * a ^ 2 * b +
c + 2 * a * b * c + a ^ 2 * b ^ 2 * c)))))
```

We have a few more parentheses than we need, but they don't hurt anything.

There is one important simplification we need to make before going on. The form of the radicand is $n^2 + m$. We want m as a function of a, b, c, and n. You will note that the coefficient of n^2 in the numerator is the same as the denominator, as it must be to have the form $n^2 + m$. So we can get the formula for m by the simple expedient of setting n^2 to 0 (m is not a function of n^2). This can be done by adding the argument pair

```
"n ^ 2", "0"
```

as the last replacement in replacem(). Then the result of replacement is

```
(((((2 * b + b ^ 2 * c + 2 * n + 4 * a * b * n + 2 * b *
c * n + 2 * a * b ^ 2 * c * n + 2 * a * 0 + 2 * a ^ 2 *
b * 0 + c * 0 + 2 * a * b * c * 0 + a ^ 2 * b ^ 2 * c * 0
)) / ((2 * a + 2 * a ^ 2 * b + c + 2 * a * b * c + a ^ 2 *
b ^ 2 * c)))))
```

Thus, the terms that contain n^2 will evaluate to 0 and drop out.

Here is transcf.icn in its entirety. The form of the palindrome (*abcba* in our example) and the limit on the loop for *n* are given as the command-line options –n s and –l i, respectively. The form option is mandatory.

```
link options
link strings

procedure main(args)
   local exp, line, vars, limit, c, opts, form, output
   local expr1, expr2

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

   form := \opts["n"] | stop("*** no form specified")

   limit := \opts["l"] | 50            # optional loop limit

   output := open("evalcf.icn", "w") |
      stop("*** cannot open file for program")

   exp := ""

   #  Input may be on more than one line.

   while exp ||:= pretrim(read(input))

   #  Variables are guaranteed to be lowercase
   #  letters. n is guaranteed to be the solution
   #  variable.

   vars := string((cset(exp) ** &letters) -- 'n')

   #  Perform ad–hoc replacements to convert
   # Mathematica syntax to a valid Icon expression.

   exp := replacem(exp,
      "\\ ", " * ",          # multiplication
      "\\^", " ^ ",          # exponentiation
```

```
    "\\(", "(",             # left parenthesis
    "\\)", ")",             # right parenthesis
    "\\/", " / ",           # division
    "\\!", "",              # ?
    "n ^ 2", "0"            # terms in n ^ 2
    )

  #  Remove extra surrounding parentheses.

  while exp ?:= 2("(", tab(bal(')')), pos(–1))

  #  Get numerator and denominator.

  exp ? {
    expr1 := tab(upto('/'))
    move(2)
    expr2 := tab(0)
    }

  #  Write the program to look for solutions.

  write(output, "procedure main()")
  write(output, "write(output, ", image(form), ")")
  write(output, "write()")
  writes(output, "write(, ")
  every writes(output, image(!vars), ", \"\t\", ")
  write(output, "\"m\", ", "\"\t\", ",  "\"n\")")
  write(output, "write()")
  write(output, "every n := 1 to ", limit, " do {")
  every c := !vars do
    write(output, "every ", c, " := 1 to n do {")
  write(output, "N := ", expr1)
  write(output, "D := ", expr2)
  write(output, "if (N % D) ~= 0 then next")
  write(output, "m := N / D")
  writes(output, "write(")
  every writes(output, !vars, ", \"\t\", ")
  write(output, "m,", "\"\t\", ",  "n)")
  write(output, repl("}", *vars + 1)    # close nestings
  write(output, "end")

  close(output)

  # Compile and execute evalcf.icn.

  system("icont –s evalcf –x")

  #  Clean up.

  remove("evalcf.icn")
  remove("evalcf")

end
```

Here is the complete evalcf.icn for the example input above, somewhat cleaned up to make it easier to read:

```
procedure main()
write(output, "abcba")
write()
write(, "a", "    ", "b", "  ", "c", "  ", "m", " ", "n")
write()
every n := 1 to 50 do {
every a := 1 to n do {
every b := 1 to n do {
every c := 1 to n do {
N := ((2 * b + b ^ 2 * c + 2 * n + 4 * a * b * n + 2 * b *
c * n + 2 * a * b ^ 2 * c * n + 2 * a * 0 + 2 * a ^ 2 * b
* 0 + c * 0 + 2 * a * b * c * 0 + a ^ 2 * b ^ 2 * c * 0))
D := (2 * a + 2 * a ^ 2 * b + c + 2 * a * b * c + a ^ 2 *
b ^ 2 * c)
if (N % D) ~= 0 then next
m := N / D
write(a, "       ", b, "    ", c, "    ", m,"    ", n)
}}}}
end
```

The output looks like this:

```
abcba

a       b       c       m       n

1       1       2       5       4
1       2       4       6       4
2       1       3       3       4
1       2       2       9       6
1       1       4       8       7
1       2       7       10      7
2       1       6       5       7
                …
2       2       1       12      14
4       3       7       7       15
1       1       2       19      16
1       1       10      17      16
1       2       1       24      16
1       2       16      22      16
1       7       2       29      16
                …
4       8       37      18      37
6       1       2       11      37
1       1       6       41      38
1       1       10      40      38
1       2       9       52      38
1       18      2       73      38
2       1       2       28      38
2       1       14      26      38
2       4       38      34      38
2       8       19      36      38
6       1       11      11      38
1       2       5       54      39
4       8       2       19      39
1       1       2       47      40
1       1       26      41      40
1       2       40      54      40
```

```
1     4     2     66    40
1     8     10    72    40
1     19    2     77    40
2     1     6     28    40
2     1     39    27    40
3     2     26    23    40
1     4     4     67    41
            …
```

The question now is what to do with output like this? It consists of many special cases. For example, the palindrome 1, 1, 10, 1, 1 occurs for m =17 and n = 16. Special cases, in themselves, aren't particularly interesting.

If, however, there is more than one solution for the same palindrome, we can get general formulas for n and m for that palindrome from the first two solutions.

Looking at the output above, we see that the palindrome 1, 1, 10, 1, 1 occurs both for m_1 =17 and n_1 = 16 and for m_2 = 40 and n_2 = 38. The difference $m_2 - m_1$ is 23 and the difference $n_2 - n_1$ is 22. This suggests formulas of the form

$$m = 23i + C$$
$$n = 22i + D$$

for suitable constants C and D.

For $i = 1$, m_1 is 17 so $C = -6$. For $i = 1$, n_1 is 16 so $D = -6$ also (a coincidence or not?), so the conjectured formulas for the palindrome 1, 1, 10, 1, 1 are

$$m = 23i - 6$$
$$n = 22i - 6$$

These formulas test out. In fact, we can prove that they are correct.

Here's a program that processes the output of evalcf and produces general formulas for specific palindromes:

```
link lists

global cmap
global form

procedure main()
   local names, namelist, vars, oldentry, entry, line
   local m, n, candidates, solutions, i

   form := str2lst(read()) | fail    # list palindrome

   every i := 1 to *form do          # make integers
      form[i] := integer(form[i])    # have correct type

   cmap := []                        # variable coefficients

   every i := !set(form) do
      if not integer(i) then put(cmap, i)

   read() | fail                     # discard blank line

   names := read() | fail

   read() | fail                     # discard blank line

   namelist := []                    # list of variables

   names ? {                         # no tab after "n"
      while put(namelist, tab(upto('\t'))) do
         move(1)
      }

   pull(namelist)                    # remove "m"

   vars := *namelist

   candidates := table()             # table of solutions

   while line := read() do {         # parse solutions
      entry := ""
      line ? {
         every 1 to vars do
            entry ||:= tab(upto('\t') + 1)
         m := tab(upto('\t'))
         move(1)
         n := tab(0)
         }
      put(\candidates[entry], m, n) | {
         candidates[entry] := [m, n]
         }
      }

   every entry := key(candidates) do {
      solutions := candidates[entry]
      if *solutions > 2 then solve(entry, solutions)
      }

end

procedure solve(entry, solutions)
   local delta, ndelta, mdelta, noff, moff, elist, count

   ndelta := solutions[4] – solutions[2]
   mdelta := solutions[3] – solutions[1]
   noff := solutions[2] – ndelta
   moff := solutions[1] – mdelta
   if noff >=0 then noff := "+" || noff
   if moff >=0 then moff := "+" || moff

   elist := []

   entry ? {
      while put(elist, integer(tab(upto('\t')))) do
         move(1)
      }

   writes(limage(lmap(form, cmap, elist)), "\t")
```

```
    write("n=", ndelta, "*i", noff, "; m=", mdelta,
      "*i", moff)
    return
end
```

The procedures str2lst() and lmap() are from the Icon program library module strings. The former converts a string to a list of its characters. The latter is the list equivalent of the built-in function map().

Here is the output for our example:

```
[2,1,1,1,2]     n=21*i–12; m=16*i–9
[1,2,2,2,1]     n=12*i–6; m=17*i–8
[1,1,4,1,1]     n=10*i–3; m=11*i–3
[2,1,4,1,2]     n=24*i–3; m=17*i–2
[1,1,12,1,1]    n=26*i–7; m=27*i–7
[1,1,2,1,1]     n=6*i–2; m=7*i–2
[2,1,2,1,2]     n=15*i–7; m=11*i–5
[1,2,3,2,1]     n=33*i–19; m=46*i–26
[1,1,10,1,1]    n=22*i–6; m=23*i–6
[1,4,2,4,1]     n=30*i–20; m=49*i–32
[1,1,8,1,1]     n=18*i–5; m=19*i–5
[1,2,1,2,1]     n=15*i+1; m=22*i+2
[2,1,3,1,2]     n=39*i–35; m=28*i–25
[1,3,2,3,1]     n=20*i–12; m=31*i–18
[1,2,4,2,1]     n=21*i–17; m=29*i–23
[1,1,6,1,1]     n=14*i–4; m=15*i–4
[2,1,6,1,2]     n=33*i–26; m=23*i–18
```

Such results provide the basis for more general formulas. For example, the palindrome 1, 1, c, 1, 1 occurs for c = 2, 4, 6, 8, 10, and 12:

c	n	m
2	$6i - 2$	$7i - 2$
4	$10i - 3$	$11i - 3$
6	$14i - 4$	$15i - 4$
8	$18i - 5$	$19i - 5$
10	$22i - 6$	$23i - 6$
12	$26i - 7$	$27i - 7$

It requires no great leap of faith to conjecture that for

$$n = Ai + C$$
$$m = Bi + D$$

the constants are

$$C = D = -((c / 2) + 1)$$

It also seems safe to say that $B = A + 1$, nor does it take much imagination to see that $A = 2(c + 1) = 2c + 2$.

So we have the general formulas for the palindrome 1, 1, c, 1, 1:

$$n = (2c + 2)i - ((c / 2) + 1)$$
$$m = (2c + 3)i - ((c / 2) + 1)$$

Indeed, these prove out.

We could continue with other specific palindromes with variable terms. For example, consider the palindrome 2, 1, c, 1, 2. The results above give us

c	n	m
1	$21i - 12$	$16i - 9$
2	$15i - 7$	$11i - 5$
3	$39i - 35$	$28i - 25$
4	$24i - 3$	$17i - 2$
5	?	?
6	$33i - 26$	$23i - 18$

There is no solution for c = 5 even though there is a solution for c = 6. Is there really no solution for c = 5? Or is something else going on?

The problem is that our loop limit for n, 50, was not large enough to net two solutions for c = 5. If you look at the table above, you'll see a different pattern for c even and c odd. The multipliers for c odd increase considerably faster than for c even.

In fact, it's necessary to go to n = 101 to get the second solution for c = 5. (We found this out by using the brute-force method described earlier [3].) Well, why not just increase the loop limit on n?

The number of iterations on the innermost loop is k^{v+1} where k is the loop limit on n and v is the number of different variables in the palindrome. Thus, there are serious practical limits to how large k can be made and there is no *a priori* way to determine it for a particular objective.

Another problem is that as the number of different variables in the palindrome gets larger, the numerator and denominator expressions quickly get more complicated. See Figure 4.

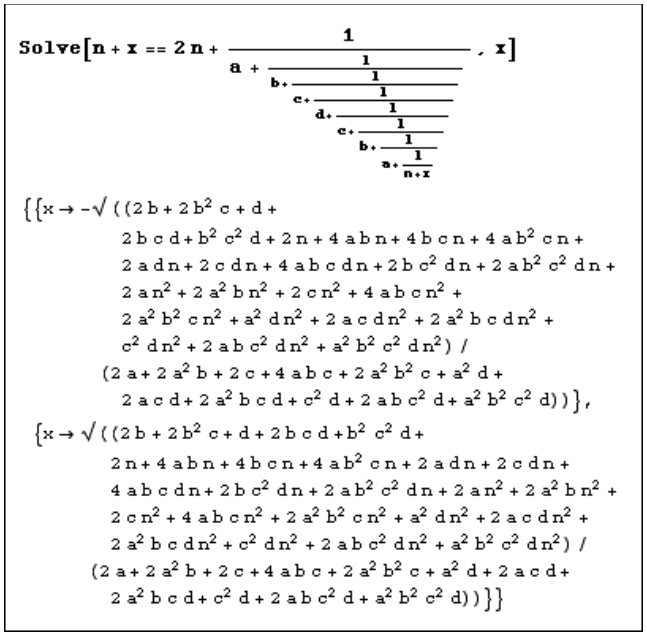


Figure 4. Continued Fraction for *a*, *b*, *c*, *d*, *c*, *b*, *a*

Of course, it's not necessary to have the initial coefficients all be different variables — or even all be variables. Figure 5 shows the continued fraction and solution for the palindrome 2, 1, *c*, 1, 2.

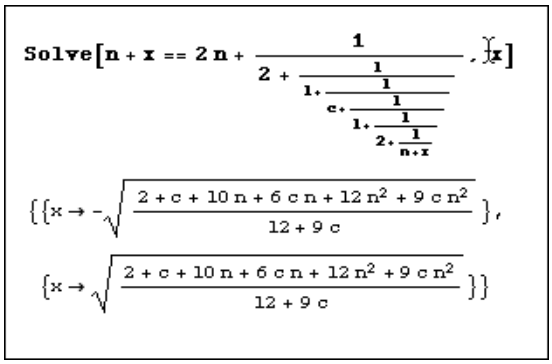


Figure 5. Continued Fraction for 2,1 ***c***, 1, 2

We tried this continued fraction for a loop limit of 200 for *n*. Here are the formulas we found:

c	*n*	*m*
1	21*i*–12	16*i*–9
2	15*i*–7	11*i*–5
3	39*i*–35	28*i*–25
4	24*i*–3	17*i*–2
5	57*i*–13	40*i*–9
6	33*i*–26	23*i*–18
7	75*i*–42	52*i*–29
8	42*i*–19	29*i*–13
9	93*i*–83	64*i*–57
10	51*i*–6	35*i*–4
11	111*i*–25	76*i*–17
12	60*i*–47	41*i*–32
13	129*i*–72	88*i*–49
14	69*i*–31	47*i*–21
15	147*i*–131	100*i*–89
16	78*i*–9	53*i*–6
18	87*i*–68	59*i*–46
20	96*i*–43	65*i*–29
22	105*i*–12	71*i*–8
24	114*i*–89	77*i*–60
26	123*i*–55	83*i*–37
30	141*i*–110	95*i*–74

It seems most likely that solutions for the missing values of *c* exist, but that we just didn't extend the search far enough to get them.

Trying to find general formulas that cover the specific ones shown above doesn't look very promising. Perhaps breaking the cases down into odd and even, or even further, would lead to results. Try your hand at this using the methods described in earlier articles. If you come up with any results, let us know in time to include them in the last issue of the Analyst, which will appear in June.

References

1. "Solving Square-Root Palindromes", Icon Analyst 64, pp. 1-6.

2. "Continued Fractions for Quadratic Irrationals", Icon Analyst 61, pp. 9-15.

3. "Constant Square-Root Palindromes", Icon Analyst 63, pp. 1-7.

Supplementary Material

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

http://www.cs.arizona.edu/icon/analyst/iasub/ia65/

The Morse-Thue Sequence

The Morse-Thue sequence is a binary fractal sequence with many interesting properties. It begins as

0, 1, 1, 0, 1, 0, 0, 1, …

This sequence was introduced in 1906 by the Norwegian mathematician Axel Thue (pronounced TOO) as an example of an aperiodic recursively computable string of symbols. Later Marvin Morse

> … proved that the trajectories of dynamic systems whose phase spaces have a negative curvature everywhere can be completely characterized by a *discrete* sequence of 0s and 1s — a stunning discovery [1].

(We quote because we don't understand it well enough to use our own words.)

Because of the importance of Morse's discovery, his name usually is listed first, although the sequence sometimes is called the Thue-Morse sequence.

Constructing the Morse-Thue Sequence

There are many ways of constructing this sequence. The one shown most often uses the *substitution map*

0 ➛ 01
1 ➛ 10

starting with the initial term 0.

This is just a simple L-System [2] and in the notation we've used for L-Systems is

```
axiom:0
0 –> 01
1 –> 10
```

The problem with producing the Morse-Thue sequence using an L-System is that each generation produces a longer initial subsequence:

01
0110
01101001
0110100110010110
01101001100101101001011001101001
…

If we want a generator that can produce an arbitrary number of terms, there's no *a priori* limit to the number of generations that will be required. This can be handled by not limiting the number of generations and using a postprocessor, but that's awkward.

Instead we can implement the substitution map directly:

```
procedure mthue1()
   local s, t

   suspend s := "0"

   repeat {
      t := ""
      every c := !s do {
         t ||:= case c of {
            "0"  :  "01"
            "1"  :  "10"
            }
         }
      suspend !right(t, *t / 2)       # new terms only
      s := t
      }

end
```

This approach requires understanding that each iteration of the loop appends to the present string and only the last half of the result is used.

This observation leads to another way to produce the Morse-Thue sequence. Start with 0 and iterate the following process: Take the present sequence and append its complement to it. (By complementing, we mean replacing 0 by 1 and 1 by 0.)

It goes like this:

0
0, 1
0, 1, 1, 0
0, 1, 1, 0, 1, 0, 0, 1
0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0
…

Here's a procedure for this method:

```
procedure mthue2()
   local s, t

   s := "0"

   suspend s

   repeat {
      t := map(s, "01", "10")
      suspend !t
```

```
      s ||:= t
      }
end
```

A third method for producing the Morse-Thue sequence is to write the nonnegative integers in binary form:

0, 1, 10, 11, 100, …

Then replace every value by its digit reduction [3] mod 2.

Here's a procedure for this method:

```
procedure mthue3()
   suspend adr(exbase10(seq(0), 2)) % 2
end
```

The procedure exbase10() is from the Icon program library module convert and converts its first argument from base 10 to the base given in the second argument. The procedure adr(), which performs the additive digital reduction, is from the Icon program library module numbers.

Before leaving the subject of implementing the Morse-Thue sequence in Icon, we feel obligated to show you this:

```
procedure mthue4()
   local i, s
   i := 0
   s := "0"
   suspend |((s[i +:= 1] |
      (s ||:= map(s, "01", "10"), s[i])) \ 1)
end
```

This procedure is from the module genrfncs in the current release of the Icon program library. It's a variant of the append-complement method.

We must have had a bad day at the Generator Factory when we wrote this procedure. (Lest you blame anyone else, this monstrosity is my work; *mea culpa*. — reg)

Properties of the Morse-Thue Sequence

The Morse-Thue sequence is self similar, as can be seen by striking out every even-numbered value, which produces the original sequence:

0, ǂ, 1, θ, 1, θ, 0, ǂ, 1, θ, 0, ǂ, 0, ǂ, 1, θ, …

Among the fascinating properties of the Morse-Thue sequence is that it is *cube-free*. This means that it does not contain the subsequences 0, 0, 0 or 1, 1, 1. But cube-free is a more general concept. In the jargon of combinatorics on words [4], a word is any sequence of characters from the alphabet being used (here, 0 and 1). Cube-free applies to all words. For example, if

W = 1, 0, 1, 1, 0

(which is a word in the Morse-Thue sequence), then W, W, W, or

1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0

does not occur in the Morse-Thue sequence.

Generalizing the Morse-Thue Sequence

The Morse-Thue sequence generalizes to bases other than 2. Using the digit reduction and residue method, it can be implemented like this:

```
procedure mthue3g(j)
   local i
   every i := seq(0) do
      suspend adr(exbase10(i, j)) % j
end
```

For example, the base-5 generalization of the Morse-Thue sequence is

0, 1, 2, 3, 4, 1, 2, 3, 4, 0, 2, 3, 4, 0, 1, 3, 4, 0,
1, 2, 4, 0, 1, 2, 3, 1, 2, 3, 4, 0, …

The procedure exbase10() can handle bases up to 36, but letters are used for bases larger than 10.

The append-complement method can be generalized also:

```
link strings
procedure mthueg2(i)
   local s, t, sub, j
   static digits
   initial digits := &digits || &lcase || &ucase
   sub := digits[1+:i] | fail
   s := "0"
   suspend s
   repeat {
      t := ""
      every j := 1 to i – 1 do
         t ||:= map(s, sub, rotate(sub, j))
```

```
    suspend !t
    s ||:= t
    }
end
```

Geometrical Interpretations of the Morse-Thue Sequence

If we visualize 0 as a black square and 1 as a white square, the Morse-Thue sequence appears graphically as shown in Figure 1:

Figure 1. The Morse-Thue Sequence

The steps for the append-complement method of construction are shown in Figure 2.

Figure 2. Morse-Thue Sequence Construction

This can be extended to two dimensions by at each step appending the complement both horizontally and vertically [5]. Figure 3 shows the first four iterations:

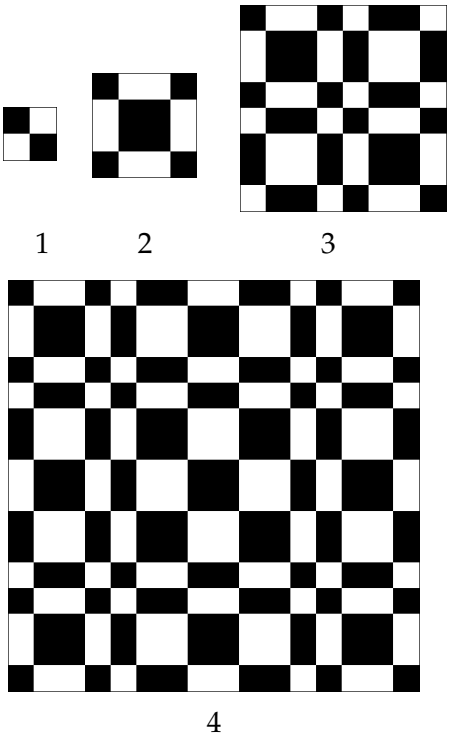


Figure 3. Constructing The Morse-Thue Plane

Like the Morse-Thue sequence itself, the Morse-Thue plane is fractal. And, despite the appearance of symmetry and regularity, there are no repetitions. That is, no finite portion of the plane can be tiled regularly to produce the whole plane.

Here's a program that produces images of the Morse-Thue plane:

```
link imrutils

procedure main(args)
  local imr1, imr2, imr3, imr4, win, i

  imr1 := imstoimr("1,g2,0")
  imr2 := imstoimr("1,g2,1")

  every i := 1 to 4 do {
    imr3 := imrcath(imr1, imr2)          # horizontal
    imr4 := imrcopy(imr3)
    imr4.pixels := map(imr4.pixels, "01", "10")
    imr1 := imrcatv(imr3, imr4)          # vertical
    imr2 := imrcopy(imr1)
    imr2.pixels := map(imr2.pixels, "01", "10")
    win := imropen(imr1)
    WriteImage(win, "mtsquare_" || i || ".gif")
    WClose(win)
    }

end
```

See Reference 6 for the procedures in the module imrutils.

The geometric interpretation of the Morse-Thue sequence extends to higher dimensions. Figure 4 shows a Morse-Thue cube.

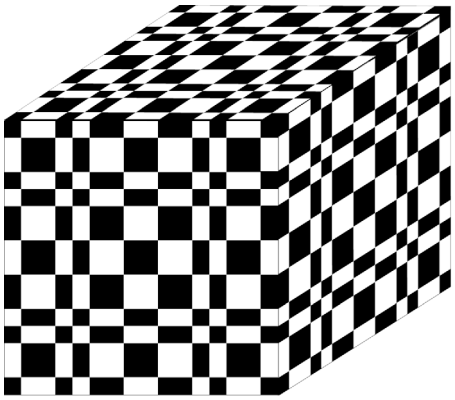


Figure 4. A Morse-Thue Cube

Try visualizing a Morse-Thue tesseract and higher-dimensional Morse-Thue cubes.

Other Applications of the Morse-Thue Sequence

The Morse-Thue sequence has applications in

many areas.

The Morse-Thue plane provides the basis for a variety of interesting weaves. Figure 5 shows a weaving draft that was "drawn up" [7] from the sixth iteration of the plane construction process shown in the last section. Notice that the Morse-Thue sequence appears in the threading and treadling.

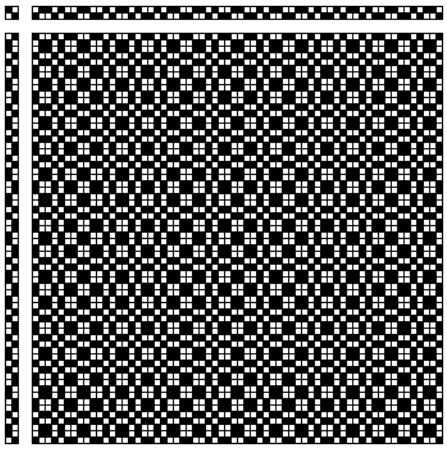


Figure 5. A Morse-Thue Weave

We have other ideas for using the Morse-Thue sequence as the basis for weaving. We'll include one of these in the next issue of the Analyst.

The Morse-Thue sequence also has been used in graphic design and in music composition [8-10].

The Fibonacci sequence is the only sequence that has more interesting, almost magical properties than the Morse-Thue sequence. Perhaps we should bring the two together.

References

1. *Fractals, Chaos, Power Laws: Minutes from an Infinite Paradise*, Manfred Schroeder, Freeman, 1991, pp. 264-269.

2. "Anatomy of a Program — Lindenmayer Systems", Icon Analyst 25, pp. 5-9.

3. "Sigma Quest", Icon Analyst 62, pp. 6-8.

4. *Combinatorics on Words: Progress and Perspectives*, Larry J. Cummings, ed., Academic Press, 1983.

5. *Gnomon: From Pharaohs to Fractals*, Midhat J. Gazalé, Princeton, 1999, pp. 223-224.

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

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

8. *Mazes for the Mind: Computers and the Unexpected*, Clifford A. Pickover, St. Martin's Press, 1992, pp. 71-77.

9. *MusiNum — The Music in the Numbers*, Lars Kindermann, http://bfws7e.informatik.uni-erlangen.de/~kinderma/musinum/musinum.html

10. *Recursion: A Paradigm for Future Music?*, Nicholas Mucherino, http://www-ks.rus.uni-stuttgart.de:/people/schulz/fmusic/recursion.html

Profile Drafting

Profile drafting [1-4] is a simple but powerful tool for designing weaves. The basic idea in profile drafting is to use a level of abstraction above individual threads and instead think of patterns of blocks, where a block consists of an unspecified sequence of threads.

For example,

A B A C A B A

is a profile sequence based on three blocks, A, B, and C. To get a conventional threading sequence, a sequence of integers is assigned to each block. Think of the blocks as being macros and the specific integer sequences as being their definitions.

For example, if A is 1, 3, 2, 4; B is 2, 3, 2, 3, 2, 3; and C is 4, 1, 3, 2, then the profile sequence above gives the threading sequence

1, 3, 2, 4, 2, 3, 2, 3, 2, 3, 1, 3, 2, 4, 4, 1, 3, 2,
1, 3, 2, 4, 2, 3, 2, 3, 2, 3, 1, 3, 2, 4

See Figure 1.

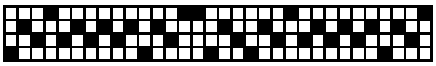


Figure 1. A Threading Sequence

On the other hand, if A is 1, 2, 3, 4; B is 3, 2, 1, 4; and C is 3, 2, 1, then the threading sequence is

1, 2, 3, 4, 3, 2, 1, 4, 1, 2, 3, 4, 3, 2, 1, 1, 2, 3,
4, 3, 2, 1, 4, 1, 2, 3, 4

See Figure 2.

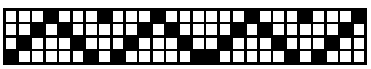


Figure 2. Another Threading Sequence

Although the two threading sequences are quite different, they have the same underlying pattern.

Actual profile drafting usually focuses on a particular kind of weave and the details of what kinds of sequences can be assigned to the blocks to produce acceptable threading sequences. This has the effect of obscuring the generality of the concept, and descriptions of profile drafting in the weaving literature often make the concept appear more difficult than it is. As we say, "think macro".

We won't attempt to get into the details of specific kinds of weaves and their requirements, such as overshot, which requires the threading sequence to have alternating parity [5].

Instead, we'll look at how profile drafting can be implemented. This will illustrate some important aspects of pointer semantics in Icon [6].

Suppose we have three blocks, A, B, and C. We can start with these blocks being empty lists:

```
A := []
B := []
C := []
```

For variety (and to make the figures easier to lay out) consider the profile sequence

```
A B A B A B C
```

We can represent this by a list of lists (a packet sequence [7]):

```
P := [A, B, A, B, A, B, C]
```

Figure 3 illustrates the data structures diagrammatically.

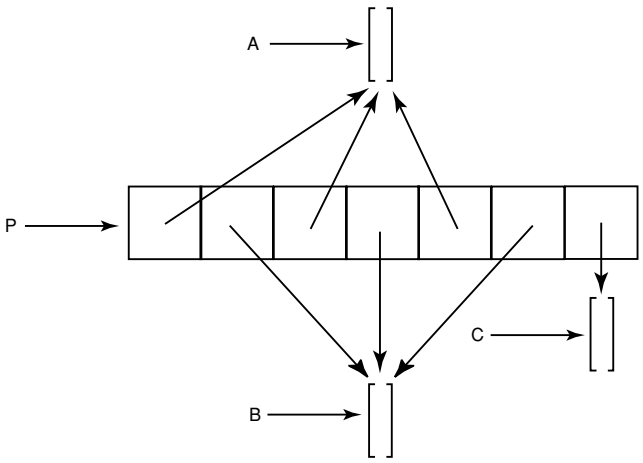


Figure 3. Data Structures for a Profile Draft

If we flatten P, we get the empty sequence because A, B, and C are empty. Now suppose we put values in A, B, and C:

```
put(A, 1, 3, 2, 4)
put(B, 2, 3, 2, 3, 2, 3)
put(C, 4, 1, 3, 2)
```

Figure 4 on the next page shows the result. Now if we flatten P, we get

1, 3, 2, 4, 2, 3, 2, 3, 2, 3, 1, 3, 2, 4, 2, 3, 2,
3, 2, 3, 1, 3, 2, 4, 2, 3, 2, 3, 2, 3, 4, 1, 3, 2

See Figure 5.

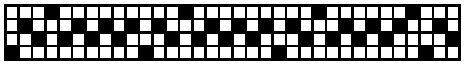


Figure 5. The Resulting Threading Sequence

The Icon Analyst

Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
U.S.A.

voice: (520) 621-6613

fax: (520) 621-4246

Electronic mail may be sent to:

icon–analyst@cs.arizona.edu

THE UNIVERSITY OF ARIZONA
TUCSON ARIZONA

and

Bright Forest Publishers
Tucson Arizona

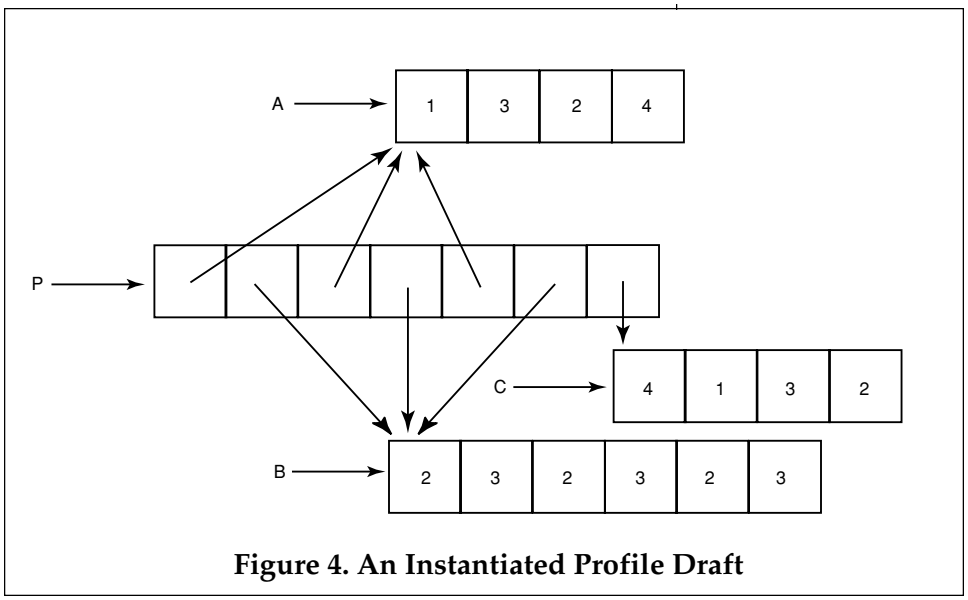


Figure 4. An Instantiated Profile Draft

We can "undefine" A, B, and C by removing their values:

```
while get(A)
while get(B)
while get(C)
```

This results in empty lists for A, B, and C, while not changing the data structures themselves. The situation again is as shown in Figure 3. The blocks can now be given different values with the same profile sequence in effect.

We can generalize the concept of profile drafting by allowing blocks to contain other blocks, as in

```
E := []
F := []
G := []
```

and then doing something like

```
put(A, E, F, E)
put(B, G, E, G)
put(C, F, E, F)
```

and finally assigning values to E, F, and G. We won't attempt to diagram the data structures, but you might try sketching them for yourself.

There are all kinds of possibilities here, including bad ones, such producing loops for which the flattening process does not terminate.

As far as we know, no weaver has attempted multilevel profile drafts. We think it's an interesting idea, although it remains to be shown that it can produce weaves that cannot be produced more easily by other methods.

References

1. *Designing with Blocks for Handweavers*, Doramay Keasbey, AltaVista Publications, 1993.

2. *Playing with Blocks*, Erica Voolich, The Cross Town Shuttle, 1977.

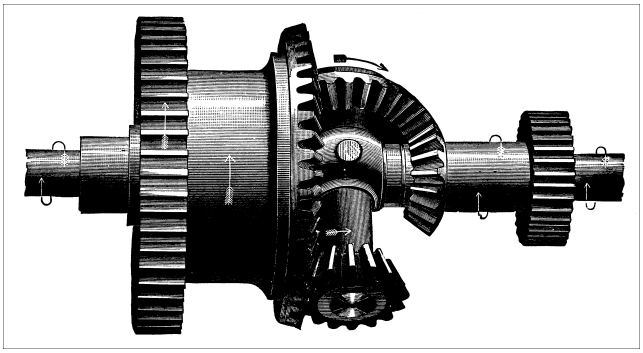
3. *Weaves and Pattern Drafting*, John Tovey, B. T. Batsford, 1969, pp. 41-64.

4. *Designing for Weaving: A Study Guide for Drafting, Design and Color*, Carol Kurtz, Hastings House, 1981, pp. 33-39.

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

6. "Pointer Semantics", Icon Analyst 6, pp. 2-8.

7. "Packet Sequences", Icon Analyst 63, pp. 7-9.



What's Coming Up

The best book on programming for the layman is "Alice in Wonderland"; but that's because it's the best book on anything for the layman.

— Alan Perlis

You'd think by the time we'd finished this penultimate issue of the Analyst that we'd know what was going to be in the final issue.

Actually, we do but we're not going to tell you. Or maybe we don't but don't want to admit it.

But we will tell you that there's not going to be anything unusual. We'll just try to finish up as many recent loose ends as we can.