The Icon Analyst

In-Depth Coverage of the Icon Programming Language

June 1999
Number 54

In this issue …

Backtracking and Bounded Expressions

Backtracking

Early artificial intelligence programming languages that incorporated search and backtracking facilities did not limit backtracking to previous computations when a portion of a search failed [1]. This had two unpleasant consequences: (1) the amount of data that had to be retained to allow backtracking limited the problems that could be handled, and (2) unexpected failure could cause backtracking arbitrarily far back in program execution and produce unexpected and sometimes mysterious results.

In subsequent artificial intelligence languages, this problem was dealt with by allowing the search tree to be pruned — portions discarded, freeing up the associated memory and preventing backtracking beyond that point [2,3]. Prolog introduced the cut operator for similar reasons [4].

Limiting Backtracking

Icon deals with this problem in a different way by limiting backtracking syntactically and requiring the programmer to explicitly construct backtracking that has a large scope.

Semicolons, for example, prevent backtracking to previous expressions. This is in contrast to conjunction in which there is no such limitation. Thus, in

e1; *e2*

once the evaluation of *e1* either fails or produces a result, evaluation continues to *e2*, but if it fails backtracking to *e1* for possible alternative results does not occur.

On the other hand, in

e1 & *e2*

if *e1* produces a result but *e2* fails, backtracking occurs into *e1* for possible alternative results.

Semicolon Insertion

The Icon compiler inserts a semicolon at the end of a line if the line ends in a complete expression and the next line starts with the beginning of

an expression. Consequently,

```
e1; e2; e3; … ; en;
```

can be written as

```
e1
e2
e3
…
en
```

Once the evaluation of a line is complete, evaluation goes on to the next line and failure cannot cause backtracking upward to previous lines.

Extending Backtracking

Operators (as opposed to control structures) bind expressions so that backtracking can occur between them. Conjunction, which performs no computation, illustrates this:

```
e1 &
e2 &
e3 &
… &
en
```

Here conjunction allows backtracking up a "ladder" to previous lines. Indentation make this binding more obvious:

```
e1 &
  e2 &
    e3 &
              … &
               en
```

The same binding of expressions for the purposes of backtracking applies to all operations, as in

```
e1 +
  e2 –
    e3 +
              … –
               en
```

Limitation as Opposed to Binding

By requiring explicit binding to obtain backtracking between expressions written on separate lines, Icon favors limiting expressions to at most one result. Limiting backtracking has two advantages alluded to earlier: (1) information that must be kept to allow backtracking can be discarded once it is no longer needed, and (2) the chances of unexpected backtracking are reduced.

Limiting backtracking is very natural when successive expressions are not generators or are not dependent on the success of successive ones. An example is

```
i := j
j := i + k
write(i)
if k > i then write(j)
```

Note the imperative nature of this code.

Bounded Expressions

Icon goes beyond limiting backtracking between semicolon-separated expressions (explicit or implicit). In specific syntactic contexts, expressions are *bounded*; once their evaluation is complete, backtracking cannot occur into them.

For example in

```
if e1 then e2 else e3
```

e1 is bounded, and whether *e1* succeeds or fails, subsequent failure of *e2* or *e3* (whichever is selected as the result of evaluating *e1*) does not cause backtracking into *e1*. If it did, the control structure would not do what's expected of if-then-else.

In the last issue of the Analyst [5], we commented on a principle used in the design of Icon: Don't do something unless there is a reason. There clearly is a reason to bound *e1*, but is there a reason to bound *e2* and *e3*? In discussions at the time, one participant was concerned that "something bad might happen" if *e2* and *e3* weren't bounded. In the absence of a concrete example of "something bad", we followed the design principle, which might be rephrased as "no gratuitous features". Nothing bad happened and if-then-else can be a generator. This may seen unusual, but it sometimes is useful, as in

```
if j > i then i to j else j to i
```

Similar reasoning was applied to other contexts for expression evaluation; expressions are bounded only if there is a good reason to do so.

Here's a list of syntactic contexts in which expressions are bounded, as indicated by underlining:

```
case e of {
  e1 : e2
  e3 : e4
    …
```

}

every *e1* do *e2*

if *e1* then *e2* else *e3*

not *e*

repeat *e*

return *e*

suspend *e1* do *e2*

until *e1* do *e2*

while *e1* do *e2*

{ *e1*; *e2*; …; *en* }

Limitation

The limitation control structure, *e1* \ *e2*, limits *e1* to at most *e2* results. This is, of course, more general than bounded expressions. Since the value of *e2* can be 1, limitation could have been used to get the effect bounded expressions, and it would produce the same result, as in

```
if e1 \ 1 then e2 else e3
```

This, however, would be cumbersome and error prone.

It's worth noting that *e2* is not bounded and can be a generator, as in

```
seq() \ (1 to 3)
```

with is equivalent to

```
(seq() \ 1) | (seq() \ 2) | (seq() \ 3)
```

for which the sequence is 1, 1, 2, 1, 2, 3.

References

1. *Planner: A Language for Manipulating Models and Proving Theorems in Robots*, C. Hewitt, AI Memo 168, MIT, 1970.

2. *Why Conniving is Better than Planning*, D. V. McDermott and G. J. Sussman, AI Memo 255A, MIT, 1972.

3. *The CONNIVER Reference Manual*, G. J. Sussman and D. V. McDermott, AI Memo 259, MIT, 1972.

4. *Implementing Prolog — Compiling Logic Programs*, D. H. D. Warren, D.A.I Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

5. "Built-In Generators" Icon Analyst 53, pp. 16-18.

Analyst Directions

A new subscriber to the Analyst recently expressed some disappointment with the content of the first few issues he received, especially articles on things like versum sequences that appeared to have no connection to Icon. Of course, he entered in the middle of the performance, as it were.

We try to cover a variety of topics in the Analyst, ranging from articles about various aspects of Icon itself to programming techniques to applications written in Icon that demonstrate Icon's value in such areas.

The Analyst now is ending its ninth year of publication. Over the years, we've covered most aspects of the language proper in some detail, including some of its more obscure nooks and crannies. We'll still continue to have articles about the language itself, looking at things from different perspectives.

At the opposite end of the range, we get into explorations like versum numbers and weaving where try to show how programming can be an integral part of problem solving. We sometimes get into areas that we had not anticipated when we started — the long series of articles on versum numbers being the most notable example. Although not all such articles feature programming directly, they are part of a larger picture that does.

To compensate for more space devoted to graphics and articles that may not be of much interest to some readers, we've increased the size of the Analyst from 12 pages to 16, and sometimes to 20 (and without increasing the price of subscriptions).

We're always interested in what our subscribers would like to see in the Analyst. We can't always oblige, and we won't repeat material that already has been covered. In addition, much of what appears in the Analyst is planned far in advance. For example, we can see at least a year of articles related to weaving.

But let us know what you like, don't like, or would like to see. We'll do what we can.

A Weaving Case Study

The value of weaving is in the work.

— Lili Blumenau [1]

Having started explorations of weaving, we decided it would be worthwhile to study a few weaving drafts in depth. We started with one of Painter's built-in weaves [2]. We got more than we bargained for.

Figure 1 shows the Painter dialog for the draft and Figure 2 shows the resulting weave.

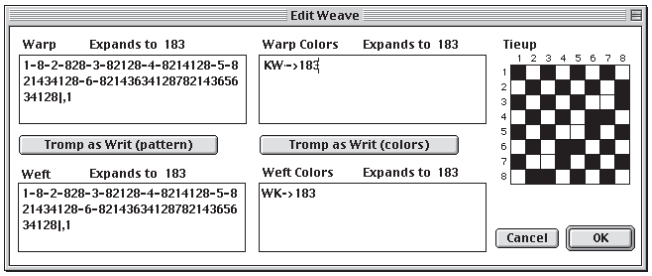


Figure 1. Painter Weaving Dialog

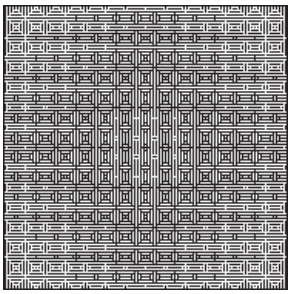


Figure 2. Weaving Image

It is a shadow weave [3, 4]. Weaves of this type produce the appearance of shadows (which are more obvious on actual woven fabrics than in images) by alternating light and dark threads in reverse orders in the warp and weft.

The threading and treadling expressions for shadow weaves typically are the same — "tromp as writ", as is the case here. Therefore we need only consider the threading expression.

The color letters have no meaning in and of themselves in Painter, but the usual associations are W for white and K for black.

The threading expression is a (true) palindrome. This follows from the fact that the pattern palindrome operator, |, has very low precedence and the expression groups like this:

((1–8–2–828 …436341287821436563412 8)|,1)

The concatenated 1 at the end converts a pattern palindrome into a true one. The weave looks better when repeated if this last character is omitted, leaving a pattern palindrome. We'll do that here.

The threading expression consists of a sequence of domain runs — "ups and downs" — between other shaft sequences. This is easier to understand graphically than in terms of numbers. Figure 3 shows the threading for the first half of the sequence. The bar at the top shows the colors.

If we look at the operand of the pattern palindrome operator, we see that it has a definite structure:

1–❶–2–❷–3–❸–4–❸–5–❺–6–8214363412878214365634128

where the components in circles have their own structure:

❶ = 8
❷ = 828
❸ = 82128
❹ = 8214128
❺ = 821434128

Note that these all are true palindromes.

After –6–, the pattern appears to break down, although there are similarities with the earlier parts. In fact,

8214363412878214365634128

is equivalent to

82143634128–7–8214365634128

So we have

1–❶–2–❷–3–❸–4–❸–5–❺–6–❻–7–❼

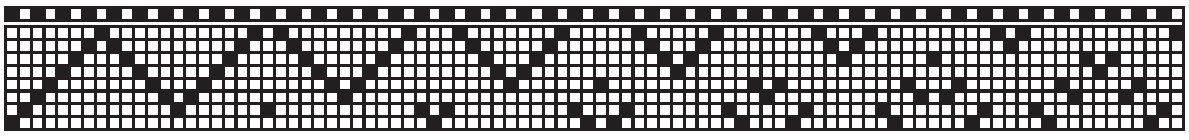


Figure 3. The Threading

with the continuation of the palindromes between:

❻ = 82143634128
❼ = 8214365634128

These palindromes can be represented using pattern forms, which makes the underlying structure more evident:

❶ = [!8]
❷ = [8!2]
❸ = [82!1]
❹ = [821!4]
❺ = [8214!3]
❻ = [82143!6]
❼ = [821436!5]

The sequence 8241365 runs not only across but also down the center of these palindromic forms — patterns within patterns.

One way to view the overall pattern is as a sequence of anchors for domain runs, which are connected by palindromes. Figure 4 shows the threading draft with the anchors indicated by vertical bars and the palindromes by horizontal bars.

We might ask several questions at this point. The first ones that come to mind are:

- If we modify this pattern in various ways, what kinds of weaves result?
- Is the threading pattern somehow special or just one of a class of patterns that produce interesting weaves?
- If so, how can this class be characterized?

We'll start with the first question — it leads to more than enough to occupy us for now.

We'll take the domain runs as given and concentrate on the sequence of anchors and palindromes. For this, it is easier to deal with character sequences. We'll retain digits for labeling the shafts and use the letters A though G to label the palindromes. Thus, the sequence can be represented as

1A2B3C4D5E6F7G

In terms of pattern forms, this is an interleaving:

[1234567~ABCDEFG]

More formally, we can label the anchor sequence $\mathfrak{a}$ and the palindrome sequence $\mathfrak{P}$, giving

$[\mathfrak{a} \sim \mathfrak{P}]$

Given transformations τ_1 and τ_2 on sequences, we can consider

$[\tau_1(\mathfrak{a}) \sim \tau_2(\mathfrak{P})]$ *general transformations*

One possibility is coupling the anchors and the palindromes, that is $\tau_1 \equiv \tau_2$:

$[\tau_1(\mathfrak{a}) \sim \tau_1(\mathfrak{P})]$ *coupled transformations*

An example of this, using our original notation, is the permutation

6–❻–3–❸–1–❶–4–❹–5–❺–2–❷–7–❼

Another possibility is using the identity transformation ι on one but not the other component:

$[\tau_1(\mathfrak{a}) \sim \iota(\mathfrak{P})]$ *anchor transformations*

or

$[\iota(\mathfrak{a}) \sim \tau_2(\mathfrak{P})]$ *palindrome transformations*

Respective examples are the permutations

5–❶–4–❷–3–❸–2–❹–1–❺–7–❻–6–❼

and

1–❺–2–❻–3–❼–4–❹–5–❸–6–❷–7–❶

We are of course, not limited to permutations. Examples of transformations that are not permutations are the coupled transformation

1–❶–2–❷–3–❸–4–❹–4–❹–3–❸–2–❷

and this transformation, which increases the length of the sequence

1–❺–2–❻–3–❼–4–❹–5–❹–6–❷–7–❶–1–❺–2–❻

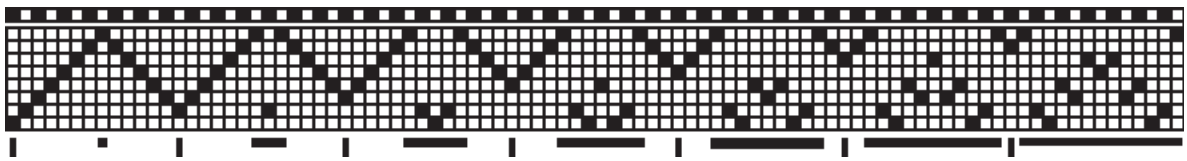


Figure 4. Threading Draft Showing Anchors and Palindromes

It is, of course, impossible to explore all such transformations. For permutations alone, there are $14! \cong 8.7 \times 10^{13}$ possibilities for the general case.

There are, however, only 7! = 5,040 permutations for the coupled anchor and palindrome cases. We tried all the anchor-sequence permutations to get a feel for how the weaves differ.

No two of the weaves are the same, although many are so similar that the differences cannot be detected without detailed examination. All are visually attractive, at least to us, and the range of design variations is relatively small. The 8 weaves in Figure 5 represent the visual extremes. We would say that the underlying pattern is aesthetically robust with respect to coupled anchor permutations.

Notice that there is some difference in the size of the weaves. This is to be expected, since the lengths of the domain runs change when the anchors do. The size is determined solely by the first anchor. If the first anchor is *i*, then the weave is 180 + 2*i* threads on a side.

This only touches on the possibilities we've already mentioned. There are many other possibilities, including:

- allowing the threading and treadling sequences for a weave to be different — removing the "tromp as writ" constraint.
- trying different tie-ups
- trying different color sequences — not just different colors but different warp and weft color sequences

To go on along these lines, we need a tool for controlled experimentation. We'll explore this and give the results in a later article.

For now, we leave you with the following program, called shadow, which produces pattern-form drafts for variations on the original shadow weave:

```
link options
link strings

global anchor_indices
global palindrome_indices
global palindrome_basis
global palindromes

procedure main(args)
   local expression, name, opts, tie_up, warp_colors
   local weft_colors, palette, i, anchor_vector
   local palindrome_vector

   opts := options(args, "b:n:t:c:d:p:")

   anchor_vector := \args[1] | "1234567"
   palindrome_vector := \args[2] | anchor_vector

   palindrome_basis := \opts["b"] | "8214365"
   weft_colors := \opts["c"] | "01"
```

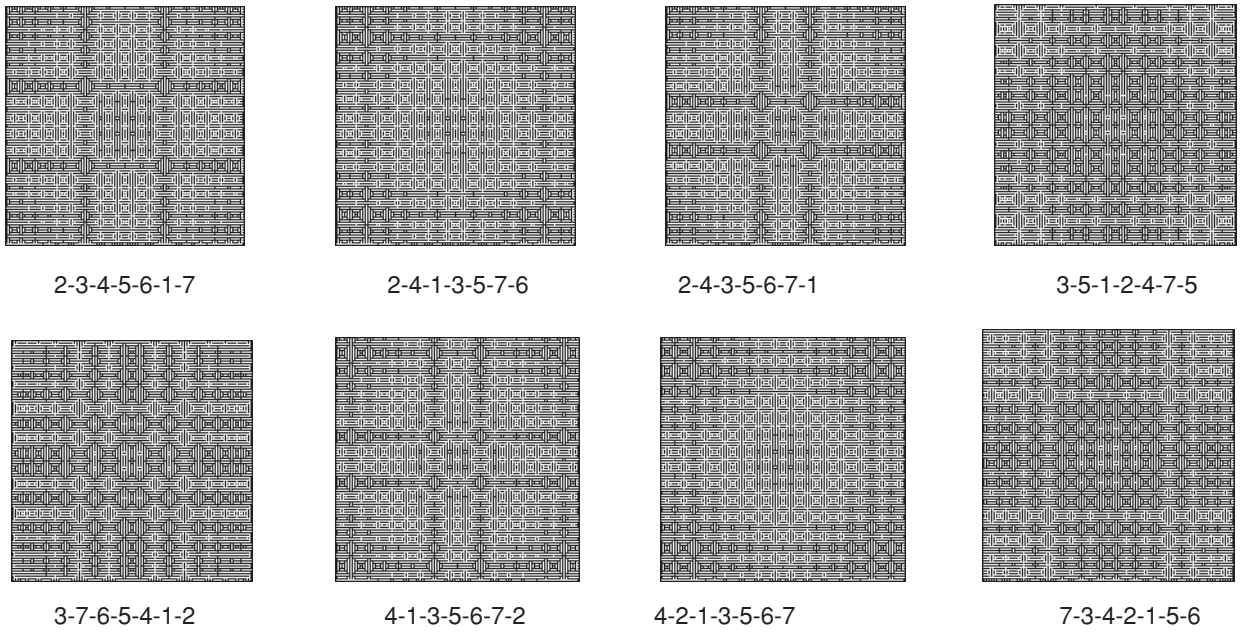


Figure 5. Example Weaves for Anchor-Sequence Permutations

```
   warp_colors := \opts["d"] | "10"
   palette := \opts["p"] | "g2"
   name := \opts["n"] | "untitled_shadow_weave"
   tie_up := \opts["t"] |
      "8;8;1010101001010101101010010101010110_
       1010010101011010100101010110101010"
   anchor_indices := transpose(
      "1234567",
      "1234567",
      anchor_vector
      )
   palindrome_indices := transpose(
      "1234567",
      "1234567",
      palindrome_vector
      )
   palindromes := list(*palindrome_basis)
   every i := 1 to *palindrome_basis do
      palindromes[i] := "[" || palindrome_basis[1:i] ||
         "!" || palindrome_basis[i] || "]"
   expression := "[" || threading(anchor_indices[1]) ||
      "|]"
   write(name)
   write(expression)
   write(expression)
   write(warp_colors)
   write(weft_colors)
   write(palette)
   write(tie_up)
   write()
end
procedure threading(i)
   local result
   if i > *palindrome_basis then return ""
   result := "–[" || anchor_indices[i] || "–[" ||
      palindromes[anchor_indices[i]] ||
         threading(i + 1) || "]]"
   if i = 1 then result := result[2:0]
   return result
end
```

The command-line options allow the specification of non-default values for the draft name, warp and weft colors, palette, palindrome basis string, and tie-up. Transformations are specified by two other optional command-line arguments. The first is a transformation for the anchor sequence and the second a transformation for the palindromes. If the latter is omitted, it defaults to the anchor-sequence transformation.

References

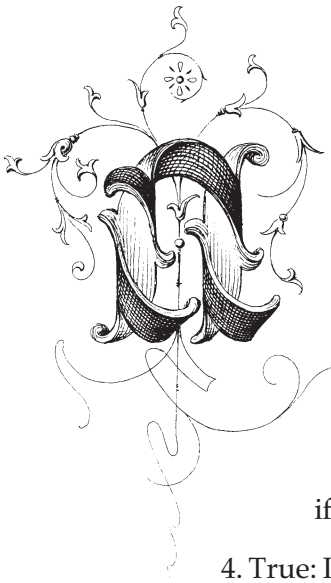
1. *The Art and Craft of Hand Weaving*, Lili Blumenau, Crown Publishers, 1955.

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

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

4. "Putting the Shadow in Shadow Weave", Donna Muller, *Handwoven*, Vol. XIX, No. 4, September/October 1998, pp. 34-40.

Answers to Expression Evaluation Quiz



See Icon Analyst 53 for the quiz questions.

1. False; a repeat loop can be terminated by return, fail, suspend, exit(), stop(), or a run-time error.

2. False.

3. True; by the case expression failing, the evaluation of a selected case clause failing, or if no case clause is selected.

4. True: If the selected case clause is a generator, the case clause generates its results.

5. True: break can have an argument expression, as in

```
break 1 to 10
```

which generates 1, 2, ..., 10 if the break expression is evaluated.

6. Assigns 1 to i; there is no failure to cause the to expression to be resumed.

7. True.

8. True in a sense: If x is a file stream that never terminates.

9. False. If x is empty, ?x fails and |?x produces no result.

Note: In the following answers, values written on separate lines are shown with separating blanks to save space.

10. The first expression fails, since every fails after all the results are generated and write() inherits the failure. The second expression writes

```
7  7  7  7  7
```

11.

(a)
```
1  2  3  4  5  5  4  3  2  1
```

(b)
```
6  5  4  3  2  7  6  5  4  3  8  7  6  5  4
9  8  7  6  5  10  9  8  7  6
```

(c)
```
1  2  1  3  2  1  4  3  2  1
```

12.

(a)
```
1  2  3  4  5  6  7  8  9
```

(b)
```
1  2  3  4  5  6  7  8  9
```

(c)
```
1  2  3  4  5  6  7  8  9  1  2  3
```

(d)
```
1  1  2  1  2  3  1  2  3  4  1  2  3  4  5
```

(e)
```
1  2  3  4  5  6  7
```

(f)
```
1  1  2  1  2  3  1  2  3  1  2  1
```

13. The first expression fails. The other three are "black holes" that never terminate or produce results.

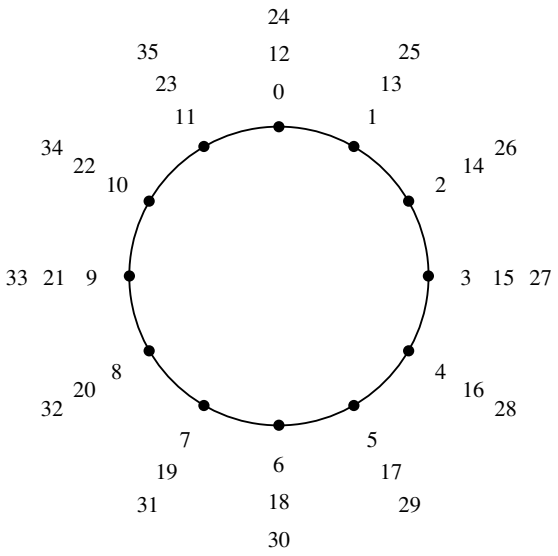
Exercises

1. Assume seq_old is a list of integers. What does the following code do? Does it always terminate? If it terminates, what does patterns contain?

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

(lequiv(), which is in the Icon program library module lists, compares two lists and succeeds if and only if their elements are the same.)

2. Recently we were trying to explain modular arithmetic to a non-technical audience. To help get the idea across, we showed what happens if you wrap the integers around a circle. Here's the kind of diagram we used:



Write a program that produces diagrams of circles with equally spaced, numbered points around the perimeter, such as the one above. Take the number of points around the circle (modulus) from a command-line argument. *Hint:* If you think about the problem in the right way, placing the labels properly is not hard.

Your program should produce both image and PostScript files [1]. A good solution would allow various parameters, such as the diameter of the circle, to be specified.

For overkill, provide a visual interface that allows such diagrams to be designed interactively.

Other things you might explore are

- adding spokes from the center of the wheel to the dots on it
- putting the first round of numbers in outlines circles where the dots now appear
- different geometric shapes

Reference

1. "From the Library — PostScript Graphics" Icon Analyst 52, pp. 17-20.

Graphics Corner
— Exploring for Tiles

The hidden harmony is better than the obvious one.

— Heraclitus, 6th–5th century B.C.

Editors' Note: The application described in this article was adapted from an earlier one shown in the Icon graphics programming book [1].

In order to appreciate the images that follow, you should view colored versions. You can find them on the Web page for this issue of the Analyst. See page 10 for the URL.

Background

In a previous article [2], we described the tilings of motifs (*generating tiles*) to create repeat patterns that cover surfaces with designs. Repeat patterns have been used since prehistoric times to decorate otherwise drab surfaces. Our environment abounds in man-made repeat patterns, which occur most predominantly in clothing and interior decoration. You may find it instructive to explore in detail the contents of a room to find as many examples of such patterns as you can.

With the advent of personal computers and affordable graphics, repeat patterns have become in increasing demand for desktop patterns, backgrounds for Web pages, and surfaces for 3D models.

The generating tiles for such patterns almost always are rectangular and come in many forms. Ones that tile seamlessly — in which no border between adjacent tiles is evident — are in demand because they allow the creation of decorations in which there is an illusion of a pattern larger than its underlying tile and provide continuous patterned surfaces without jarring or distracting discontinuities.

The application described in this article is designed for discovering interesting tiles in existing images. It originally was intended to be tool to find the smallest generating tiles in repeat patterns. It was soon clear that the application had more interesting capabilities — a small portion of a non-tiled image often produces an interesting repeat pattern. It turns out that you can find interesting tiles of this kind in almost any image.

The Application

The idea is to allow the user to select a rectangular area of an image and immediately see what that selection looks like when it is tiled.

The interface, shown in Figure 1, allows various ways of specifying a selection with a precise location and size.

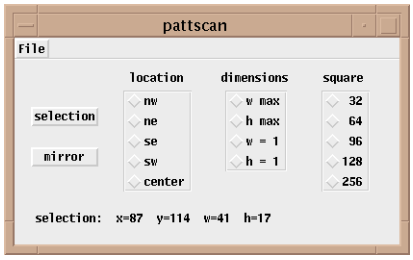


Figure 1. The Application Interface

The selection button brings up a dialog for entering selection information textually. See Figure 2.

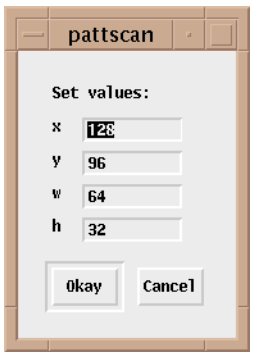


Figure 2. The Selection Dialog

A selection also can be made by clicking and dragging out a rectangle on the image being explored. An existing selection can be nudged in one-pixel increments using the arrow keys. Arrow keys in combination with the meta key nudge the dimensions.

When a selection is made, it is tiled in another window. Figures 3 shows a source image and

Figures 4 and 5 the results of tiling selections from it.

Figure 3. An Image for Tile Exploration

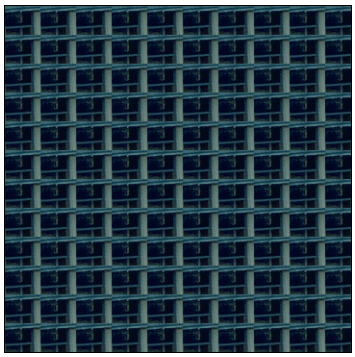


Figure 4. Tiling of a Selection Near the Center

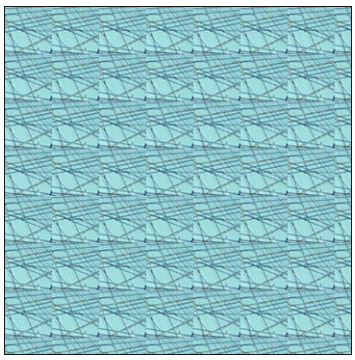


Figure 5. Tiling of a Selection in the Rigging

Tiling is a fast process. When a selection is made, the corresponding tiling appears almost immediately. When a selection is made interactively by dragging out a rectangle, the tiling tracks it. Unless the selection is large, the tiling tracks the rectangle as the mouse is dragged. If the selection is too large for that, the tiling may lag behind the changing selection and only catch up when the selection stops changing.

A selection often produces a more interesting tiling if it is mirrored [3] when tiled. This can be done using the mirror button show in Figure 1. Figures 6 show the mirrored tiling for the selection used in the non-mirrored tiling shown in Figure 5.

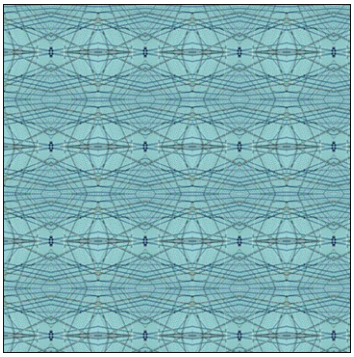


Figure 6. A Mirrored Tiling

Suggestions for Exploration

Sometimes interesting patterns can be found by using dimenions for selections that might not come about naturally just by dragging out a selection rectangle by hand.

For example, if the width or height is set to one pixel, the result is vertical or horizontal stripes accordingly (provided the other dimension is large enough). Figure 7 shows the results of a selection rectangle that is one pixel high and extended the full width of the tug boat image. In the case of a detailed image like this one, the result resembles a complex gradient more than stripes.

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/ia54/

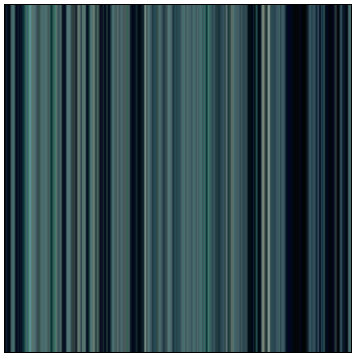


Figure 7. A Gradient Pattern

By increasing the small dimension from one pixel to several, textured stripes such as those found in woven fabrics often result. Figure 8 shows the results of a selection in which the long dimension is reduced and the height is six pixels.

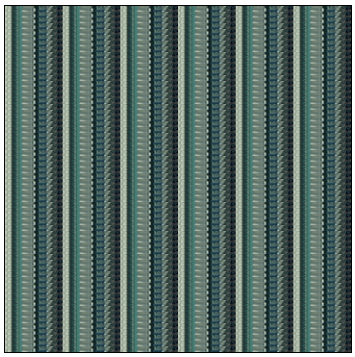


Figure 8. Textured Stripes

The nature of the source image of course effects the kinds of tiles that are found. The image in Figure 9 is, like the tug boat image, from a photograph. Most of this image is very dark, but the lights reflected on the water provide ample hunting ground for tiles.

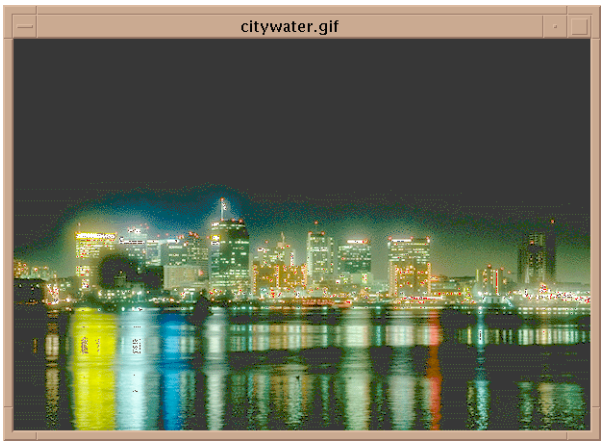


Figure 9. City Lights Reflected in the Water

Figure 10 shows one such tiling.

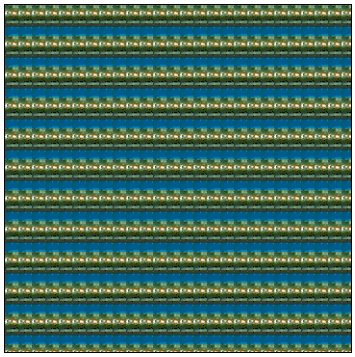


Figure 10. Tiling from City Lights

Figure 11 shows another tiling, this time mirrored.

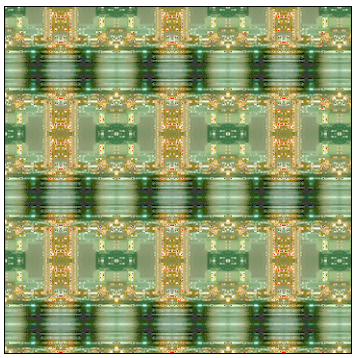


Figure 11. Mirrored Tiling from City Lights

Cartoons, which usually don't have the subtle gradations of color typically found in photographs of natural scenes, also can produce interesting tiles. They usually are more stylized that tilings from photographs of natural scenes

Figure 12 shows the cartoon from which the mirrored tilings in Figures 13 and 14 were derived.

Figure 12. Cartoon

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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

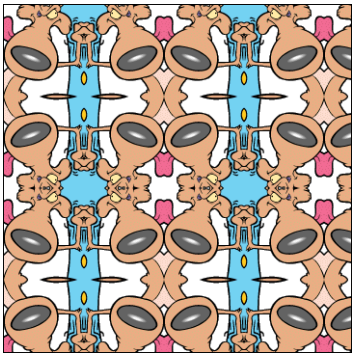


Figure 13. Mirrored Tiling from the Cartoon

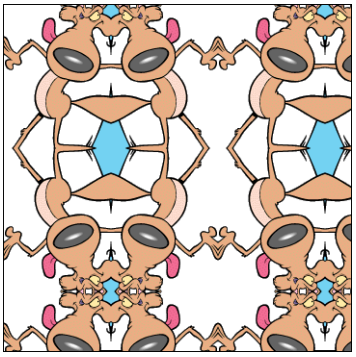


Figure 14. Mirrored Tiling from the Cartoon

Perhaps surprisingly, patterns make good material for finding tiles. Figure 15 shows a numerical carpet [4]. Figures 16 and 17 show two tilings derived from it.

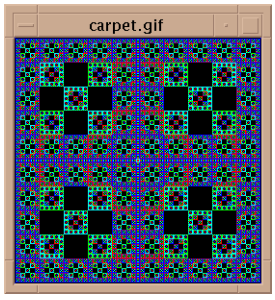


Figure 15. A Numerical Carpet

Icon on the Web

Information about Icon is available on the World Wide Web at

http://www.cs.arizona.edu/icon/

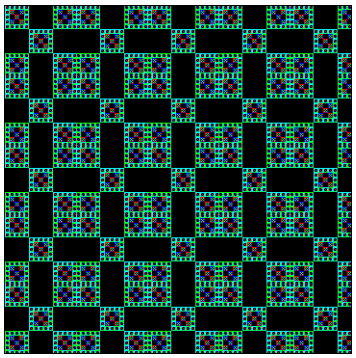


Figure 16. Tiling from a Numerical Carpet

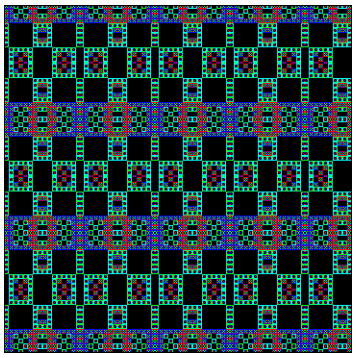


Figure 17. Tiling from a Numerical Carpet

Text also can be used as a basis for tiling. Figures 18 and 19 show two examples.

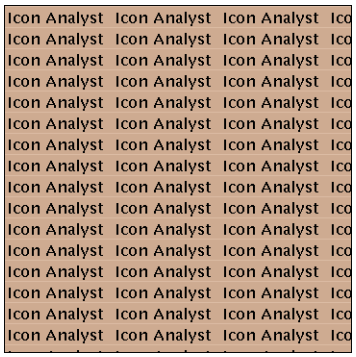


Figure 18. Tiling from Text

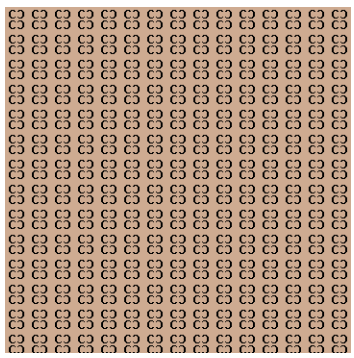


Figure 19. Mirrored Tiling of a Letter

Figure 20 shows other tilings obtained by searching within various images.

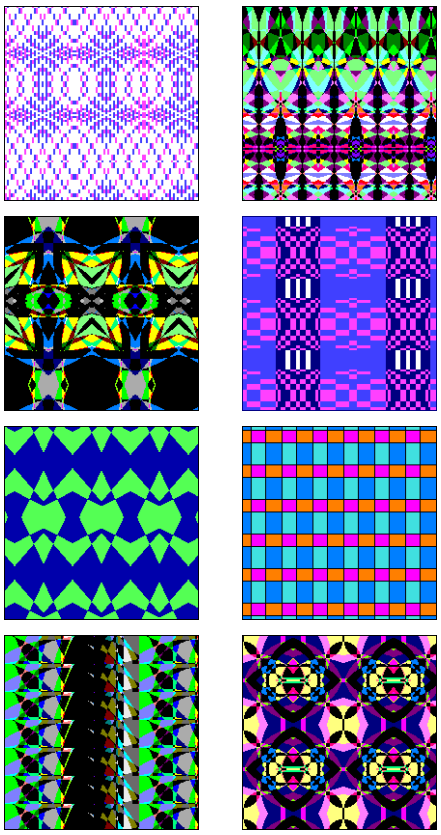


Figure 20. Various Tilings

Conclusion

To fully appreciate what it's like to explore for tiles, you have to do it. It's especially interesting when tiles are found interactive by dragging out a selection rectangle and watching the tiled results.

The application for tile exploration is on the Web site for this issue of the Analyst so that you can try it yourself.

In the next issue of the Analyst, we'll described the more interesting aspects of the program itself.

References

1. *Graphics Programming in Icon*, Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend , Peer-to-Peer Communications, Inc., 1998, Plate 13.2.

2. "Graphics Corner" Icon Analyst 44, pp. 8-9.

3. "Graphics Corner — Seamless Tiling", Icon Analyst 45, pp. 10-12.

4. "Anatomy of a Program — Numerical Carpets", Icon Analyst 45, pp. 1-10.

Generating Sequences

In the last issue of the Analyst, we started a series of articles on generators and sequences by reviewing Icon's repertoire of built-in generators [1].

In this article, we'll focus on how sequences can be produced using only the built-in repertoire — we'll save (declared) procedures for a later article. We'll also bar co-expressions, which also will be treated separately.

Several years ago, we posed some problems related to this subject [2]. We'll start here with that approach and then go on to a general model for writing expressions that produce sequences.

Sequence Generation Problems

All the sequences that follow are integer sequences. Sequences can, of course, contain values of any type, but integer sequences are adequate for showing principles, and they avoid some distracting complexities associated with values of other types.

Here are seven infinite sequences. We'll show how to write generators for them in the next section, but you might give them a try before looking ahead.

1. The positive integers; 1, 2, 3, …

2. The squares: 4, 9, 16, …

3. The integers raised to their own powers: 1, 4, 27, 256, …

4. The even squares: 4, 16, 36, …

5. The integers with alternating signs: 1 –2, 3, –4, 5, –6, …

6. The Fibonacci numbers, in which each term is the sum of the two preceding ones, starting with 1, 1: 1, 1, 2, 3, 5, 8, 13, …

7. The "self-replicating" sequence, with each term repeated its own number of times: 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, …

Solutions

1. Generating the positive integers is trivial:

```
seq()
```

This function provides the basis — a "driver" — for many other sequences: sequences in which the values of terms are functions solely of their positions. Put another way, seq() indexes the terms in a sequence.

2. The squares follow immediately from seq():

```
seq() ^ 2
```

This introduces the idea of applying an operation to the results of a generator.

3. Generating the integers raised to their own powers can be done as follows:

```
(i := seq()) & (i ^ i)
```

or cast as mutual evaluation

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

We'll use mutual evaluation in the following examples, since it is more compact, especially for complex expressions.

Note that this example introduces the need for an auxiliary variable.

4. Generating the even squares can be done by filtering out the odd ones:

```
(i := seq(), if i % 2 = 0 then i) ^ 2
```

Since there is no else clause, if-then fails if its control expression fails. This generator also can be cast as

```
(i := seq(), i % 2 = 0, i) ^ 2
```

If the comparison fails, the following expression, i, is not evaluated.

A little insight suggests a simpler expression:

```
seq(2, 2) ^ 2
```

5. One way to generate integers with alternating signs is to use the method of the preceding example:

```
(i := seq(), if i % 2 = 0 then i else –i)
```

Another formulation maintains a state between successive results:

```
(j := –1, seq() * (j *:= –1))
```

The concept of maintaining a state (the value of j in this example) has more general applicability.

6. Generating the Fibonacci numbers introduces an initial, finite sequence followed by the rest:

```
1 | 1 | expr
```

Recall that the alternation of sequences produces the concatenation of their results.

The initial values are, however, needed in *expr* to start it up, so they need to be assigned to auxiliary variables:

```
(i := 1) | (j := 1) | |(k := i + j, j := i, i := k)
```

7. Generating the self-replicating sequence intro-

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

duces the concept of using more than one generator:

```
(i := seq(), |i \ i)
```

A somewhat more cryptic form is:

```
(i := seq(), (1 to i) & i)
```

That is, (1 to i) & i generates i copies of i.

A General Model

A schema for generating any sequence is

$\mathcal{G} \mid \mathcal{F}$

where $\mathcal{G}$ produces an initial, finite sequence (possibly empty) and $\mathcal{F}$ generates the rest of the (possibly infinite) sequence. $\mathcal{G}$ also may set the values of variables used in $\mathcal{F}$.

The components of $\mathcal{F}$ can be given in the form

(*i*, *d*, *c*, *r*)

or

(*i* & *d* & *c* & *r*)

where *i* does initialization, *d* is a generator that drives the rest of the expression, *c* performs some computation, usually on a result produced by *d*, and *r* produces the results of the expression.

Both *c* and *r* may be (finite) generators. (If either is an infinite generator, no prior components are resumed and the computation could be cast in a different way.)

In most situations, some component expressions can be eliminated and the remaining expressions can be collapsed. Figure 1 shows the general formulations for the seven sequences shown earlier. The numbers in parentheses refer to the specific expression given above when there is more than one.

Next Time

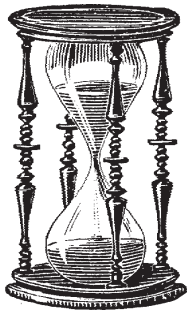
The next article in this thread will be about using procedures as generators. There will be other articles on generating sequences that will run in parallel.

References

1. "Built-In Generators" Icon Analyst 54, pp.16-19.

2. "Exercises" Icon Analyst 12, pp. 1-2.

Subscription Renewal



For many of you, this is the last issue in your present subscription to the Analyst. If so, you'll find a renewal form in the center of this issue. Renew now so that you won't miss an issue.

Your prompt renewal helps us by reducing the number of follow-up notices we have to send. Knowing where we stand on subscriptions also lets us plan our budget for the next fiscal year.

n	$\mathcal{G}$	$\mathcal{F}$			
		i	*d*	*c*	*r*
1.	&fail	&null	i := seq()	&null	i
2.	&fail	&null	i := seq()	i ^:= 2	i
3. (2)	&fail	&null	i := seq()	i ^:= i	i
4. (2)	&fail	&null	i := seq()	if i %2 = 0	i
5. (2)	&fail	j := –1	i := seq()	i *:= (j *:= –1)	i
6.	(i := 1) | (j := 1)	&null	|(k := i + j, j := i, i := k)	&null	i
7. (1)	&fail	&null	i := seq()	&null	|i \ i

Figure 1. General Formulations

Quiz

Using only Icon's built-in repertoire but no co-expressions, give expressions that generate the following sequences.

1. The palindromic integers:

1, 2, … 9, 11, 22, … 99, 101, …

2. The integers with the order of successive values reversed:

2, 1, 4, 3, 6, 5, …

3. The "sawtooth" sequence, which consists of runs up to each successive integer:

1, 1, 2, 1, 2, 3, 1, 2, 3, 4, …

4. The "mountain peak" sequence, consists of runs up to each successive integer and back down to 1:

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

5. The digit-sum sequence, in which the *i*th term is the sum of the digits in *i*:

1, 2 , 3, …, 9, 1, 2, 3, …, 9, 10, 2, 3, …, 10, 11, 3, …

6. The "sigma" sequence in which the digit sum of an integer is replied repeatedly until there is only one digit (for example, 28 ➛ 10 ➛ 1):

1, 2 , 3, …, 9, 1, 2, 3, …, 9, 1, 2, 3, …, 1, 2, 3, …

7. What do the following expressions generate?

(a) `seq() \ seq()`

(b) `(seq() ^ 2) \ seq()`

(c) `seq() \ (seq() ^ 2)`

(d) `(seq() ^ 2) \ (seq() ^ 2)`

(e) `(seq() \ seq()) \ seq()`

(f) `((seq() \ seq()) \ seq()) \ seq()`

(g) `(1 to 10) ^ seq()`

(h) `seq() | seq()`

(i) `(–1) ^ seq()`

(j) `(5 * (seq()) % 9`

8. For each of the following sequences, determine a rule that produces it and give an expression that generates it.

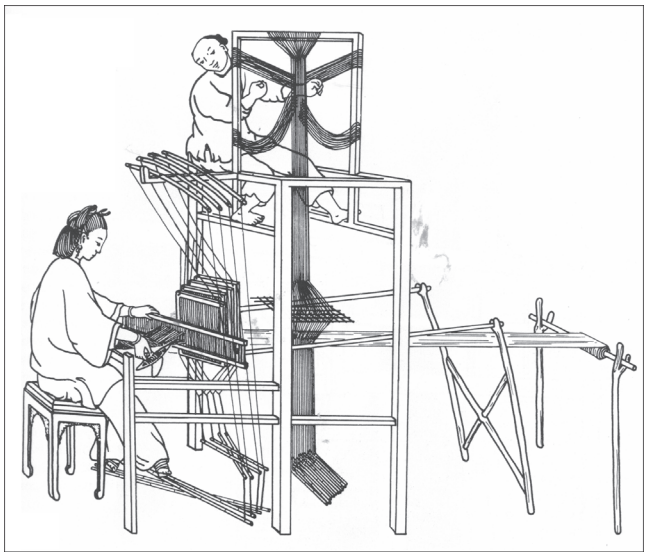
(a) 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 0, 1, 1, 1, 2, 1, 3, 1, 4, 1, 5, 1, 6, 1, 7, 1, 8, 1, 9, 2, 0, 2, 1, 2, …

(b) 1, 2, 3, 4, 5, 6, 0, 1, 2, 1, 0, 1, 1, 1, 2, 1, 3, 1, 4, 1, 5, 1, 6, 1, 0, 1, 1, 1, 2, 2, 0, 2, 1, 2, …

(c) 1, 1, 2, 3, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, …

(d) 1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489, 10000000000, 285311670611, 8916100448256, …

(e) 3, 81, 19683, 4304721, 847288609443, 150094635296999121, …



What's Coming Up

The purpose of computing is insight not numbers.
— Richard Hamming

In the next issue of the Analyst we plan to have the article on animation by image replacement that didn't make this issue.

We'll follow up the article on exploring for tiles with a discussion of some of the more interesting aspects of the program's implementation.

There will be solutions to the exercises and answers to the quiz in this issue, as well as another quiz.

The series of articles on weaving will continue with one on shadow-weave wallpaper and another article on looms and weaving drafts.

In the series of articles on generators and sequences, we'll review co-expressions and programmer-defined control operations. **From the Library** will describe some of the sequence resources that are available.