Andrew Glassner's Notebook
http://www.glassner.com

Digital Weaving, Part 2

Andrew Glassner

The process of weaving is a great model for making beautiful patterns. The basic ideas are conceptually simple, and the resulting fabrics can be lovely to see.

In Part 1 of this series I talked about the basics of looms and drafts, and I showed how to model the basic weaving process mathematically. I also showed a few examples from my digital loom, which implements those ideas.

This time I'll talk about how we can run the weaving process backward and deduce the draft from a fabric sample. I'll also discuss a language that helps us create and explore complex woven patterns.

Picking up the thread

Let's quickly summarize the basics that we discussed last time.

To produce a piece of fabric, you set up threads on a *loom*. One set of threads, called the *warp* threads, are drawn through *shafts* that are in front of you. Another set of threads, called the *weft* threads, pass through *treadles* located on the right side. These are woven under and over the warp threads to produce a row of fabric.

When we pick up a weft thread and run it to the left, we need to know if it should pass over or under each of the warp threads. The instructions for this are given in a diagram called the *draft*, as in Figure 1. The draft contains four binary matrices.

In the upper left, matrix ***F*** is painted black (that is, it has the value 1) if the vertical warp thread is on top; otherwise it's white (or has the value 0). This is the woven fabric, resulting from setting up a loom using the information in the other three matrices and then weaving. To weave, we need to determine the color of each cell in ***F*** given the other three matrices.

Typically we work one row at a time, from right to left. Given a particular horizontal weft thread, we look first to the treadling matrix ***R*** in the upper right. This matrix contains a single black cell in each row. We find the column with the black cell, and look downward to that column of matrix ***T*** in the lower right.

This matrix, called the *tie-up*, can be an arbitrary collection of 1s and 0s (or black and white cells). We look down the identified row of the tie-up and find those cells that are black. For each black cell, we look to the matrix ***S*** in the bottom-left of the draft.

Looking at the shaft matrix ***S***, we locate the cell directly under the warp thread that the weft is to be woven with next. If that cell of ***S*** is black, then the weft thread should pass under the warp. We then continue our way down the column of the tie-up, consulting each column of ***S*** corresponding to a black cell in ***T***. If any of the identified cells in ***S*** are black, then the weft passes under the warp. If they're all white, the weft goes on top.

We can capture this in a set of nested summations that I call *the weaving equation*. I noted in my November/December 2002 column that you can also play some tricks with binary numbers and tables to eliminate a lot of the complexity from the weaving equation, reducing it to just a few look-ups and a single binary operation.

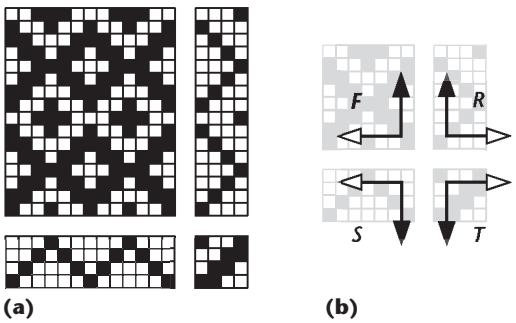
Figure 1 also shows the conventional indexing scheme for these four matrices. Each one has its origin at their common corner. To find an entry (a, b) for any matrix, we first count vertically a units and then horizontally b units.

This was a very fast and schematic run-through. You can find a more complete description with examples in my November/December 2002 column.

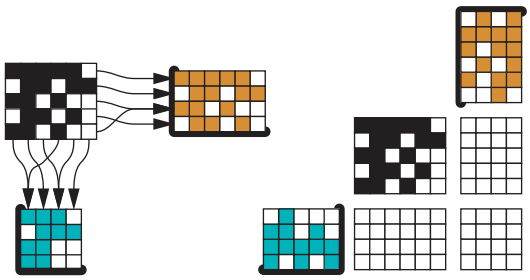
Deducing the draft

In my last column I spoke exclusively about weaving patterns from a given draft. It's natural to wonder if we can go the other way. Given a piece of fabric, can we deduce the draft? In other words, can we invert the weaving equation?

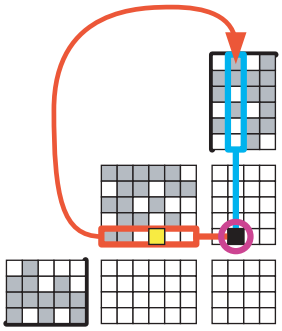
The answer is yes. The process is deductive, rather than inductive, because we're not creating any new information. The information we need to find the draft is already contained in the weaving. In other words, the piece of fabric and the draft are two versions of the same thing, and we can turn either into the other. To use the terminology of the weaving equation, last time we were



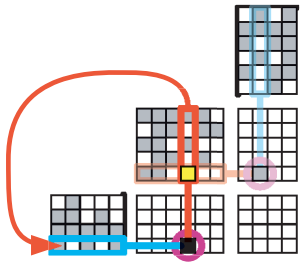
1 (a) A simple draft. (b) The matrices are indexed as labeled, first in the black direction, then the white.



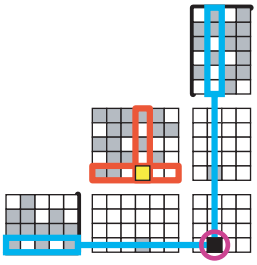
2 Getting ready to deduce a draft from a fabric. (a) Given the fabric ***F***, we create L_R, a collection of all the unique rows, shown in gold, and L_C, a collection of all the unique columns, shown in blue. (b) To make the process easier, I created a blank draft around the fabric and moved the two collections from Figure 2a around the draft. The heavy black lines around two edges of each collection show how I rotated them.



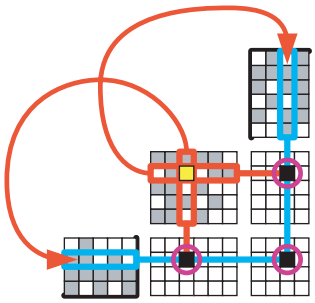
3 Step 1 of the deduction algorithm. We select a black cell from ***F***, here marked in yellow. We extract its row, outlined in red, and look it up in the collection of rows L_R. The row it matches is outlined in blue. The intersection of these two rectangles identifies a cell in the weft pattern, which we set to 1.



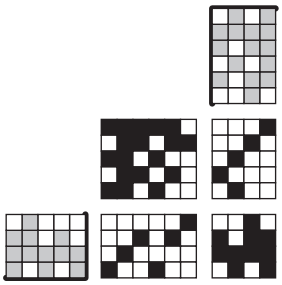
4 Step 2 of the deduction algorithm. Using the same cell from ***F*** that we used in step 1 (and still marked in yellow), we find the column that contains it, marked in red, and find the index of that column in the list of columns. The match is marked in blue. The intersection of these two colored rectangles identifies a cell in ***S***, which we set to 1.



5 Step 3 of the deduction algorithm. The intersection of the row and column found in the first two steps identifies a cell in the tie-up ***T***, which is then set to 1.



6 Processing of another black cell from ***F***, again marked in yellow.



7 The completed draft resulting from the deduction algorithm.

given the warp and weft patterns ***S*** and ***R*** and the tie-up ***T***, and from them computed the fabric ***F***. The deduction algorithm runs this the other way—it takes the fabric-overlapping matrix ***F*** as input, and finds ***S***, ***R***, and ***T***.

In this discussion, I'll ignore color, thickness, and spacing, since these can all be read off the original fabric by eye. Our goal is to find the three binary matrices that determine the overlapping structure of the threads.

Note that determining ***F*** from a real, physical sample can be tricky: you need to look at the fabric closely and determine for every overlap which thread is on top. For closely woven fabrics with fine threads, this can require magnifying lenses, a steady hand, and a lot of patience. But in the end, you'll have a matrix that represents one complete unit of the repeating pattern that makes up the overall cloth.

Once we have ***F***, we can use a nice algorithm for finding the other matrices developed by Ralph Griswold (I reference this in the "Further Reading" sidebar). It's surprisingly easy. I'll describe the process verbally, but take a look at Figures 2 through 7 for a graphic representation.

I'll begin by creating two helper data structures, which I'll call L_C and L_R. These will contain lists of columns and rows from ***F***, and both begin empty.

To build L_C, I'll scan through the fabric F and look at each column. If that column isn't yet in the collection L_C, I'll add it in. Then I'll do the same thing for L_R by scanning through the rows, adding in any row that's not yet in the list.

Once we finish this process, L_C has one copy of each unique column in ***F***, and L_R has one copy of each unique row in ***F***. Figure 2a shows an example of this. Note that it ultimately doesn't matter in which order the rows and columns are added. The arrangement of 1s in the three matrices will move around, but they'll still generate the same fabric. In other words, more than one set of ***S***, ***R***, and ***T*** matrices can create a given ***F*** matrix. You might find it interesting to think about the relationships between these equivalent representations, but I won't go into it any further here.

When we finish this scanning step, L_C contains one copy of each unique column in ***F***, and L_R will contain one copy of each unique row. Let's use the notation $|L_C|$ to refer to the number of entries in L_C, and similarly $|L_R|$ is the number of unique rows in L_R.

We can now create and initialize our three matrices. Starting with the weft pattern ***R***, we note that it's as high as the height of ***F*** and $|L_R|$ wide. Similarly, ***S*** is as wide

Further Reading

The deduction algorithm for finding the matrices from a weaving was originally presented in "From Drawdown to Draft—A Programmer's View," by Ralph E. Griswold, April 2000 (http://www.cs.arizona.edu/patterns/weaving/). In this article, Griswold provides fragments of source code in the Icon language. You can download Icon for free from http://www.cs.arizona.edu/icon/index.htm. The complete program for the deduction algorithm is available at http://www.cs.arizona.edu/patterns/weaving/FA/index.html.

Perl is another programming language well-suited to the deduction algorithm. It's available for free from http://www.activestate.com.

Procreate's painting program, Painter 7 (http://www.procreate.com), has had a weaving language built into it for quite a while, and it lets you design your own weaves and then use them as patterns for filling. Their weaving language is a bit tricky to use. I've written a set of notes that you may find useful if you want to use their system. You can find them at http://www.glassner.com.

My weaving language (AWL) draws very strongly on Procreate's Painter weaving language and monographs by Ralph Griswold.

The `growblock` operator is based on a discussion by Ralph Griswold in his paper, "Variations On A Shadow Weave," April 1999 (http://www.cs.arizona.edu/patterns/weaving/). The idea of using integer sequences was presented by Griswold in "Drafting with Sequences," March 2002 (also available on his Web site), and a number of other other monographs that deal with specific sequences. His Web site contains a wealth of useful information, including scanned-in copies of out-of-print but fascinating reference books.

Integer sequences for the `eis` command come from *The Encyclopedia of Integer Sequences* by Simon Plouffe and Neil J. A. Sloane, Academic Press, 1995. The contents of that book are also available in a terrific online resource that lets you look up sequences by name or even by the numbers in the sequence. Go to http://www.research.att.com/~njas/sequences/ to use this database of more than 76,000 sequences. I've only provided a small handful of these directly in my code. An AWL interpreter connected to the Internet could do a real-time query to the online database to pick up any sequence named.

as $\boldsymbol{F}$, and $|L_C|$ rows high. Finally, the tie-up matrix $\boldsymbol{T}$ is $|L_C|$ rows high by $|L_R|$ columns wide. I'll initialize all the elements of all three matrices to zero. Take a look at Figure 2b to confirm that these numbers all line up this way. I positioned the two collections just outside a blank draft. Note that I rotated them from how they're drawn in Figure 2a for convenience.

With that preparation finished, it's time to populate the matrices with 1s where they're needed. Because of all this preparation, the job itself is pretty easy. Essentially we scan the matrix $\boldsymbol{F}$ one cell at a time, looking for black cells (that is, elements with the value 1). Each time we find a black cell we execute the same procedure. To illustrate the process, let's follow what happens to a black cell in the bottom row of $\boldsymbol{F}$, third from the right: $\boldsymbol{F}_{0,2}$. I've marked this cell in yellow in Figure 3.

The first thing we do is look up the row containing this cell in the collection of rows, L_R. I marked the fabric row containing our yellow cell in red, and the corresponding row in the collection in blue. As we can see, it's row number 1 from the collection that we matched. The intersection of these two rectangles is a cell in the weft pattern $\boldsymbol{R}$, and we set that to black, as in Figure 3. In symbols, $\boldsymbol{R}_{0,1} \leftarrow 1$.

Now we look up the column containing this cell, marked in red in Figure 4, and find its entry in the collection of columns L_C. From the figure we can see that this is column number 3, again marked in blue. This time the intersection of the red and blue rectangles identifies a cell in the warp pattern $\boldsymbol{S}$, so we set that to black: $\boldsymbol{S}_{3,2} \leftarrow 1$.

Now for the tie-up. We found entries for row 3 and column 1 from the sets L_C and L_R, so we set the corresponding cell from the tie-up to black: $\boldsymbol{T}_{3,1} \leftarrow 1$, as in Figure 5.

That's it for this cell from $\boldsymbol{F}$. Now we march along for another black cell, and handle it the same way. Figure 6 shows the process for another black cell from $\boldsymbol{F}$. We repeat this process until we've run it for every black cell in $\boldsymbol{F}$. When we've done that, the matrices are fully populated and we're done. Figure 7 shows the draft that results from this deduction process.

This algorithm is a natural for languages that support associative arrays, because then you can look up a row or column just by using it directly as an index, rather than searching through a list.

My description of the deduction algorithm was optimized for simplicity and an appealing visual presentation, rather than efficiency. Given the speed of today's computers, and the fact that even mechanized drafts are rarely larger than a few hundred cells on a side, this algorithm effectively runs instantaneously.

Weaving language

Procreate's art program Painter contains an interesting weaving language. The basic ideas are terrific and let you create great-looking patterns.

The Painter weaving language has no name that I know of, so here I'll call it PWL. Although this language is innovative and powerful, using it can be tricky. The interface is buried deep inside the program, you have to hunt online for the documentation, and it has a lot of idiosyncracies and unexpected limitations. On the other hand, it's inside the Painter program, which means you can use it as another art tool along with all the other tools in the system. In the "Further Reading" sidebar I provided a pointer to a document I wrote to help people use this resource.

For my digital loom, I implemented a variation on PWL. I added a bunch of new commands, changed the syntax a bit, and generalized it in several directions. Because it woudn't be fair to the Painter folks to change

```
T    1 0 0 1 0 0 1 1 0 1 1 0 1 1 0 0
S    3 2 2 0 0 1 3 3 0 1 2 3 2
R    0 1 2 1 0 3 2 3 0 0 2 2 3 3 1 3
```

8 Specifying the different matrices with lists.

their system and still use their name (they might not like the changes I've made), I christened my variant Andrew's Weaving Language, or *AWL*. From now on I'll stick to AWL, but be aware that it's just an extension of the Painter language.

The purpose of AWL is to create a sequence of numbers that can go into the ***S*** and ***R*** matrices. Recall that each column of ***S*** can have only a single 1, and each row of ***R*** has the same restriction. We can specify each of these matrices with just a list of integers. As Figure 1 shows, we count ***S*** down and to the left, and ***R*** up and to the right. Each list entry for ***R*** is in the range $[0, |L_R|-1]$, and each entry for C is a number in the range $[0, |L_C|-1]$. Figure 8 shows an example of how to specify each of these matrices with a list; the ***S*** and ***R*** matrices only need a single number per column and row, respectively, while the matrix has all of the 1s and 0s spelled out. For the tie-up, anything except a 0 is considered a 1.

I implemented AWL as a *postfix* language, which uses a technique known as *stack* notation, or *reverse Polish notation*. Jan Lukasiewicz, a Polish mathematician (1878–1956), invented this style. In postfix notation, we write the *operands* first, and then the *operator*. So rather than writing `2 + 3`, we'd write `2 3 +`. Postfix is nice because we don't need parentheses. The traditional, or *infix*, expression `2 + 3 * 4` is ambiguous. If we multiply first we get `2 + (3 * 4) = 14`, but if we add first we get `(2 + 3) * 4 = 20`. Usually we use *precedence rules* to determine how to proceed. The convention is that multiplication has a higher priority than addition, so that expression would conventionally evaluate to 14. If we wanted to add first, we'd have to write `(2 + 3) * 4`. Postfix gets around this problem. To add first, you could write `2 3 + 4 *`, or even `2 3 4 + *`.

These last two forms are the same because the evaluation uses a *stack*. The standard metaphor for a stack is a pile of cafeteria trays: you remove them one at a time from the top, and new ones get added to the pile one at a time, to the top. Each time we see a number, we **`push`** it onto the top of a stack. When an operator comes along, it **`pops`** its operands off the stack, computes with them, and then does a **`push`** to put the result back on. In a well-formed expression, when you reach the end there's just one entry remaining in the stack, holding the final value of that computation. Probably the best-known and most widely used postfix language used today is PostScript, the language that's used by computers to communicate with printers and specify page layouts.

Postfix expressions are appealing from a programming point of view because they're much easier to implement than more traditional *infix* expressions.

Getting started

Before we get going, let's define a few terms. To make things easier for this discussion, I'll speak only about matrix ***S***, which describes the warp threads and their associated treadles. Everything is the same for the weft and treadle matrix ***R***, with just the obvious change to the matrix sizes. It also applies to the tie-up matrix ***T***.

It's hard to present a language like this without it resembling a big shopping list. Each command deserves a moment's explanation, however, and that inevitably turns into a big list. But we can make that list as succinct as possible by establishing some conventions first, and then using them to keep the discussion focused just on what each operator does.

The *domain* is a pair of numbers that specify a range of the available shafts. The domain is initialized to the range $[0, |L_S| - 1]$. Though I start at 0 for convenience, many weavers start counting at 1. The purpose of the domain is to let us conveniently create *runs*, or sequences of numbers that count up or down. A *complete run* cycles through the entire domain. For example, suppose that we specified that we have seven treadles. Then a complete run might be `0 1 2 3 4 5 6`. But we can start anywhere, so another complete run is `3 4 5 6 0 1 2`. If we set the domain to the range [2, 5], then a complete run might be `3 4 5 2`.

As you've probably guessed, all numbers are adjusted to the domain using modulo arithmetic for the current domain when they're generated. So if we have seven treadles, no matter how we calculate them, the only numbers that actually come out at the end of the process are in the range [0, 6]. If we compute the sequence `4 5 6 7 8 9`, then that would become `4 5 6 0 1 2`.

The language has only three different elements: operators (which are identified by name, like **`reverse`**), and two different kinds of operands: *scalars* (or individual numbers), and *sequences*, or lists of numbers.

To create a sequence we can just list the numbers, separated by spaces. Now suppose that we want to follow a sequence by another operand that's just a single number. For example, the **`rotate`** command takes a sequence and a number, and rotates the sequence that many steps. If the sequence is `1 2 3 4` and we want to rotate it two steps, in an infix language we could write `1 2 3 4 rotate 2`. But in postfix we can't write `1 4 3 2 2 rotate`, because the list `1 4 3 2 2` looks like a single big list. In this case we might say that the last element on the list is the one we want, but many operators take two lists as input, so we need to distinguish where one list ends and the next begins.

The trick is to use the **`push`** command after a list. That tells the system that all the numbers that have been given since the last operator are to be interpreted as a single operand, and get pushed on the stack that way. So we'd write `1 4 3 2 push 2 rotate`. Note that we didn't need a **`push`** in front of **`rotate`**, since the command implicitly ends the operand that precedes it. Adding a **`push`** there wouldn't hurt, but we don't need it.

Figure 9 shows a screen shot of my digital loom in action. We type in an AWL expression into the Warp pattern window in the AWL form, and press the associated button. It evaluates the expression, and copies the

resulting sequence into the Warp pattern field in the weaving form. We can also type an expression into any of the other AWL fields and they get copied into the weaving form. We can use AWL, then, to specify the spacing, thickness, and color patterns for the threads (in addition to the ***S***, ***R***, and ***T*** matrices).

Finally, I'd like to cover an idea called *reshaping*. Some operators take two lists and combine them in a way that only makes sense if both lists have the same length. For example, the operator `interleave` creates a new list by folding together two others. If the first operand is `1 2 3` and the second is `7 8 9`, we'd write this as `1 2 3 push 7 8 9 interleave`, resulting in `1 7 2 8 3 9`. The similar expression `2 3 push 3 4 5 interleave` wouldn't make sense, since the left operand `2 3` has length 2, while the right operand `3 4 5` has length 3. This kind of thing comes up a lot, and in many cases the right thing to do is to repeat the shorter operand until it's as long as the longer one. So in this case, we'd just repeat the operand `2 3` to create `2 3 2`. This process is called *reshaping*. If both lists are the same size, nothing happens to either. Many of the operators automatically perform a reshaping step before they go to work. This means that our earlier expression `2 3 push 3 4 5 interleave` would be evaluated as though it was `2 3 2 push 3 4 5 interleave`.

There's no provision for turning reshaping off, since if the operands were of different sizes and we didn't reshape, it would raise an error.

Most operators take one or more elements off of the stack, process them, and then push a result back on the top. If the operand is a sequence, I'll write it as **A** or **B**. If it's a single number, I'll write that as *c* or *d*. Individual elements of sequence **A** are written $\mathbf{A}_i$. So $\mathbf{A}_0$ is the first number. I'll write $\mathbf{A}_L$ to represent the last number in the sequence and $|\mathbf{A}|$ to refer to the length of **A**. So if **A** is `1 3 5 7`, then $\mathbf{A}_0 = 1$, $\mathbf{A}_L = 7$, and $|\mathbf{A}| = 4$.

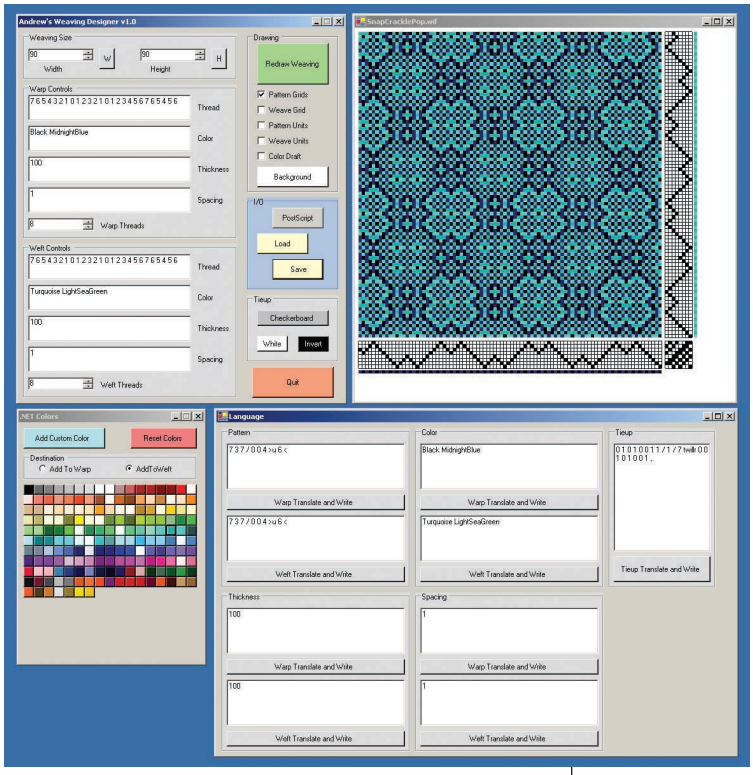
Operands are popped off the stack in the reverse order from the definition of the operator. For example, suppose we define an operator named `combine` as **A B** `combine`. If we enter an expression `1 2 push 3 4 5 combine`, then we first pop `3 4 5` and it becomes **B**, and then we pop `1 2` which is treated as **A**.

In all of the following examples, the domain is [0,7]. Many commands have symbolic shortcuts, which are given in parentheses right after the command. Note that the shortcuts are made out of symbols, so that they don't eat up any more words (because we can use AWL to create color expressions that use color names, any word used by an AWL command becomes unavailable as the name of a color). Many of the shortcuts are similar to those in PWL, but there are a few changes and lots of additions.

Basic operators

There are a few operators that I think of as "basic," because they're pretty straightforward.

A *d* `extend (+)`. If $|\mathbf{A}| > d$, truncate **A** to *d* elements. If $|\mathbf{A}| < d$, repeat **A** as needed until there are at least *d* elements, and then truncate that result after *d* elements.



9 A screen shot of my digital loom.

```
1 2 3 4 5 push 3 extend = 1 2 3
      1 2 push 3 extend = 1 2 1
```

A *d* `repeat (*)`. Repeat **A** a total of *d* times.

```
1 2 3 push 3 repeat = 1 2 3 1 2 3 1 2 3
```

A `reverse (@)`. Reverse the order of the elements of **A**.

```
1 3 5 7 reverse = 7 5 3 1
```

A *d* `rotater (>>)` and **A** *d* `rotatel (<<)`. Rotate the elements of **A** to the right (or left) by *d* steps.

```
1 2 3 4 5 push 2 rotater = 4 5 1 2 3
```

A *c* `nth`. Build the new sequence from element 0 of **A**, then skip *c*–1 elements, take the next, skip another *c*–1, take the next, and so on.

```
1 2 3 4 5 6 7 push 2 nth = 1 4 7
```

A `palindrome (|)`. The output is **A** followed by the reverse of **A**, except that the first and last elements of **A** aren't included in the reversed version. We don't repeat the first and last elements because we want to avoid flat spots both when making the new sequence, and when we repeat one after the other. For example, if we didn't do this then `1 2 3 palindrome 2 repeat` would be `1 2 3 3 2 1 1 2 3 3 2 1` rather than `1 2 3 2 1 2 3 2`, which is almost always more appropriate for creating weaving drafts.

```
1 2 3 4 5 palindrome = 1 2 3 4 5 4 3 2
```

Up and down

We use these operators to create runs, or ascending and descending integers within the current domain. As we saw last month, many drafts consist of these runs, so it's useful to have a bunch of convenient ways to specify them.

A B `down (>)` and **A B** `up (<)`. For `down`, the result is **A**, followed by a run descending from the last element of **A** to the first of **B**.

```
1 5 3 push 6 3 down = 1 5 3 2 1 0 7 6 3
```

The command `up` is the same, but the run ascends.

A B *c* `downloop (>l)` and **A B** *c* `uploop (<l)`. The command `downloop` is like `down`, but inserts *c* complete runs in addition to the single descending run.

```
1 2 3 push 6 3 push 1 downloop
  = 1 2 3 2 1 0 7 6 5 4 3 2 1 0 7 6 3
```

The command `uploop` is the same, but each run ascends.

A B `downup (>u)` and **A B** `updown (<d)`. Reshape the inputs. For `downup`, take the first element of **A** and insert a descending run to the first element of **B**. Now ascend to the second element of **A**, descend to the second element of **B**, and so on.

```
1 2 3 push 6 7 downup
  = 1 0 7 6 7 0 1 2 1 0 7 0 1 3 2 1 0 7 6
```

The command `updown` is the same, but the alternation begins with an ascending run.

A B *c* `downuploop (>ul)` and **A B** *c* `updownloop (<dl)`. The command `downuploop` is like `downup`, but inserts *c* complete runs in each inserted sequence.

```
1 push 5 push 2 downuploop
  = 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5
```

The command `updownloop` is the same, but the alternation begins with an ascending run.

A B `ramp (-)`. This creates a run from $\mathbf{A}_L$ to $\mathbf{B}_0$, but it does so entirely within the domain. Therefore, if $\mathbf{A}_L < \mathbf{B}_0$, it creates an ascending run. Otherwise, it's a descending one.

```
2 push 5 ramp = 2 3 4 5
7 push 5 ramp = 7 6 5
```

A B *c* `ramploop (-l)`. This command is like `ramp`, but like `uploop` or `downloop` it includes *c* full runs.

Advanced operations

The previous commands were all designed to do basic operations on sequences, or create simple runs. This next batch of commands lets us make more complex patterns.

A `binary0` and **A** `binary1`. Treat **A** as the length of alternating sequences of `0`s and `1`s. The command `binary0` starts with `0`, while `binary1` starts with `1`. This is a convenience command mostly useful for specifying tie-ups.

```
3 1 4 2 2 binary0 = 0 0 0 1 0 0 0 0 1 1 0 0
```

A B `block (#)`. Reshape the inputs. Each entry $\mathbf{A}_i$ is repeated $\mathbf{B}_i$ times.

```
3 4 5 1 2 push 2 3 block
  = 3 3 4 4 4 5 5 1 1 1 2 2
```

A B `blockpal (#p)`. The command `blockpal` is like a `block`, but it first processes its inputs by making a palindrome of them, treating the inputs as *pairs*. So in the following example, the pair `1 2` is considered the first element so it's not repeated at the end, and the pair `2 2` is the last element, so it's not repeated in the middle.

```
1 3 2 push 2 4 2 blockpal
  = 1 1 3 3 3 3 2 2 3 3 3 3
```

name c d `eis`. The value of *name* is a string, typically a single letter followed by digits. It specifies an index number in the *Encyclopedia of Integer Sequences*, which is a massive reference work containing tens of thousands of interesting integer sequences (see the "Further Reading" sidebar for more information). From that named sequence, we skip the first *c* entries, and then extract the next *d* values.

A B `growblock (=)`. First, reshape the inputs. Each element $\mathbf{A}_i$ is followed by a ramp to $\mathbf{B}_0$ (recall that goes up or down, as needed, to stay within the run). At the end of the run, a palindrome of the first *i* elements of **B** is inserted, and then a run to $\mathbf{A}_{i+1}$ is made.

This diagram shows the result for input sequences with three elements each. The dash (–) stands for a ramp.

```
A0–B0–A1–B0B1B0–A2–B0B1B2B1B0
0 1 2 push 4 5 6 growblock
  = 0 1 2 3 4 3 2 1 2 3 4 5 4 3 2 3 4 5 6 5 4
```

The command `growblock` was inspired by an analysis of shadow weaves by Ralph Griswold in one of his monographs (see the "Further Reading" sidebar).

A `iblock (i#)` and **A** `iblockpal (i#p)`. Like `block`, but the values are interleaved in one operand. Thus element $\mathbf{A}_0$ is repeated $\mathbf{A}_1$ times, element $\mathbf{A}_2$ is repeated $\mathbf{A}_3$ times, and so on.

The command `iblockpal` is like `blockpal`, building palindromes from its input pairs, except that they're taken in interleaved fashion.

A B `interleave (%)`. This command first

reshapes its inputs. Then it creates a new string by taking each element of **A** and then **B** in turn.

```
1 2 3 4 push 9 7 5 interleave
  =1 9 2 7 3 5 4 9
```

A B permute. First, repeat **A** until its length is an integer multiple of the length of **B**. That is, create a new vector **A′** by choosing the smallest r such that $r|\mathbf{A}|/|\mathbf{B}|$ is an integer.

Now create a new vector **B′** by repeating **B** so that it is the same length as **A′**, but with a twist. Add 0 to the first repeat of **B**, and $|\mathbf{B}|$ to the second, $2|\mathbf{B}|$ to the third, and so on. For example, if **A** = `1 2 3 4` and **B** = `3 1 2`, then we choose $r = 3$ and create

```
A′=1 2 3 4 1 2 3 4 1 2 3 4
B′=3 1 2 (3+(3 1 2)) (6+(3 1 2))
  (9+(3 1 2))=3 1 2 6 4 5 9 7 6 12 10 11
```

The output is then found by using each element of **B′** as the index from **A′**. That is,

$$\mathbf{A}'_{\mathbf{B}'_0}, \mathbf{A}'_{\mathbf{B}'_1}, \mathbf{A}'_{\mathbf{B}'_2}, \ldots \mathbf{A}'_{\mathbf{B}'_p},$$

where p is the length of either of the new vectors. The entries of **B′** are taken modulo p.

```
1 2 3 4 5 6 push 1 0 permute=2 1 4 3 6 5
3 4 5 push 2 1 permute=5 4 4 3 3 5
```

A B pbox. This is a convenience for the following operation, where $|\mathbf{A}|$ is the length of **A**:

```
A B |A| extend permute
```

A B tartan and **A B tartanpal**. These are a small variation on the **iblock** and **iblockpal** commands. Some tartan descriptions double the thread count for each entry, and AWL expects single counts. So after reshaping, each $\mathbf{A}_i$ is repeated $\mathbf{B}_i/2$ times.

A B template (:). Replace each entry in **A** with a little pattern based on the elements of **B**. Weavers call this process *subarticulation*.

First, create a new vector **C** with the same length as **B** (that is, $|\mathbf{C}| = |\mathbf{B}|$) and initialize **C** to all `0`s. Now compute $\mathbf{C}_i = \mathbf{B}_i - \mathbf{B}_0$ for all $i = [1, |\mathbf{B}|]$. Thus the first element of **C** is 0, and all other elements are the signed distance of each $\mathbf{B}_i$ from $\mathbf{B}_0$. So if **B** = `6 7 5` then **C** = `0 1 – 1`, and if **B** = `3 4 5 1 2` then **C** = `0 1 2 – 2 – 1`.

Compute the output by replacing each element $\mathbf{A}_i$ with the new $|\mathbf{C}|$-length vector $\mathbf{A}_i + \mathbf{C} - 1$.

```
0 1 2 3 push 2 template=1 2 3 4
0 3 6 push 2 3 1 template
  =1 2 0 4 5 3 7 0 5
```

A *c d* **twillr (t>>)** and **A** *c d* **twilll (t<<)**. Make *c* repeats of **A**, each time rotating it to the right (or left) one more time than before.

```
1 2 3 4 push 3 push 1 twillr
  =1 2 3 4 4 1 2 3 3 4 1 2
```

Utilities

This last batch of commands are for utility, stack management, and bookkeeping functions.

- **clear**. Erase the entire stack.
- **concat (,)**. Take the top two elements from the stack, create a new sequence by placing the second after the first, and push that result back on the stack.
- **dup**. Get the top item on the stack and push a new copy of it onto the stack.
- **pop**. Discard the top element of the stack.
- **push (/)**. Take everything up to now and treat it as a single element of the stack.
- *c d* **domain**. Set the limits on the domain to [*c*,*d*].
- **swap**. Swap the top two entries on the stack.
- **veclen**. Find the length of the top element on the stack and push that value.
- **vmax** *and* **vmin**. Push the value of the largest (or smallest) element in the sequence on top of the stack.

A summary of all these commands appears in Table 1 (next page).

Tie-up and example

The tie-up is a matrix of 1s and 0s. When applying the result of an AWL expression to a tie-up, the language treats anything that's not a 0 as a 1. The output is automatically extended as necessary to make it the correct size. We apply elements to the tie-up using Figure 1's indexing.

Even short AWL expressions can easily become very long, impenetrable sequences of digits. The ability of languages like this to symbolically represent such lists is one of its biggest advantages.

That also poses a challenge for demonstration, since examples of expressions easily expand into giant strings of numbers. So Tables 2 through 4 (on page 85) provide some short examples. In these tables, I assume that the domain is set to [0,7]. In Table 2, the left column is the command that's just been processed, the column to its right is the top of the stack, and to its right is the second item in the stack. Following the expression, it reads **1 push 4 up palindrome 2 push 7 up interleave 2 rotate**. In Table 3, following the expression, it reads **2 3 push 3 2 block 4 5 3 template**. In Table 4, following the expression **1 3 push 6 2 updown 2 nth reverse**.

Languages like AWL abound in idioms, or little constructions that seem to recur frequently. Let's look at one. Suppose we want to create a sequence that uses alternating members of **A** and **B**, giving us $\mathbf{A}_0\ \mathbf{B}_1\ \mathbf{A}_2\ \mathbf{B}_3 \ldots$. The general approach would be like this:

```
A 1 nth B 1 rotatel 1 nth interleave
```

Table 5 (on page 85) shows this in action. Following the expression, it reads **0 1 2 3 push 1 nth A B C D push 1 rotateL 1 nth interleave**.

Using the **domain** command, you can change the

Table 1. Summary of AWL commands.

Command	S*	R*	Summary
A *d* `extend`	+		Repeat or clip **A** to *d* elements
A *d* `repeat`	*		Repeat **A** a total of *d* times
A `reverse`	`@`		Reverse the elements of **A**
A *d* `rotatel`	<<		Rotate **A** left by *d* steps
A *d* `rotater`	>>		Rotate **A** right by *d* steps
A *d* `nth`			Take every *d*th element of **A**
A `palindrome`	|		**A** followed by a near-reversal
A B `down`	>		**A**, descending run from $\mathbf{A}_L$ to $\mathbf{B}_0$, **B**
A B *c* `downloop`	>l		Like `down` but include *c* runs as well
A B `downup`	>u	√	Alternating down and up runs
A B *c* `downuploop`	>ul	√	Like `downup` but include *c* runs as well
A B `up`	<		**A**, ascending run from $\mathbf{A}_L$ to $\mathbf{B}_0$, **B**
A B *c* `uploop`	<l		Like `up` but include *c* runs as well
A B `updown`	<d	√	Alternating up and down runs
A B *c* `updownloop`	<dl	√	Like `updown` but include *c* runs as well
A B `ramp`	–		Go up or down as needed to stay in domain
A B *c* `ramploop`	–l		Like `ramp` but include *c* runs as well
A `binary1`			Treat **A** as lengths of alternating `1`s and `0`s
A `binary0`			Treat **A** as lengths of alternating `0`s and `1`s
A B `block`	#	√	Each $\mathbf{A}_i$ is repeated $\mathbf{B}_i$ times
A B `blockpal`	#p	√	`block` with an internal `palindrome`
name c d `eis`			Extract *d* elements, starting at *c*, from EIS *name*
A B `growblock`	=	√	Interleave **A** with growing palindromes of **B**
A `iblock`	i#	√	Like `block` but the inputs are interleaved in **A**
A `iblockpal`	i#p	√	`iblock` with internal `palindrome`
A B `interleave`	%	√	Take alternating elements of **A** and **B**
A B `permute`			Use elements of **B** to index **A**
A B `pbox`			Shortcut for **A B** |**A**| `extend permute`
A `tartan`		√	Like `iblock` but repeats are $\mathbf{B}_i/2$
A `tartanpal`		√	`tartan` with internal `palindrome`
A *c d* `twillr`	t>>		Make *c* repeats of **A** rotating each by *d* more
A *c d* `twilll`	t<<		Like `twillr` but rotate left
A B `template`	:		Create a subarticulation using **B** as a template
`clear`			Erase the stack
`concat`	,		Concatanate the top two stack elements
`dup`			Pop the top of stack and push it back twice
`pop`			Discard top of stack
`push`	/		Consider all since last command a single sequence
c d `domain`			Set the domain to [*c*, *d*]
`swap`			Exchange top two stack elements
`veclen`			Push length of list on top of stack
`vmax`			Push largest element in sequence on top of stack
`vmin`			Push smallest element in sequence on top of stack

* The S column provides the symbolic shortcut, if available. If the R column is checked, the command reshapes its inputs.

domain as often as you like during an expression. So for example, in this expression

```
0 / 7 domain 4 / 2 up 1 / 5 domain 4 /
  2 up concat
```

the domain is first set to [0,7], and then an upward sequence `4 5 6 7 0 1` is generated (I used the shortcut `/` for `push`). Then the domain is set to [1,5] and the same run is specified, but this time the domain limits it, and we get `4 5 1 2`.

Sometimes there are several good ways to write a sequence. In the following discussion, I'll use AWL's symbolic shortcut names for simplicity; recall that `push` is `/`, `repeat` is *, `concat` is `,`, `block` is #, `interleave` is %, and `template` is `:`. Let's try to find a compact way to write the following:

```
7 6 7 6 7 6 7 6 5 4 5 4 5 4 5 4 3 2 3 2
  3 2 3 2
```

One way to do this is to note that we have four repeats of the sequence `7 6`, then four of `5 4`, and four of `3 2`, so we might write

```
7 6 / 4 * 5 4 / 4 *, 3 2 / 4 *,
```

We might instead notice that this is four 7s interleaved with four 6s, and the other pairs follow the same pattern, leading us to write

```
7 5 3 / 4 # 6 4 2 / 4 # %
```

We could also treat each eight-element chunk as a subarticulation on the starting values:

```
7 5 3 / 8 7 8 7 8 7 8 7 :
```

where I used the wraparound feature of modulo arithmetic to get the effect we're looking for. We could further encode that pattern of 8s and 7s:

```
7 5 3 / 8 7 / 4 * :
```

All five of these expressions, from the explicit list of numbers to this most compact result, evaluate the same thing. They're just different ways of looking at and expressing how we see the structure of the patterns. This example wasn't exhaustive by any means—there are several other ways to write this pattern.

We can also play around with pattern languages. For example, suppose you had a sequence `1 2 3` and you wanted to make a new sequence that had each element repeated four times. How would you do it? Remembering that the `block` operator reshapes its inputs, we only need to say `1 2 3 / 4` # and we're done. There are lots of cool tricks like this.

Colors, spacing, and thickness

In my last column, I talked about each thread's thickness as a number between 0 and 100. I also talked about spacing as a number from 0 to 1, but we could just as easily set it to 0 to 100, and divide by 100 internally.

Thus all the language elements discussed above can be typed into the spacing or thickness fields just as easily as they can be typed into the warp and weft pattern fields.

Color is a slightly different issue. All of the examples in the last section were in terms of integer sequences. For some operators (like `up`) that's the only kind of argument that makes sense. But other operators don't care what their operands look like. For example, `interleave` just takes out elements from one input sequence and then the other. Those input sequences can be integers, of course, but they could be anything. In particular, they can be text strings.

In my system, I provide access to color by name. You can use any of the 140 built-in colors in Microsoft's C# programming environment, any of the roughly 40 colors commonly used in Scottish tartans, or any custom colors you create and name yourself.

Thus you can type in something like `Blue Gold Red reverse` and get back the list in opposite order, or something more ambitious like `Blue Gold Red push 2 3 2 block`.

Table 2. An example of stack processing.

Operation	Top of Stack	Second Item in Stack
1 push 4 up	1 2 3 4	
palindrome	1 2 3 4 3 2	
2 push 7 up	2 3 4 5 6 7	1 2 3 4 3 2
interleave	2 1 3 2 4 3 5 4 5 3 7 2	
2 rotate	7 2 2 1 3 2 4 3 5 4 5 3	

Table 3. A second example of stack processing.

Operation	Top of Stack
2 3 push	2 3
3 2 block	2 2 2 3 3
4 5 3 template	5 6 4 5 6 4 5 6 4 6 7 5 6 7 5

Table 4. A third example of stack processing.

Operation	Top of Stack
1 3 push 6 2 updown	1 2 3 4 5 6 5 4 3 4 5 6 7 0 1 2
2 nth	1 4 5 4 7 2
reverse	2 7 4 5 4 1

Table 5. How to create a simple sequence that uses alternating values from two others.

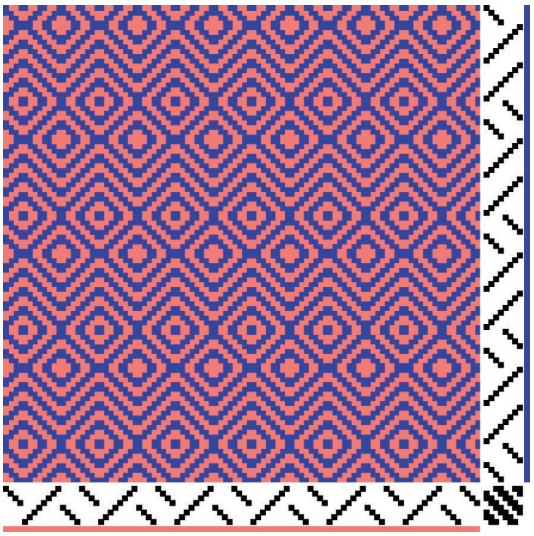
Operation	Top of Stack	Second Item in Stack
0 1 2 3 push	0 1 2 3	
1 nth	0 2	
A B C D push	A B C D	0 2
1 rotatel	B C D A	0 2
1 nth	B D	0 2
interleave	0 B 2 D	

If you type in nonnumerical data into any of the fields except for the color field, the system will raise an error.

Weavings

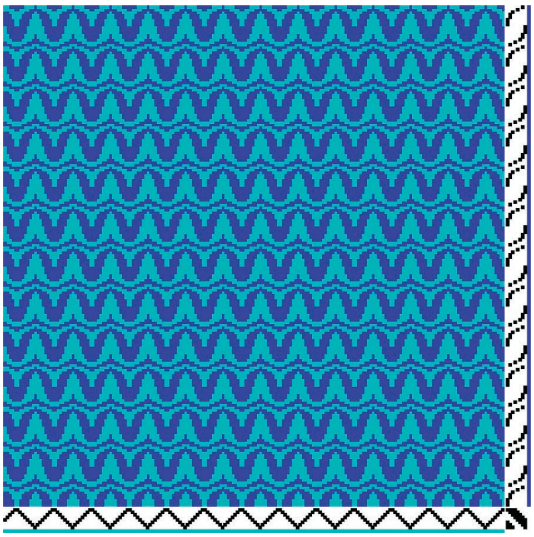
Of course, the whole reason for creating these interesting patterns is to use them to create attractive weavings. Figures 10 through 29 show a variety of different weavings, along with the AWL expressions for the tie-up, warp, and weft. Many beautiful patterns come from simple specifications. Sometimes the expressions are bulky, but they're still a lot shorter than simply listing all the numbers for a given matrix.

```
T  00110011/1/8t<<
S  3/4>0/7<1*,
R  3/4>0/7<2*,
```



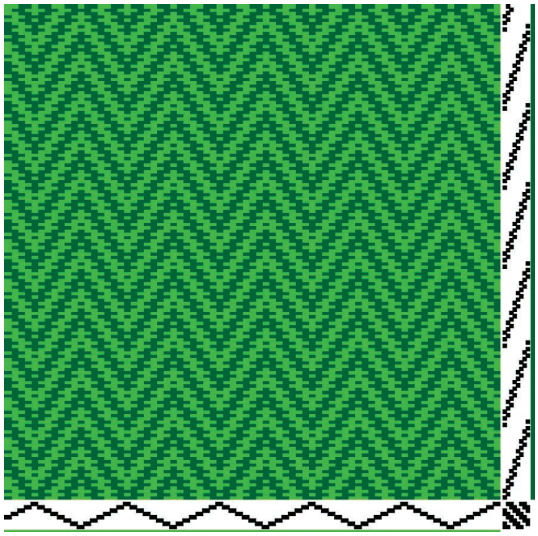
10 *Surveillance.*

```
T  11110000/1/8t<<
S  0/7<0>
R  0001124613566777
```



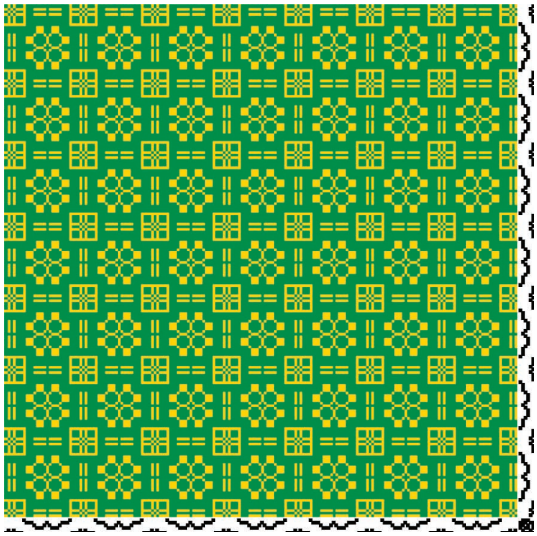
12 *High Seas.*

```
T  01100110/3/8t>>
S  0/7<1>00:
R  0/7<010:
```



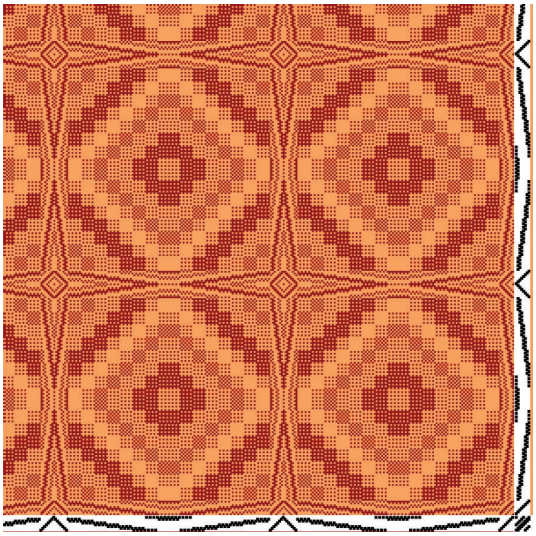
11 *Garden Party.*

```
T  13231411121211112123 11 binary0
S  45/1*45012321/13131311#p,
R  same as S
```



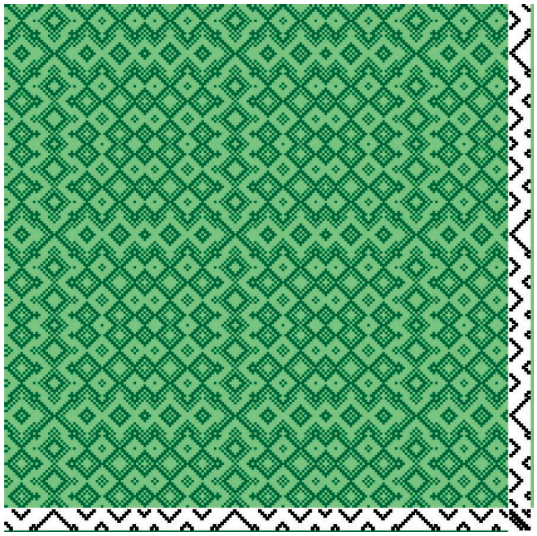
13 *House of Cards.*

T `10001001/1/8t>>`
S `0/7<0/7−12/7+:,|`
R same as ***S***



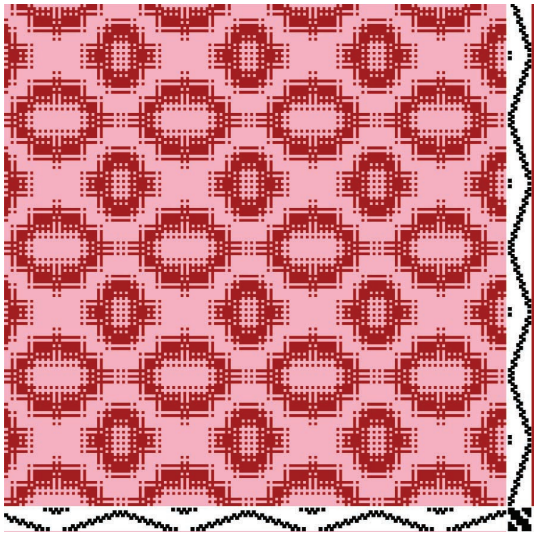
14 *Memories of Sand.*

T `10000101/1/8t<<`
S `243/656>u7/0>,|`
R same as ***S***



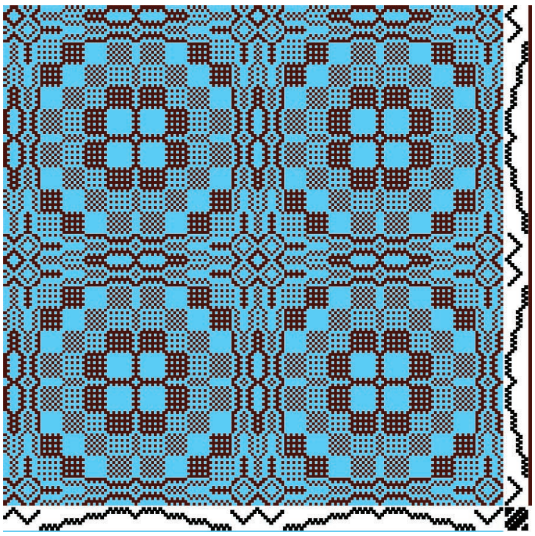
16 *Caffeine Buzz.*

T `237221331146411241236 23 binary1`
S `0/7<121:|`
R `0/7<010:|`



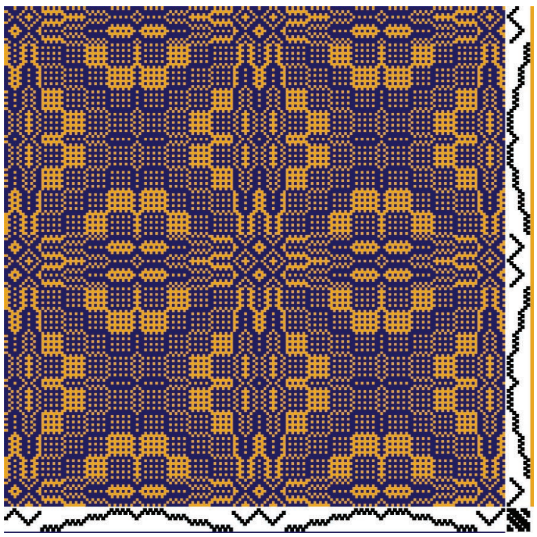
15 *Kiss Me, You Fool.*

T `1100110/1/8t>>`
S `0/5<|753/87/4*:,1,01/4*23,,|`
R same as ***S***



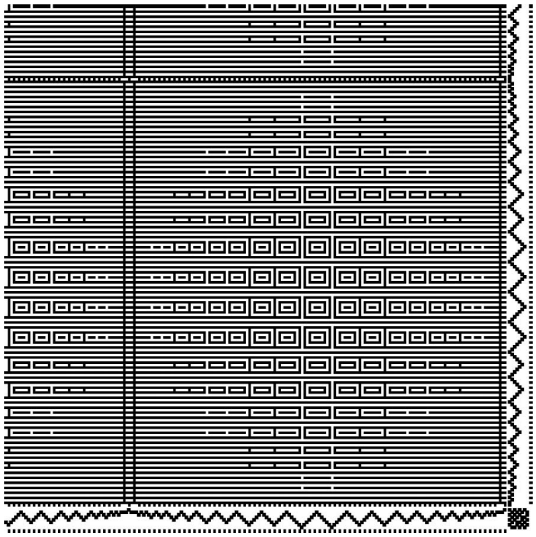
17 *Box Suite #1.*

```
T  11100110/1/8t<<
```



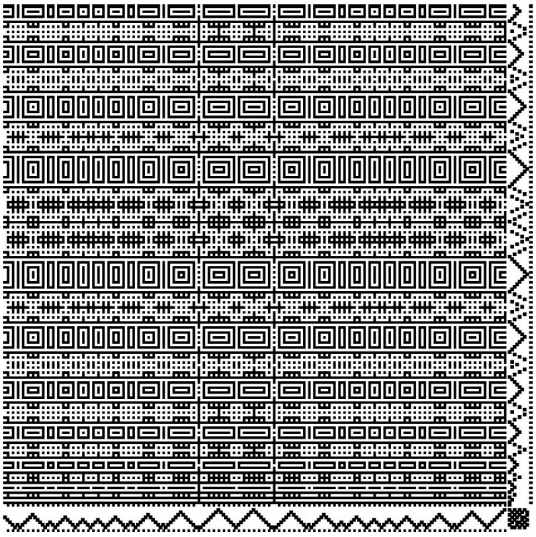
18 *Box Suite #2.* A variation on Figure 17. Only the tie-up has changed.

```
S  0/6-1/7- =|
R  0/6-0/6- =|
```



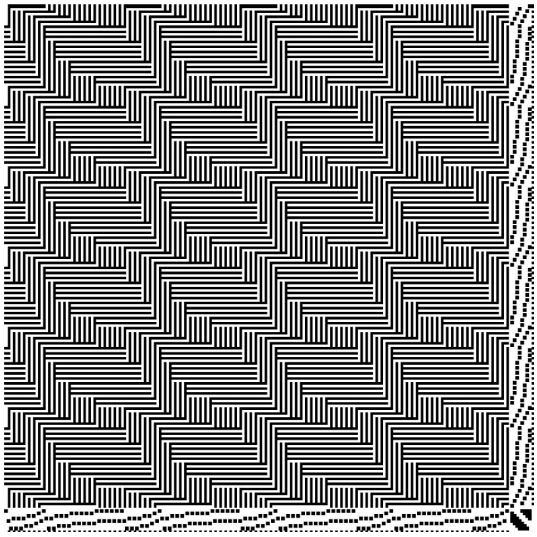
20 *Breaking Out.* A variation on Figure 19. The tie-up is unchanged.

```
T  12/7/1614/14+15,,# binary1
S  0/7<7/0> =|
R  0/7<02461357 =|
```



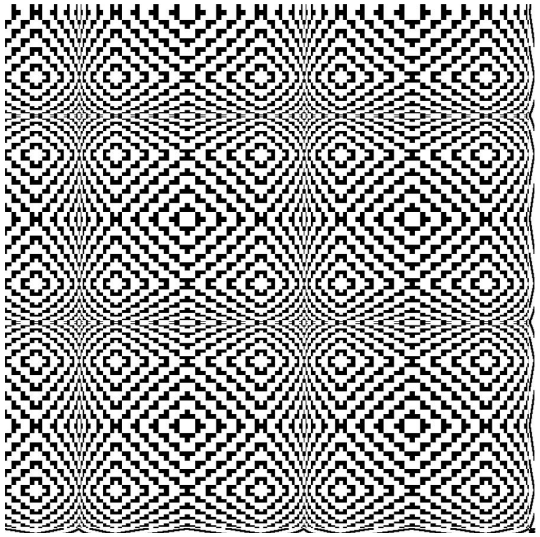
19 *Imperfect Edges.* A shadow weave. Both sets of threads alternate black and white, the warp starting with black and the weft starting with white.

```
T  00001110/1/8t<<
S  0/7-11236532#04:
R  0/7-11112343#04:
```



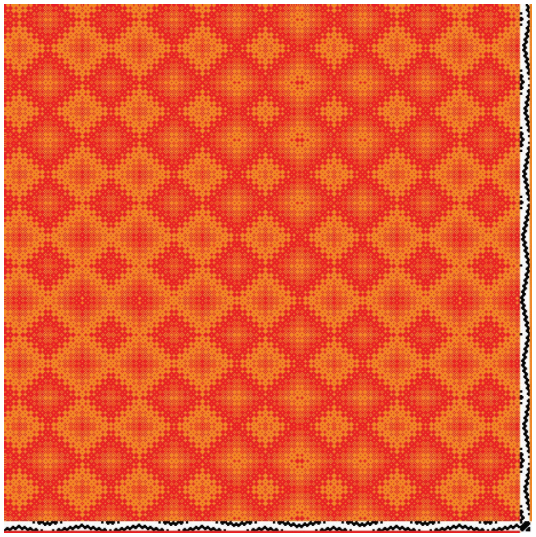
21 *Nude Descending a Fabric.* Another shadow weave.

T `222221311 binary0 22221313 binary1,2*`
S `0/7−2*|12345678/12224568#|1,#11,`
R `0/7−2*|12345678/1234578#|1,#11,`



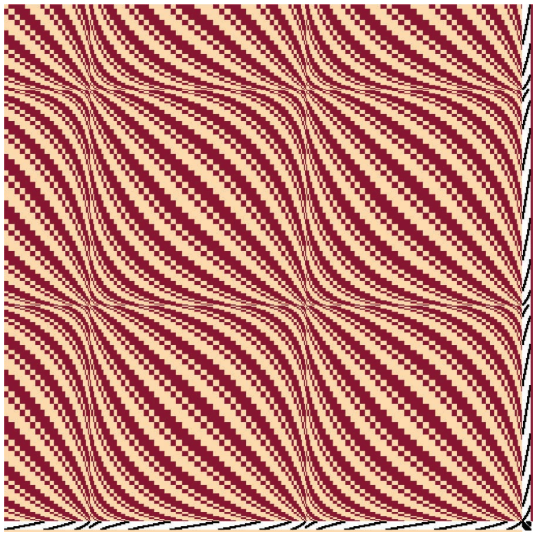
22 *Fly's Eye.*

T `10001110/1/8t>>`
S `0123|3*12345654321:32123:`
R `7654|3*87654345678:12321:`

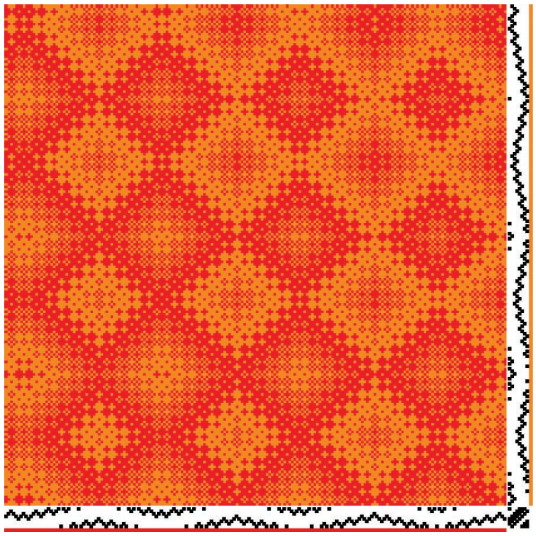


24 *Lucky's Last Chance.* **Subarticulation can be used to make large and subtle patterns.**

T `10001110/1/8t<<`
S `0/7−2*/11212232334 34445455555656666| #`
R same as ***S***

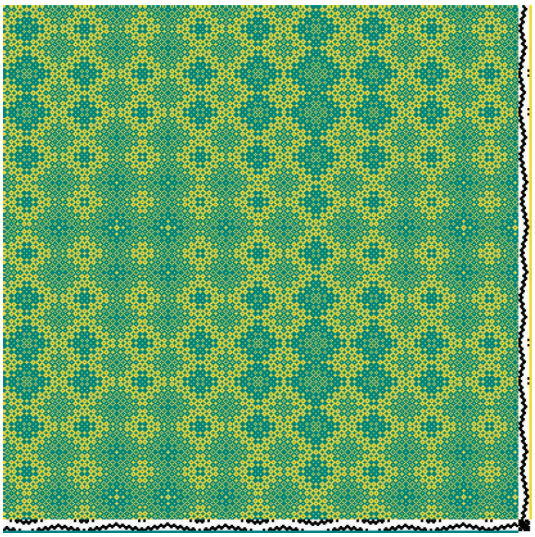


23 *Sandworms.*

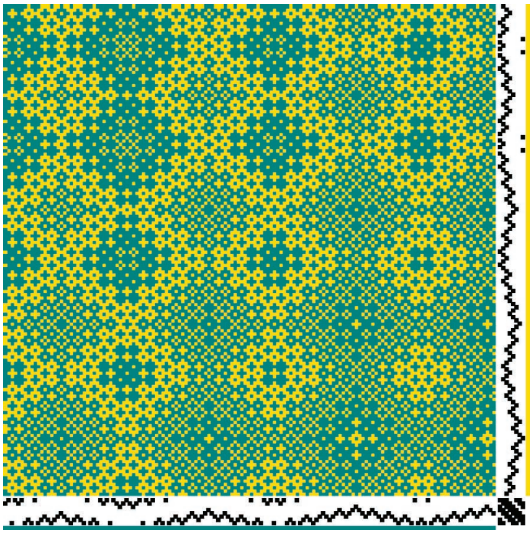


25 **Detail of Figure 24.**

```
T  11100110/1/8t<<
S  0123|3*12345|1,:32123:
R  7654|3*2345432:12321:
```

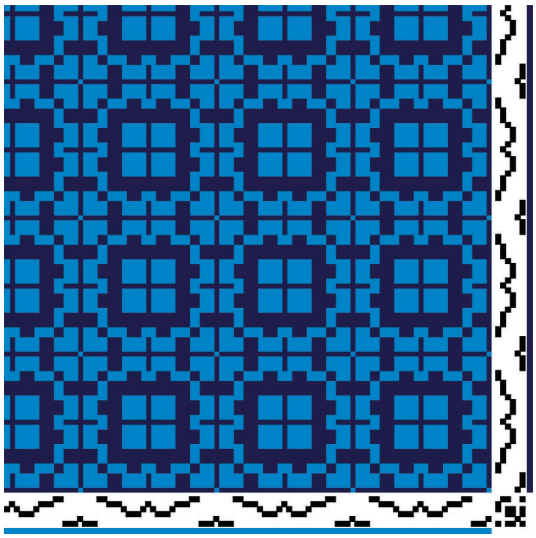


26 *Long Green.* Another use of multiple subarticulations.



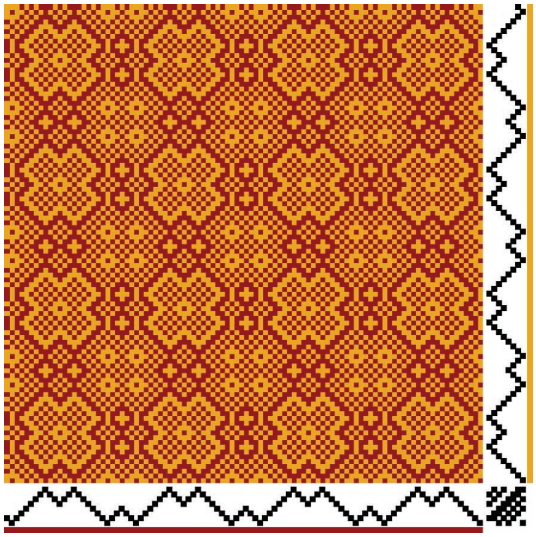
27 Detail of Figure 26.

```
T  5115115113111122112 binary0
S  0/5<|
R  0/5<|
```



28 *The Pen.*

```
T  32332211121214114l1 binary1
S  45012321/13231311#p
R  same as S
```



29 *Waffle Iron.*

The real beauty of having a symbolic language is that it's easy to try experiments. Once you have a pattern you like, you can easily explore variations. How about a subarticulation here? Or a palindrome there? Try starting with something pretty and play with it for a while to come up with something else that you like as much, if not better.

Experimenting in a digital loom is a lot faster than doing it in real life, or even with paper and pencil. Just type in the expression, push the button, and see the result. Once you've created a draft you like, you can then take it to your loom and produce a textile that's both functional and beautiful.

Next time we'll talk about Scottish tartans, and how to implement a digital loom. ■

Readers may contact Andrew Glassner at andrew@glassner.com.