The Icon Analyst

In-Depth Coverage of the Icon Programming Language

August 1998
Number 49

In this issue ...

Character Patterns (continued)

My life is made of patterns that can scarcely be controlled. — Paul Simon [1]

In the last issue of the Analyst, we introduced the concept of character patterns [2], which are sequences of characters in which each different character serves as a label for some value. The values so represented might be numbers, words, colors, nucleotides, musical notes — anything that can be represented in one fashion or another in Icon. A sequence of values might be a chant, a textile design, a DNA sequence, a tune ... .

Character patterns allow Icon's string-processing repertoire to be used to manipulate a sequence of possibly complex values by manipulating their labels.

The assumption is that the such sequences have interesting properties, which we'll call patterns without trying to define the concept precisely.

In the previous article we introduced two concepts related to character patterns:

- *pattern forms*, such as repetitions and reversals.
- *pattern grammars*, which allow the structure of a character pattern to be represented hierarchically.

Limitations

Characters

One obvious limitation of character patterns is the small number of characters available to serve as labels. Of the 256 possible characters, some are reserved for pattern characters — that is, characters used in pattern forms. In addition to the labels for values, characters also are needed for grammar symbols. Others are "u" characters: unsuitable, undesirable, or unavailable. Some of these have no associated glyphs (graphics) and hence cannot be distinguished on-screen. Even some characters with glyphs may be impossible to keyboard directly. The "u" characters vary from platform to platform, of course.

The number of different values that can be successfully handled in terms of character patterns depends to some extent on the nature of the problem domain, but in most cases, a practical limit is about 50.

This, more than anything, determines what kinds of sequences of values can be manipulated as character patterns.

Length

Character patterns, which are just strings with special interpretations, can be very long. The limit on the length of an Icon string, which requires one 8-bit byte per character, is $2^{27}-1 \approx 1.3\times10^{8}$ for 32-bit hardware and $2^{31}-1 \approx 2.1\times10^{9}$ for 64-bit hardware. In other words, you'll run out of memory before you exceed Icon's limit on string length.

Manipulating long strings, however, can be time-consuming, especially for operations that involve pattern matching with bad combinatorial properties.

We have successfully worked with character patterns with tens of millions of characters, but it sometimes was painful and care had to be taken to limit pattern matching.

Pattern Forms

In the first article on character patterns, we introduced only three pattern forms:

s	a specific string
[s,i]	i repetitions of s
<s>	the reversal of s

Note that we changed from parentheses to brackets for repetitions. We'll need parentheses for other, more conventional purposes later.

Reversals can be used to represent palindromes, as in s<s> and sc<s> where c is a single character. They also can be used to represent "near palindromes", such as st<s>, where *t > 1 and t is not a palindrome. We call such patterns *palindroids* (sorry; we couldn't resist).

We started with these forms because they had proved useful in practice. There are many other possibilities, which we'll introduce from time to time.

For this article, we added one more pattern form for *decollation*. In its simplest form, the decollation of a string consists of separating it into two parts: the odd-numbered characters and the even-numbered ones. For example, the decollation of "ababacbacb" produces "aaabc" and "bbcab". If the string is of odd length, the first part is one character longer than the second. As a pattern form, the parts appear enclosed in braces and the parts separated by a comma, as in

```
{aaabc,bbcab}
```

The *collation* of the two parts produces the original string.

Decollation may seem like a strange form to add to the repertoire: it adds three characters for pattern forms, and it increases, not decreases, the number of characters needed to represent the string (or does it?). It also adds two pattern characters and hence reduces the number of available token symbols by two.

But the real question is why add decollation as a pattern form? As for the other pattern forms now in the repertoire, the reason was practical — in some instances it allows the structure of a character string to be made clear. And, it can reduce the number of characters needed.

To see how this can come about, consider the character pattern

```
babcbabcbabcbabcbabcbabc
```

the corresponding decollation is

```
{bbbbbbbbbbbb,acacacacacac}
```

which in turn yields

```
{[b,12],[ac,6]}
```

In this case we get both structure and compactness.

It is easy to extend this idea to allow decollation into more than two parts. Here's a procedure in which s is the string to be decollated and i is the number of parts:

```
procedure decol(s, i)
   local parts, j, form

   parts := list(i, ",")

   s ? {
      repeat {
         every j := 1 to i do {
            (parts[j] ||:= move(1)) | break break
            }
         }
      }

   form := ""

   every form ||:= !parts

   return "{" || form[2:0] || "}"

end
```

We were tempted to further generalize and extend the decollation pattern form to allow for the number of successive characters taken from the for each part to be specified. This can be represented schematically by

$$j_1, j_2, j_3, \ldots, j_n$$

where j_1 characters go to the first part, j_2 to the second, and so on, cyclically. (This is one of those things that's harder to describe than it is to understand. In other words, "you know what I mean".)

For example, the 1, 3, 2 decollation of

```
abcdefghijklmnopqr
```

is

```
agm,bcdhijnop,efklqr
```

There is, of course, the question of what to do if "things don't come out even". We'll take the easy way out: whatever is easiest to code.

The programming is relatively easy. Here's a

procedure that takes a string and a list of decollation widths and returns the corresponding pattern form:

```
procedure decol(s, widths)
   local parts, i, form

   parts := list(*widths, ",")

   s ? {
      repeat {
         every i := 1 to *widths do {
            if pos(0) then break break
            parts[i] ||:= move(widths[i]) | tab(0)
            }
         }
      }

   form := ""

   every form ||:= !parts

   return "{" || form[2:0] || "}"

end
```

We'll leave it to you to figure out what happens if "things don't come out even".

The problem with this is that it's not possible to collate the result to get the original string without also knowing the values of $j_1, j_2, j_3, \ldots, j_n$. In other words, these values would have to be encoded in the pattern form. We decided to defer this until there is evidence that the additional complexity is justified by need.

Grammars

In the previous article on character patterns, we illustrated how formal grammars can be used to represent the structure of character patterns. We used Lindenmayer Systems [3-4], but with the very limited facilities needed to represent character patterns, almost any kind of formal grammar would do.

We'll stick with Lindenmayer Systems (L-Systems), since we have software to manipulate them. We need a little more terminology, which is somewhat specific to character patterns, in order to talk about grammars.

A grammar consists of rules that describe a set of strings, which is called a language. A grammar consists of symbols. We can distinguish three kinds of symbols:

- *pattern characters*, the bracketing characters, comma, and the digits in repetition counts.
- *tokens*, which are the labels that represent values.
- *variables* , whose values are strings of symbols of any of these three types.

In the parlance of formal language theory, pattern characters and tokens are *terminal symbols*, which variables are *nonterminal symbols*. Incidentally, L-systems themselves make no distinctions between different kinds of symbols.

A grammar consists of a set of *definitions* that associate variables with their values. Definitions are called *productions* in formal language theory.

In L-Systems, definitions are written as

$$S \rightarrow S_1 S_2 S_3 \ldots$$

to indicate that the value of the variable S is the string of symbols $S_1 S_2 S_3 \ldots$.

One variable is designated as the *goal symbol*, which is called the *axiom* in L-System terminology. It is the goal symbol that defines the language. Other variables define subgrammars.

L-Systems have parameters, one of which specifies the axiom. It has the form

```
axiom:A
```

which designates A to be the goal symbol.

Here is an example, which includes pattern forms:

```
axiom:A
A –> [B<B>,3]
B –> [CD,12]
C –> [b,4]
D –> [a,4]
```

In this grammar, A, B, C, and D are variables, a and b are tokens, and the brackets, commas, and digits are pattern characters.

Figure 1 shows this grammar graphically.

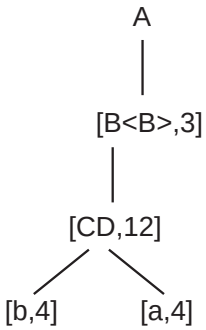


Figure 1. A Grammar Tree

Notice this is a tree. In most applications, grammar graphs have loops that allow the representation of languages with an infinite number of strings. Here there is a finite number of strings and only one that consists entirely of nonterminal symbols.

The Problem

The example above shows the end result of analyzing a character pattern. Using lindsys from the Icon program library, this grammar generates the following string showing pattern forms:

```
[[[b,4][a,4],12]<[[b,4][a,4],12]>,3]
```

To get the original character string, it is necessary to expand the pattern forms. Here's a procedure that does this:

```
procedure expand(pattern)
   local result, i, j, slist, s
   static ochars, cchars

   initial {
      ochars := '[<{'
      cchars := ']>}'
      }

   result := ""

   pattern ? {
      while result ||:= tab(bal(ochars)) do {
         case move(1) of {
            "[" : {
               result ||:= repl(expand(tab(bal(',',
                  ochars, cchars))), (move(1),
                  expand(tab(bal(']', ochars, cchars)))))
               }
            "<" : result ||:= reverse(expand(tab(bal('>',
                  ochars, cchars))))
            "{" : {
               s := tab(bal('}', ochars, cchars))
               slist := []
               s ? {
                  while put(slist, expand(tab(bal(',',
                     ochars, cchars) | 0))) do
                        move(1) | break
                  }
               every i := 1 to *slist[1] do {
                  every j := 1 to *slist do
                     result ||:= slist[j][i]
                  }
               }
            }
         move(1)
         }

      return result || tab(0)
      }

end
```

Using this procedure to expand the pattern forms in example above produces the original character string:

```
bbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbb
baaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbaa
aabbbbaaaaaaaabbbbaaaabbbbaaaabbbbaaaabbbba
aaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaa
bbbbaaaabbbbaaaabbbbbbbbaaaabbbbaaaabbbbaaa
abbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabb
bbaaaabbbbaaaabbbbaaaabbbbaaaaaaaabbbbaaaab
bbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbb
aaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbbbb
baaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbaa
aabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaab
bbbaaaaaaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaa
bbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbb
baaaabbbbaaaabbbb
```

The problem, however, is to go the other way — to convert this 576-character string to a grammar such as the one shown in Figure 1.

Two kinds of facilities are needed: pattern matching procedures to find pattern forms and a mechanism to build grammars from them.

Finding pattern forms cannot be done entirely automatically. There often are many different ways to represent a character pattern by pattern forms, and finding a "good" one often is an intellectual challenge.

On the other hand, building a grammar from

pattern forms can be done automatically. The idea is to start with a grammar that has a single variable, the goal symbol, whose initial value is a character pattern. Then, every time a pattern form is found, it is assigned to a new variable and the corresponding string is replaced by the new variable in all other definitions.

For the example above, the steps are:

1. *Initial grammar:*

```
A –> bbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbb
aaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbb
aaaabbbbaaaabbbbaaaaaaaabbbbaaaabbbbaaaa
bbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaa
bbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbbbbb
aaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbb
aaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbb
aaaabbbbaaaaaaaabbbbaaaabbbbaaaabbbbaaaa
bbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaa
bbbbaaaabbbbaaaabbbbaaaabbbbbbbbaaaabbbb
aaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbb
aaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbb
aaaaaaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaa
bbbbaaaabbbbaaaabbbbaaaabbbbaaaabbbbaaaa
bbbbaaaabbbbaaaabbbb
```

2. *Finding* bbbbaaa *repeated 12 times in* A *(reversals follow automatically):*

```
A –> B<B>B<B>B<B>
B –> [bbbbaaaa,12]
```

3. *Finding* B<B> *repeated 3 times in* A*:*

```
A –> [B<B>,3]
B –> [bbbbaaaa,12]
```

4. *Finding* b *repeated 4 times in* B*:*

```
A –> [B<B>,3]
B –> [Caaaa,12]
C –> [b,4]
```

5. *Finding* a *repeated 4 times in* B*:*

```
A –> [B<B>,3]
B –> [CD,12]
C –> [b,4]
D –> [a,4]
```

Icon on the Web

Information about Icon is available on the World Wide Web at

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

Incidentally, this character string is not contrived. It represents a 6×6 checkerboard pattern in which each square is 4×4. The tokens a and b represent two colors. Figure 2 shows this checkerboard for two shades of gray.

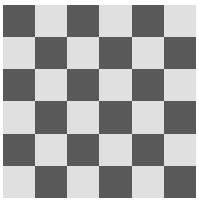


Figure 2. An Image from a Character Pattern

The grammar is not surprising when you look at the character string. There obviously are many occurrences of a and b repeated four times. However, there are 8-character repetitions, 16-character repetitions, and maybe 32-character ones.

You get different grammars depending on what you pick for the initial repetition. Here are a few examples.

```
A –> [DBCCDB,3]
B –> [b,4]
C –> [a,4]
D –> [BC,11]
```

```
A –> [B<B>,3]
B –> [C,6]
C –> [DE,2]
D –> [b,4]
E –> [a,4]
```

```
A –> [DCECB,3]
B –> ED
C –> [B,11]
D –> [b,4]
E –> [a,4]
```

Character patterns with more tokens and more complex structure often have a bewildering number of quite different grammatical representations. It can be an absorbing puzzle to try to find ones that are the smallest, show structure most clearly, and so on.

Next Time

Since the determination of pattern forms requires both pattern-matching procedures and intelligent guidance, a capable application is needed. In addition, in order to "see" patterns and evaluate results, a good visual interface is necessary.

In the next article on character patterns, we'll describe such an application.

References

1. *Patterns*, from *Parsley, Sage, Rosemary and Thyme*, lyrics by Paul Simon, 1966.

2. *Character Patterns*, Icon Analyst 48, pp. 3-7.

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

4 "Anatomy of a Program — Lindenmayer Systems", Icon Analyst 26, pp. 4-9.

Versum Deltas

> *Not being a mathematician, I am not obligated to complicate my explanations by excessive mathematical rigor.* — Petr Beckmann [1]

When we first started working with versum numbers, we computed the deltas (differences) between the first few hundred. This is trivial to do:

```
procedure main()
   local i, j
   i := read() | exit()                # starting value
   while j := read() do {
      write(j – i)
      i := j
      }
end
```

There are obvious patterns both in the actual deltas and in the sequence in which they occur. See Figure 1.

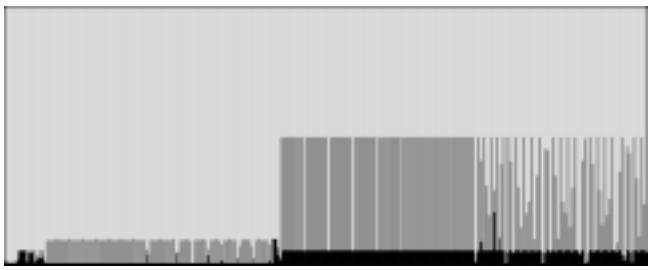


Figure 1. The First 500 Versum Δs

At the time we couldn't see any obvious rule underlying the patterns, and we put the subject aside in order to deal with other aspects of versum numbers.

We recently reviewed some notes made at the time and started to investigate the subject more seriously. By the time we got around to this, we'd learned a lot about versum numbers and also had developed useful tools.

The most important principle we'd learned was that it's particularly helpful to deal with versum numbers according to the number of digits they have. We'd also learned that versum numbers that begin with a 1 are more complicated and difficult to characterize than versum numbers that begin with other digits — and also that those that begin with 3 through 9 differ from those that begin with a 2 by only a constant [2].

If we'd known these things when we first looked at versum deltas, we might have understood more than we did — by looking at the first few hundred deltas, we not only viewed versum numbers with different numbers of digits, but we also intermixed those that begin with a 1 with those that begin with other digits. Look at Figure 1 again in light of what we now know about versum numbers.

Some Notation

Before going on, we need to establish some notation to avoid complicated and repetitive verbiage.

First we'll call versum numbers *versums*. This is a barbarism, but it's easy to remember. And we'll use the Greek letter Δ for "delta".

In what follows, we'll often refer to versums with n digits, for which we'll use n: versums. We'll also need to differentiate between even and odd n, for which we'll use n_e and n_o, respectively. (A more standard way to make this distinction would be to refer to $n = 2m$ and $n = 2m+1$, but we prefer to have the distinction explicit.)

To identify versum numbers that start with the digit i, we'll use the notation i… versums. In some cases the ending digit is significant, for which the natural extension is $i…j$. When needed, we'll add additional digits in the initial and final positions, as in n_e:2…22 versums, which stands for versums with an even number of digits that start with 2 and end with 22. Finally, we'll make use of regular expression notations, such as [i–j]… to denote versums that begin with the digits i though j.

Dividing Up the Problem

As expected, 1… versum Δs are more complicated (and interesting) than 2… versum Δs. 1…versum sequences also have many more *different* Δs than 2… versum sequences. See Figures 2 and 3, which are typical.

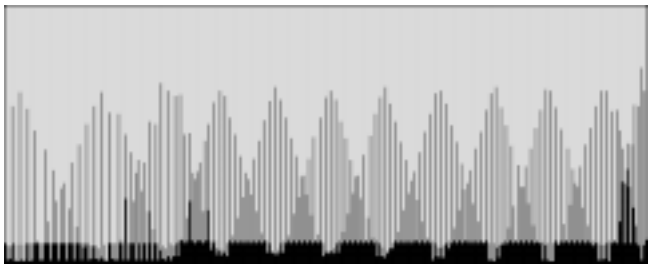


Figure 2. 7:1… Versum Δs

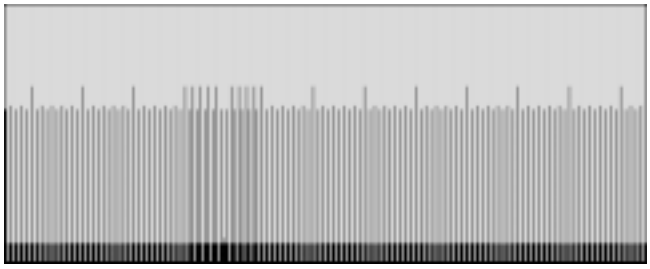


Figure 3. 7:2… Versum Δs

These observations led us to divide the study of versum Δs into 1… versum Δs and 2… versum Δs. Since [3-9]… versum Δs are the same as 2… versum Δs, we only need to consider the additional deltas between each group.

We started with 2… versum Δs, tackling the simpler problem first — and it's hard enough. We'll come back to 1… versum Δs later.

There are surprisingly few *different n*:2… Δs:

n	*no. of Δs*	*no. of different Δs*
1	0	0
2	0	0
3	9	2
4	18	2
5	189	4
6	360	4
7	3609	8
8	6858	7
9	68589	14
10	130320	11
11	1303209	22
12	2476098	16

There also are patterns in the values of versum Δs and in their counts. The patterns for n_e and n_o are different, as shown in Figure 4 on the next page.

The information shown in Figure 4 was obtained from Δ sequences using the following short program:

```
procedure main()
   local tabul

   tabul := table(0)

   while tabul[integer(read())] +:= 1

   tabul := sort(tabul, 3)

   while write(right(get(tabul), 8), right(get(tabul), 8))

end
```

This program expects valid data. What happens if a line of input is not numeric? See the bottom of this column for the answer.

There are patterns everywhere. Where do we start? Or better, what do we hope to accomplish?

This is a recreational problem with no evident connection to "important" problems or the "real world". Consequently, trying to characterize the patterns in a concise way is a reasonable goal. If the experience here is like that for other studies of versum numbers, achieving a deep understanding is unlikely — but we can hope.

It is more motivating to have a concrete goal. If we can find a simple rule for characterizing 2… versum Δ sequences, we could produce versum sequences for larger *n* than allowed by the present brute-force method of testing many candidates. This will be our focus.

The Brute Force Method

Before going on, it's worth looking at the so-called brute-force method of producing 2… versum sequences.

If a line of input is not numeric, integer(read()) fails, terminating the while loop. The program proceeds without comment to produce a tabulation for the previously read lines. In other words, it produces invalid results without any indication there was a problem. This is about the worst kind of a bug there is. The program can be fixed by separating read() and integer() to avoid unexpected failure of the latter:

```
while line := read() do {
   line := integer(line) | stop("*** invalid data")
   tabul[line] +:= 1
   …
```

	Δ	no.		Δ	no.
n=3	1	5	n=4	11	9
	19	4		99	9
n=5	10	90	n=6	11	99
	89	50		99	171
	91	32		891	72
	101	17		990	18
n=7	1	810	n=8	11	999
	99	1710		99	3249
	100	90		891	1368
	121	9		990	342
	791	400		8811	720
	811	320		9801	162
	890	100		9900	18
	911	170			
n=9	10	14580	n=10	11	9999
	89	8100		99	61731
	99	17910		891	25992
	121	9		990	6498
	891	13680		8811	13680
	901	2430		9801	3078
	990	1800		9900	342
	1000	90		88011	7200
	1111	90		97911	1620
	7811	4000		98901	162
	8011	3200		99000	18
	8801	900			
	8900	100			
	9011	1700			
n=11	1	131220	n=12	11	99999
	99	617310		99	1172889
	100	14580		891	493848
	121	9		990	123462
	791	64800		8811	259920
	890	16200		9801	58482
	891	143280		9900	6498
	990	35820		88011	136800
	1111	90		97911	30780
	8811	136800		98901	3078
	8911	24300		99000	342
	9801	16200		880011	72000
	9900	1800		979011	16200
	9901	810		988911	1620
	10000	90		989901	162
	11011	900		990000	18
	78011	40000			
	80011	32000			
	87911	9000			
	88901	900			
	89000	100			
	90011	17000			

Figure 4. Tabulations of *n*:2… Versum Δs

This method consists of generating successive candidates and testing each one to see if it's versum. Here's a naive version:

```
link options
link vpred

procedure main(args)
   local n, i,  first, last, candidate, opts

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

   n := \opts["n"] | 6
   i := \opts["i"] | 2

   if n < 4 then stop("*** invalid digit specification")
   if not (1 <= i <= 9) then
      stop("*** invalid initial digit specification")

   first := i * 10 ^ (n – 1) + i

   last := (i + 1) * 10 ^ (n – 1)

   if i = 1 then last –:= 2            # past, need to reduce
   else if n % 2 = 0 then last –:= 10 – i
   else last –:= 21 – i

   candidate := first

   repeat {
      if vpred(candidate) then write(candidate)
      candidate +:= 1
      if candidate > last then exit()
      }

end
```

The command-line option –n specifies the number of digits and the option –i specifies the initial digit. The initial digit may be 2 through 9. Just 2 would suffice, but the additional generalization does not complicate the computation.

The first and last versum numbers in the sequence, which are easy to determine but different in form i_e and i_o, provide the starting and ending points.

Having initialized the data, the computation consists of generating candidates and testing.

The naive part is incrementing by 1. We know that the last digit is i or $i-1$ [2]. If the last digit of the last number produced is i, it's silly to add 1 — instead add 9. More specifically, if the last digit of the last number produced is i, add 1, otherwise 9. Furthermore, since adding 1 takes the last digit to i, and adding 9 takes it to $i-1$, we can just alternate.

We can do much better than this for i_e because all i_e versums are divisible by 11. Consequently all i_e versums Δs are divisible by 11. The leads to

alternate increments of 11 and 99. The code looks like this:

```
        ...
    delta := 9
    delta_alt := 1
    if n % 2 = 0 then {
      delta := 11 * delta
      delta_alt := 11 * delta_alt
      }
        ...
    repeat {
      if vpred(candidate) then write(candidate)
      candidate +:= delta
      if candidate > last then exit()
      delta :=: delta_alt
      }
        ...
```

We have, therefore, reduced the number of candidates by a factor of 4.5 for i_o and 45 for i_e. The method still deserves to be labeled brute-force, if somewhat more subtle.

It might seem that this improvement would be adequate for producing 2... versum sequences for moderately large *n*. For just *n* = 14, however, the number of 2... versums is 470,458,449. In order to get these, vpred() must be called this number of times *and succeed*, even if every candidate produced a versum number. Of course, most don't. Clearly, we need a method that does not involve calling vpred() an astronomical number of times.

By the way, do not ask why one would want to compute versum sequences in the first place. The problem is a puzzle and an end in itself. Its existence is enough reason — the "because it's there" response.

Versum Δ Grammars

One of the problems in studying versum Δ sequences is that they are very long — too long to get overall insight by using images like those shown in Figures 2 and 3.

For 2... versum Δs, the number of different Δs is small enough to allow each one to be represented by a character and hence deal with versum Δ sequences as strings. This leads us to analyze character patterns in terms of pattern forms and grammars, as described in the last issue of the Analyst [2]. That article used the 6:2... versum Δ sequence as an example.

That sequence has a near-palindromic form. This is not an accident. All n_e:2... versum Δ sequences have the *same* form. Moreover, all n_o:2... versum Δ sequences have a true palindromic form. Here are the top-level productions. The symbol A is just a label; its structure is different for n_e and n_o, and of course for different values of *n*.

n_e:2...	Ab<A>a	$n_e \geq 6$
n_o:2...	Ad<A>	$n_o \geq 5$

For smaller *n*, the patterns are degenerate — there's not enough "room" for them to be fully articulated.

Furthermore, a, b, and d always represent the same Δs:

a →	11	$n_e \geq 6$
b →	99	$n_e \geq 6$
d →	121	$n_o \geq 7$

The significance of these observations is that it's only necessary to compute about half of a 2... versum Δ sequence: the sequence corresponding to A in the grammar. The rest can be constructed from the grammatical forms. A factor of two may not seem like much, given the previous discussion, but it's welcome nonetheless.

Here are some specific examples. Starting with n_e, for *n* = 6, a grammar is:

```
axiom:*
* –> Ab<A>a
A –> BD
B –> [bda,9]
C –> [babc,9]
D –> [Cba,4]
```

The tree for A in this grammar is shown in Figure 5.

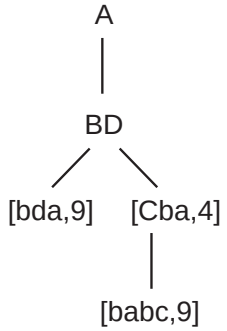


Figure 5. An A Tree for the 6:2... Δ Sequence

For *n* = 8, a grammar is:

```
axiom:*
* –> Ab<A>a
A –> BDEbdaG
```

```
B –> [bdga,9]
C –> [bdabde,9]
D –> [Cbda,8]
E –> [bdabf,9]
F –> [abcbfb,9]
G –> [bFabcbaI<F>,4]
H –> [bcbebcba,9]
I  –> [Hbcba,8]
```

The tree for A in this grammar is shown in Figure 6.

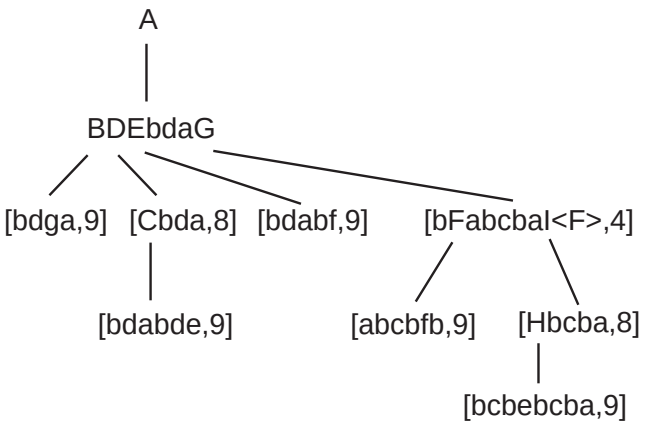


Figure 6. An A Tree for the 8:2… Δ Sequence

Notice the similarities between the definitions for B, C, and D in Figures 5 and 6. There is, in fact, a pattern to the first few productions for all $n_e \geq 6$ and probably for later ones. Notice also the similarity between the subtrees below D and I in Figure 6.

As *n* gets larger, other new productions emerge, but always at the end (at least the way we've chosen to create them). That is, A –> BD for *n*=6, A –> BDEbdJa for *n* = 8. For *n* = 10, there is a similar extension.

One of the difficulties in getting a more precise characterization of such grammars is that there are many possible grammars for a given *n*; when constructing them, it's easy to wander away from what might be the desired ones.

n_o grammars are noticeably different from n_e ones (as Figure 4 shows, there also are more n_o Δs than n_e Δs for comparable *n* as *n* gets larger). Here's a grammar for *n* = 5:

```
axiom:*
* –> Ad<A>
A –> BDE
B –> [bad,5]
C –> abacabaca
D –> [Ed,3]
E –> Cb<C>
```

For *n* = 7, a grammar is:

```
axiom:*
* –> Ad<A>
A –> BDEFJHIb
B –> [bgch,5]
C –> [babgcf,4]
D –> [Cbabgch,8]
E –> [babgh,4]
F –> babgdb<I>ebabhbabe
G –> [babfbabe,4]
H –> [Gbabhbabe,8]
I  –> [bhbabe,4]
J  –> [HIbdb<I>ebabhbabe,3]
```

As noted earlier, there are grammars for $n_o \geq 5$ that have the same top-level form, but similarities at lower levels are more elusive than for n_e.

Note: Many versum grammars can be made smaller by creating productions for frequently occurring strings of tokens. For example, in the *n* = 8 example above, bab occurs 13 times and could be replaced by a production such as

```
K –> bab
```

This has the effect of making the location and nature of such token strings stand out, as well as reducing the size of the grammar. Doing this, however, often obscures the overall structure of a grammar.

Despite the difficulty of finding similarities in the overall structure of versum grammars, we believe they exist. If we could produce a grammar and the Δs for a given *n* automatically, we could generate the corresponding versum sequence very efficiently. One intriguing idea is to use metagrammars to generate versum Δ grammars. For

Supplementary Material

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

http://www.cs.arizona.edu/icon/analyst/iasub/ia49/ia49sub.htm

example, one meta-grammar on each successive generation might produce the n_e:2 ... versum sequence grammars: the 2:, 4:, 6:, ... 2... versum grammars. This cannot be done entirely within the grammatical framework we've been using. If nothing else, some mechanism is needed to produce the increasing number of tokens for the Δs.

More important, we're a long way from knowing enough about versum Δ sequences to even approach the problem. Nonetheless, the idea remains in the back of our mind. Some day ... .

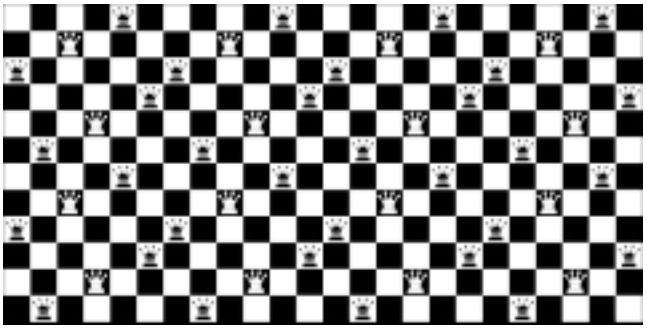
Next Time

In the next article on versum Δs, we'll start with the problem of finding the Δs for a given *n*.

References

1. *A History of Pi*, Petr Beckmann, Barnes & Noble Books, 1993.

2.Versum Numbers, Icon Analyst 35, pp. 5-11.



Graphics Corner — Drawing Images

Icon's drawing primitives allow you to produce a variety of geometric shapes. If you want to produce an irregular image, you can, of course, use DrawPoint(). If the image is large, complicated, or involves many colors, it's painful if not impractical to draw it point by point.

If you have an image file for what you want in a format supported by Icon, you can read it into a window. If you need the image, or perhaps many of them, in an application, this requires separate files at known places.

The function DrawImage() provides an alternative. DrawImage(x, y, s) fills a rectangular area with upper-left corner x,y using the *image string* s.

An image string consists of three comma-separated parts: the width of the rectangle, the name of a *palette*, and pixel data:

"*width*,*palette*,*pixels*"

A palette associates characters with colors. There are many different palettes to accommodate a variety of requirements. The pixel data consists of one character from the palette for each pixel in the image. The height of the image is implied by the number of pixel characters, which must be a multiple of the width. The pixel characters are drawn a row at a time, from left to right, top to bottom. That is, the first *width* pixels describe the top line of the image, and so on.

There are two kinds of palettes: grayscale ones whose names start with a g and color palettes whose names start with a c. There are 255 grayscale palettes, g2 though g256, with the number of entries indicated. The first and last characters in these palettes stand for black and white, respectively (black and white are considered to be shades of grays in this context).

Here is an image string for one of the queens used in the program vqueens in the Icon program library:

```
bqueen := "41,g7,
6666666666666666666666666666666666666666_
6666666666666666666666666666666666666666_
6666666666666666666666666666666666666666_
6666666666666400366666666300466666666666_
6666666666665000046666664000056666666666_
6666666666664000036666663000046666666666_
6666666666660000566666650000666666666666_
6666666666666522466666666422566666666666_
6666334666666664466666666446666666433666_
6662000466666663166666666136666666400026 6_
6660000266666664066666666046666666200006 6_
6660000366666665046666664056666666300006 6_
6664002666666666016666661066666666600466_
6666665166666666004666640066666661566666_
6666666226666666002666620066666662266666 6_
6666666036666660004663000666666306666666_
6666666403666640000220004666630466666666_
6666666620266620000000002666202666666666_
6666666650002100000000000120005666666666_
6666666663000000000000000000003666666666_
6666666666000000000000000000006666666666_
6666666666300000000000000000036666666666_
6666666666500000000000000000056666666666_
6666666666610000000000000000166666666666_
6666666666630000000000000000366666666666_
6666666666652222222222222225666666666666_
6666666666664444444444444446666666666666_
6666666666640000000000000004666666666666_
6666666666651000000000000015666666666666_
6666666666664000000000000046666666666666_
6666666666651000000000000015666666666666_
6666666666640000000000000046666666666666_
6666666666644444444444444466666666666666_
6666666653222222222222222223566666666666_
6666666660000000000000000000066666666666_
6666666640000000000000000000046666666666_
6666666630000000000000000000036666666666_
6666666630000000000000000000036666666666_
6666666630000000000000000000036666666666_
6666666630000000000000000000036666666666_
6666666666666666666666666666666666666666"
```

We've split the literal so that each line of the image string corresponds to a line of the image, and we've squeezed the lines together so that you can better

see the pattern. The g7 palette has 7 characters from 0 (black) through 6 (white). The corresponding image, which is square, is shown in Figure 1.

Figure 1. The Black Queen

There are six color palettes. The c1 palette, which has 90 colors, is based on Icon's color naming system [1]. The remaining c*n* palettes have *n* levels of each of the RGB primary colors with additional grays added. See Reference 2 for a complete description of palettes. You can view a palette with its colors and the corresponding characters using the program palette in the Icon program library. Images of the color palettes also are on the Web page for this issue of the Analyst.

While it's possible to construct image strings by hand, unless the image and the number of colors are small, the process ranges from tedious to impractical. The Icon program library has help. For example, the program giftoims converts GIF images to image strings. It uses Capture(palette) from the module gpxop.icn, which converts a rectangular portion of a window to an image string according to palette.

There are several functions related to palettes, including:

PaletteKey(palette, color), which returns a character from palette that is close to color.

PaletteColor(palette, s), which returns the color in palette that represents s.

PaletteChars(palette), which returns the string of the characters in palette.

Except in the large palettes in which they are used for colors, the characters "~" and "\377" designate "transparent" pixels in an image string. Transparent pixels are not drawn and hence do not overwrite what's already on the canvas.

This allows overlays, such as the one shown in Figure 2.

Figure 2. "No Unicorns Allowed"

The program that produced the final result is:

```
link graphics

procedure main()
   local prohibit, unicorn, width, height
   local white, mask, palette

   prohibit :=
      $include "prohibit.ims"

   unicorn :=
      $include "unicorn.ims"

   prohibit ?:= {
      width := tab(upto(','))
      move(1)
      palette := tab(upto(','))
      move(1)
      white := PaletteKey(palette, "white")
      width || "," || palette || "," || map(tab(0), white, "~")
      }

   unicorn ? {
      width := tab(upto(','))          # width for window
      move(1)
      tab(upto(',') + 1)               # skip palette
      height := *tab(0) / width
      }

   WOpen("size=" || width || "," || height) |
      stop("*** cannot open window")

   DrawImage(0, 0, unicorn)

   DrawImage(0, 0, prohibit)

   WDone()

end
```

We used $includes, since the image strings are too long to show here.

The effect shown in this example could, of course, be achieved in other ways. The interdiction symbol is simple enough that it could be drawn using DrawCircle() and DrawLine(), although you might find it a bit tricky to get it just right. The overlay could have been achieved by reading a transparent GIF.

Nonetheless, there are many things that can be done using image strings that would be awkward or impractical using other methods. We'll show some of these in another article.

References

1. *Graphics Programming in Icon*, Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend, Peer-to-Peer Communications, 1998, pp. 139-141.

2. *Graphics Programming in Icon*, Color Plate 8.1 and pp. 441-444.

Sorting

Sorting is an essential part of many programs, but there never seem to be all the facilities that you'd like to have. In this article we'll start by describing Icon's built-in facilities and then go on to additional facilities in the Icon program library in a subsequent article.

Sorting in Icon is complicated by the fact that Icon has many different data types, and a collection of values to be sorted may contain values of different types.

Sorting Rules

Icon first sorts the values by type and then among values of the same type by rules that depend on the type. The type order is:

the null value
integers
real numbers
strings
csets
files and windows
co-expressions
functions and procedures
lists
sets
tables
records

It makes some sense that the null value comes before any other value in sorting. Note that there is only one null value. Numbers come next because they are, in some sense, simple. There is no particular reason for integers to come before real numbers except that integers are simpler than real numbers.

Strings, being central to Icon's computational repertoire, come next, followed by values of the related type, csets.

The placement and order of windows, files, procedures, and co-expressions is rather arbitrary. In fact, they might better come at the end, but once a decision like this has been made, you're stuck with it. In any event, having values of these types in collections to be sorted is unlikely and you probably can program in Icon without ever needing to know where they fall in the type order.

The structure types are at the end, with records last because records themselves have types and in that sense are somewhat different from the other structure types.

Now there is the question of sorting among values of the same type. As you'd expect, sorting among numeric values is by magnitude. Strings are sorted lexically (sometimes called alphabetically, but remember that all 256 characters may occur in strings).

In lexical sorting, the numerical codes for the characters are compared. These days, almost all computers except IBM mainframes use the extended 8-bit character set based on the 7-bit ASCII standard. In ASCII, digits come before letters, uppercase letters come before lowercase ones, and "special" characters are scattered around. See Reference 1 for details about character sets. Anyway, everyone knows what alphabetical sorting is and it's easy to extend that knowledge to strings that contain characters other than letters.

Csets also are sorted lexically. Although csets themselves are unordered collections of characters, those characters have numerical values. One way to think of cset sorting is that they are sorted lexically as they would be if converted to strings.

Files and windows are sorted together by their file names and window labels, respectively. Co-expressions are sorted by serial number [2].

Functions and procedures are sorted together by their names.

Among values of the same structure type, values are sorted by serial number, which also is the order in which they are created. That is, the "oldest" structures of a given type come first. Finally records are subsorted by type name (lexically) and then by serial number.

Note: The third edition of the Icon language book failed to record the changes in sorting that had taken place since the second edition. The information given on page 162 is erroneous.

Sorting Values in Structures

Icon's primary sorting function is sort(X), where X is a structure: a list, set, table, or record. The result produced by sort() always is a list.

For a list, sort(L) produces a new list of the same length but with the elements in sorted order.

For example, if

```
potpourri := ["two", '2', 1, &null, 1.0, &null, 2, "one"]
```

then sort(potpourri) produces

```
[&null, &null, 1, 2, 1.0 , "one", "two", '2']
```

Records and sets can be sorted just like lists.

For a set, sort(S) also produces a new list with as many elements in the list as there are members in the set.

The sticky case is tables, and tables are the most frequently sorted structures. The problem is that table elements are pairs of values: keys and associated values. The keys are unique, like the members of sets, but the values need not be and often aren't.

The original solution to this was to produce a list of two-element lists, in which each two-element list corresponds to a table element, with the key being first and the value second. This approach has the virtue of being logical, preserving the structure of the table, and producing a list of the same size as the table (as it is for other structure types). For tables, sort() has an optional second argument: sort(T, 1) orders the two-element lists according to the keys, while sort(T, 2) orders them according to the associated values.

Consider this example:

```
One := table()

One["string"] := "one"
One["integer"] := 1
One["real"] := 1.0
One["cset"] := '1'
```

The result of sort(One, 1) is

```
L1 := [["cset", '1'], ["integer", 1], ["real", 1.0],
  ["string", "one"]]
```

while the result of sort(One,2) is

```
L2 := [["integer", 1], ["real", 1.0], ["string", one"],
  ["cset", '1']]
```

There are two problems with "two-level" sorting. First, since the result is a list of lists, it's necessary to use double subscripting to get to the keys and their associated values. For example, to generate the values of sort(T, 1), something like this is needed:

```
(!L1)[2]
```

This can be somewhat confusing, and it is error prone.

The other problem is storage overhead. The result of sorting a table with *n* elements is *n*+1 lists. Every list carries with it some storage overhead [3]. Tables that are to be sorted often are very large, so the memory requirements can be significant. To make matters worse, sorting a table often is the final output phase of a program that may have taken a long time to produce the table. A run-time error because of insufficient storage to produce the sorted list and get the final results has maddened more than one Icon programmer. (You may take "madden" in either of its meanings.)

This pragmatic concern led to sorting options for which the result is a "one-level" list of alternating keys and their associated values — and hence whose size is twice the size of the table. sort(T, 3) orders by key, while sort(T, 4) orders by value. For example, the result of sort(One, 3) is

```
L3 := ["cset", '1', "integer", 1, "real", 1.0,
  "string", "one"]
```

while the result of sort(One, 4) is

```
L4 := ["integer", 1,"real", 1.0, "string", "one",
  "cset", '1']
```

This type of sorting replaces the double-subscripting problem by one of keeping track of positions in the list. For example, to generate the values in L3, something like this is needed:

```
L3[2 to *L3 by 2]
```

Nevertheless, most programmers prefer one-level sorting for tables.

Sorting Structures by Field

The function sortf(X, i) sorts by "field" values. Lists and records are ordered by the values of their ith elements / fields. Other values are sorted as they are by sort().

The value of i, which defaults to 1, can be negative but not zero. Lists and records having equal i fields are ordered as they would be by sort(), but lists and records that do not have an ith element appear before those that do.

sortf() only applies to lists, sets, and records; it cannot be used for tables.

Here's an example of a way sortf() can be used. The data consists of a list of marbles:

```
record marble(type, diameter, condition, price, image)
```

```
                    ...
   spc001 := ("swirl", 1.5, "mint", 85.00, "i001.gif")
   spc002 := ("swirl", 0.5, "good", 10.00, "i002.gif")
                    ...
   spc099 := ("agate", 1.25, "mint", 15.00, "i099.gif")
   spc100 := ("agate", 2.5, "good", 12.50, "i100.gif")
                    ...
   marbles := [spc001, spc002, ..., spc099, spc100, ...]
                    ...
```

Then

```
marbles_by_value := sortf(marbles, 4)
```

and so on.

Stability

A sorting method is said to be stable if equal values always retain their positions when sorted, but unstable otherwise.

For sorting, Icon uses the C library routine *qsort(),* which typically implements quicksort [4]. The quicksort algorithm is unstable, so Icon's sort() and sortf() functions are unstable.

Is this a problem? If a structure to be sorted contains two equivalent values, how can you tell if their order is retained in sorting? For example, suppose L1, L2, and L3 were created one after the other,

```
lists := [L1, L2, L3, L4, L4, L3, L2, L1]
```

then sort(lists) produces

```
sorts := [L1, L1, L2, L2, L3, L3, L4, L4]
```

Since sort() is unstable, the first L1 in lists may be the second element of sorts. But, since Icon uses pointer semantics, the first and last elements of lists are not only equal for the purposes of sorting — they are *identical*. There is no way to tell one from the other. The same is true of multiple instances of other types of values. For example, as a result of

```
s1 := "one"
s2 := "o" || "n" || "e"
```

both s1 and s2 have the value "one". The fact that their values were created separately and in different ways does not matter. The way Icon is implemented, there actually are two copies of "one" [5], but semantically they are the same, and there's no way to tell them apart. One way to think about this is to consider the analogous case for integers:

```
i1 := 1
i2 := 2 – 1
```

Both i1 and i2 have the value 1, but there aren't two different 1s.

But is this true for all Icon values? Are there values that are different but equal for the purposes of sorting? Review the rules for sorting and see if you can find examples. See the last section of this article for a discussion of this.

The fact that Icon's sorting is unstable *is* limiting for sortf(). If sorting were stable, multiple calls of sortf() could be used to sort according to multiple fields.

Limitations of Sorting

Icon's built-in sorting facilities provide no options; there is no way in Icon's built-in repertoire to, for example, discard duplicate values, reverse the order, or other such useful things. The result always is a list with values in order from the smallest to the largest according to fixed rules. And there is no way to specify the comparison method used for sorting. Such things have to be programmed in Icon.

As you'd expect, the Icon program library has procedures for sorting in more sophisticated ways. We'll describe them in a subsequent article.

The Consequences of Unstable Sorting

The situations in which distinct values may be equal for the purposes of sorting occur when values are sorted according to names. For example, if two windows have the same label or if a window has a label that is the same as the name of a file, they are equal for the purposes of sorting. Other examples are more obscure. For example,

```
op1 := proc("–", 1)
op2 := proc("–", 2)
```

assigns the unary negation operator and the binary difference operator to op1 and op2, respectively. Operators are really functions with special syntax, so op1 and op2 sort as functions. Both have the name "–", so sorting may change their relative positions. (How can you tell their names are "–"?) Since functions and procedures sort together, it's possible to have a similar situation. Suppose a program has a procedure abs():

```
procedure abs(i)
        ...
end
```

Then as a result of

```
prc := abs
fnc := proc("abs", 0)
```

the value of prc is the procedure value and the value of fnc is the built-in function [6]. They have the same name, but they are different. The same is true of a record constructor, as in

```
record seek()
```


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

which produces a record constructor with the same name as the built-in function seek().

These examples are, of course, esoteric. The most serious limitation imposed by unstable sorting is that you can't do a multi-key sort using multiple calls of sortf().

References

1. *The Icon Programming Language*, 3rd edition, Ralph E. Griswold and Madge T. Griswold, Peer-to-Peer Communications, 1996, Appendix B, pp. 261-268.

2. *The Icon Programming Language*, pp. 191.

3."Memory Utilization", Icon Analyst 4, pp. 7-10.

4. *The Art of Computer Programming*, Vol. 3, *Searching and Sorting*, Donald E. Knuth, Addison-Wesley, 1973, p. 114-123.

5."String Allocation", Icon Analyst 9, pp. 4-7.

6."Procedure and Operator Values", Icon Analyst 29, pp. 1-3.

We have additional articles on character patterns and versum deltas planned for the next issue of the Analyst. We also plan to follow up the article on sorting in this issue with a **From the Library** article about sorting facilities in the Icon program library.

In new areas, we are working on articles on animated graphics and making "movies".

And there's the ever-present threat of another **Tricky Business** article.