The Icon Analyst

In-Depth Coverage of the Icon Programming Language

October 1998
Number 50

In this issue …

About This Issue

We have a lot of material backed up, and some of it, as you will see in this issue of the Analyst, covers a lot of real estate.

Rather than have things stack up further or sacrifice content, we've gone to a 20-page issue. (Because of the way the Analyst is bound, the page count has to be a multiple of four.)

We do not expect to produce this many pages on a regular basis. In fact, all we promise is 12 pages, although recently, most have been 16.

Analyzing Character Patterns

The distinguishing characteristic of a string is that it is one of the most compact ways of storing vast amounts of data in a way in which information can be replicated. —Michio Kaku [1]

In two previous articles on character patterns [2-3], we explained how character patterns can be useful in representing sequences of values, and we used pattern forms and grammars to identify the structure of sequences.

This article describes an application that helps in analyzing character patterns — finding pattern forms and building pattern grammars.

Overview

The application is named charpatt. Any similarity between the pronunciations of *charpatt* and *carpet* can be attributed to space aliens.

The application supports importing character strings, identifying pattern forms, and building

Back Issues

Back issues of The Icon Analyst are available for $5 each. This price includes shipping in the United States, Canada, and Mexico. Add $2 per order for airmail postage to other countries.

grammars. The interface is shown in Figure 1 with a partially developed grammar.

At the top are menus and below them information panels.

The scrolling text list at the left is a workspace that shows one of the definitions in the current grammar. The workspace allows the detailed examination of a definition and can be used to identify a specific substring. It also is the place where most searches for pattern forms are made.

The scrolling text list at the right shows the grammar. Clicking on a definition places the definition in the workspace.

The information panels give the name of the grammar file, its size in bytes, the variable associated with the current definition, the number of symbols remaining that can serve as variables, and finally the depth of the grammar, which is the number of L-System generations necessary to produce the character pattern.

The menus provide the following services:

- File: opening and saving grammars, importing character strings, and other file-related activities.
- Search: looking for various pattern forms and related operations.
- Workspace: operations related to the current definition.
- Grammar: operations related to the grammar as a whole.
- Tokens: operations related to the values associated with the labeling characters in character patterns.
- Options: configuration settings.

Loading and Saving Data

The File menu provides facilities related to bringing in data and saving it. Rather than occupy space with an image of each menu, we'll explain instead in terms of the keyboard shortcuts for menu items.

@O: Open a previously saved grammar.

@S: Save the current grammar.

@C: Load a character pattern to create a new grammar.

@V: Create a character pattern and a new grammar from a sequence of values.

@E: Create a grammar by entering a character pattern manually.

@A: Revert to last saved grammar.

@U: Undo last change to the current grammar.

@T: Redo the last undo.

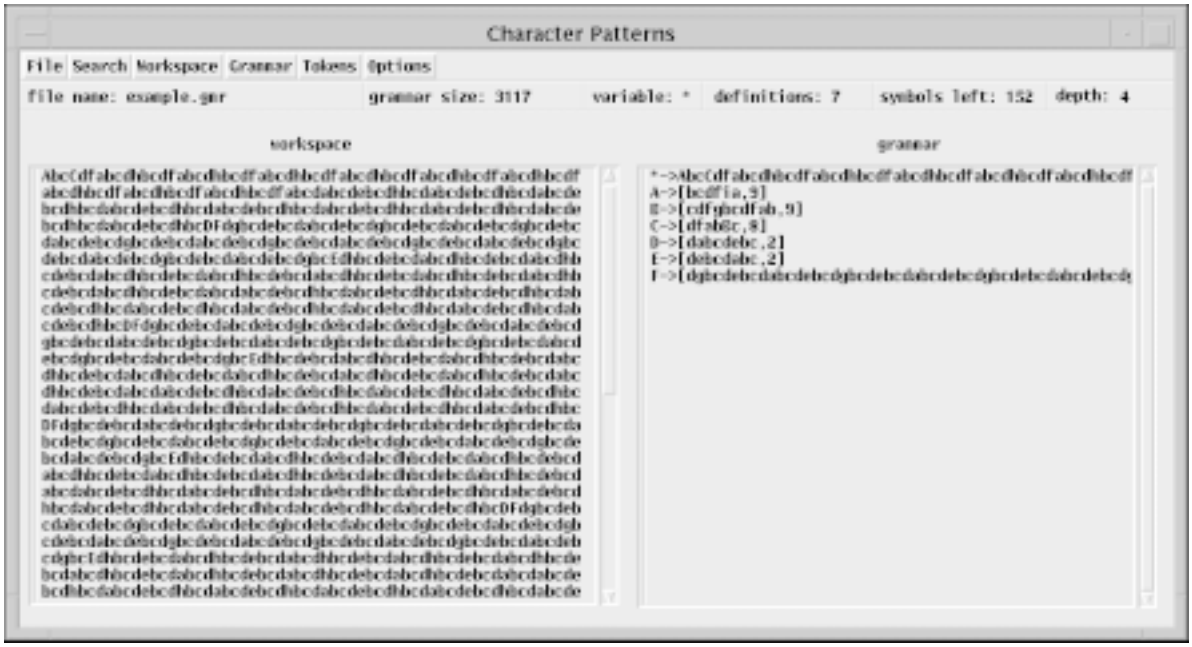


Figure 1. The charpatt Interface

For @O, @C, and @V, the navigation interface described in Reference 4 is used.

Grammars are saved as L-Systems so that they can be expanded by existing programs.

For @C, the character pattern is taken from a specified range of lines in a file. For @V, values can be imported in two ways: as one value per line of a file or lines with blank-separated values. (At present, values can only be strings or values that can be represented by strings, which includes numbers.)

For @E, a text-entry dialog is presented for typing a character pattern. Only short grammars can be created in this way; it's mainly useful for testing.

Searching for Pattern Forms

The Search menu provides several ways of searching for pattern forms:

@F: Find a specific substring.

@K: Find a substring by range.

@P: Find palindromes.

@R: Find repetitions by length.

@H: Find repetitions of a specific string.

@N: Find *n*-grams.

@L: Find locations at which a string occurs.

Except for @N, which processes all definitions, all searches take place in the workspace.

Except for @L, which provides information that may be useful in other searches, all successful searches produce pattern forms for the strings that are found. These then can be used to create new definitions in the grammar and replace instances of the strings by the variables for the new definitions.

The simplest kind of search is for a specific string. Figure 2 shows a typical dialog for @F.

Figure 2. Dialog for @F

The string to be searched for is entered in the first text-entry field. The range specification, in the style of Icon character-position numbering, applies to the workspace and allows the search to be restricted to a portion of it. In Figure 2, the entire workspace is specified.

The decollation field provides for the result to be decollated into a number of fields [3]. The default is 1, specifying no decollation (the usual setting).

The final text-entry field limits results to those that produce at least the specified reduction in the size of the grammar — 10 characters in this case. (The savings can be negative to allow pattern forms that increase the size of the grammar.)

Figure 3 shows the result for the grammar shown in Figure 1.

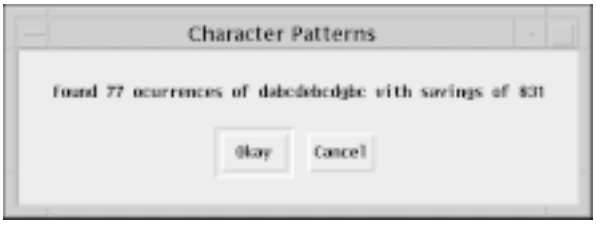


Figure 3. Results of a Search

If the result is accepted, a new definition is created and all occurrences of the string are replaced by the new variable.

Searching for repetitions by length provides an example of a more complicated kind of search. Figure 4 shows the dialog box for @R.

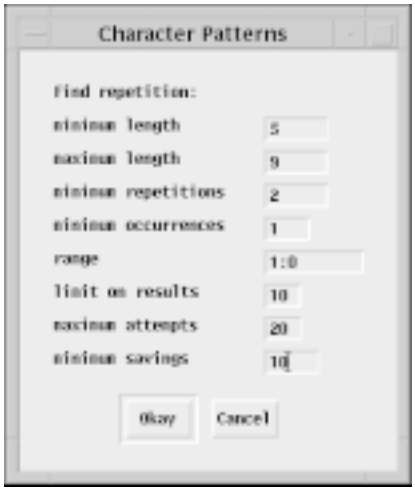


Figure 4. Dialog for @R

The first two text-entry fields specify limits for the number of characters in a repetition — that is, *s in [s,i]. In Figure 4, s can range from 5 to 9 characters.

The next two text-entry fields specify the minimum number of repetitions (i) and the minimum number of times the repetition occurs in the grammar. Although the search is based on the workspace, once a repetition is found, all other definitions are searched for instances of it.

The dialog as shown in Figure 4 limits the number of results to 10, while the limit on the attempts to find repetitions is limited to 20 (some attempts may fail). The limit on attempts can be used to curtail otherwise time-consuming searches.

When the search is complete, the results are shown in descending order of the savings they would produce. The dialog resulting from the specifications in Figure 4 is shown in Figure 5.

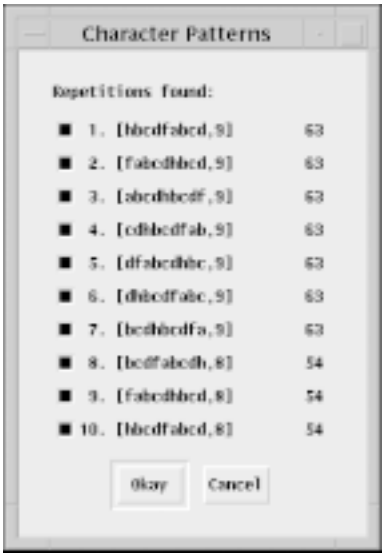


Figure 5. Dialog from a Successful Search

The results are presented as pattern forms in a toggle dialog with all pre-selected (this can be changed in the Options menu). The savings that would result from selecting them are shown at their right. Unwanted ones can be deselected. If the dialog is dismissed by Okay (or a return), the selected pattern forms are made into new definitions and all occurrences of the corresponding strings are replaced by the variables for the new definitions. Since the strings found may overlap, not all of them may result in new definitions (they are processed from the top down). In the situation here, all of the results overlap and there would be only one new definition.

Preloading Search Dialogs

Many searches are undertaken based on visual inspection of the workspace. In Figure 1, for example, it is evident that there are many occurrences of the substring dabc, among others. Often, however, it is difficult or impossible to pick good candidates for searches out of the maze of characters in the workspace. There are two features that provide help in such situations.

One feature brings up a segment of the workspace in a text-entry dialog where it can be examined and edited. This is accomplished by @-clicking on a line of the workspace (the meta modifier is required to avoid accidentally activating this feature).

Figure 6 shows the result of @-clicking on a line of the workspace shown in Figure 1.

Figure 6. A Segment of the Workspace

In this more limited context, and glancing at the workspace as a whole, it is evident that the string dabcdebcdghc occurs frequently. The text-entry field can be edited to isolate this string, as shown in Figure 7.

Figure 7. An Edited Segment

Accepting this string does not change the workspace — that is not allowed in the present application. It does, however, preload fields for search dialogs with values based on this string. For searches that take a specific string (@F, @H, and @L), their string fields are preloaded with dabcdebcdghc. This saves re-entering the string, which can be difficult if it is long and complicated. For searches that specify lengths (@R and @N), the length fields are preloaded with the length of the string (12). For example, a subsequent @R would have 12 preloaded in its minimum and maximum length fields.

Another way to preload search dialogs is to use @L, which shows the locations of successive instances of a specific string and the distances between them (deltas). Figure 8 shows a location dialog for the string bc. A typical result is shown in Figure 9.

Figure 8. A Location Dialog

Figure 9. A Location Result

Clicking on Next produces the next location of bc, if any. Clicking on Done preloads search dialogs that have length fields with the current delta. They are preloaded with the last delta when the search terminates because there are no more instances of the string.

Grammar Information

In addition to the information panels on the face of the application window, there are several methods for getting information about the current grammar. @I from the Workspace menu provides information about the current definition. See Figure 10.

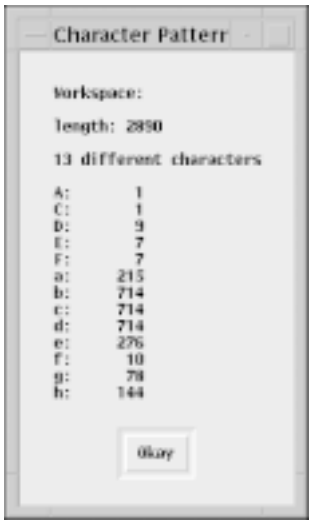


Figure 10. Workspace Information

@Y from the Grammar menu provides information about the symbols. See Figure 11.

Figure 11. Grammar Information

The available symbols are used for new variables in the order given. The symbols from the high part of extended 8-bit ASCII, which are used last, are as they appear for the default X-Window font.

Finally, the values associated with tokens, if any, are displayed by @1 from the Tokens menu. See Figure 12.

Figure 12. Tokens and Their Values

The values can be changed by editing the text-entry fields.

The Final Result

The end result of working on the partially developed grammar shown in Figure 1 is shown in Figure 13 on the next page. The results of searches used in earlier examples were not used to get this grammar. Note that the size of the grammar has

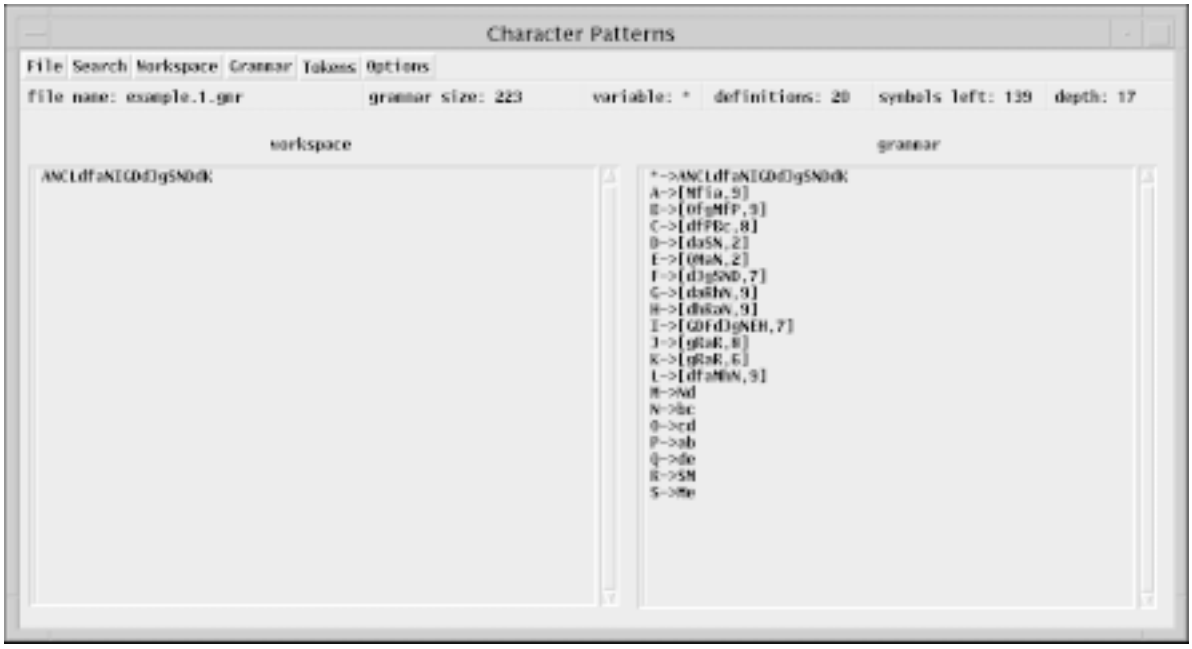


Figure 13. A Completed Grammar

gone from 3117 characters to 223 characters while adding 13 definitions and increasing the depth by 13.

Many other small grammars exist for this example. This is the smallest one we were able to find.

Comments

What we've described here only covers the basic operations in charpatt. It is a large and capable application with many sophisticated features (that is, way too big and way too complicated).

Over time we'll develop a user's manual and put it on the Web. Even without detailed instructions, you can accomplish a lot by experimenting with it. Finding structure in character patterns is challenging even with charpatt to help. It can be a fascinating, even addicting, puzzle game.

As a program, it is large by Icon standards — over 2,500 lines. It would take more than 30 pages to even list it in the Analyst using a barely readable type size, let alone describe it in detail. The implementation does, however, have some interesting features, which we intend to describe in a future article and in **Programming Tips**.

Too big and too complicated notwithstanding, we plan to add some additional features These include:

- Allowing the grammar to be edited manually.
- Allowing definitions to be added and deleted manually.
- Allowing tokens to be added and deleted manually.
- Allowing search specifications to be given in terms of pattern forms.
- Allowing string transformations, such as reversal, to be applied to definitions.
- Providing filters to modify values as they are imported.
- Creating grammars from subsets of existing grammars.
- Merging grammars.

Among concepts we are considering in this context are meta-grammars and character patterns of unlimited length.

What's Next?

Aside from loose ends, the next big topic related to character patterns is synthesis: constructing pattern forms and building grammars directly, not just as a by-product of analysis. Although we've yet to explore this topic in detail, we know that much of the framework in charpatt will be useful in an application to aid character-pattern synthesis. And it may prove that analysis and

synthesis facilities are needed in combination. Whether we can do this without creating a monstrosity remains to be seen.

In the meantime we'll lay the groundwork with an article on a weaving language that uses many pattern forms we've not considered in analysis.

References

1. *Hyperspace: A Scientific Odyssey Through Parallel Universes, Time Warps, and the Tenth Dimension*, Michio Kaku, Anchor Books, 1994.

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

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

4."File System Navigation Using VIB", Icon Analyst 48, pp. 10-14.

Versum Deltas (continued)

> *Certainly, let us learn proving, but also* ***let us learn guessing****.* — George Pólya [1]

In the last issue of the Analyst [2], we started to explore versum deltas (Δs) — the differences between successive versum numbers. In addition to the intrinsic recreational interest, we hoped to find a way of generating versum numbers efficiently by producing versum deltas algorithmically.

We tried using versum grammars to characterize versum sequences. One problem with that approach was that we had no way to produce the Δs for a given class of versum numbers. Since then, we've been working on that problem — if we can't produce the sequence of Δs, maybe we can at least produce the Δs.

Recall that we used the notation *n*:*i*… to denote *n*-digit versum numbers whose first digit is *i*. We started by dividing the problem into two parts, *i* = 1 and *i* = 2 (the Δs for *i* = 3 through 9 being the same as for *i* = 2).

We noted that the Δs for *n* even (n_e) and *n* odd (n_o) are different both in number and nature. In this article, we'll look more closely at *n*:2… Δs. The "problem tree" is shown in Figure 1 on the next page, with the current topics underlined.

George Pólya

1893-1989

George Pólya, a Hungarian-born mathematician, made many contributions in diverse areas: probability theory, geometric symmetry, combinatorics, and differential equations <1>.

When the political situation in Europe degenerated, he moved to the United States and finished his career at Stanford University.

Despite his notable contributions to mathematics, he is most remembered for his teaching and books on problem solving [1-2], which are still in print <2> and modestly priced.

References

1. *How to Solve it: A New Aspect of Mathematical Method,* George Pólya, Princeton University Press, *1945.*

2. *Mathematics and Plausible Reasoning,* George Pólya, Princeton University Press, 1954 (two volumes, *Induction and Analogy in Mathematics* and *Patterns of Plausible Inference*).

Links

1. http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Polya.html

2. http://www.amazon.com/

Editor's Note:

I majored in physics as an undergraduate at Stanford but had a considerable interest in mathematics. I recall taking a stupefyingly dull course in "pure mathematics'" and wondering why I ever thought mathematics was interesting. Near the end of the semester, George Pólya, who had recently retired, gave a guest lecture. The students, who generally competed for the most-bored expression, became excited and enthralled. I don't recall the specific topic, but his lecture made such an impression on me that I still have a mental image of Pólya drawing and gesticulating at the blackboard. I was momentarily tempted to change my major to mathematics, but lack of ability and "math fear" fortunately prevailed.

— reg

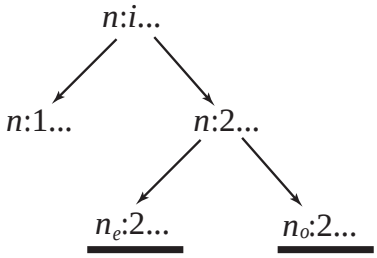


Figure 1. The Versum Δ Problem Tree

We'll tell you in advance that we've had more success with n_e:2… Δs than with n_o:2… Δs.

The Nature of n_e:2… Versum Δs

There are clear patterns in n_e:2… Δs:

	Δ	count
n = 4	11	9
	99	9
n = 6	11	99
	99	171
	891	72
	990	18
n = 8	11	999
	99	3249
	891	1368
	990	342
	8811	720
	9801	162
	9900	18
n = 10	11	9999
	99	61731
	891	25992
	990	6498
	8811	13680
	9801	3078
	9900	342
	88011	7200
	97911	1620
	98901	162
	99000	18
n = 12	11	99999
	99	1172889
	891	493848
	990	123462
	8811	259920
	9801	58482
	9900	6498
	88011	136800
	97911	30780
	98901	3078
	99000	342
	880011	72000
	979011	16200
	988911	1620
	989901	162
	990000	18

n_e:2… Δs

All of these numbers are divisible by 11, as they must be, since all n_e:2… versum numbers are divisible by 11 [3]. Note that all Δs for *n* are included in the Δs for *n* + 2 — and that the additional Δs are all larger. This strongly suggests that there is an underlying rule. The number of different Δs is given by

$$\eta(n) = 2 \qquad n = 4$$
$$\eta(n) = \eta(n-2) + (n/2) - 1 \qquad n > 4$$

Pattern, patterns. What a fertile guessing ground.

We did not figure out was going on until we had the wit to put the Δs in increasing order (as shown in the list above) and take *their* Δs. Here's what they look like for *n* = 16:

88
792
99
7821
990
99
78111
9900
990
99
781011
99000
9900
990
99
7810011
990000
99000
9900
990
99
78100011
9900000
990000
99000
9900
990
99

From there, all it takes is to work out the form of the numbers and recognize that 88 is a degenerate case, as often happens at the beginning of a sequence. $780_i11 = 11 \times (71 \times 10^{i+1} + 1)$, which gives the otherwise apparently anomalous 792 for *i* = 0. The 990_j are obvious. Here it is cast as a procedure:

```
procedure deltas(n)
   local i

   suspend 88

   every i := 0 to ((n / 2) – 3) do {
      suspend 11 * ((71 * (10 ^ i)) + 1)
      suspend 99 * (10 ^ (i to 0 by –1))
      }

end
```

Now we can recast the "brute-force" method [2] by limiting Δs to ones generated in this fashion. We also can divide them into two classes as before.

Here's the program, which takes an even integer greater than 4 on the command line:

```
link vpredrep

procedure main(args)
   local n, first, last, vers, new_vers, dels, delta
   local deltas_1, deltas_2

   n := integer(args[1]) | 6          # default for testing

   ((n > 4) & (n % 2 = 0)) |
      stop("*** invalid digit specification")

   deltas_1 := [11]                   # starting values
   deltas_2 := [99]                   # for each case

   delta := 99

   #  Build the delta lists.

   every delta +:= deltas(n) do {
      put(deltas_1, delta)
      if delta % 2 = 0 then put(deltas_2, delta)
      }

   first := integer("2" || repl("0", n – 2) || "2")
   last := integer("2" || repl("9", n – 2) || "2")

   new_vers := vers := first

   repeat {
      dels := case vers[–1] of {
         "1"  :  deltas_1
         "2"  :  deltas_2
         }
      write(vers)
      if vers >= last then break
      every new_vers := vers + !dels do {
         if vpred(new_vers) then {
            vers := new_vers
            break
            }
         }
      }

end
```

This approach dramatically reduces the number of Δs that have to be tried. Here are comparisons of failed calls of vpred() for the "gentler" brute-force method in the last article [2] and the improved version here:

n	*old*	*new*	*ratio*
6	1,481	99	14.96
8	174,960	4,752	36.82
10	18,051,498	146,232	123.44

The program, however, still requires vpred(). The fundamental problem remains.

The Nature of n_o:2… Versum Δs

Since this approach worked so well for n_e shouldn't it work for n_o?

Dream on. Here are the Δs for n_o.

n = 3	1	5
	19	4
n = 5	10	90
	89	50
	91	32
	101	17
n = 7	1	810
	99	1710
	100	90
	121	9
	791	400
	811	320
	890	100
	911	170
n = 9	10	14580
	89	8100
	99	17910
	121	9
	891	13680
	901	2430
	990	1800
	1000	90
	1111	90
	7811	4000
	8011	3200
	8801	900
	8900	100
	9011	1700
n = 11	1	131220
	99	617310
	100	14580
	121	9
	791	64800

890	16200
891	143280
990	35820
1111	90
8811	136800
8911	24300
9801	16200
9900	1800
9901	810
10000	90
11011	900
78011	40000
80011	32000
87911	9000
88901	900
89000	100
90011	17000

These Δs are, of course, not all divisible by 11 or any single number (except 1). Nor is there the inclusion property as there is for n_e. And the approach to taking Δs of the Δs does not lead anywhere.

We tried various other things, such as subdividing the problem into n_o:2…1 and n_o:2…2 versums. That seemed to make things simpler, but we didn't get the brass ring.

We only mention the n_o:2… problem in hopes it will suggest something to one of our readers.

About Guessing

We do a lot of guessing in our work on versum numbers. Mathematicians guess too — it's just that usually you only see the end result in a theorem. Guesses usually start with perceiving patterns in empirical data. As Pólya remarks about Euler's work, "by observation, daring guess, and shrewd verification" [1].

We have an amazing collection of guessed formulas relating to versum Δs. The guess we're proudest of is the position in a n_e:2… Δ sequence at which the first Δ occurs that does not occur in sequences for smaller n_e. There is no *a priori* reason to believe that such a formula exists, but it does:

$$(10^{((n/2)-2)} - 99) \times 10^{((n/2)-1)} + 89$$

which in our digit notation is

$$9_{((n/2)-4)}010_{((n/2)-3)}89$$

and as a pattern form

[9,*n*/2–4]01[0,*n*/2–3]89

— a totally useless guessoid (which we made no attempt to prove but did check through $n = 14$).

Next Time

In the next Analyst, we have an article that shows how to generate the *n*:2… versum numbers efficiently and without using vpred().

References

1. *Induction and Analogy in Mathematics*, George Pólya, Princeton University Press, 1954.

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

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

Graphics Corner — Fun with Image Strings

In the last **Graphics Corner**, we introduced image strings and palettes [1]. In this article, we'll show some of the things that can be done with image strings. The important point is that image data is represented by strings of characters and Icon's computational repertoire can be used to manipulate them. This often is faster and easier than dealing with images themselves. Furthermore, image strings support transparency, which can be obtained otherwise only by using prepared GIF89a image files.

Since most manipulations are performed on the pixel data and not on the heading information, it is useful to set up a structured approach to handling image strings by representing them with image records:

```
record ImageRecord(width, palette, pixels)
```

Here's a procedure to convert an image string to an image record:

```
procedure imstoimr(ims)
   local imr

   imr := ImageRecord()
```

```
   ims ? {
      imr.width := tab(upto(',')) | fail
      move(1)
      imr.palette := tab(upto(',')) | fail
      move(1)
      imr.pixels := tab(0)
      }
   return imr
end
```

A procedure to convert the other way is simple:

```
procedure imrtoims(imr)
   return imr.width || "," || imr.palette || "," || imr.pixels
end
```

Drawing from an image record is straightforward:

```
procedure drawimr(win, x, y, imr)
   return DrawImage(win, x, y, imrtoims(imr))
end
```

However, to handle omitted and defaulted arguments in the way that Icon's built-in drawing repertoire does, more needs to be done. The first, window, argument can be — and often is — omitted, indicating the subject window, &window, is to be used. One way to handle this is to check if the first argument is of type "window". If not, shift the given arguments to their proper positions using a trick described in an earlier Analyst article [2]. Then win can be set to &window.

```
   if type(win) ~== "window" then {
      win :=: x :=: y :=: imr
      win := \&window | runerr(140, &window)
      }
```

Finally, if x and y are null (not omitted), they default to the upper-left corner of the window:

```
   /x := –WAttrib("dx")
   /y := –WAttrib("dy")
```

So the otherwise simple procedure becomes this:

```
procedure drawimr(win, x, y, imr)
   if type(win) ~== "window" then {
      win :=: x :=: y :=: imr
      win := \&window | runerr(140, &window)
      }
   /x := –WAttrib("dx")
   /y := –WAttrib("dy")
   return DrawImage(win, x, y, imrtoims(imr))
end
```

The devil and the bulk of the code are in the details.

Another useful procedure opens a new window based on an image record:

```
procedure openimr(imr)
   win := WOpen("size=" || imr.width || "," ||
      (*imr.pixels / imr.width)
   drawimr(win, 0, 0, imr)
   return win
end
```

Given this mechanism, what can we do with it?

One way to approach this is to review Icon's computational repertoire and see what might be relevant to image manipulation. (You'll sometimes find features in a computer application that are there only because they are easy to do, not because there is any particular need for them. In rare cases, a really original and useful feature originates in this way — perhaps found to be worthwhile only after it is implemented.)

To apply Icon's string processing functions to pixel strings, it's important to remember that all the image pixels are represented in one long string, not in individual row segments as it's useful to view them.

In what follows, we'll assume that image strings have no punctuation characters (spaces and commas for palettes that do not use these characters to label colors). Removing such punctuation characters might best be done in imstoimr().

There are two main two kinds of operations on image strings: rearranging the characters (pixels) and changing the colors associated with pixels. These are typified by reverse() and map(). We'll describe some of the things that can be done by rearrangement of characters in this article and continue with changing colors in the next **Graphics Corner**.

At first glance, reverse() may sound like an improbable operation on the pixels of an image string. In fact, it rotates the image 180°. Here is a procedure to do this:

```
procedure rot180imr(imr)
   imr.pixels := reverse(imr.pixels)
```

```
   return imr
end
```

Rotation by 90º is not so simple. Here's one way of rotating an image 90º clockwise:

```
procedure rot90cwimr(imr)
   local height, columns, i, row

   height := *imr.pixels / imr.width
   columns := list(height, "")

   imr.pixels ? {
      while row := move(imr.width) do
         every i := 1 to imr.width do
            columns[i] := row[i] || columns[i]
      }

   imr.pixels := ""

   every imr.pixels ||:= !columns

   imr.width := height

   return imr
end
```

Rotation by an arbitrary amount is not feasible within this framework.

Note that the last two procedures change their arguments and do not return new image records. To return a new record without changing its argument, all that's needed is

```
imr := copy(imr)
```

at the beginning of the procedure.

The issue of modification versus creating a new structure comes up often. There is no cut-and-dried answer. Independent of the issue of storage utilization, there are questions about which is best from a users viewpoint. Copying generally is "safer" than modification, but it requires more coding, as in

```
imr := rot180imr(imr)
```

instead of just

```
rot180imr(imr)
```

Of course if you expect imr to be changed, not copied, the latter does nothing (well, nothing useful). On the other hand, even if imr is changed, the former doesn't do any harm, since all the procedures that modify image records also return them. This is necessary to support nested calls such as

```
imr := rot180imr(imstoimr(ims))
```

Here are procedures to flip an image vertically and horizontally (about the horizontal and vertical axes), respectively.

```
procedure fliphimr(imr)
   local pixels

   pixels := ""

   imr.pixels ? {
      while pixels ||:= reverse(move(imr.width))
      }

   imr.pixels := pixels

   return imr
end

procedure flipvimr(imr)
   local pixels

   pixels := ""

   imr.pixels ? {
      while pixels := move(imr.width) || pixels
      }

   imr.pixels := pixels

   return imr
end
```

The Icon program library provides more ready-to-use facilities, most of them in the core module strings. For example, rotate() shifts an image horizontally, wrapping around from one end to the other:

```
procedure shifthimr(imr, i)

   imr.pixels := rotate(imr.pixels, i)

   return imr
end
```

The value of i determines how many pixels the image is shifted. Positive i shifts to the left, negative i to the right. The procedure rotate() defaults i to 1. A small value of i is needed to give the impression of smooth motion.

Here's a loop that scrolls an image leftward.

```
until WQuit() do {
   drawimr(0, 0, imr)
   shifthimr(imr, 1)
   }
```

The effect that you get depends on the image. Images whose left and right edges meet seamlessly generally work best. A simple form of seamlessness

is a solid border, as in the image of a unicorn shown in Figure 1.

Figure 1. A Unicorn

The image is 200×200 pixels. We left off the border we usually add to outline images, since we don't want a border moving across the screen. A succession of snapshots taken at 20-pixel increments is shown in reduced form in Figure 2, with borders added so you can see the separate frames.

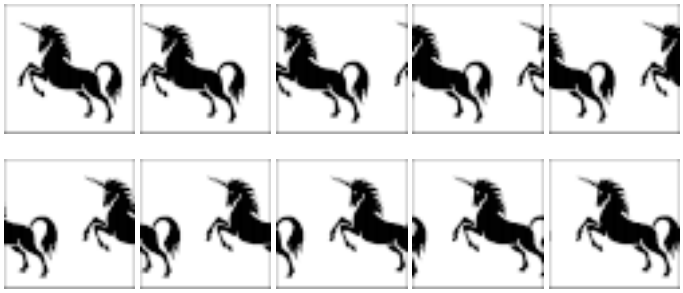


Figure 2. Frames From the Scrolling Display

The visual effect is of an endless line of unicorns entering at the right and leaving at the left. You can imagine it as panning over a larger image as shown in Figure 3.

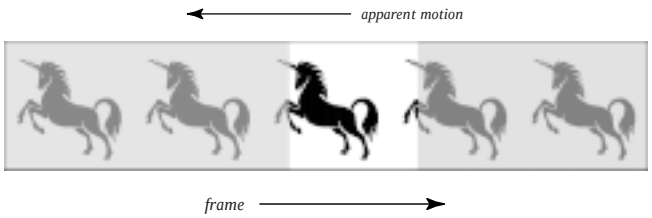


Figure 3. Unicorn Parade

There is a problem with using rotate() for shifting that is not obvious in the example above. Shifted-out pixels re-enter one row higher (for left shifts). One full cycle across the width raises the image one pixel. Eventually the unicorn's head goes off the top to reappear at the bottom. Unless the animation goes on for a long time, this artifact is barely noticeable. The problem can be circumvented by going back the original image record after a full cycle.

As is usually the case, vertical operations are more complicated than the corresponding horizontal ones. The comments above suggest a method for vertical shifting:

```
procedure shiftvimr(imr, i)
   local rows

   /i := 1

   imr.pixels := rotate(imr.pixels, i * imr.width)

   return imr

end
```

There are lots of other things you can do by rearranging the characters in image strings. And you can get silly. Here's a procedure that scrambles a string:

```
procedure randizimr(imr)

   imr.pixels := scramble(imr.pixels)

   return imr

end
```

The procedure scramble() is from the strings module in the Icon program library.

Although we can't think of a good reason for scrambling an image, we can imagine ways of encoding images by encoding their image strings.

We'll leave you with that thought.

References

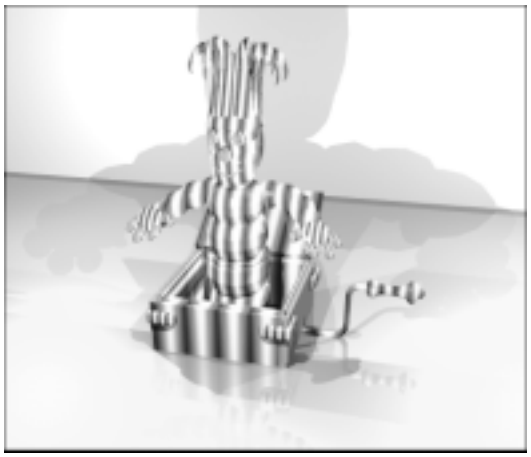
1."Graphics Corner — Drawing Images", Icon Analyst 49, pp. 11-13

2."Idiomatic Programming", Icon Analyst 14, pp. 4-8.

Supplementary Material

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

http://www.cs.arizona.edu/icon/analyst/iasub/ia50/ia50sub.htm



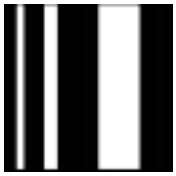
Tricky Business — Image Grammars

One of the problems with image strings is that they are bulky. The width and palette information is insignificant, but the pixel data has one character for every pixel in the image. For a 128×128 image, which might cover about two square inches on a high-resolution monitor, this amounts to 16,384 characters.

When such image strings are incorporated in a program, they not only make the program large, but they also make it hard to read and edit. Include files can be used to take image strings out of line, but this produces auxiliary files that have to be maintained and packaged with the program. So does reading image string files during program execution. The logical thing to do is compress image strings in a form that can be included in a program. This is, after all, what image file formats such GIF do to image data.

Many image strings, especially those for patterns and those that have large areas of solid colors, have a large amount of redundancy. That is, they contain only a small amount of information. A complex "photographic" image and a simple design of the same size have image strings of the same size, but the former contains much more information than the latter. See Figure 1.

sheriff



stripes

Figure 1. Different Image Types

The image of the sheriff has 256 different colors, while the image of stripes has only two. Independent of the number of colors, it's also obvious that unlike the image of the sheriff, very little information is needed to describe the stripes. In other words, much of the stripes image is redundant.

There are many ways to compress image strings so that they can be included in a program. We have a kinky idea on how this might be done: Use pattern grammars. A grammar then can included in the program as a string or list of strings and then expanded into an image string when needed.

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

So far, in the Analyst we've only used pattern grammars to show the structure of strings. The results often are very compact, but our focus has been on structure. With "image grammars", the focus is on compactness.

Suitable Images for Image Grammars

In order for an image to be a candidate for compression, it can have only a modest number of different colors. Grammars need characters for variables and pattern forms as well as for the tokens that stand for colors. Thus, the 256-color image of the sheriff is ruled out. How many colors an image can have and still qualify for representation as an image grammar depends on the complexity of the image. For example, an image of stripes with 100 different colors may be tractable, but for an image with randomly scattered colors, even 10 different colors may not have a useful image grammar.

We'll have more to say about what constitutes useful image grammars later — it's not just their size.

Examples

Figure 2 shows images that we used for testing our ideas about image grammars. They range in size from 128×128 to 360×360, and have from 1 to 47 different colors.

The test01 image is just a solid color, while test02 is the image of stripes from Figure 1.

The next three images are stripes with several colors and different orientations. The images test06 through test15 are numerical carpets [1].

Next the unicorn from the **Graphics Corner** article in the last Analyst [2] appears in grayscale and in black-and-white.

We threw in the next two images to have something different. The last image consists of randomly distributed colors. Here are the details.

image	*size*	*colors*	*pixels*
test01	128×128	1	16384
test02	128×128	2	16384
test03	445×445	9	198025
test04	445×445	9	198025
test05	445×445	9	198025
test06	128×128	2	16384
test07	128×128	8	16384
test08	128×128	8	16384
test09	128×128	4	16384
test10	128×128	5	16384
test11	128×128	8	16384
test12	128×128	3	16384
test13	128×128	5	16384
test14	128×128	9	16384
test15	128×128	4	16384
test16	211×204	13	43044
test17	211×204	2	43044
test18	206×206	13	42436
test19	130×130	47	16900
test20	128×128	14	16384

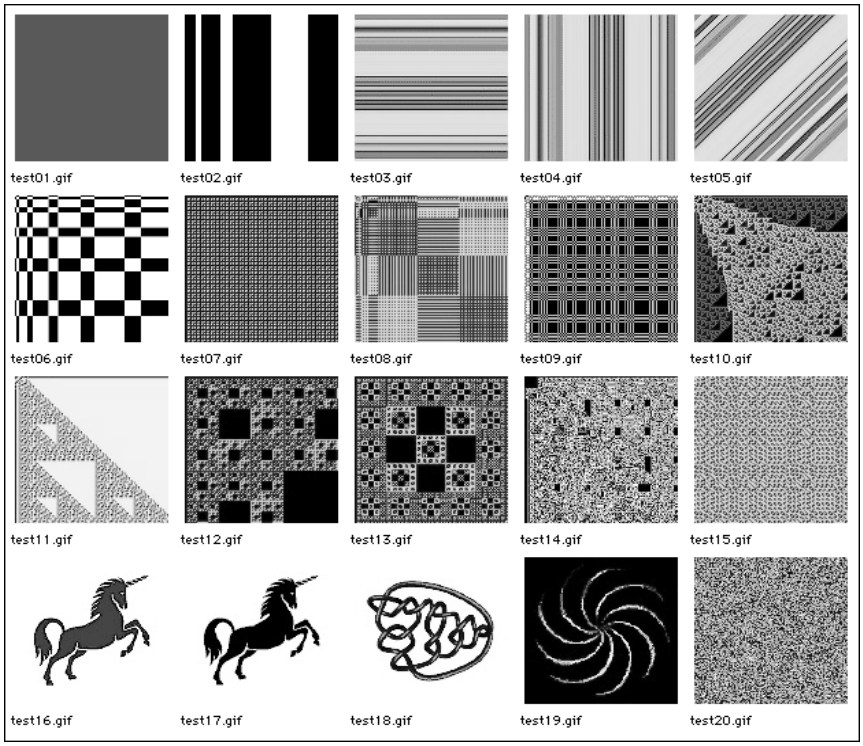


Figure 2. The Test Images

Test Image Information

We make no claim that these examples are, in any sense, representative. We just tried to get some diversity within the limits in which image grammars can be constructed.

So far we've said nothing about how well the concept of image grammars works or what degree of compression they give. If you've read this far, do you expect spectacular results or no compression at all?

Results

The image grammars for which results are shown here were constructed by hand using the application that is described in the article on analyzing character patterns that starts on page 1 of this issue of the Analyst.

The most obvious information of interest is the size of a grammar compared to the size of its pixel data.

image	*grammar size*	*% of pixel data size*
test01	54	0.33
test02	99	0.66
test03	421	0.21
test04	334	0.17
test05	3950	1.99
test06	446	2.72
test07	330	2.01
test08	626	3.82
test09	597	3.64
test10	5194	31.70
test11	2786	17.00
test12	2398	14.64
test13	2984	18.21
test14	8362	51.04
test15	897	5.47
test16	3848	8.94
test17	1745	4.05
test18	5264	12.40
test19	2681	15.86
test20	10592	64.65

Compression Results

It was surprising to us that there was significant compression for most of the test images. More surprising was the fact that for some nontrivial images, the grammar was less than 1% the size of the pixel data.

It's revealing to compare the sizes of the grammars with the sizes of the corresponding GIF files:

image	*grammar*	*GIF*	*% of GIF*
test01	54	188	28.72
test02	99	812	12.19
test03	421	18352	2.29
test04	334	60684	0.55
test05	3950	66888	5.90
test06	446	1138	39.19
test07	330	1535	21.50
test08	626	1327	47.17
test09	597	1589	37.57
test10	5194	3772	132.66
test11	2786	2566	108.57
test12	2398	2134	112.37
test13	2984	3187	93.63
test14	8362	6701	124.79
test15	897	1980	45.30
test16	3848	2987	128.82
test17	1745	1254	139.15
test18	5264	4143	127.06
test19	2681	2463	108.85
test20	10592	9540	111.03

Grammar and GIF Sizes

Again, we were surprised. We expected, instead, that except for the horizontal and vertical stripes, the GIF images would be smaller than the corresponding grammars.

What Image Grammars Look Like

For the most part, you're not going to want to look at image grammars: Most are too large and contain too many "odd-ball" characters for human consumption. However, here are two smaller ones, and a large one with most of it elided as indicated by ellipses. Long lines are split to fit. The "odd-ball" characters are as they appear in the Helvetica type face on a Macintosh.

test01:

```
axiom:∗
gener:1
∗–>[m,16384]
```

test02:

```
axiom:∗
gener:3
∗–>[BCACCAAD1AB,128]
A–>[0,8]
B–>[0,9]
C–>[1,5]
D–>[C,6]
```

test19:

```
axiom:*
gener:49
∗–>C1°N12°YHZX?=¥Xppppx}////vZ2XHX\i¥ …
!–>5B1d
"–>dY
$–>`c
%–>;1
&–>`0
'–>id
—>dc
      …
≠–>Mb
Æ–>dN
Ø–>oo
∞–>dP
±–>8F
≤–>MY
≥–>BB
¥–>11
```

```
µ–>55
∂–>iY
∑–>`P
∏–>FF
```

Other Aspects of Image Grammars

There's more to an image grammar than just its size. Image grammars that require a lot of characters beyond those that are "printable" cause problems in text editors.

Another consideration is how long it takes to convert an image grammar to an image string. First the grammar has to be run through an L-system processor to create its terminal string. This is a character pattern that typically contains many pattern forms. The pattern forms then have to be expanded to yield the desired pixel data.

Expanding a character pattern with pattern forms is not a problem; that's fast. The difficulty lies with getting the character pattern from the grammar. The time this takes depends largely on the number of generations required — the depth of the grammar. The time can be quite significant for grammars with large depth.

The number of characters used by a grammar often is related to the depth. Characters used in addition to those for tokens and meta-characters are used for variables. A grammar with a large number of variables usually has a large depth. The depth is bounded by the number of definitions that contain other variables (as opposed to those that just define a string of tokens) plus one for all the rest.

Here is the information on the number of variables used and the grammar depths:

image	*variables*	*depth*
test01	1	1
test02	3	2
test03	28	24
test04	13	3
test05	162	153
test06	39	31
test07	23	16
test08	59	49
test09	60	57
test10	165	147
test11	163	140
test12	167	155
test13	165	147
test14	162	100
test15	92	87
test16	145	112
test17	84	81
test18	160	135
test19	61	49
test20	160	2

Grammar Symbols and Depths

In most of these numbers, you'll see the expected relationship between the number of variables used and the depth. The numbers for test20 are exceptions. How could a grammar with 160 variables have a depth of 2? Obviously, all but one of the definitions themselves contain no variables. Impossible? No: All but one of the definitions are of the form

$$v \text{–>} t_1\, t_2$$

where t_1 and t_2 are tokens. That is, the grammar is composed almost entirely of digrams (2-grams).

Notice also that image grammars that have a large number of variables also are those that give poorer compression. Basically, this says something about the kinds of images for which image grammars might play a useful role.

Creating Image Grammars

It's hard to imagine image grammars being useful if they have to be constructed by hand. Even with charpatt, the process is tedious and slow.

Somewhat surprisingly, relatively compact image grammars can be created following a fixed program. Except for the most unusual images, the only pattern forms worth considering are repetitions and constants. And unless there are significant repetitions, it's unlikely that an image is suitable for representation as an image grammar.

Repetitions capture horizontal areas of solid color and repeating patterns. Constants that yield savings in grammar size can be found by getting *n*-grams.

After trying many different approaches, we settled on the following strategy: First get repetitions and then *n*-grams. There are several reasons for this order. In the first place, repetitions of significant length produce more savings than *n*-grams. In addition, finding repetitions is fast, while getting *n*-grams is relatively slow.

The number of pattern matches required to find an *n*-gram in a string of length *l* is

$$l - n + 1$$

which is essentially *l* for cases of interest. Thus, locating *n*-grams is linear in the current size of the grammar.

The more serious problem is that it is necessary to determine the potential savings for every *n*-gram. This requires computing the number of *non*-overlapping occurrences of each *n*-gram and using these numbers in the savings computations. If there are *k* different characters in a string, the number of potential *n*-grams is k^n. The actual number of different *n*-grams produced by a search almost always is much less than this, but the number still can be very large.

Within repetitions, we found that it usually works best to start with long repetitions of a single character (this takes care of long strings that represent areas of solid color). Then we proceed to 2-character, 3-character and larger repetitions.

For the *n*-gram "end game", we tried various approaches, but discovered that just looking for digrams works as well or better in most cases than looking for longer *n*-grams. (Recall that the image grammar for test20 consists almost entirely of digrams; it is smaller than any grammar we could find in other ways.)

You might think that looking for the longest *n*-grams first would yield better results, but that's not necessarily so. First note that the amount of space saved by creating a definition for an *n*-gram that occurs *m* times is

$$(m - 1) \times n - d$$

where *d* is the number of characters required for a new definition: one for the variable symbol, two for the –> separator, and one or two characters for a line terminator, depending on the platform [3]. So, for one-character terminators, *d* is 4, the value we'll use in the following example. The (*m* –1) factor comes from replacing the *m* occurrences but subtracting one for the *n*-gram's appearance in the new definition.

Suppose, for sake of example, that abcabc occurs three times in the grammar and abc occurs 20 additional (non-overlapping) times.

If we create a definition for abcabc, we get a savings of 5. There now are 22 occurrences of abc — the 20 original ones and the two in the new definition. Adding a second definition for abc yields a savings of 37, for a total of 43.

If instead we create a definition for abc, for which there are 26 occurrences altogether — six in the three occurrences of abcabc plus the other 20 — we get a savings of 45. And this adds only one definition.

This is, of course, a contrived example, but situations like it occur frequently. There are, of course, many other intertwining considerations. This is what makes finding the smallest possible grammar so fascinating.

In any event, we found it usually is best to work from smaller to larger *n*-grams: 2-grams (digrams), 3-grams (trigrams), and so on.

A more sophisticated approach would be to analyze the pixel data before searching for pattern forms. Even knowing the number of different characters and how many times each occurs can be used to steer the compression program.

Conclusion

We've not implemented a program for automatically generating image grammars. We're not convinced it's worth the effort. Are image grammars a great idea or just tricky business? Tricky business, we think, but with some surprisingly interesting properties.

And you could add them to your bag of tools for obfuscating your programs. Look at any but the simplest image grammar and try to imagine what image it represents.

A Note on Image Compression Techniques

The techniques used to create compact image grammars are of the same nature as those used by many image-compression algorithms.

The repetition pattern form is a type of run-length encoding, used in RLE image files and supported by many other bitmap image formats [4].

The representation of constant strings in grammars is a form of dictionary lookup, which is used in a variety of ways in image compression algorithms, including GIF [5].

Of course, image grammars employ these techniques in unsophisticated ways. Conversely, they can produce smaller representations than standard image-compression techniques because they can take advantage of structural relationships that general-purpose algorithms do not.

References

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

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

3. “Line Termination”, Icon Analyst 48, pp. 1-3.

4. *Encyclopedia of Graphics File Formats*, James D. Murray and William vanRyper, O’Reilly & Associates, 1994, pp. 132-142.

5. *Encyclopedia of Graphics File Formats*, pp. 142-148.

From the Library — Sorting

We described Icon’s built-in sorting repertoire in the last issue of the Analyst [1]. As in many cases, the Icon program library provides functionality to augment the built-in functionality. Two procedures stand out:

- sortff(), which is like the built-in function sortf() except that sortff() can sort on more than one field.
- isort(), which allows a procedure to be specified for ranking values.

Multi-Field Sorting

sortff(x, fields[]) sorts any “subscriptable” structure (records, lists, and tables but not sets) by fields given as a list of integers in fields. The structure first is sorted by fields[1], and then for elements with identical values, by fields[2], and so on. For example,

```
sortff(customers, 2, 1, 3)
```

sorts the values in customers first by field 2, then by field 1, and finally by field 3. sortff() is called with a variable number of arguments and all but the first are put in a list when the procedure is called.

Here’s the code:

```
procedure sortff(X, fields[])

   *X <= 1 & (return copy(X))

   return sortff_1(X, fields, 1, [])

end

procedure sortff_1(X, fields, k, uniqueObject)
   local sortField, cachedKeyValue, i
   local startOfRun, thisKey

   sortField := fields[k]
   X := sortf(X, sortField)     # initial sort using fields[k]
   #
   #  If more than one sort field is given, use each
   #  field successively as the current key, and,
   #  where members in X have the same value for
   #  this key, do a subsort using fields[k+1].
   #
   if fields[k +:= 1] then {
      #
      #  Set the equal–key–run pointer to the start
      #  of the list and save the value of the first key
      #  in the run.
      #
      startOfRun := 1
      cachedKeyValue := X[startOfRun][sortField] |
         uniqueObject
      every i := 2 to *X do {
         thisKey := X[i][sortField] | uniqueObject
      if not (thisKey === cachedKeyValue) then {
         #
         #  We have an element with a sort key
         #  different from the previous.  If there's a
         #  run of more than one equal keys, sort
         #  the sublist.
         #
         if i – startOfRun > 1 then {
            X := X[1:startOfRun] |||
               sortff_1(X[startOfRun:i], fields,
               k, uniqueObject) ||| X[i:0]
```

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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

```
        }
      # Reset the equal–key–run pointer to this
      #  key and cache.
      startOfRun := i
      cachedKeyValue := X[startOfRun][sortField]
        | uniqueObject
      }
    }
    #
    #  Sort a final run if it exists.
    #
    if i – startOfRun > 1 then {
     X := X[1:startOfRun] ||| sortff_1(X[startOfRun:0],
        fields, k, uniqueObject)
      }
    }

  return X

end
```

Customized Sorting

The procedure isort(x, p, y) sorts the elements of x using the procedure p for ranking them. x can be any value to which the element generation operation ! applies (strings and structures). For example,

```
isort(database, proc("*", 1))
```

sorts database by the sizes of its elements.

If p() takes more than one argument, y is passed to it as the second argument. The argument p also can be an integer, in which case it supplies a subscript that is applied to each element.

Here's the procedure:

```
procedure isort(x, keyproc, y)
  local items, item, key, result

  if y := integer(keyproc) then
    keyproc := proc("[]", 2)
  else /keyproc := 1

  items := table()

  every item := !x do {
    key := keyproc(item, y)
    (/items[key] := [item]) | put(items[key], item)
    }

  items := sort(items, 3)

  result := []

  while get(items) do every put(result, !get(items))

  return result

end
```

Other Sorting Facilities

The Icon program library contains other facilities for sorting. Most are intended for specialized applications such as managing address lists. Two programs have general applicability:

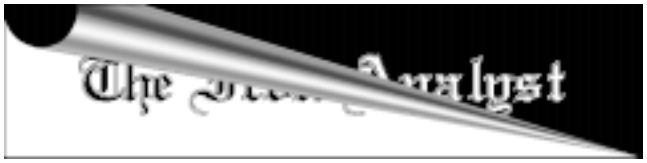
- grpsort, which sorts a file composed of multiline "records" with a designated separator.
- ipsort, which sorts an Icon program, putting the main procedure first, followed by the other procedures in alphabetical (lexical) order by name.

Acknowledgments

Bob Alexander and Richard Goerwitz wrote sortff(), Bob Alexander wrote isort(), and Tom Hicks wrote grpsort. If you're wondering about ipsort, it was written by an Analyst editor. It is our policy not to acknowledge editors' contributions.

Reference

1."Sorting", Icon Analyst 49, pp. 13-15.



What's Coming Up

We still have a big backlog of material (not that it's all in shape for publication).

Our expectations for the next issue are an article on pattern-form metrics, an article on generating versum numbers, and the first of a series of articles on animation.

We also expect to have a **Graphics Corner** that explores what can be done by changing the colors associated with image strings. Another **Tricky Business** article is a possibility.

Icon on the Web

Information about Icon is available on the World Wide Web at

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