The Icon Analyst

In-Depth Coverage of the Icon Programming Language

April 1999
Number 53

In this issue …

Weaving Drafts 1
Graphics Corner 4
A Small Programming Problem 10
Built-In Generators 16
Answers to Structure Quiz 19
Quiz — Expression Evaluation 19
What's Coming Up 20

Weaving Drafts

We now know that handwork is a heritage which no machine can ever take from us; we are adjusting our needs to this knowledge.

— Marguerite P. Davison [1]

The term *draft* is used in weaving for any description of a weave that can be used to produce it on a loom. For a treadle loom, a draft has five parts:

threading sequence
treadling sequence
warp color sequence
weft color sequence
tie-up

There are many other aspects of a weave that we won't consider here, such as thread thickness.

Draft Formats

Drafts traditionally have been hand-drawn on grid paper [2]. Most computer weaving programs display an on-screen grid that is familiar to weavers, but weaving drafts are stored in proprietary formats. There is also a program-independent format for exchanging drafts between programs. We'll describe it later in this article.

None of these formats is suitable for our exploration of weaving, which focuses on patterns. Instead, we use pattern-forms [3], which include the Painter weaving language repertoire [2,4].

Pattern-Form Drafts

It's easy enough to represent the five parts of a draft by strings. The threading and treadling sequences (T-sequences) can be composed from characters that label the shafts and treadles, respectively. The warp and weft color sequences (C-sequences) can be composed from characters that label colors.

The tie-up is a matrix that can be represented by, say, concatenated rows composed of zeros and ones. It's also necessary to add dimension information, since the matrix need not be square.

There is one missing ingredient: the colors themselves. To be general, we'd need the actual color values. For our purposes, however, Icon's built-in color palettes do nicely. There are two reasons for this: (1) the number of different colors in actual weaves is small, and (2) color fidelity is not necessary for exploring patterns in weaves; in fact, it is not even achievable in actual weaving.

There is one potential problem with using palettes. The color palettes span color space in various ways. No color palette can come even close to representing, say, 50 shades of blue (although the grayscale palettes can represent up to 256 shades of gray). We've not seen this kind of problem in practice, but we're working on programmer-defined palettes to supplement the built-in ones. Such defined palettes could handle situations the built-in ones cannot. Although programmer-defined palettes do not exist yet, they at least give us an out for now.

The name of the palette from which the color labels in C-sequences are composed adds a line to pattern-form drafts. We added a line for the draft name, making a pattern-form draft seven lines in all. Figure 1 on the next page shows an example of a pattern-form draft.

Grammar Drafts

We have used grammars to find and characterize patterns in various kinds of strings [3, 5-8]. To use this approach for pattern-form drafts, we need a corresponding grammar draft. The problem is that a weaving consists of several relatively unrelated parts.

The T-sequences naturally go together, since they often have patterns in common — in fact, in many cases the two T-sequences are the same: "tromp as writ" in weaving parlance.

The C-sequences have similar relationships and use the same labeling characters. But the characters in T- and C-sequences mean different things: shafts and treadles versus colors. To combine them all in one grammar invites chaos.

There are also patterns in tie-ups, but they tend to be two-dimensional in nature and searching for such patterns in a one-dimensional representation does not work well.

To keep a draft together in one file, we designed a grammar-draft format that is composed of three sub-grammars — one for the T-sequences, one for the C-sequences, and a token grammar in which the name, tie-up, and palette are hidden away. The sequence pairs are connected in definitions that use variables not contained in the sequences.

A grammar draft is shown in Figure 2 with long sequences truncated. It is the grammar from which the pattern-form draft in Figure 1 was derived.

WIFs

WIFs (Weaving Information Files) provide a program-independent way of specifying weaves. WIFs are widely used to exchange drafts between weaving programs <1>, and there are several standalone programs for converting between proprietary forms and WIFs <2-5>.

```
aquadesign                                                        name
[[1>8]∗5][[7<1]8∗4]765432[[1>8]∗5][[7<1]8∗4][7<1]                  threading
[[1>8]∗10]                                                        treadling
[G∗20][H∗39][G∗20][G∗19][H∗39][G∗20]                              warp colors
[0–>80]                                                           weft colors
c1                                                                palette
8;8;1001001111000001111000000111000000111001100101000100101000100101   tie-up
```

Figure 1. A Pattern-Form Draft

```
name:T–aquadesign.wgr
comment:
axiom:@
genr:1
@–>X;Y
X–>1234567812345678123456781234567812345678765432187654321876543218765432...
Y–>1234567812345678123456781234567812345678123456781234567812345678123456...
end:
name:C–aquadesign.wgr
comment:
axiom:@
genr:1
@–>U;V
U–>GGGGGGGGGGGGGGGGGGGGHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH...
V–>0
end:
name:t–aquadesign.wgr
n–>aquadesign
p–>c1
t–>8;8;1001001111000001111000000111000000111001100101000100101000100101
end:
```

Figure 2. A Grammar Draft

The WIF format is text-based, very general, and extensible. Its syntax is based on the Windows INI format [9], which uses named sections that contain lists of keywords and associated values. Figure 3 shows portions of a typical WIF.

WIFs have properties that make them inappropriate for direct use in our investigations:

- The format is very verbose; a typical WIF contains a large amount of redundant information.
- Parsing a WIF file is a messy process and not something to do repeatedly.
- The format provides no mechanism for describing patterns — it's just raw data.

On the other hand, WIFs provide a ready source of material for studying patterns in weaves. Some WIFs are available on the Web <6-8> and there is a moderately priced CD-ROM that contains thousands of WIFs [10].

To use WIFs, we need to be able to convert them to pattern-form drafts. This is the kind of task for which Icon is well suited. As is typical with format-conversion programs, the process is messy and infested with special cases. We won't list the program here: It would take up a lot of space without providing much in the way of insight. We will, however, include it with the Web material for this issue of the Analyst, along with a considerably simpler program to convert pattern-form drafts to WIFs.

Further Articles

We have several articles on weaving in the works, including

- a weaving case study
- finding patterns in draft sequences
- creating images from drafts
- creating drafts from images
- name and algebraic drafting
- network drafting
- name and algebraic drafting
- sequence drafting
- dobby devices and liftplans
- weavability
- a weaving program

We're also investigating Jacquard loom technology and exotic topics like three-dimensional weaving.

```
[WIF]
Version=1.1
Date=April 20, 1998
Developers=WIF@mhsoft.com
Source Program=QD WIF
Source Version=0.9.2

[CONTENTS]
Color Palette=yes
Weaving=yes
    …

[COLOR PALETTE]
Entries=16
Form=RGB
Range=0,255

[WEAVING]
Shafts=16
Treadles=25

[THREADING]
1=1
2=2
3=1
4=2
5=3
6=4
    …

[TIEUP]
1=4,5,6,7,8,9,10,11,12,13,14,16
2=1,2,6,11,12,13,14,15
3=1,2,3,4,8,12,16
4=1,2,3,4,5,6,10,15
5=1,2,6,11,14
6=1,2,3,4,5,6,7,8,12,16
    …

[TREADLING]
1=17
2=18
3=17
4=18
5=17
6=18
    …

[COLOR TABLE]
1=192,192,192
2=152,152,152
3=0,0,0
4=176,144,144
5=116,100,152
6=152,100,132
    …

[WARP COLORS]
1=6
2=5
3=3
4=3
5=3
6=8
    …

[WEFT COLORS]
1=14
2=12
3=8
4=14
5=3
6=14
    …
```

Figure 3. A WIF

The situation with respect to weaving is reminiscent of versum sequences — the more we explore, the more we find to explore. Incidentally, we're not finished with versum sequences — we're just taking a breather. And don't be surprised to see a versum weave.

Acknowledgment

We owe thanks to Jane Eisenstein, an experienced handweaver. She uses several weaving programs for the Macintosh and is the author of QD WIF for that platform <5>. She has patiently answered our naive questions as we've been learning about weaving. She's also given us pointers to other useful resources related to weaving.

References

1. *A Handweaver's Pattern Book*, Marguerite P. Davison, Marguerite P. Davison, Publisher, 1994.

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

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

4. "A Weaving Language" Icon Analyst 52, pp. 1-3.

5. "Character Patterns" Icon Analyst 48, pp. 1-7.

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

7. "Analyzing Character Patterns" Icon Analyst 50, pp. 1-7.

8. "Tricky Business — Image Grammars" Icon Analyst 50, pp. 14-18.

9. *Inside Windows: 3.11 Edition*, Jim Boyle, et al., New Riders, 1994.

10. *Over 7000 Draw Downs in Color*, Eleanor Best, multi-platform CD-ROM, EnGBest@aol.com,1998.

Links

1. WIF Specification:

   http://www.mhsoft.com/wif/wif.html

2. WeaveConvert (Windows) for Patternland 5 and Fiberworks 3:

   http://www.blarg.net/~ender/weaveconvert/

3. ptn2wif (Windows and source code) for WinWeave:

   http://www.gac.edu/~max/weaving/ptn2wif/

4. QD WIF (Macintosh) for Macintosh ProWeave , WeaveMaker One, WeaveNet, and WeavePoint:

   http://www.softweave.com/html/QD_WIF.html

5. WIFCNVT (Windows) for WeaveIt:

   http://www.weaveit.com/support.htm

6. Weaving File Exchange:

   http://www.wyellowstone.com/users/ww/weaving.htm#File Exchange

7. Association of Northwestern Weavers Guilds:

   ftp://anwg.org/pub/weaves/

8. Maple Hill Software:

   http://www.mhsoft.com/wif/files/

Graphics Corner — Changing Image Colors

There is no room for frivolity where color is concerned.
– Color Marketing Group [1]

In the last issue of the Analyst [2], we showed how mutable colors can be used in to create animations. This article describes a quite different use of mutable colors — changing colors in an image interactively.

The Application

The idea is simple — create a copy of an image in which each color is mutable and provide an interface through which a user can select a color and change it. Figures 1 and 2 show the interface and an example image.

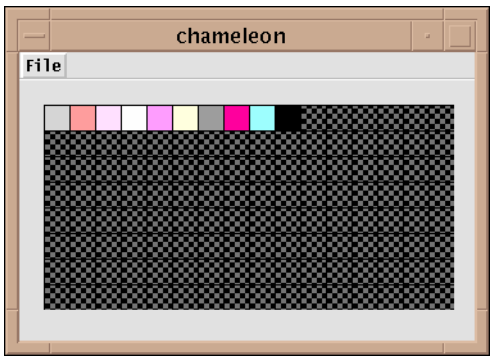


Figure 1. The Application Interface

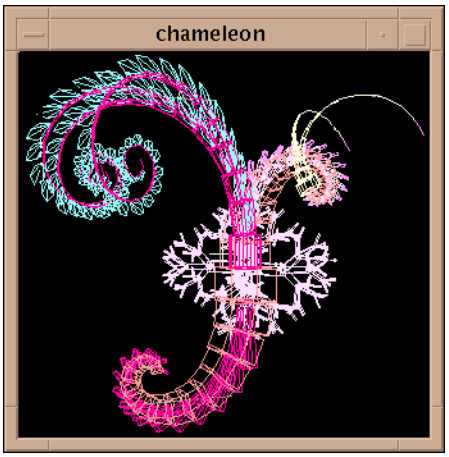


Figure 2. An Example Image

The palette on the interface contains a cell for each color in the image. When the user clicks on cell, a color dialog is presented, as shown in Figure 3.

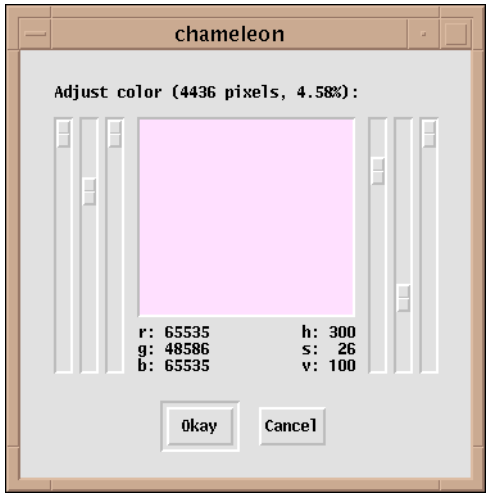


Figure 3. Color Dialog

Note that the dialog provides information about the use of the selected color in the image.

When the user changes the color in the dialog, as shown by the example in Figure 4, that color changes in the image.

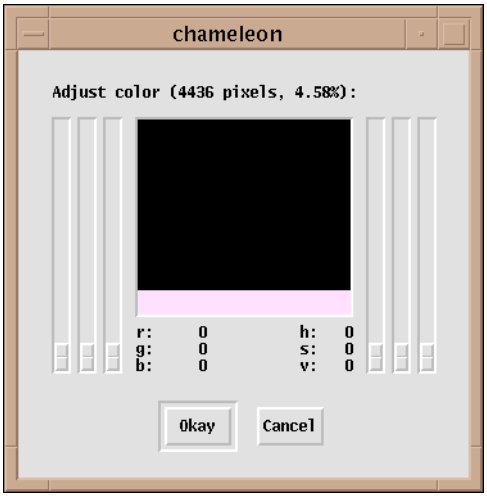


Figure 4. Modified Color Dialog

In Figure 4, the selected color has been changed to black, the background color for the image, causing the ornamentation in the center of the image to disappear. The result is shown in Figure 5.

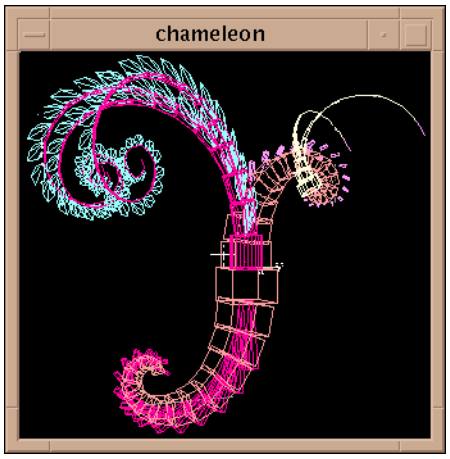


Figure 5. The Modified Image

If the user dismisses the color dialog with Cancel rather than Okay, the previous color is restored.

The application also provides for saving a changed image and reverting to the original colors.

Many other features could be added. See the **Extensions** section at the end of this article.

The Program

The complete program is shown in Figure 6 on following pages. The comments about the program are keyed to the numbered circles in the listing.

```
link graphics
link interact
link numbers
link tables

global cellsize              # size of palette cell
global colors                # mutable color list
global count                 # table of pixel counts
global image_window          # window for user image
global mutant                # image with mutable colors
global orig_colors           # list of original colors
global palette               # color selection palette
global panel                 # palette window
global pixels                # number of pixels in image window
global mutant_posx           # location of mutant window
global mutant_posy

$define ColorRows    8       # palette rows
$define ColorCols    16      # palette columns
$define FrameWidth   24      # allowance for window–manager frame (ad hoc)

procedure main()
   local atts, vidgets

   atts := ui_atts()

   put(atts, "posx=0", "posy=0")                                          ❶

   (WOpen ! atts) | stop("*** cannot open application window")

   mutant_posx := WAttrib("posx") + WAttrib("width") + FrameWidth
   mutant_posy := WAttrib("posy")                                         ❷

   vidgets := ui()

   palette := vidgets["palette"]

   cellsize := palette.uw  / ColorCols                                    ❸

   panel := Clone("bg=black", "fg=black", "dx=" || palette.ux, "dy=" || palette.uy)
   Clip(panel, 0, 0,  palette.uw, palette.uh)                             ❹

   clear_palette()

   GetEvents(vidgets["root"], , shortcuts)                                ❺

end

#  Set up empty palette grid.

procedure clear_palette()
   local x, y

   Fg(panel, "black")
   EraseArea(panel)
   WAttrib(panel, "fillstyle=textured")                                   ❻
   Pattern(panel, "checkers")
   Bg(panel, "very dark gray")

   every x := 1 + (0 to ColorCols – 1) * cellsize do
      every y := 1 + (0 to ColorRows – 1) * cellsize do
         FillRectangle(panel, x, y, cellsize – 1, cellsize – 1)           ❼

   WAttrib(panel, "fillstyle=solid")
   Bg(panel, "black")

   return

end

# Handle File menu.
```

Figure 6. The Program

❶ The posx and posy attributes are added to the attributes VIB provides for the application window. The intent is to position the application window at the upper-left corner of the screen. This generally is not possible, since the window manage provides a frame around the window that must fit on-screen. Most window managers do the best they can to position the window as requested. However, the actual coordinates may not be the same as the requested ones.

❷ mutant_posx and mutant_posy are set to position the window for the mutable image to the right of the application window. In general, it's not possible to compute the window manager's frame width. An *ad hoc* constant is used to approximate it.

❸ The cell size is computed from the "usable" size of the region, which is inside its visual frame. We assume here that the region size has been set up in VIB so that this works out properly.

❹ A window for the palette grid is cloned from the application window and the clipping is set so that drawing on the grid cannot overwrite areas outside it.

❺ GetEvents() is used because this application is entirely event driven. That is, computation is done only as a direct result of user requests. GetEvent() never returns; user action causes program termination.

❻ The cells are filled initially with a dark checkered background. This makes it easy to distinguish unused cells from cells filled with colors, which are solid.

❼ Notice that the filled rectangles are one pixel less on a side than the cell size. This provides the black borders around the cells.

```
procedure file_cb(vidget, value)
   case value[1] of {
      "open     @O"  :  image_open()
      "quit     @Q"  :  quit()
      "revert   @R"  :  image_revert()
      "save     @S"  :  image_save()
      }
   return
end

# Open new image.

procedure image_open()
   local i, x, y
   WClose(\image_window)                                          ❽
   repeat {
      if OpenDialog("Open image:") == "Cancel" then fail
      image_window := WOpen("canvas=hidden", "image=" || dialog_value) | {
            Notice("Cannot open image.")                          ❾
            next
            }
      break
      }
   mutate(image_window) | fail                  # create the mutant window
   Raise()                                                        ❿
   clear_palette()                              # clear old color cells
   colors := vallist(orig_colors)                                 ⓫
   i := 0
   every y := 1 + (0 to ColorRows – 1) * cellsize do
      every x := 1 + (0 to ColorCols – 1) * cellsize do {
         Fg(panel, colors[i +:= 1]) | break break
         FillRectangle(panel, x, y, cellsize – 1, cellsize – 1)
         }
   return
end

#  Save current image, which is in the mutant window.

procedure image_save()
   snapshot(\mutant) | {                                          ⓬
      Notice("No image to save.")
      fail
      }
   return
end

#  Restore original image colors.

procedure image_revert()
   local old, color
   every old := key(orig_colors) do {                             ⓭
      color := orig_colors[old]
      Color(panel, color, old)
      }
```

Figure 6 (continued). The Program

❽ Before an image is opened, the previous image, if any, is closed. The nonnull test fails if there is not a previous image.

❾ The image is opened with its canvas hidden. A window for the mutable image will be created, and there is no need to produce a distraction by having two windows visible on screen. If the image cannot be opened, WOpen() fails and the user is notified.

❿ After the mutant window is created, Raise() is used to bring the application window to the front and make it the "focus" from which events are then accepted. (The focus may not be set by all window managers.)

⓫ orig_colors is a table created by mutate(). Its keys are the mutable color numbers and the corresponding values are the original image colors. The palette cells are filled using the mutable colors, which at this point have the same color values as those in the image.

⓬ If there is no mutant window, an attempt to save an image results in a notification to the user. This can only happen if the user tries to save an image before opening one, but an application needs to handle aberrant user actions (not to mention crazed beta-testers) in a proper way. The use of fail instead of return here and in other similar situations is primarily for documentation purposes and for tracing. It has no actual affect on program behavior.

⓭ The original color values are in orig_colors. To revert to them, the corresponding mutable colors in the color panel are reset. Changing the color value associated with a mutable color changes the color value in all windows of an application. It therefore is not necessary to change the color values in the mutant window explicitly.

```
   return
end
# Get mutable colors and window from image.
procedure mutate()
   local c, width, height, n, x, y
   WClose(\mutant)
   orig_colors := table()
   count := table(0)
   width := WAttrib(image_window, "width")
   height := WAttrib(image_window, "height")
   pixels := width * height
   mutant := WOpen("width=" || width, "height=" || height,
      "posx=" || mutant_posx, "posy=" || mutant_posy) | {
         Notice("Cannot open mutant window.")
         fail
         }
      every y := 0 to height – 1 do {
         x := 0
         every c := Pixel(image_window, 0, y, width, 1) do {
            if not(n := \orig_colors[c]) then {
               orig_colors[c] := n := NewColor(c) | {
                  Notice("Too many colors in image.")                    ⓮
                  WClose(mutant)
                  fail
                  }
               }
            count[n] +:= 1
            Fg(mutant, n)
            DrawPoint(mutant, x, y)                                      ⓯
            x +:= 1
            }
      }
   return
end
# Handle callbacks on the palette.
procedure palette_cb(vidget, e, x, y)
   local color, new
   if e === (&lpress | &mpress | &rpress) then {                         ⓰
      color := Pixel(x, y, 1, 1)   # get pixel color
      if not integer(color) then fail                  # not mutable
      new := Color(panel, color)                       # get color
      if ColorDialog(
         "Adjust color (" || count[color] || " pixels, " ||
            frn((100.0 * count[color]) / pixels, , 2) || "%):",
         Color(panel, color),
         track,                  # procedure to track color changes      ⓱
         color                   # color being mutated
         ) == "Okay" then new := dialog_value
      Color(panel, color, new)
      Color(mutant, color, new)
      }
```

Figure 6 (continued). The Program

⓮ NewColor() fails if no more mutable colors are available. This means that the image has too many colors for this application. Only 256 colors are available altogether. Two are needed for every color in the image — one for the color itself and another for its mutable counterpart. Consequently, images that can be processed cannot have more than 128 characters. The actual number is less in most cases, since black and white are reserved and the application interface uses two other colors. Note that if there are too many colors, the partially completed mutable window is closed. Opening another image causes this also, but it's better not to leave the screen cluttered, even temporarily.

⓯ For every pixel in the image, a corresponding pixel in the mutant window is drawn. Drawing an image of more than trivial size, point by point, is time-consuming. The practical size of images for which this application can be used is limited.

⓰ The user presses a mouse button with the mouse pointer positioned on a cell on the palette grid to indicate that the corresponding color is to be adjusted. Note that if the color of the pixel on which the user clicks is not an integer, it is not a mutable color. This can happen if the user clicks on an empty cell or, more likely, accidentally clicks on a border pixel between cells. If this happens, the event is ignored.

⓱ The third argument in a call of ColorDialog() is a procedure to call for every change the user makes to the color. This

```
   return
end

#  Quit the application.

procedure quit()
   snapshot(\mutant)
   exit()                                                    ⓲
end

#  Handle keyboard shortcuts.

procedure shortcuts(e)
   if &meta then case(map(e)) of {
      "o"  :  image_open()
      "q"  :  quit()
      "r"  :  image_revert()
      "s"  :  image_save()
      }
   return
end

#  Track the color in the color dialog.

procedure track(color, s)                                    ⓳
   Color(panel, color, s)
   Color(mutant, color, s)
   return
end

#===<<vib:begin>>===        modify using vib; do not remove this marker line
procedure ui_atts()
   return ["size=355,225", "bg=pale gray", "label=chameleon"]
end

procedure ui(win, cbk)
return vsetup(win, cbk,
   [":Sizer:::0,0,355,225:chameleon",],
   ["file:Menu:pull::1,0,36,21:File",file_cb,
      ["open   @O","save   @S","revert @R","quit   @Q"]],
   ["menubar:Line:::0,21,357,21:",],
   ["palette:Rect:invisible::19,41,320,160:",palette_cb],
   )
end
#===<<vib:end>>===          end of section maintained by vib
```

Figure 6 (concluded). The Program

allows user changes to be tracked in the mutant image so the user can see the effect of changes as they are made and before they become permanent.

⓲ The application should keep track of whether or not the image has been changed since it was last saved and offer to save it only if it has changed.

⓳ When the tracking procedure is called, its first argument is the fourth one provided when ColorDialog() is called (in this case, the mutable color being changed). The second argument to the tracking function is the current color from the dialog. Both the pixels in the palette grid and the image itself are changed.

Extensions

The functionality of the program described here could be extended is several ways. Some suggestions are:

- selection of a color by clicking on a pixel in the mutant image
- palette color sets that can be saved and loaded
- an undo facility
- freeing colors when the mutant image is closed so that they can be reused for another image
- multiple mutant windows with color transfer between them.

You probably can think of other possibilities … contributions to the Icon program library always are welcome.

Limitations

Recall that mutable colors only work with monitors set to 8-bit depth or less and that they do not work properly in Windows Icon.

Acknowledgment

The images used in the examples in this article and on the last page were adapted from an "air horse" created by Laurens Lapré, using his very sophisticated Lparser application <1,2>.

Next Time

In the next **Graphics Corner**, we'll return to patterns and describe an application for finding interesting tiles in even the most mundane image.

References

1. This remark is from a 1995 press release by The Color Marketing Group <3> in which they announced the new color names for 1995, including Coralando and Cricket. Other new colors this organization has "created" in recent years include Vinaigrette, Stetson, Canyon Cloud, Elephant's Breath, Highway, Red Haute, and Treasure. The new colors they announced for 1999 include Cosmetique Peach, Beignet, Blue Moon, Par Four Green, Pink for Sure, Hip Hop Yellow, and Mineola Orange.

2. "Animation — Mutable Colors" Icon Analyst 52, pp. 11-16.

Links

1. Lparser:

   http://www.xs4all.nl/~ljlapre/lparser.htm

2. Lparser for the Macintosh:

   http://www.geocities.com/CapeCanaveral/9376/meshula.html#LParser

3. Color Marketing Group:

   http://www.colormarketing.org/

A Small Programming Problem — Sorting Digits

The chief cause of problems is solutions.
— unknown

The Problem

Late last year we posted the following problem to members of icon-group:

> Write a procedure digsort(i) that returns the integer that results from sorting the digits of i, preserving sign. For example, digsort(201) should return 12 and digsort(–1042) should return –124. You may assume i is an integer.

We later had to clarify this with the additional provision that duplicate digits should be preserved, so that, for example, digsort(10881) should return 1188. (Without this provision, the problem is essentially trivial — integer(cset(i))).

We received a number of solutions, some though icon-group, others directly, and we added a few of our own. Two solutions were erroneous: They failed to handle zero correctly.

We originally planned to present the results exactly as received, preserving the author's style, variable names, and so on. This approach, however, made it difficult to compare the solutions, so we've modified the solutions to use similar styles and names. We've also eliminated comments in the few cases they were provided in favor of out-of-line remarks. In some cases we've combined very similar solutions.

We've not associated authors' names with particular solutions (no glory; no embarrassment). The authors are, however, listed at the end of this article.

We've classified the solutions by the kinds of data structures and operations that were used.

Basic String and Cset Operations:

Several of the solutions used only basic string and cset operations.

Solution 1:

```
procedure digsort(i)
   local c, s

   s := ""

   every c := !"–0123456789" do
      every find(c, i) do
         s ||:= c

   return integer(s)

end
```

This is about as straightforward as you can get. It could be made slightly faster, on average, by handling 0 differently:

```
   every c := !"–123456789" do
      every find(c, i) do
         s ||:= c

   return integer(s) | 0
```

This saves looking for zeros in the integer, leaving s as the empty string only if i is 0. This is handled by the alternative in the return expression, since integer() fails and 0 is returned. Note that the alternative is evaluated only if i is 0; there is no overhead otherwise.

Incidentally, some solutions similar to this

one used upto(c, i) instead of find(c, i). This requires converting the *string* c produced by element generation to a cset for every call of upto() and hence is slower.

Solution 2:

```
procedure digsort(i)
   local c, s

   s := ""
   i := string(i)

   every c := !cset(i) do
      every s ||:= (find(c, i) & c)

   return integer(s)

end
```

This solution, as simple as it appears, has several clever aspects. The first is the conversion of i to a string, so that it doesn't have to be done in every call of find(). This solution also uses cset(i) to remove digits that do not occur in i. For some data, this may be faster, but it requires the creation of a cset and conversion back to a string for the element generation operator. Note that there is an implicit recognition that "–" occurs before the digits in the collating sequence (for both ASCII and EBCDIC). This fact was used in many solutions.

Solution 3:

```
procedure digsort(i)
   local c, s

   s := ""
   i := string(i)

   every c := !cset(integer(cset(i))) do
      every s ||:= (find(c, i) & c)

   return integer(s)

end
```

This slight variation on Solution 2 contains an obscure trick (that is, it's really clever). The expression

```
!cset(integer(cset(i)))
```

removes any zeros in i by converting all the characters to an integer. It would be (slightly) faster to use

```
!integer(cset(i))
```

since in this case the integer, which contains at most one instance of every digit in i, is converted directly from an integer to a string for element generation rather than via an intermediate cset.

But what about this?

```
!(i -- '0')
```

This removes any zeros and produces a cset of the remaining digits. The result is not the same as that for the previous expression, however, if i is 0. Therefore a procedure that uses it might end as shown in the remarks following Solution 1:

```
return integer(s) | 0
```

String Scanning

Solutions that used string scanning were similar to those that used basic string operations. They generally were longer, if (perhaps) somewhat easier to formulate.

Solution 4:

```
procedure digsort(i)
   local s, c

   s := ""

   i ? {
      every c := !cset(i) do {
        while s ||:= (tab(find(c)) & move(1))
        tab(1)
        }
    }

   return integer(s)

end
```

Compare this solution to Solution 2.

Solution 5:

```
procedure digsort(i)
  local s, s1, s2

  s := string(i)
  s2 := ""

  every s1 := !cset(s) do s ? {
    while tab(find(s1)) do
      s2 ||:= move(1)
    }

  return integer(s2)

end
```

This solution differs from the previous one in that string scanning is inside the character-generation loop rather than outside it.

Solution 6:

```
procedure digsort(i)
   local   s

   s := ""
   every s ||:= (i ? (tab(find(!cset(&subject))) &
     move(1)))

   return integer(s)

end
```

Here string scanning is inside the concatenation that is building the result, while the element-generation operation is inside string scanning. This formulation seems to us to be a bit strange, but it *is* different.

Lists

The solutions that used lists did so to accumulate characters or character counts and then sort() to get them in the right order.

Solution 7:

```
procedure digsort(i)

   local   L, s

   L := list()
   every put(L, !i)

   s := ""
   every s ||:= !sort(L)

   return integer(s)

end
```

This solution illustrates the basic method — straightforward and clear.

Solution 8:

```
procedure digsort(i)
   local L, s

   every put(L := [], !i)

   every (s := "") ||:= !sort(L)

   return integer(s)

end
```

This solution is essentially the same as the last one, but it is made slightly shorter, in some sense, by nesting expressions.

Solution 9:

```
procedure digsort(i)
   local L, j, k

   L := []
   j := abs(i)
   while j ~= 0 do {
      put(L, 0 ~= j % 10)
      j /:= 10
      }

   every k := !sort(L) do
       j := j * 10 + k

   if i < 0 then j := –j

   return j

end
```

Surprise: Actual arithmetic!

Solution 10:

```
procedure digsort(i)
   local L, s

   L := []

   i ? {
      s := tab(any('–+')) | ""
      while put(L, move(1))
      }

   L := sort(L)
   while s ||:= get(L);

   return integer(s);

end
```

This solution initializes the string of sorted digits in string scanning. Note that the plus sign is spurious; an integer (assumed), when converted to a string, never has a leading plus sign.

Solution 11:

```
procedure digsort(i)
   local L, j, d, s

   L := list(9, "")

   every d := !abs(i) do
      L[d] ||:= d

   s := ""

   every s ||:= !L

   return if i < 0 then –s else (integer(s) | 0)

end
```

This solution differs from the preceding ones by using failure on an out-of-bounds subscript to

dispense with zeros. Since a minus sign would cause an error, it works on the absolute value. Notice that unary minus operation converts numeral strings representing integers to integers.

Solution 12:

```
procedure digsort(i)
   local L, j, s

   L := list(9, 0)

   every L[!abs(i)] +:= 1

   s := ""

   every j := 1 to 9 do
      s ||:= repl(j, L[j])

   return if i < 0 then –s else (integer(s) | 0)

end
```

This solution is a variant of the preceding one, counting digits instead of concatenating them one at a time. Then repl() is used to build the string of digits. This solution should be faster for large integers.

Tables

Tables were used for basically the same reasons as lists.

Solution 13:

```
procedure digsort(i)
   local T, s

   T := table("")

   every s := !i do
      T[s] ||:= s

   s := ""

   every s ||:= T[!cset(i)]

   return integer(s)

end
```

Notice the similarity between this solution and Solution 11. Using tables, however, allows a minus sign to be handled like any other character.

Solution 14:

```
procedure digsort(i)
   local T, L, d, s

   if i = 0 then return 0

   T := table(0)

   every T[!i] +:= 1

   L := sort(T, 3)

   s := ""

   while d := get(T) do
      s ||:= repl(d, get(L))

   return integer(s)

end
```

Like Solution 12, this one keeps counts of characters and uses repl() rather than repeatedly concatenating digits.

Solution 15:

```
procedure digsort(i)
   local L, s

   if i = 0 then return 0

   L := table(0)

   every L["0" ~== !i] +:= 1

   L := sort(L, 3)

   s := ""

   while d := get(L) do
      s ||:= repl(d, get(L))

   return integer(s) | 0

end
```

This solution filters out zeros while counting other characters. Is it worth the time and trouble?

Solution 16:

```
procedure digsort(i)
   local T, L, d, s

   T := table(0)

   every T[!i] +:= 1

   delete(T, "0")

   L := sort(T, 3)

   s := ""

   while d := get(L) do
      s ||:= repl(d, get(L))

   return integer(s) | 0

end
```

This solution counts zeros but removes the table entry before constructing the final string.

The Different

Here are two solutions that are quite different in character from the others.

Solution 17:

```
procedure digsort(i)

   return if i < 0 then –digsort1(–i) else digsort1(i)

end

procedure digsort1(i)
  local pt1, pt2

  if i[pt1 := 2 to *i] < i[pt2 := 1 to pt1 – 1] then
     return digsort1(i – (i[pt2] – i[pt1]) *
      (10 ^ (pt1 – pt2) – 1) * 10 ^ (*i – pt1))

  return i

end
```

The author has this to say about the solution: "This won't win any brevity or obscurity awards, but it is another approach". We're not so sure about the obscurity part, at least in the context of Icon.

Solution 18:

```
procedure digsort(i)
   local j, sw

   i := string(i)

   sw := 1

   while \sw do {
      sw := &null
      every j := 1 to *i – 1 do {
         if i[j] >> i[j + 1] then {
            i[j] :=: i[j + 1]
            sw := 1
            }
         }
      }

   return integer(i)

end
```

We suppose there had to be one actual character sort. In this case it is the classic bubble sort, which is one of the least efficient sorting methods, on average. Note that a minus sign is handled properly, since string comparison is used.

Evaluation

Clarity and style are subjective. You can make your own judgement of these. Two more or less objective measurements can be made: size and speed.

Size

The procedures all are small, so some measures that would be meaningful for large programs are not so meaningful here. For what it's worth, here are the sizes measured in terms of non-blank lines, bytes of source code, and the number of syntactic tokens. The latter probably is the most significant measure. The smallest values are underlined.

soln.	*lines*	*bytes*	*tokens*
1	8	162	<u>28</u>
2	8	176	36
3	8	188	40
4	11	220	41
5	10	208	43
6	<u>6</u>	147	30
7	8	152	32
8	<u>6</u>	<u>137</u>	29
9	13	257	58
10	11	207	45
11	9	207	50
12	9	219	54
13	9	175	39
14	11	235	57
15	11	243	59
16	11	232	57
17	10	301	71
18	15	297	60

Speed

We tested the solutions with 5,000 instances of seven kinds of integers: positive and negative 5-digit, 10-digit, and 15-digit ones and one of all zeroes — 35,000 calls in all. Except for the zeros, the integers were produced at random. The times in milliseconds from runs on a 233MHz DEC Alpha are shown in Figure 1 on the next page. The times for the fastest solutions are underlined.

The apparent anomaly for Solution 15 with all zeroes is easily explained — it's the only solution that tests for zero before doing anything else. It's not surprising that the last two solutions did not fare well. The magnitude of the difference, especially for Solution 17, was surprising to us — it runs nearly 154 times slower than Solution 2.

But what about large integers? There are none for the timings above (the Dec Alpha has 64-bit

soln	*0*	*5-digit +*	*5-digit –*	*10.digit. +*	*10-digit –*	*15-digit +*	*15digit –*	*total*
1	650	934	1000	1233	1283	1517	1584	8201
2	250	567	617	850	916	1100	1184	5484
3	300	617	650	850	916	1100	1184	5617
4	400	1167	1333	1933	2116	2650	2850	12449
5	367	1100	1283	1883	2050	2650	2800	12133
6	350	900	1017	1517	1650	2100	2234	9768
7	300	600	667	1017	1066	1400	1467	6517
8	300	600	633	983	1016	1367	1434	6333
9	267	1217	1217	2167	2116	3034	3034	13052
10	450	934	967	1550	1566	2150	2200	9817
11	484	884	833	1200	1166	1517	1517	7601
12	1234	1584	1517	1817	1783	2084	2084	12103
13	434	1084	1217	1750	1883	2367	2500	11235
14	34	1350	1533	2083	2283	2700	2917	12900
15	17	1317	1517	2000	2233	2617	2817	12518
16	450	1317	1533	2033	2266	2600	2817	13016
17	100	4417	4483	35250	35833	131717	133034	344834
18	184	2684	3050	10967	11716	25467	26767	80835

Figure 1. Timings

words). We tried timing the programs for some large integers, using 5,000 terms in the versum sequence for 196. These integers get large quickly. Their average length (number of digits) is more than 1,048 and the last one has 2,088 digits. Most of these numbers are large enough for the quadratic complexity of conversion between integers and strings to have a significant impact [1].

Here are the results:

soln.	*last*	*all*
1	1883	3188534
2	233	402050
3	216	380184
4	433	759684
5	266	495000
6	250	506350
7	250	468850
8	250	452734
9	416	782234
10	266	487050
11	233	433734
12	216	399484
13	433	771584
14	250	450917
15	233	439267
16	233	435567
17	?	–
18	343933	–

We didn't attempt to perform the tests for the entries that have a dash. We did try Solution 17 for the last integer. We ran it in the background expecting it to complete in a large but reasonable amount of time. It had been running over 25 *days* (more than 2×10^9 milliseconds) when its process was accidentally terminated. You might think it (or Icon) was in a loop. But we tried a much smaller

Supplementary Material

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

http://www.cs.arizona.edu/icon/analyst/iasub/ia53/

large integer for which Solution 17 ran a very long time but completed successfully. There is a lesson here.

Contributors

The following persons contributed solutions to the digit-sorting problem: Wade Bowmer, Michael Glass, Ralph Griswold, Nevin Liber, Todd Proebsting, Gregg Townsend, Ken Walker, Steve Wampler, and Cheyenne Wills.

Although we've not identified the authors of the particular solutions, it seems only fair to give Steve Wampler kudos for submitting the fastest solution.

Reference

1. *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms,* Donald E. Knuth, Addison-Wesley, 1969.

Built-in Generators

If a language doesn't affect the way you think about programming, it's not worth knowing.

— Alan Perlis

One of the defining characteristics of Icon is its extensive use of sequences — values in order.

Strings are sequences of characters. Lists are sequences of values that may be of any type. Files are sequences of characters or lines, depending on how they are interpreted.

Data objects that are sequences are nothing new, although in most programming languages the concept is not stressed. Generators that produce sequences of values in *time* are central to computation in Icon.

Since most programming languages don't have generators, programmers who come upon Icon with experience in other languages often overlook the possibilities generators offer — or, in an attempt to understand them in terms of what they already know, misunderstand them.

Articles about generators and sequences have appeared in past issues the Analyst [1-4], and they've been used in central ways in articles about applications such as numerical carpets [5-6]. And generators have appeared in some context in almost every issue of the Analyst. Nevertheless, generators and sequences have not been given the coverage they deserve. This article is the first in a series in which we plan to explore generators and sequences in depth. We're starting with this article, which reviews Icon's built-in generator repertoire. Most of the material that follows is elementary and designed to provide a foundation for future articles.

Element Generation

Element generation, !x, is one of the most versatile operations in Icon's repertoire.

!x applies to any value that has "elements". This covers much territory — strings, lists, records, sets, tables, and files. In addition, the element generation operation can be applied to any value that can be converted to a string, such as a cset.

Strings, lists, and files are sequences of values and generation from them is from the first element to the last. Note that element generation treats files as sequences of lines in the fashion of read(). Records can be viewed as a sequence of values in the order their fields are given in the declaration.

The elements of sets and tables have no inherent order, but their values can be generated, which imposes an order of sorts on them. As noted in the quiz answers on page 19, the order is not

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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

random, but it is unpredictable and depends on the inner workings of the implementation [7,8].

The generation of values from a table is further complicated by the fact that a table element consists of a pair of values, a key and it's corresponding value (for which we have no better name).

!T generates the values corresponding to the keys, not the keys themselves. The function key(T) generates the keys. There probably is a better way to design element generation for tables, but it's far too late now to change anything in the main computational repertoire of Icon.

Integer Generators

There are two ways to generate integers by fixed increments in increasing or decreasing numerical order. The more familiar is

```
i to k by j
```

The function seq(i, j) does the same thing, but with no limit.

String-Analysis Generators

Three string-analysis functions generate the positions of substrings within strings: bal(), find(), and upto().

These functions produce integers in increasing numerical order. The number of values these functions can produce is, of course, limited.

Keywords

There are five keywords that are generators:

```
&features
&regions
&storage
&allocated
&collections
```

The values generated by &features depend on the features that are available in a particular implementation, such as co-expressions and graphics.

&regions and &storage generate three values related to storage utilization, while &allocated and &collections generate four values related to storage management.

Miscellaneous

function() generates the names of the built-in functions. The values depend on the platform and the features supported.

There is one generator in Icon's graphics repertoire: Pixel(). It generates the color values of the pixels in a specified rectangular area. The values are either strings — comma-separated RGB triples — or for mutable colors, small negative integers.

Control Structures

There are two control structures for composing generators from expressions: alternation and repeated alternation. These control structures are not generators in and of themselves. Instead they evaluate expressions in ways that result in generation.

Alternation, *e1* | *e2*, is one of Icon's most useful control structures. It produces the sequence of *e1* followed by the sequence for *e2*. Note that if *e1* and *e2* are not generators but produce only one value each, their alternation is a generator (that produces two values).

Repeated alternation, |*e*, is rarely used and its potential is easily overlooked. |*e* repeatedly produces the sequence produced by *e*. It can be thought of as

e | *e* | *e* | ...

However, repeated alternation treats one case specially. If *e* produces no result (fails), |*e* fails. This handling of failure is designed to prevent repeated alternation from going on endlessly without ever producing a result — going over a sort of evaluation event horizon.

It's important to realize that an expression in repeated alternation may produce values for a while and then fail, terminating the repeated alternation. For example, |read(f) generates lines from f but terminates when the end of the file is reached and read(f) fails.

Perhaps the most important use of repeated alternation is to create a generator from a non-generator, as in the example above. Another example is |?0, which generates an unending sequence of real numbers in the range 0.0 to 1.0.

The Rationale for Generators

Why are some operations generators and others not?

The general design philosophy for Icon was to make an operation a generator only if there was a good reason. This was partly a conservative attitude toward language design, and it was not motivated by other considerations, such as pos-

sible implementation problems.

The best reason to cast a computation as a generator is when the computation naturally produces a sequence of values *and* it's necessary to maintain an internal state to get from one value to the next. This is the case for element generation, integer generation, and the string-analysis functions mentioned earlier. Note that where the number of possible values is limited, a list containing all of them is an alternative formulation. This, of course, does not work for generators with a potentially unlimited number of results, such as seq() and |e. Another advantage of generators is that values are only produced as needed — a form of lazy evaluation. And, in practice, it's often the case that not all of the possible values are needed.

There are two ways to cast a computation that does not meet these criteria as a generator. One is to generalize it so that it does. The other is to just have the computation done repeatedly.

An example of the first way is the string-analysis function match(s1, s2), which succeeds if s1 is an initial substring of s2 and returns the position i in s2 after s1. In the initial design of Icon, we considered generalizing match() so that if resumed, it would generate i – 1, i – 2, … 1. We tried this with disastrous results — straightforward string analysis produced unexpected results and it was necessary to use limitation to prevent them. The lesson here is that we did not have a good reason for this generalization — nothing clearly useful to offset the complications.

Casting a computation as a generator simply by having it do the same operation repeatedly can have unexpected consequences. Consider this example, which takes different actions depending on whether or not a value read is "end".

```
if read() == "end" then … else …
```

If read() were a generator but the value read not "end", read() would be resumed to read another line. If no line was "end", the entire file would be consumed, which probably would not be the intended effect.

The limitation control structure could be used to avoid this, but if there were many such generators, programs would have to be littered with limitations to assure there was no unexpected generation.

You might argue that programmers shouldn't write the kind of code shown above. But a language designed with reliance on good programming practice would not get very far.

The other side of the coin is that it is easy enough to make any non-generator into a generator using repeated alternation.

Some of Icon's operations were cast as generators for reasons other than those stated above. In the case of the keywords and function(), it was easier to implement them as generators than to, say, produce lists of values. This probably was a mistake from a design viewpoint, but at least these generators are needed infrequently.

Pixel(x, y, w, h) is a generator for entirely different reasons. The operation could be formulated (and originally was) to return the color value of a single pixel. In this formulation, Pixel() would have to be called many times for a large rectangular area. But that was not the problem. In a client-server environment, each call of Pixel() produces a request to the server. With a slow communication link, performance could be intolerably bad. As it is cast, one call of Pixel() requires only one server request, which produces all the color values in the rectangle, which then are generated.

References

1. "Generators" Icon Analyst 3, pp. 8-10.

2. "Result Sequences" Icon Analyst 7, pp. 5-8.

3. "Programming Tips (element generation)" Icon Analyst 8, p. 12.

4. "Programming Tips (recursive generators)", Icon Analyst 13, pp. 10-12.

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

6. "Exploring Carpet Space", Icon Analyst 47, pp. 5-10.

7. "The Design and Implementation of Dynamic Hashing for Sets and Tables in Icon", William G. Griswold and Gregg M. Townsend, *Software — Practice & Experience*, Vol. 23, No. 4, April 1993, pp. 351-367.

8. *Supplementary Information for the Implementation of Version 8 of Icon*, Ralph E. Griswold, Icon Project Document 112b, The University of Arizona, April 7, 1996.

Answers to Quiz on Structures

See Icon Analyst 52 for the quiz questions.

1. False. L[0] is an attempt to reference an element beyond the end of L, which fails.

2. True.

3. They both produce the same results, but the first leaves L unchanged, while the second removes all its elements, leaving it empty.

4. It adds a null-valued element to the right end of L.

5. They all leave L unchanged, although some differ in intermediate steps.

6. L := sort(set(L)).

7. True.

8. False.

9. False.

10. False. If R has a field named center, they produce the same results, but if R doesn't have a field named center, R.center causes a run-time error, while R["center"] fails.

11. True.

12. True.

13. False.

14. False; the order is unpredictable but not random.

15. *sort(S).

16.

```
procedure elim_str(S)
   every x := !S do
      if type(x) == "string" then delete(S, x)
   return S
end
```

17. set(R), set(S), set(T), and list(S) cause run-time errors. table(S) creates a table whose default value is S.

18. True.

19. False in general.

20.

```
procedure keyset(T)
   S := set()
   every insert(S, key(T))
   return S
end
```

21. Values can be obtained from keys, as in T[key(T)]. The converse is not true.

22. True, although this is a property of the implementation and not specified in the language.

23. False. Although there is no language or implementation limit on the number of elements in a table, it must fit in memory, which is limited.

Quiz — Expression Evaluation

1. True or false: A repeat loop only can be terminated by evaluating a break expression within it.

2. True or false: In

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

   if *e1* is a generator and produces a result but *e2* fails, *e1* is resumed.

3. True or false: A case expression can fail.

4. True or false: A case expression can generate a sequence of values.

5. True or false: A break expression can generate a sequence of values.

6. What does the following expression do?

```
i := 1 to 5
```

7. True or false: The number of results that *e1* | *e2* can generate is the sum of the number of results

that *e1* and *e2* can generate separately.

8. True or false!x can produce an unlimited number of results.

9. True or false|?x always produces an unlimited number of results.

10. What do the following expressions do?

```
write(every (1 to 5) & 7)
every write((1 to 5) & 7)
```


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

11. What do the following expressions do?

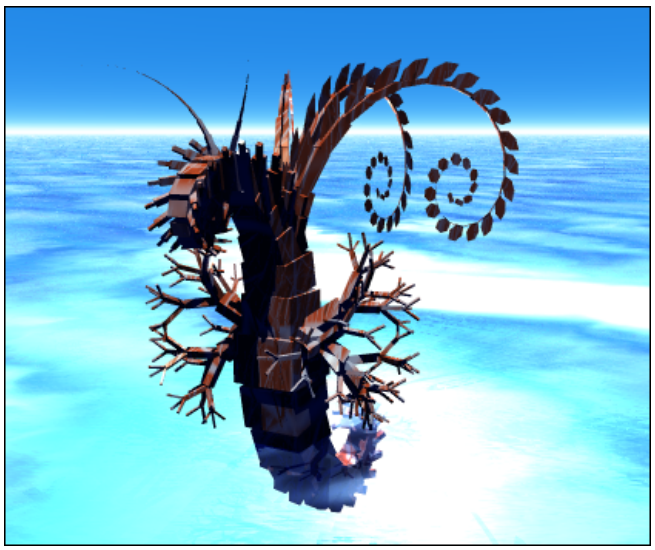
```
every write((1 to 5) | (5 to 1 by – 1))
every write((1 to 5) + (5 to 1 by – 1))
every write((1 to 5) > (5 to 1 by – 1))
```

12. What do the following expressions do?

```
every write(seq() \ 9)
every write((1 to 20) \ 9)
every write(seq() \ (9 | 3))
every write(seq() \ (1 to 5))
every write((seq() \ 9) \ 7)
every write((seq() \ ((1 to 3) | (3 to 1 by –1))))
```

13. What are the consequences of evaluating the following expressions?

```
|(2 = 3)
|2 = 3
2 = |3
|2 = |3
```



What's Coming Up

Machinery is aggressive. The weaver becomes a web, the machinist a machine. If you do not use the tools, they use you. — Ralph Waldo Emerson

In the next issue of the Analyst, we plan to have a weaving case study, an application for discovering interesting tiles in images, an article on animation by image replacement, and another article on generators and sequences.

We'll also have answers to the quiz on expression evaluation and another quiz, probably on Icon's preprocessor facility.