The Icon Analyst

In-Depth Coverage of the Icon Programming Language

June 1994
Number 24

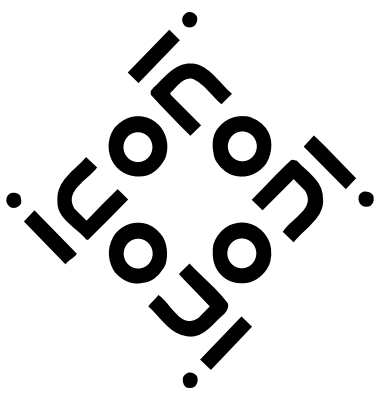
In this issue ...

Handling Images in X-Icon

Editors' Note: This is the last of a series of articles on graphics in Icon. Like previous articles, this one describes the facilities found in Version 8.10 of Icon. The next version of Icon will substantially enhance Icon's graphics facilities and change the way some things are done. You'll probably first hear about these changes in the Icon Newsletter. Later on, we plan to have a few articles in the Analyst on some of the new graphics features.

All or a portion of the contents of a window can be saved in an image file. Conversely, image files can be read into a window.

Version 8.10 of Icon supports two image formats: XBM and XPM. XBM is a black-and-white *bitmap* format that's supported by X. XPM is a color *pixmap* format. XPM isn't yet officially supported by X, but it's widely used and may be supported in the future. By convention, XBM files are identified by the suffix .xbm, and XPM files by .xpm. Although XPM can be used for black-and-white images, XPM files are larger than their XBM counterparts.



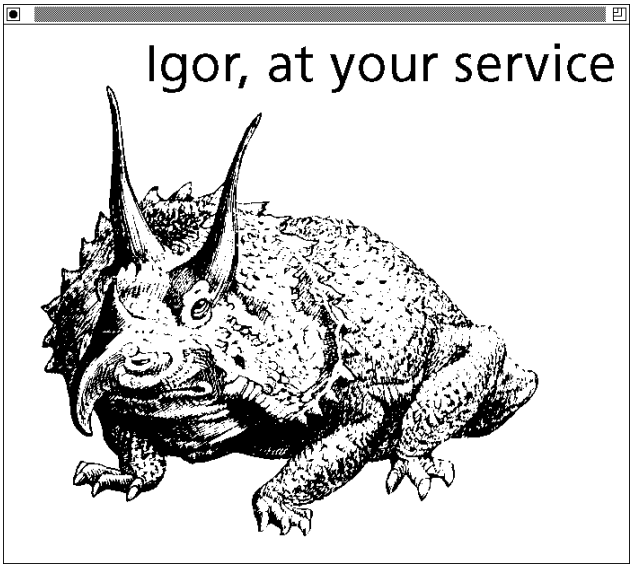
The next version of Icon will support CompuServe Graphics Image Format (GIF). GIF is widely used in many different computing environments, so this addition will make it much easier to use images in Icon.

There are many other file formats for pixel-based images in common use, such as TIFF [1]. Utility programs can be used to convert between different formats.

An image can be loaded into a window when it's opened by using the image attribute and a file name, as in

```
&window := open("igor", "x",
  "image=igor.xbm")
```

which might produce a start-up window such as this:



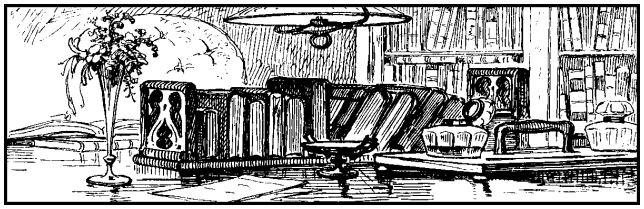
Another possible use for image files is customized entry forms, as in

Alternatively, an image can be read into a window by XReadImage(s, x, y), which reads the image named s into &window. The upper-left corner of the image is placed at x and y, which default to 0. Any portion of the image that does not fit into the window is discarded. XReadImage() fails if the image file cannot be opened or is not in a supported format.

The function XWriteImage(s, x, y, w, h) writes the contents of &window to the file named s, starting at x and y and extending by w and h. x and y default to 0 and if w or h is omitted, the extent is to the edge of the window in that direction. Thus, XWriteImage(s) writes the entire contents of &window to the file named s. If s ends in .xpm, the image file is written in XPM format. For all other file names, the image is written in XBM format and the color information is lost (all pixels that are not white are written as black).

Reference

1. *Graphic File Formats*, David C. Kay and John R. Levine, TAB Books, 1992.



From the Library

This is the first of a series of articles that feature the Icon program library. In this article, we'll start by giving a general description of the library for those of you who may not be familiar with it. Then we'll give you some tips on finding things in the library. In each of these articles, we'll describe a program or procedure from the library that is particularly useful.

Background

The Icon program library has a history that is nearly as long as Icon itself. The library started as a way for us to keep track of useful programs and procedures for our own use. We soon realized that other Icon programmers would benefit from the library, and it was first released publicly in 1983. There have been several revisions to the library since then and in 1990 we started a subscription update service that allows Icon programmers to get new library material several times a year.

At the time we started the library, we didn't anticipate how large it would become. What started as a handful of programs and procedures written by three or four local Icon programmers has grown to 230 complete programs and 1224 procedures designed for use in other programs — the work of 45 different persons. We've also added useful data, supporting documentation, and "packages" that are sufficiently complex they require special treatment. With everything accounted for, the library presently weighs in at nearly 5.4MB. And it's constantly growing.

We originally planned to prepare all the code for the library in a consistent style and typographical format. To help maintain the library, each file was given a standardized header and we established guidelines for documentation contained in the file. An example is shown in Figure 1 on the next page. The original header format survived with only minor changes, but it soon became impractical for us to adapt user-contributed code to a common typographical format, much less a consis-

```
############################################################
#
#       File:       post.icn
#
#       Subject:    Program to post news
#
#       Author:     Ronald Florence
#
#       Date:       October 2, 1991
#
############################################################
#
# This program posts a news article to Usenet.  Given an optional
# argument of the name of a file containing a news article, or an
# argument of "–" and a news article via stdin, post creates a
# follow-up article, with an attribution and quoted text.  The
# newsgroups, subject, distribution, follow-up, and quote–prefix can
# optionally be specified on the command line.
#
#    usage: post [options] [article | –]
#       –n newsgroups
#       –s subject
#       –d distribution
#       –f followup–to
#       –p quote–prefix (default " >")
#
# See the site & system configuration options below.  On systems
# posting via inews, post validates newsgroups and distributions in
# the "active" and "distributions" files in the news library directory.
#
############################################################
#
# Links:    options
#
############################################################
#
# Bugs:     Newsgroup validation assumes the "active" file is sorted.
#           Non-UNIX sites need hardcoded system information.
#
############################################################
```

Figure 1. Typical Library File Header

tent programming style. The multitude of coding styles currently represented in the library probably is a good idea — it allows users to see how different persons write Icon code.

We have had several problems with maintaining the library as it has grown besides those associated with its sheer size. Although many Icon programs will run on any platform, some programs inevitably require platform-specific features, such as pipes. Since we don't have the resources to customize the library for a variety of different platforms, we settled for including platform-specific programs along with the rest.

When we added graphics facilities to Icon, this problem took on an entirely different character. On one hand, graphics facilities are only supported on a few of the platforms on which Icon runs; at the present time a minority of Icon programmers can used its graphics facilities. On the other hand, the number of programs and procedures that use graphics is large and growing. At present, the graphics portion of the library comprises nearly half its bulk and it certainly will be in the majority in a year.

It would be easiest for us to combine the graphics and non-graphics portions of the library in the manner we've done for other platform-specific components, but it seems unfair to burden the majority of library users with a large amount of material they can't use.

At present we're living with a somewhat unhappy compromise in which the part of the library that requires graphics facilities is segregated from the rest. This allows persons who can't use the graphics portion of the library to discard it easily. So far, we've had no complaints about this approach.

The Library Hierarchy

The Icon program library is organized in a hierarchy as shown in Figure 2 on the next page. The directory names reflect their content:

data	data
docs	documentation
packs	packages
procs	procedures
progs	complete programs

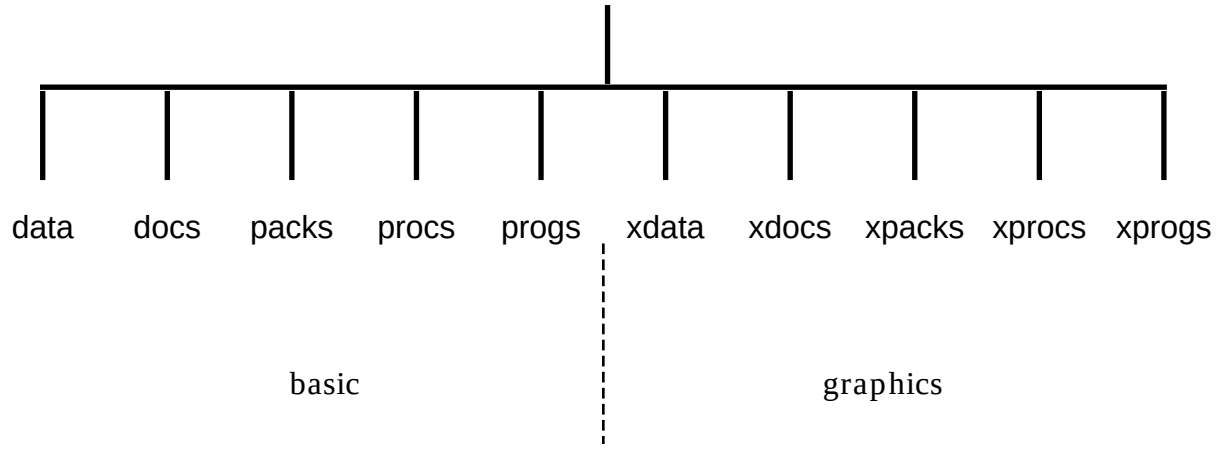


Figure 2. Icon Program Library Hierarchy

There are subdirectories in packs but not in the other directories. The x prefix on directories in the right half of the tree identifies "X-Icon" material that relates to graphics. We're in the process of "de-Xing" our terminology now that Icon's graphics facilities are no longer tied to the X Window System. You may see the xs become gs some time soon.

Finding Things in the Library

Suppose you need a program to count the number of instances of each different character in a file or a procedure to compute binomial coefficients. Programming these kinds of things in Icon is sufficiently easy that you may decide to write your own. But both the program and the procedure mentioned above are in the library, tested, and have useful features that you might not think of if you were to write your own. We won't bore you with a sermon on the virtues of reusable code, but if you're like us, you'd prefer to use an existing program rather than writing one your own — if you could find an existing one. That's the problem — finding things in the library.

This is a long-standing problem, and one that has become steadily worse as the library has grown in size. In early versions of the library, when there were only a few files, you could guess from file names or scroll through likely code on the screen — or even print the whole library and leaf through it. File names are problematical. Even with the best effort at being descriptive, it's hard to get much useful information with only the eight-character base name limit that cross-platform file transfer imposes. Even selected scrolling is impractical with the present library, and printing it all is out of the question for most of us, if for no other reason than because of environmental concerns.

There are, however, ways to find things in the library or least narrow the search. The obvious starting point is the listing of files with brief descriptions in the manual that accompanies the library. With a couple of good guesses, you probably quickly can find the program and procedure we mentioned earlier. From there, a quick glance at the documentation at the beginning of the two files will tell you if they are what you want.

Keyword-in-context listings provide permuted indexes of the library's content that are easier to use than the listing in the manual. There is a listing for programs, progs.kwc, and one for procedures, procs.kwc. Both are in the docs directory. There are corresponding listings for the graphics part of the library. See Figure 3 on the next page for an example of the contents of a permuted index.

These permuted indexes are made up from

```
                                .
                                .
                                .
icalls.icn                tabulate Icon  calls
findstr.icn          find embedded       character strings
ruler.icn                 write a        character ruler
xtable.icn                   show        character code translations
delamc.icn  delaminate file using tab    characters
fileprnt.icn                 display     characters in file
tablc.icn                  tabulate      characters in a file
adlcheck.icn                             check for bad address list data
sing.icn       sing The Twelve Days of   Christmas
                                .
                                .
                                .
```

Figure 3. Portion of a Permuted Index

the subject lines of the file headers. As such, they are only as good as the header lines that authors provide. Nonetheless, these permuted indexes often let you quickly tell if the library has something you want — or doesn't.

What you'd really like, no doubt, is a highly intelligent on-line help facility with an expert knowledge of everything in the library. We don't have that, but there is an interactive browsing program for the library.

Browsing in the Library

A program, called ibrow, written by Bob Alexander, makes browsing in the library easy. You'll see Bob's name often in these articles on the library; he's made some of the best-written and most useful contributions to it.

The program ibrow requires UNIX, both for pipes and because it is capable of invoking UNIX utilities. The program could, however, be adapted to other platforms (hint, hint).

If ibrow is called without any arguments, it processes all the Icon files in the current directory. Alternatively, the files to be processed can be given on the command line, as in

```
ibrow complex.icn rational.icn
```

The first thing ibrow does is provide a numbered, alphabetical listing of all the procedures and record declarations in the programs. The names of procedures are followed by parentheses, while the names of records are followed by periods. ibrow then presents the user with options for further processing, as shown in Figure 4.

As you'd expect, typing q followed by a return at this point ends the session with ibrow. The characters nn refer to the numbers given in the declaration listing. Typing a number optionally followed by f and then a return shows the name of file containing the declaration and gives the first line of the declaration. Following the number by m displays the entire declaration using the UNIX utility *more*. The letters e and v invoke the *ex* and *vi* editors, respectively, positioned at the beginning of the declaration.

From there, you're on your own.

Next Time

In the next article on the Icon program library, we'll feature a procedure that produces a very nicely formatted image of any Icon data object. It's especially useful for structures — lists, sets, tables, and records. It can even handle circular references.

Not only is this program very useful, but the code itself is instructive, so we'll make it the subject of an article in our **Program Anatomies** series.

```
Icon Browser -- scanning files:
  complex.icn
  rational.icn

 1. addrat()    5. cpxmul()    9. mpyrat()    13. reciprat()
 2. complex.    6. cpxstr()    10. negrat()   14. str2rat()
 3. cpxadd()    7. cpxsub()    11. rat2str()  15. strcpx()
 4. cpxdiv()    8. divrat()    12. rational.  16. subrat()

q,nn,nn[fmev],<return> (? for help):
```

Figure 4. Typical Browsing Session

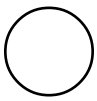
Turtle Graphics

Turtle geometry [1, 2] was developed as a way of introducing children to mathematics by presenting geometry in an interesting metaphor that encourages personal involvement.

Geometry can be viewed in many ways. Traditionally, it's taught first in the axiomatic style of Euclid and later developed in the algebraic style of Descartes.

Human beings have excellent spacial intuition for geometry, presumably because of their highly developed visual system and its value for survival in evolutionary competition.

Consider something as simple as a circle [3]:



We all understand that. But what about the algebraic relationship

$$x^2 + y^2 = r^2$$

that describes the coordinates of a circle in terms of its radius. If you didn't already know what a circle was, would this algebraic description give you an intuitive notion of what a circle is?

The parametric equations for a circle,

$$x := r \cos(\theta) \qquad 0 \le \theta \le 2\pi$$
$$y := r \sin(\theta)$$

suggest a way of plotting a circle, but they do not appeal to our geometric intuition either.

Verbal descriptions of a circle don't help that much either. If you go to textbooks, you'll find definitions like these: "A circle is the locus of points in a plane that are equidistant from a given point called the center" [4] and "A circle is a plane continuous curve all of whose points are equidistant from a fixed coplanar point" [5] — definitions only a mathematician could love.

Turtle geometry takes a navigational approach to geometry. A circle can be described in terms of actions a child can perform. To trace out a circle, move forward a little, turn right a little, move forward a little, turn right a little, and so on. With this, a child can understand what a circle is all about. Granted, there are problems with this formulation. One is knowing when to stop. Another is that a circle constructed in this way isn't perfect, even if all the moves and turns could be made precisely the same . But perfect circles exist only as mathematical abstractions, and the insight gained by creating a circle this way not only is educational for children, but it also encourages them to explore variations and other geometrical figures.

In turtle geometry, as the name suggests, a turtle is an agent for navigation. It moves in response to instructions like "move forward a step". For our purposes, the turtle is conceptual, although robot turtles have been used in educational experiments. In fact, the idea of using a turtle, which is a captivating focus for children, originated in a very early robot tortoise called *Machina speculatrix* that was constructed by W. Grey Walter [6].

Thus, a child gives the turtle instructions and the turtle moves as a result, perhaps tracing out an approximation of a circle. Usually, of course, this is done with a computer and the navigational space is the computer screen.

Turtle graphics comes by adding a drawing capability, whereby the turtle may draw a line as it moves, producing figures as the result of its travels.

So far, we've been talking about teaching children about geometry. The ideas apply to adults also, although they usually have so much "baggage" from previous experience that the turtle metaphor is not as captivating as it is for children. The navigational model of turtle graphics nonetheless provides an easy and natural way of producing many kinds of interesting drawings. That's what we'll explore in this article.

Turtle graphics originally appeared in the programming language Logo [7], which has many additional capabilities in addition to graphics. Turtle graphics, however, can be implemented in almost any imperative programming language that has graphics capabilities. In the case of Icon, turtle graphics have been implemented by a package of procedures that are included in the Icon program library.

Some aspects of turtle graphics are fundamental and are supported in essentially the same way in all implementations. Other aspects can be cast in various ways, depending on the capabilities desired, the amount of effort invested in the implementation, and the taste of the designer. As a result, all implementations of turtle graphics share a common foundation, but no two are identical (as far as we can tell).

At any time, a turtle is at a specific location on a surface, usually a plane with limited extent. The

turtle also has a heading — the direction it faces. The turtle moves or changes its heading in response to commands. When it moves, it may draw a line, or it may not, depending on the command. Beyond that, it's a question of what commands are available.

Some turtle graphics systems support multiple turtles, color drawing, and even fancy features that have little to do with the original concepts. Icon's version of turtle graphics is comparatively simple and straightforward. There is only a single turtle and there are no commands related to color.

Turtle procedures provide the commands that manipulate the turtle and its drawing. Icon programs that use these procedures of course have all the rest of the capabilities of Icon available, such as changing the color of drawing.

The next section describes the most important aspects of Icon's turtle graphics. There are other facilities in addition to those described here; see turtle.icn in the Icon program library for a complete description.

Icon Turtle Graphics

The turtle always draws on &window. If &window is null when the first turtle procedure is called, a 500×500 window is opened and assigned to &window.

The turtle, which is invisible, initially is in the center of the window, facing toward the top.

Angles are measured in degrees. The positive direction is clockwise. Although specific angles rarely occur in turtle graphics programs, it may be useful to know that 0° is in the positive x direction, so that the turtle initially faces -90°.

Distances are measured in pixels but are multiplied by a scaling factor that defaults to 1 but can be set to another value.

The primary turtle graphics procedures are:

TDraw(n) moves the turtle forward n units in the direction it's facing, drawing a line from where it was to where it winds up. The value can be negative to move and draw backwards. TSkip(n) is like TDraw(n) except that the turtle does not draw a line.

TDrawto(x, y) turns the turtle toward the location (x,y) and moves the turtle there while drawing a line. TGoto(x, y) moves the turtle to (x,y) without drawing a line or changing its heading.

TLeft(d) turns the turtle left d degrees. Its location is not changed and nothing is drawn. TRight(d) is like TLeft(), except the turtle is turned to the right.

TFace(x, y) turns the turtle to face toward (x,y). Nothing is drawn.

TScale(n) *multiplies* the scaling factor by n. The turtle location and heading are not changed and nothing is drawn.

THome() returns the turtle to its original position at the center to the window, facing upward. Nothing is drawn.

TReset() clears the window and starts over.

TSave() saves the state of the turtle. The state consists of the turtle's location, its heading, and the current scaling factor. TRestore() restores the state of the turtle.

The last two procedures greatly increase the power of turtle graphics. It's worth noting, however, that these procedures go well beyond the original framework of turtle geometry. Human beings, much less turtles, have very limited capacity for remembering and recalling levels of information like this. A computer, of course, has no trouble with this, and it's trivial to implement in Icon.

Drawing with Turtle Graphics

The capabilities of turtle graphics suggest the kinds of drawings for which it's best suited: Those that can be easily cast in terms of simple movements and angle changes.

Consider a "random walk" in which the direction in which the turtle moves is chosen at random:

```
repeat {
  TDraw(1)
  TRight(?30 + 1 – 15)
  }
```

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

FTP: cs.arizona.edu (cd /icon)

The turtle moves forward and draws for one unit. It then turns right an amount in the range –15° and +15° and repeats. This goes on until the program is interrupted. An example of the result is shown in Figure 1.

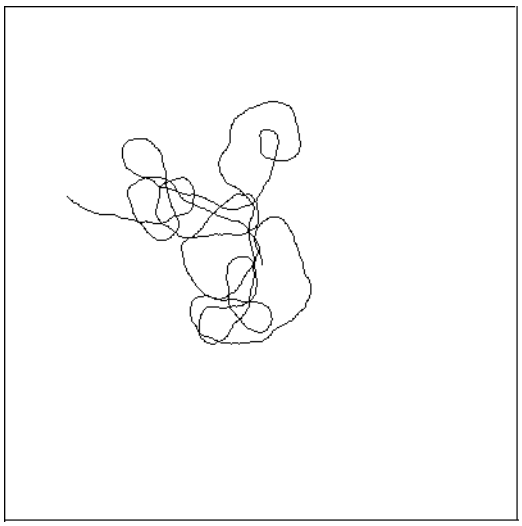


Figure 1. A Random Walk

The amount of movement and the angular range used here are rather arbitrary; it's easy to make a more general procedure in which the amount of movement also is chosen at random and different parameters can be specified.

The complexity and variety of figures that can be produced by simple rules is illustrated by these few lines of code that draw "spirals":

```
angle := 30 + ?149
incr := sqrt(4 * ?0) + 0.3
side := 0

while side < 500 do {
  TDraw(side +:= incr)
  TRight(angle)
  }
```

Here both the angle of drawing and the increment of movement and drawing have random components. Some examples of the results are shown on the next page.

The value of being able to save and restore the state of the turtle is illustrated by the following procedure, which draws a random "bush":

```
procedure bush(n, len)
  TSave()
  TRight(?70 + 1 – 35)
  TDraw(?len)
  if n > 0 then {
    every 1 to ?4 do {
      bush(n – 1, len)
      }
    }
  TRestore()
end
```

This procedure might be used as follows:

```
TSkip(–120)                  # position root
bush(n := 4 + ?4, 300 / n)
```

An example of the results is shown in Figure 2.

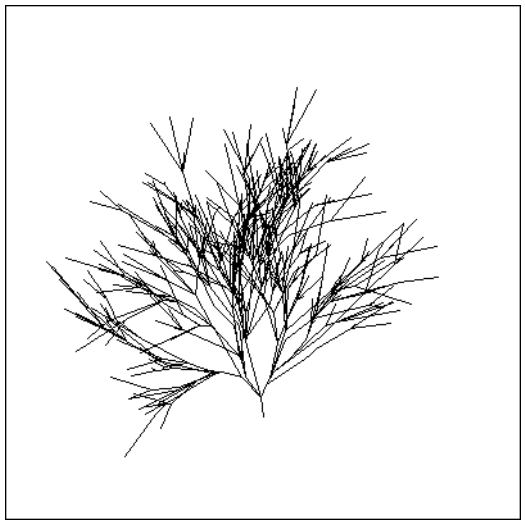


Figure 2. A Bush

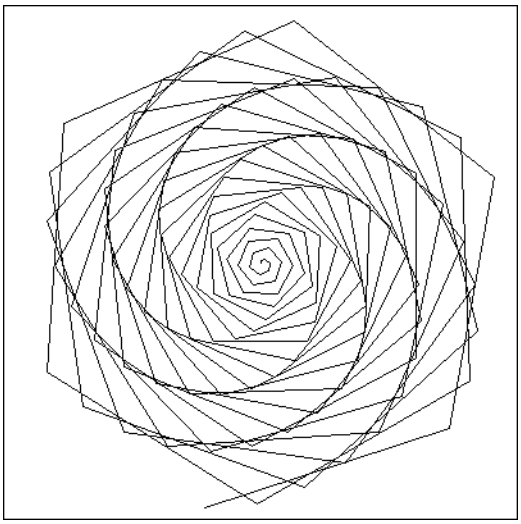
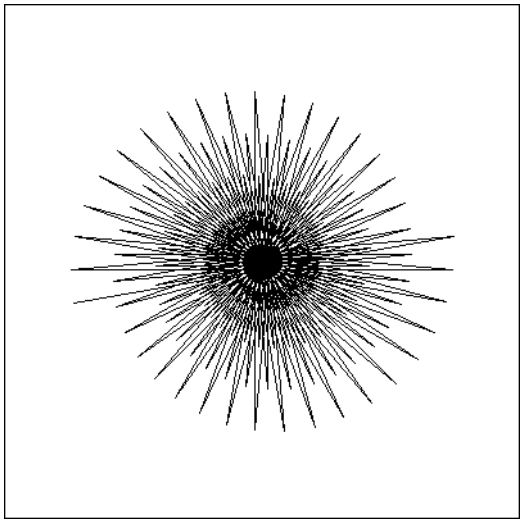
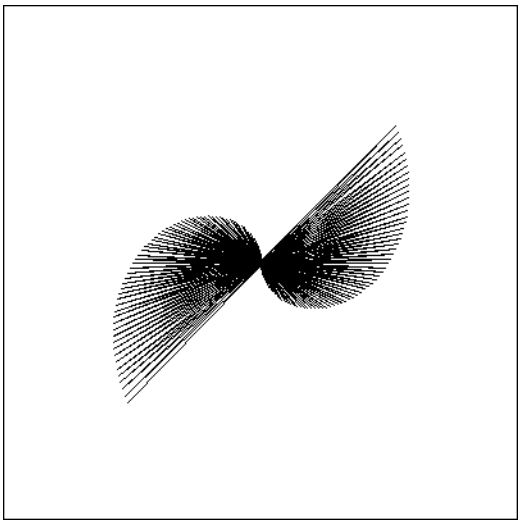
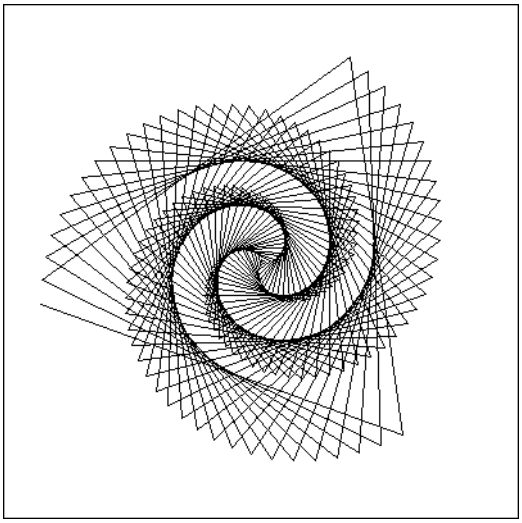
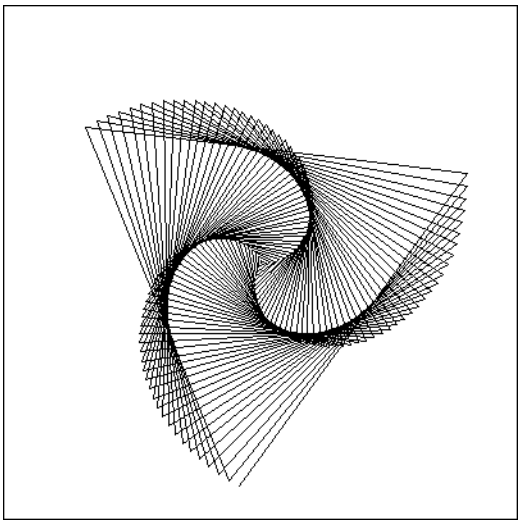
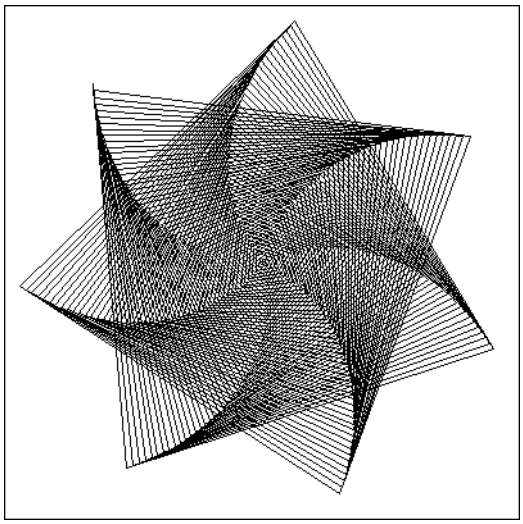
Conclusions

Figures of this kind can be drawn using coordinate computations and Icon's built-in drawing functions. That is, of course, how the turtle procedures are implemented. The advantages of turtle graphics lie in its conceptual framework and navigational metaphor.

In the next issue of the Analyst, we'll describe another situation in which turtle graphics are particularly apt.

Acknowledgment

The turtle graphics package in the Icon program library was designed and implemented by Gregg Townsend. The examples given in this articles are based on ones contained in his demonstration package, tgdemo.icn.

“Spirals”

References

1. *Mindstorms; Children, Computers, and Powerful Ideas*, Seymour Papert, Basic Books, Inc., New York, New York, 1980.

2. *Turtle Geometry; The Computer as a Medium for Exploring Mathematics*, Harold Abelson and Adrea diSessa, The MIT Press, Cambridge, Massachusetts, 1980.

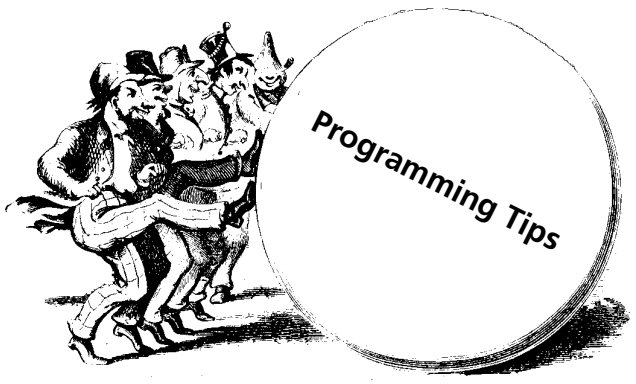
3. "Computer Recreations", Brian Hayes, *Scientific American*, February 1984, pp. 14-20.

4. *Brief Analytic Geometry*, second edition, Thomas E. Mason and Clifton T. Hazard, Ginn and Company, Boston, Massachusetts, 1947.

5. *A Handbook of Curves and Their Properties*, Robert C. Yates, J. W. Edwards, Ann Arbor, Michigan, 1947.

6. *The Living Brain*, W. Grey Walter, W. W. Norton & Co., Inc., New York, New York, 1953.

7. *Visual Modeling with Logo; A Structured Approach to Seeing*, James Clayson, The MIT Press, Cambridge, Massachusetts, 1988.



Assignment Expressions

In most imperative programming languages, assignment is a statement, not an expression. Hence assignment has no value associated with it and it cannot be used where expressions are expected. Icon, on the other hand, has no statements, only expressions, and assignment is just another kind of expression. It produces a value and can be used where expressions are expected.

If you learned to program in a language in which assignment is a statement, you may not have thought of how useful it can be to have assignment expressions.

Before going on, we'll review the properties of assignment expressions in Icon. An assignment is just an operation with two arguments. The first (left) argument is the variable to which the second (right) argument value is assigned. As in all binary operations in Icon, the left argument is evaluated before the right one.

Both arguments can be arbitrarily complex expressions as long as the left one produces a variable and the right one produces a value of a type that is appropriate for assignment to the variable. Although ordinary variables can be assigned any kind of value, keywords like &random can only be assigned integer values. A value assigned to &random therefore must be an integer or a value that can be converted to an integer. Similarly, &subject can only be assigned a string value.

The evaluation of either argument of assignment can, of course, fail. If an argument fails, no assignment is performed and the assignment expression fails.

Assignment groups from right to left (unlike most binary operations) and produces its left argument as a variable. Consequently,

```
i := j := 1
```

groups as

```
i := (j := 1)
```

and assigns 1 to both i and j.

Such multiple assignments are fairly common in Icon programs. They seem natural and may not even raise issues about statements versus expressions.

Augmented assignment, in which assignment is syntactically combined with another binary operation, is, of course, an expression also. Note, however, that

```
i := j +:= 1
```

does not increment both i and j; it increments j and assigns the resulting value to i.

It's fairly common to see augmented assignment in subscripting, as in

```
args[i +:= 1]
```

in which the value of i is incremented and the result is used to subscript args.

In most cases, the use of assignment within another expression is a convenience with minor benefits in terms of program brevity and some loss in readability. There are situations, however, when the use of assignment within other expressions has a substantial benefit.

Consider the following procedure for drawing a regular polygon with n sides in a circle of radius r, centered at xc and yc:

```
procedure polygon(n, r, xc, yc)
   incr := 2 * &pi / n
   angle := 0
   x := r * cos(angle)                 # just r
   y := r * sin(angle)                 # just 0
   every 1 to n do {
      angle +:= incr
      xnew := r * cos(angle)
      ynew := r * sin(angle)
      XDrawLine(xc + x, yc + y,
         xc + xnew, yc + ynew)
      x := xnew
      y := ynew
      }
   return
end
```

Successive lines are drawn from previously computed points to newly computed ones. The variables x and y hold the previously computed coordinates, while the new ones are in xnew and ynew.

Here's a case where the amount of code can be substantially reduced by using assignment at the places where new values are needed. The main loop can be recast as:

```
   every 1 to n do {
      angle +:= incr
      XDrawLine(xc + x, yc + y,
         xc + (x := r * cos(angle)),
         yc + (y := r * sin(angle)))
      }
```

Not only is the code shorter, but this form eliminates the need for two local variables to hold the new values.

Incidentally, care is needed when assignment is used to provide the values of arguments in a call. Arguments that are variables are not dereferenced until all argument expressions have been evaluated. That's not a problem in the example above, since the first argument expression, xc + x, uses the value of x before it is changed in the computation of the third argument, and similarly for the second and fourth arguments. If the expression had been

```
XDrawLine(x, y,
   x := r * cos(angle), y := r * sin(angle))
```

the new values of x and y computed in the third and fourth arguments would have been used for the first and second arguments also.

This problem can be avoided by dereferencing the first and second arguments, extracting their values before going on to evaluate the third and fourth argument expressions:

```
XDrawLine(.x, .y,
   x := r * cos(angle), y := r * sin(angle))
```


The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
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
Gould-Simpson Building
The University of Arizona
Tucson, Arizona 85721
U.S.A.

voice: (602) 621-8448

fax: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

...uunet!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

The Bright Forest Company
Tucson Arizona

A Word of Thanks

There's more to the production of a newsletter than you might realize if you haven't done it. As the editors, we do the writing, layout, text entry, copyediting, program testing, and so forth.

The Department of Computer Science here provides assistance with lots of "little" things – things like taking subscription renewals, helping with the mailing, and so on.

The most important help we get comes from Gregg Townsend, who reads every issue before we send it to be printed. He does an excellent job and with amazing quickness. Over the years he's caught many errors ranging from typographic mistakes to incorrect programs. And he's often made valuable suggestions for general improvements. We secretly hope that someday we'll produce an issue for his approval in which he can find nothing wrong. But that probably will never happen.

We just want to take this opportunity to express our sincere thanks to Gregg for all his help; help that benefits not just us but all our readers.

Reflections

It seems appropriate, as we plan to start the fifth year of the Analyst, to take stock and give some thought to where we are going.

When we initiated the Analyst, we had a general idea of things that we wanted to do and what prospective readers might want. Our original intention was to provide a mix of material ranging from tutorials for persons new to Icon to advanced material for the most experienced and technically oriented readers.

As the Analyst has evolved, we've tended toward more advanced material. This reflects, in part, what our readers have told us. We've also made an effort to put more elementary material in the *Icon Newsletter*, whose readership includes a many persons who are new to Icon. But the trend in the Analyst also reflects our own interests and research directions.

We've long known that documentation is an integral part of the process of doing research. Writing about a technical subject tends to bring out problems and expose gaps that are not otherwise apparent. And writing often leads to new ideas. The Analyst, with its relatively informal tone and (we hope) friendly readership, is a much more facile vehicle for this kind of interaction between technical work and its documentation than conventional forms of academic publication. So we benefit too.

Obviously, over a period of years, the nature of the material in a newsletter like this changes simply because some topics have been thoroughly covered while new topics come up that we'd not anticipated. For example, when we started the Analyst, we had no idea that there would be a major addition to Icon in the form of graphics facilities. With this issue, we've concluded a series on graphics. We've been somewhat concerned about how well this material would be received, since Icon presently supports graphics on only a few platforms. As a result, we've deliberately spaced out the articles so they wouldn't overwhelm readers with no capability for using graphics in Icon.

We'll continue to have articles from time to time on graphics, and if our present implementation efforts go well, more persons may be able to use graphics in the future. But we'll also try to maintain a focus on the "mainstream" of Icon programming.

What's Coming Up

We usually have a lot of material for the Analyst in the "pipeline". In fact, we sometimes have nearly a full year of issues complete, or nearly so, in advance. We're not quite that far ahead at the moment, but we have a pretty good idea of most of the articles for the next few issues. We have not, however, decided on order, so we can't be as specific about the next issue as we usually are.

But look for another article on the Icon program library and a related program anatomy, as well as an article on the use of turtle graphics for rendering figures described by Lindenmayer Systems. If that doesn't mean anything to you, stay tuned. And renew your subscription if you haven't already.