The Icon Analyst

In-Depth Coverage of the Icon Programming Language

October 1994
Number 26

In this issue …

Random Numbers … 1
Trivia Quiz … 3
Lindenmayer Systems … 4
Answers to the Trivia Quiz … 9
Cheap Tricks … 9
Programming Tips … 11
What's Coming Up … 12

Random Numbers

Introduction

This is the first of a series of articles on random numbers in Icon.

Almost every programming language has some way of producing random numbers. Random numbers are needed to model physical phenomena that have random properties, to simulate processes like demands for service, to provide unbiased test data, to pick alternative computational paths to test algorithms fairly, and in many games. We suspect random numbers are used most often in game programs.

When we talk about random numbers in a program, we're not talking about numbers produced in a truly random fashion — that takes a physical process, like the flip of a coin or the decay of an unstable atom. Instead, we're talking about pseudorandom numbers, which are produced in a deterministic fashion but that occur in a sequence that has the statistical properties of a truly random sequence. With that understanding, we'll drop the "pseudo" for the remainder of this article.

Random Selection in Icon

In Icon, the random selection operation ?x produces a randomly selected value that depends on the type of x. If x is a positive integer, ?x produces a randomly selected integer in the range 1 to x. If x is 0, it produces a randomly selected real number in the range $0.0 \leq r < 1.0$. If x is a string, it produces a randomly selected one-character substring. For structure types, ?x produces a randomly selected element of the structure.

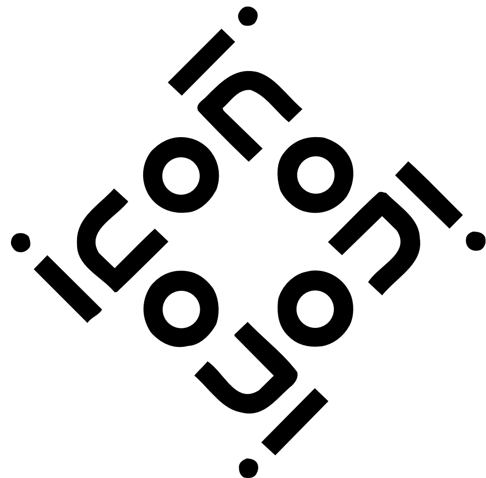
There are some details that are worth knowing. A real number is converted to an integer for the purposes of random selection, while a cset is converted to a string. In the case of a string that is given as a variable, ?x produces a variable and a value can be assigned to it. For example, if chars is a string,

```
?chars := "xxx"
```

replaces a randomly selected character of chars by xxx. Similarly, a value can be assigned to a randomly selected element of a list or record. For example, if grades is a list,

```
?grades := 0
```

sets a randomly selected element of grades to 0. Random selection for a table is similar; the value corresponding to a randomly selected key is set. A randomly selected value of a set can be obtained, but no variable is associated with an element of a



set, so a set cannot be changed by assignment to a randomly selected element of it.

That may be all you need to know about random numbers in Icon for most purposes. The rest of this article explores what is going on behind the scenes.

Random Sequences

It is important to understand that random selection is based on a sequence of random numbers that is produced by a single random number generator. Any random selection operation produces a new value in this sequence. This value then is used in the selection. The values for Icon are in the range of 0 to 2^{31}–1. This is the range of nonnegative integer values for most implementations of Icon. The range of integers is larger on 64-bit platforms, but it is never smaller, and the range of random numbers is the same for all implementations of Icon. This allows programs to be transported between platforms without affecting the behavior of random selection.

Icon's random number generator is capable of producing all the values in the range 0 to 2^{31}–1. At the end of a sequence it repeats. It's very unlikely that any program will go through all possible values: 2^{31}–1 is a very large number, 2,147,483,647 to be explicit. On a Sparc IPX, it would take weeks of steady computing to go through the entire sequence, even with a program that does nothing else.

The keyword &random is the current value in the random sequence. The initial value of &random is 0. The first few values in the sequence starting at 0 are:

```
         0
 453816694
 885666996
 678165018
1096161928
 905669982
 656467580
 170957890
1583830416
 108920774
1539632324
 295778538
 721762584
1144737966
1333202828
1237514258
```

You'll notice that the rightmost digits are anything but random. This is a characteristic of the "linear congruential" method Icon uses for random number generation. The leftmost digits, which are distributed more randomly, are used in selection.

&random sometimes is called the seed for random-number generation, since it determines where in the sequence subsequent random numbers start. For example, to get a sequence different from the default one, you can assign a value other than 0 to &random at the beginning of program execution. Since the sequence is so long, however, the effect is about the same as getting a different sequence. For example, 10 is the 708,384,987th value in the sequence. So if you start your program with

```
&random := 10
```

the effect is virtually the same as if you were able to use an entirely different random sequence. The first few values starting at 10 are:

```
        10
 751550904
1327774926
 760568108
 228004146
1783119808
1841059382
 440785524
2098334938
 641898056
 853581342
 504440892
2097486594
1241501008
 694516870
2094500484
```

Again, you'll notice that the rightmost digits are not random.

Since &random starts at 0, unless you change it, a program will produce the same results of random selection for the same data. This is, of course, anything but random behavior, but it's handy when you're developing and debugging a program.

Programs that need to exhibit random behavior can accomplish this in several ways. One is to allow the user to specify an initial value for &random as a command-line option. This allows the user to get repeatable results that nonetheless may be different from those for the default value of &random.

In other cases, you may want to set &random so that it's different every time a program is executed. Doing this is tricky; in fact, it's theoretically impossible, although it can be done so that the chance of the same value for different executions of a program is vanishingly small.

One method is to set &random from &clock, which is the time of day. This assures that you'll get different values unless different program executions start at the same second of the day.

The value of &clock is a string of the form "*hh*:*mm*:*ss*", where *hh* is the hour, *mm* the minute, and *ss* the second. Thus, the seed could be set by

```
&random := &clock[1:3] || &clock[4:6] ||
  &clock[7:9]
```

An alternative approach is string scanning, but there's another, if less obvious method:

```
&random :=
    map("HhMmSs", "Hh: Mm:Ss", &clock)
```

This use of map() to delete characters from a string is one of many unusual ways map() can be used. See Reference 1.

Using just &clock to set &random reduces the probability of the same value for different runs to about 0.0000116, assuming all times are equally probable. The probability is greater for typical use, but still very small. You can do much better by using randomize() in the Icon program library (link randomiz; the file name is truncated to meet the requirements of some operating systems). For your amusement, here's the procedure:

```
procedure randomize()
  static ncalls

  initial ncalls := 0

  ncalls +:= 1

  &random :=
    map("sSmMhH", "Hh:Mn:Ss", &clock) +
    map("YyXxMmDd", "YyXx/Mm/Dd",
    &date) + &time + 1009 * ncalls

  return

end
```

Notice, among other things, that the value this procedure produces depends on how many times it has been called.

Next Time

In the next article on random numbers, we'll look at methods by which random numbers can be generated, the particular method used by Icon, and how good it is.

You may have noticed that the values produced for &random starting at 0 and 10 are all even. This is not just a property of the linear congruential method Icon uses; it's a story in itself, which we'll also cover in the next article.

Acknowledgment

Gregg Townsend provided valuable technical assistance with this article. Bob Alexander wrote the original version of randomize(). Gregg Townsend contributed to the present version.

Reference

1. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, pp. 200-205.

Trivia Quiz

When giving tests in the courses we teach, we sometimes include "trivia" questions about obscure matters. We only assign a point or two of credit for such questions and assure students that their performance on such questions will not affect their course grades.

Despite these disclaimers, students often spend a considerable amount of time and effort on such questions. There is something about "trivia" that fascinates some persons. We put the word trivia in quotes, because many of the questions relate to potentially useful information. But, then, different persons have different ideas of what's trivial. No doubt, some of our students think all of the material in a course is trivial from the point of view of relevance to their interests and ambitions.

Assuming you're more interested in Icon than some of our students, we thought we'd put a "trivia" quiz in the Analyst. All the answers can be found in previons issues of the Analyst. The answers, with citations, are given on page 9. If you like this silliness, let us know and we'll do more.

1. How much allocated strorage does a record with three fields take?

2. How much allocated storage does the string literal "hello" take?

3. On IBM mainframes is the internal representation of characters in EBCDIC or ASCII?

4. Are find() and upto() matching functions?

5. How large is a list value compared to an integer value?

6. Is there any difference in the results of evaluating stop() and exit()?

7. What environment variable is used to locate files for the link declaration?

8. What does

```
suspend &fail
```

do?

9. Give an example of a run-time error that cannot be converted to failure by error conversion.

10. Give an example of expressions for which

expr1 +:= *expr2*

and

expr1 := *expr1* + *expr2*

are not equivalent.

11. When is it necessary to use a semicolon to separate expressions in Icon?

12. What is the difference between an expression and a statement in Icon?

13. What is RTL?

14. What is a thread in MT Icon?

15. How is Rebus implemented?

16. What is the attribute that specifies the distance bewteen the base lines of text written to a window?

17. What is a stream in Seque?

18. What does the following expression do?

```
&subject := &subject
```

19. Suppose the ith number in the Fibonacci sequence is computed using

```
procedure f(i)
  if i = (1 | 2) then return 1
  else return f(i – 1) + f(i – 2)
end
```

How many calls of f() does it take to compute the 10th Fibonacci number?

20. What units are used for angles in turtle graphics?

Anatomy of a Program — Lindenmayer Systems (continued)

In the last issue of the Analyst, we described Lindenmayer systems, which were invented to characterize the development of plants by strings of symbols and rewriting rules.

The original intent of L-systems was purely formal. Of course, to determine if a string of symbols describes a plant, it helps if you can “look at” what the string represents. Early work was done by hand, associating symbols with plant cells and deciding what a string represented in terms of a plant. A breakthrough came with the realization that the symbols could be interpreted as simple commands to draw pictures of plants.

Consider the following L-system:

X	axiom
X → F–[[X]+X]+F[+FX]–X	replacement rules
F → FF	

This is the same L-system given in the previous article, but with different symbols.

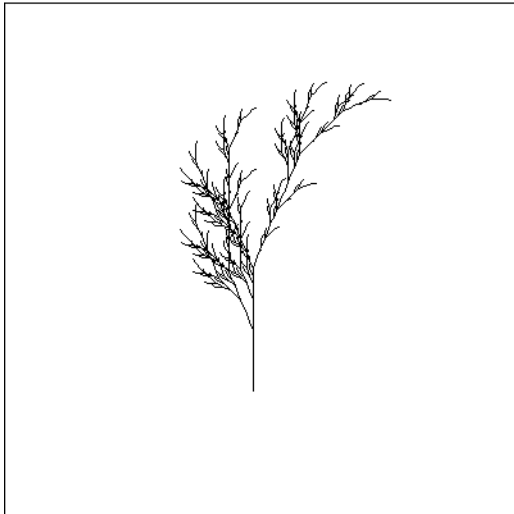
The choice of symbols is not arbitrary; instead the symbols have meanings when considered as drawing commands:

- F move forward a specified length, drawing a line
- f move forward a specified length without drawing a line
- \+ turn right a specified number of degrees
- – turn left a specified number of degrees
- [save the current position and direction
-] restore the previously saved position and direction

The symbol f is not used in the L-system above, but it is needed in some others. The symbol X in the L-system above is a marker that is ignored in drawing.

The specified length is constant for a drawing and determines its scale. The specified angle for turns also is a constant for a drawing, and it plays a fundamental role in the nature of the drawing.

All this is rather abstract. To make it concrete, here's what the L-system above produces for an angle of 22.5° at 5 generations:



By the way, it's a lot more fun to watch an L-system being drawn than it is to look at the final result.

You can imagine the excitement that resulted from converting a string of apparently meaningless symbols into a drawing that looks very much like a plant.

More insight into L-systems and the "plants" they produce can be obtained by comparing successive generations. The picture at the top of the next page shows the first six generations for the L-system above. The scale has been reduced and a base line has been provided to make it easier to compare the generations. Notice that the height and width of the plants double with each generation. You may also detect some fractal characteristics (self-similarities) in the successive generations.

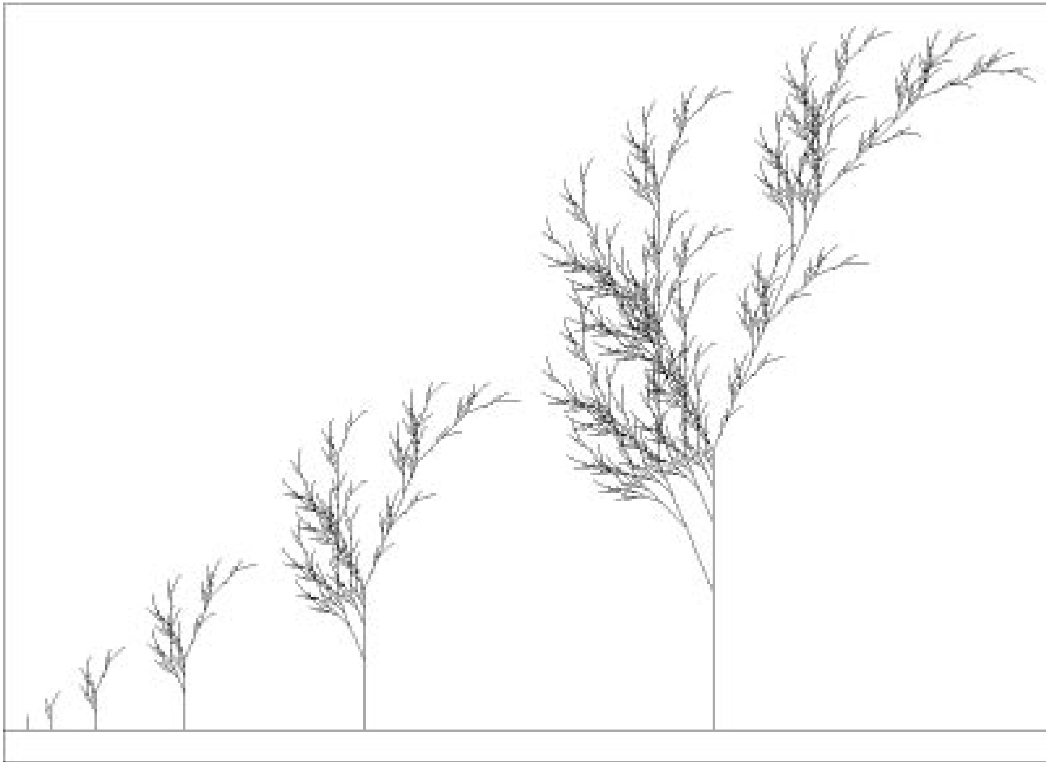
Implementation

The commands used to produce such a drawing are navigational in nature and have direct correspondences in turtle graphics [1]:

```
F    TDraw(length)
f    TSkip(length)
+    TRight(angle)
–    TLeft(angle)
[    TSave()
]    TRestore()
```

The program given in the previous article simply writes the symbols that result from rewriting. With the interpretation of symbols as drawing commands, it's easy to convert that program into one that draws. We need to add the length and angle to the L-system parameters and interpret the symbols using turtle procedures:

```
link turtle                # turtle graphics package

procedure main()
  local rule, line, sym, new, axiom, gener, angle
  local length, keyword, value, allsyms, replace

  rule := table()

  allsyms := ''            # initially empty cset

  while line := read() do
    line ? {
      if sym := tab(find("–>")) then {
        move(2)
        replace := tab(0)
        rule[sym] := replace
        allsyms ++:= replace
        }
      else if keyword := tab(find(":")) then {
        move(1)
        value := tab(0)
        case keyword of {
          "axiom":   {
            allsyms ++:= value
            axiom := value
            }
          "gener":   gener := value
          "angle":    angle := real(value) |
            stop("*** invalid angle: ", line)
          "length":   length := integer(value) |
            stop("*** invalid length: ", line)
          default:
            stop("*** invalid keyword: ", line)
          }
        }
      else stop("*** invalid specification: ", line)
      }

  if /axiom then stop("*** no axiom")

  /length := 5             # defaults
  /gener := 4
  /angle := 90.0

  every sym := !allsyms do
    /rule[sym] := sym
```



```
  every sym := lgen(!axiom, rule, gener) do
    case sym of {
      "F":  TDraw(length)
      "f":  TSkip(length)
      "+":  TRight(angle)
      "–":  TLeft(angle)
      "[":  TSave()
      "]":  TRestore()
      }

  Event()                # wait to dismiss window

end

procedure lgen(sym, rule, gener)

  if gener = 0 then return sym
  suspend lgen(!rule[sym], rule, gener – 1)

end
```

Note that symbols with no interpretation, such as X, are ignored.

That's all there is to it — a few additional lines of code, and you can draw all kinds of interesting "plants", as shown on page 7. As you can see, some of these pictures look more realistic than others. Some are plantlike, but look like they might have come from prehistoric times or another planet. You'll find more examples in several books on the subject [2-4].

Although L-systems were invented to model plant development, they can be used to generate many other kinds of figures, including some fractals and tilings, as shown on page 8.

The Icon program library contains not only a program, linden.icn, for writing and drawing L-systems, but a collection of L-systems that include the ones used to draw the pictures in this article. See linden.dat.

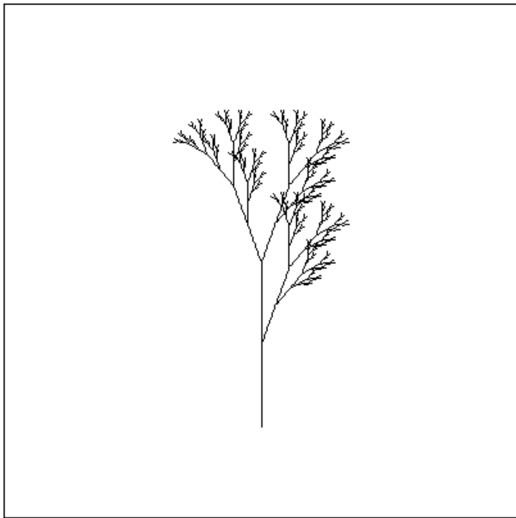
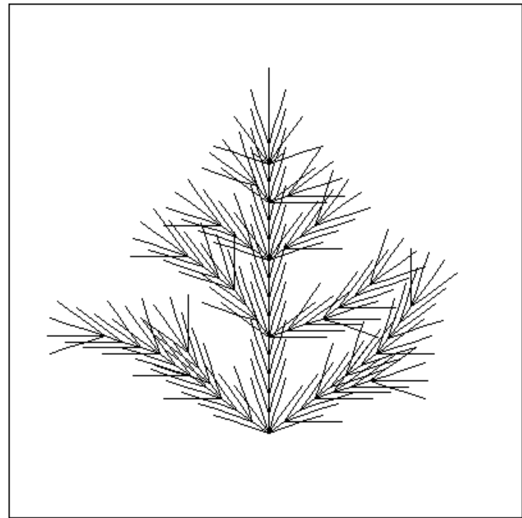
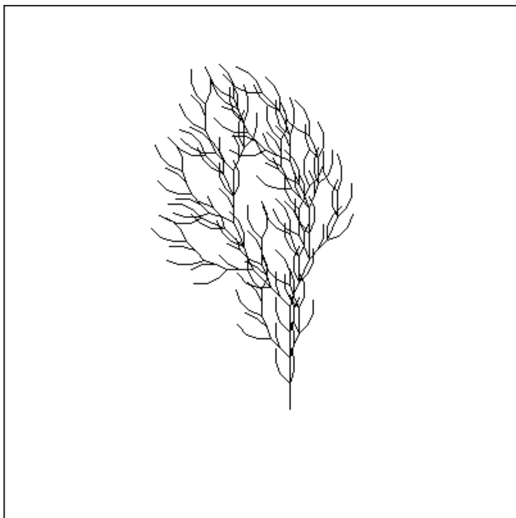
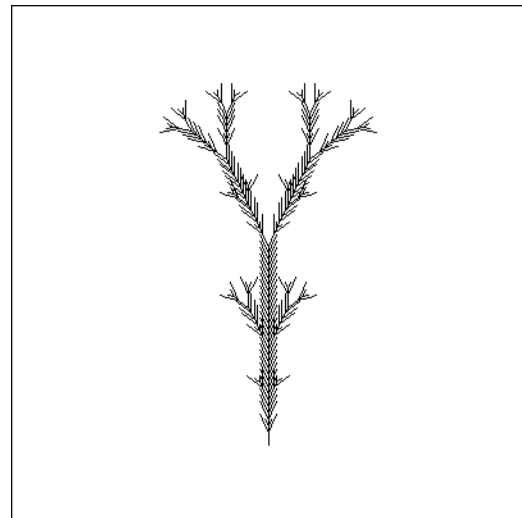
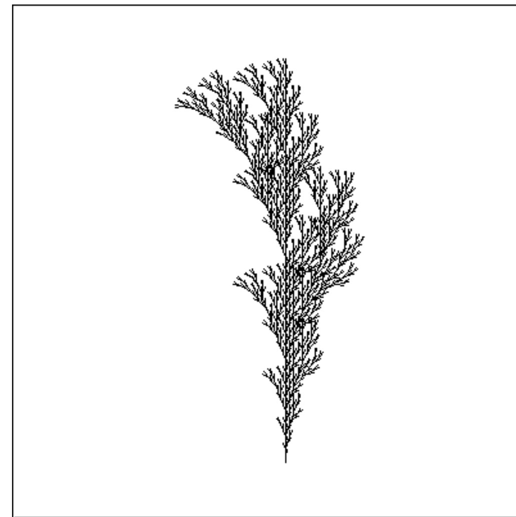
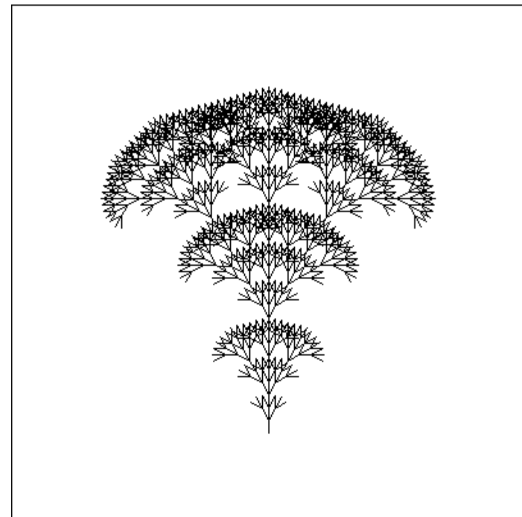
Conclusion

There are many kinds of L-systems. The kind we have described are context-free and deterministic — called 0L-systems in the literature. More powerful L-systems are needed to describe plant development more accurately and to model certain characteristics of plants. Unfortunately, these more complicated L-systems are not as easy to implement as 0L-systems and different approaches

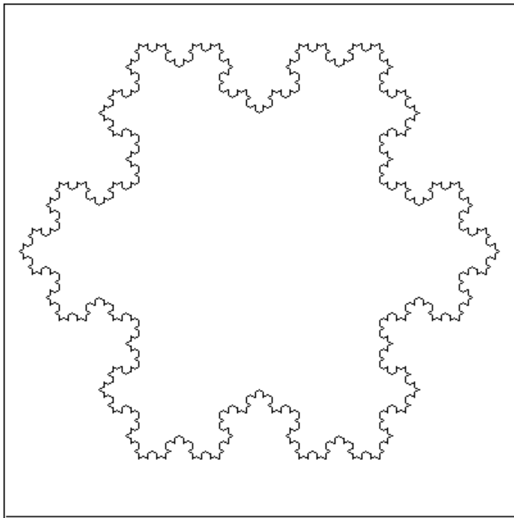
Downloading Icon Material

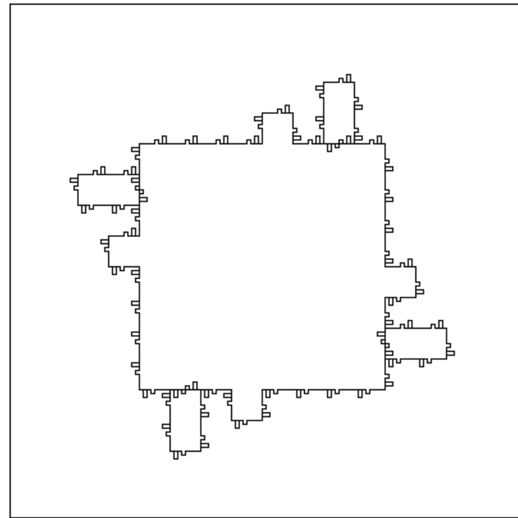
Most implementations of Icon are available for downloading via FTP:

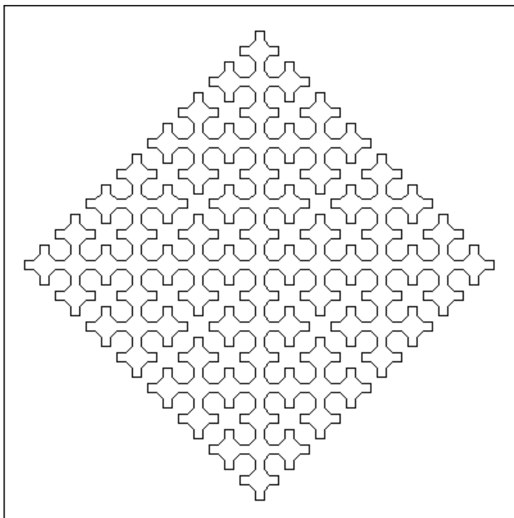
cs.arizona.edu (cd /icon)

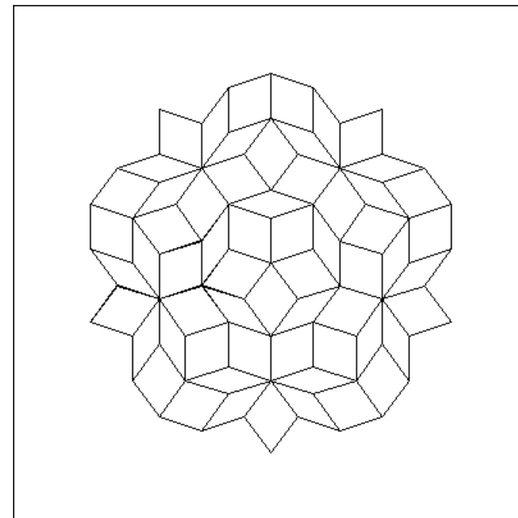







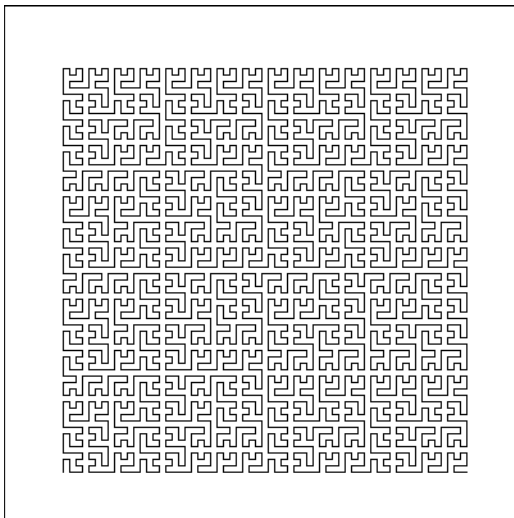
Lindenmayer “Plants”

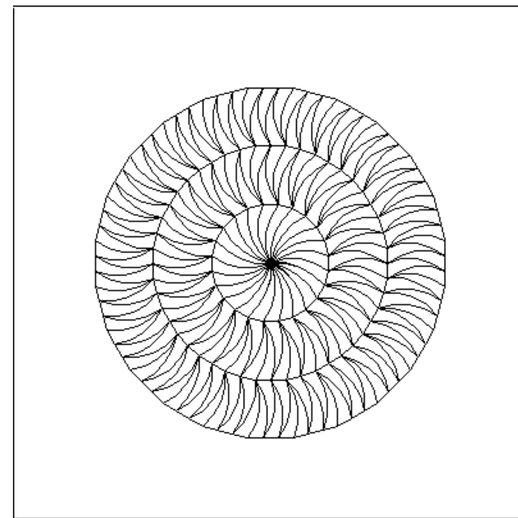












Other Lindenmayer Drawings

are needed. If you want to try your hand at these, see Reference 2 or 3. We'd welcome the addition of more capable L-system programs to the Icon program library.

Although the more complicated kinds of L-systems are not easy to implement, there are several things that can be done to extend 0L-systems to enable them to make more interesting drawings.

We'll describe these, as well as a different approach to implementing 0L-systems, in the future articles on Lindenmayer systems.

References

1. "Turtle Graphics", Icon Analyst 24, pp. 6-10.

2. *Lindenmayer Systems, Fractals, and Plants*, Przemyslaw Prusinkiewicz and James Hana, Springer-Verlag, New York, 1989.

3. *The Algorithmic Beauty of Plants*, Przemyslaw Prusinkiewicz and Aristid Lindenmayer, Springer-Verlag, New York, 1990.

4. *The Science of Fractal Images*, Hienz-Otto Peitgen and Deitar Saupe, Springer-Verlag, New York, 1988, pp. 272-286.

Answers to the Trivia Quiz

The places in the Analyst where you can find more material about the answers are given with the issue and page numbers following the answer. In some cases, you may have to dig a little to see how the answer is derived.

1. 48 bytes (4, 9).

2. None; string literals are included in the icode file produced when an Icon program is translated (4, 8). The situation for the compiler is similar.

3. There is one implementation of Icon for IBM mainframes that uses EBCDIC and another that uses ASCII (3, 3).

4. No, only `tab()` and `move()` are matching functions (3, 5).

5. All Icon *values* are the same size (6, 3).

6. Yes; `stop()` writes a line to standard error output but `exit()` doesn't (6, 11).

7. `IPATH` for the Icon interpreter (7, 9); `LPATH` for the Icon compiler (7, 11).

8. Nothing (9, 1).

9.

```
x := "abc"
x[2] := (x := [] & "B")
```

(10, 11).

10. For a list of numbers `L`

```
?L +:= 1
```

and

```
?L := ?L + 1
```

(8, 7).

11. Never (2, 3).

12. Icon has no statements (11, 11).

13. RTL is a superset of C in which the Icon run-time system is written (12, 5).

14. A set of co-expressions that share a program state (14, 8-9).

15. By a variant translator that produces SNOBOL4 code (18, 3).

16. `leading` (18, 6).

17. A stream is a data object that is capable of producing a sequence of values (19, 1).

18. It sets `&pos` to 1 (20, 2).

19. 109 (21, 8).

20. Degrees (24, 7).

Cheap Tricks

You can have a lot of fun writing clever expressions in Icon — while making your code hard to understand. We're not sure where we stand on this; we've both encouraged and discouraged such "idiomatic" programming in Icon. The problem is particularly difficult when teaching Icon. Students can learn a lot about Icon by writing clever but

Back Issues

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

opaque expressions, yet we want to encourage clear, readable coding techniques.

For the Analyst, we feel safe in at least showing examples of clever ways to use Icon; you have to make your own decisions about style. We titled this short article **Cheap Tricks** because we didn't want to dignify what follows as a programming tip or something from the wizards.

The case in point involves calling the same function with different argument lists in succession. This situation occurs fairly often in graphics operations involving symmetry, as in

```
plot(figure, x + i, y + j)
plot(figure, x + i, y – j)
plot(figure, x – i, y + j)
plot(figure, x – i, y – j)
plot(figure, x + j, y + i)
plot(figure, x + j, y – i)
plot(figure, x – j, y + i)
plot(figure, x – j, y – i)
```

Many graphics functions take an arbitrary number of argument sets so that multiple drawings can be done in one call. If our hypothetical plot() supported this, the eight calls above could be replaced by

```
plot(
  figure, x + i, y + j,
  figure, x + i, y – j,
  figure, x – i, y + j,
  figure, x – i, y – j,
  figure, x + j, y + i,
  figure, x + j, y – i,
  figure, x – j, y + i,
  figure, x – j, y – i
  )
```

For this article, we'll suppose plot() does not provide this capability. What can be done, then, to do the plotting with a more compact expression? It's tempting to write

```
every plot(figure, x + (i | – i | j | –j),
  y + (i | –i | j | –j))
```

but this produces 16 calls — half of which are not wanted.

You can limit the argument lists to the desired ones by using

```
every plot(figure, x + (i | –i), y + (j | –j))
every plot(figure, x + (j | –j), y + (i | –i))
```

It seems, though, that there should be a way to do this with one every. Another temptation is

```
every plot(
  (figure, x + (i | –i), y + (j | –j)),
  (figure, x + (j | –j), y + (i | –i))
  )
```

This expression, however, does not do what's intended. The two arguments of plot() are mutual evaluation expressions. A mutual evaluation expression produces its last argument. So,

```
(figure, x + (i | –i), y + (j | –j))
```

produces y + j and

```
(figure, x + (j | –j), y + (i | –i))
```

produces y + i. The resulting call of plot() is

```
plot(y + j, y + i)
```

which is hardly what's wanted, not to mention there would be 15 more erroneous calls because of the every.

If you're intent on being clever, don't give up. It *is* possible to do this in a single every loop. The trick is list invocation:

```
every plot ! (
  [figure, x + (i | –i), y + (j | –j)] |
  [figure, x + (j | –j), y + (i | –i)]
  )
```

You can be sure a casual reader of code like this will not understand it or what you had in mind. This reminds us of a system programmer in the early days of computing at Bell Labs who wrote totally bogus code that was overwritten by the correct code when the program started execution. (This was before the days of "pure" code. In fact, the machine language in use then required code to be overwritten even to perform common operations.)

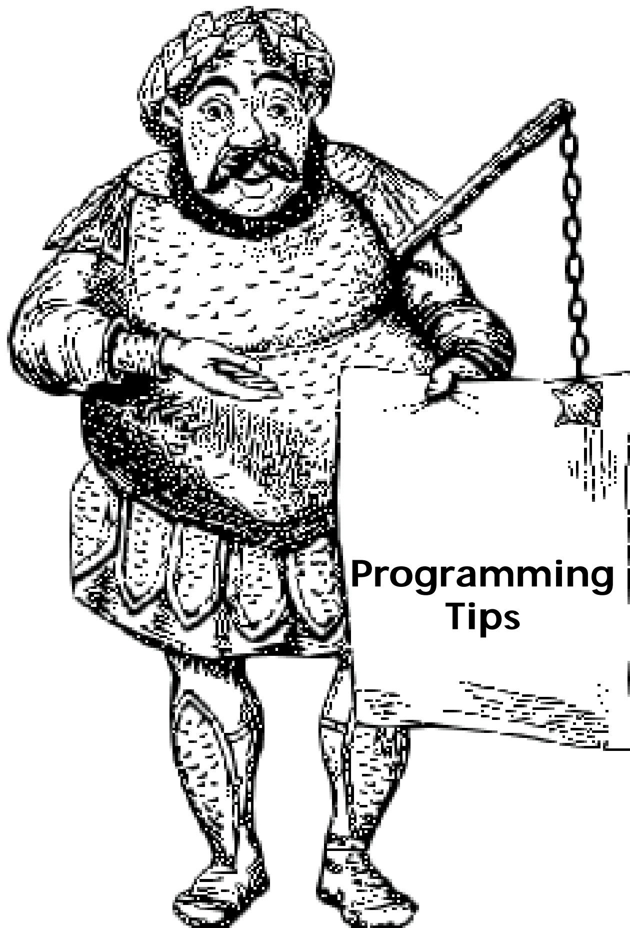
There's more wrong with this approach than obscurity. A list is created for each set of arguments — eight lists in all every time the expression is evaluated. That adds up to a lot of storage allocation, plus the time that may be spent in garbage collection as a result. These problems can be mitigated somewhat by using records instead of lists, since records take less space than lists. (Version 9 of Icon is required to use records in invocation.) Something like this will do:

```
record args(x, y, z)
        ...
```

```
every plot ! (
    args(figure, x + (i | –i), y + (j | –j)) |
    args(figure, x + (j | –j), y + (i | –i))
    )
```

This is, if anything, more obscure than the list version.

Okay, you've seen it. Now forget it.



Returns from Procedures

This is the second of three tips on ways to avoid bugs in your Icon programs. It starts with a question on language design: What should happen when control flows off the end of a procedure without an explicit return, fail, or suspend?

One possibility is to treat this situation as an error. This is, in some sense, the safest choice, but it runs contrary to Icon's philosophy that favors ease of programming over error checking and rigidity.

Another possibility would be to return the result of the last expression in the procedure that was evaluated, as some other programming languages do. This choice has some merit, but since an Icon procedure can return, fail, or suspend, things get murky. For example, what should happen if the last expression is a generator? Should it generate all its values or return just the first one? This alternative also has the disadvantage of making it hard to understand what some procedures do, since the last expression evaluated may not be the last physical expression in the procedure (consider, for example, an if-then-else expression).

A third possibility, and one that was chosen for early versions of Icon, is to return a null value if control flows off the end of a procedure. This is a reasonable choice and the null value is the obvious value to return if none is specified.

A fourth possibility, and the one chosen for later versions of Icon, is to have the procedure fail if control flows off its end. The argument for this choice is that if a procedure doesn't explicitly return a value, it should not return any value at all, which fits nicely with Icon's concepts of success and failure.

We can think of other possibilities, such as a command-line option to the Icon translator to change the treatment of returns or a directive for doing this that could be included in the program itself. We don't think these are good ideas, but there's not much chance of changing Icon at this point in its life in any event.

Either the third or fourth possibility mentioned above can lead to bugs. If the null value were returned, a procedure that generates a sequence of values would have to provide and explicit fail after generating all its results to avoid returning an extra, spurious null value.

On the other hand, with the present scheme, a procedure that is used only for side effects and computes no value fails unless an explicit return is provided. And the problems this can lead to are what this tip is about.

Consider this procedure:

```
procedure error(message)
   write(&errout, "***", message)
end
```

Here, the procedure serves as a wrapper to simplify producing diagnostic messages. It doesn't compute a value and doesn't return one — it just lets control flow off the end, resulting in failure.

Ordinarily, the fact that such a procedure fails doesn't cause any problems. Because no value is expected, such a procedure usually is called in places where success and failure are not important, and the failure goes undetected.

Sometimes, however, this goes awry. Consider the following code:

```
   while line := read() do
     filter(line)
       ...
procedure filter(line)

   if line ? {
      # analysis of line to see if it should
      # be written
     }
   then write(line)

end
```

Since control flows off the end of filter() whether or not it writes a line, it fails. That isn't a problem, since the outcome of an expression in a do clause is ignored.

But now suppose you observe that the loop above can be written more compactly as

```
while write(filter(read()))
```

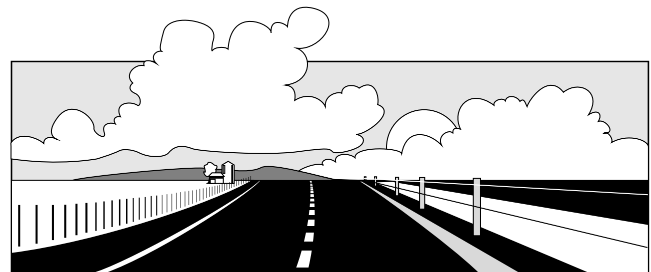
This looks perfectly reasonable, but only one line is read, since the failure of the first call of filter() terminates the while loop.

The cause of a problem like this may be hard to locate — especially in the context of a large program, or when someone who did not write the original program tries to improve it.

The solution is simple: Just add a return at the end of filter():

```
procedure filter(line)

   if line ? {
      # analysis of line to see if it should
      # be written
     }
   then write(line)

   return

end
```

You'll save yourself a lot of grief if you make it a practice always to put a return at the end of a procedure that does not have a value to return.



What's Coming Up

In the next issue of the Analyst, we'll have an article on the static analysis of Icon programs. We'll also have another tip on how to avoid bugs in Icon programs.

We're also working on the next articles on random numbers and Lindenmayer systems.

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

© 1994 by Madge T. Griswold and Ralph E. Griswold
All rights reserved.