The Icon Analyst

In-Depth Coverage of the Icon Programming Language

April 1995
Number 29

In this issue ...

New Area Code

Most of Arizona, including Tucson, now has a new area code, 520. Until July 23, 1995, our old area code, 602, also will work. After that, you'll have to use 520 to reach us.

A word of warning: Some automated switchboards only can dial area codes whose second digit is a 0 or 1. If you try to call us at 520 after March 19 and are unable to reach us, that may be the problem.

Procedure and Operator Values

String invocation, described in the last issue of the Analyst, allows you to invoke procedures and operators using their string names. It's also possible to get procedure and operator values from their string names.

In past articles we've mentioned that procedures are Icon values, although it's seldom necessary to use such values explicitly. For example, the global variable trim has a procedure (function) value at the beginning of execution, which can be seen by executing

```
write(type(trim))
```

which writes procedure. It's this value that's looked up if you execute

```
"trim"(line)
```

If you're using trim() explicitly, there's no reason to use string invocation, but suppose you use

```
p := read() |
  stop("procedure name expected")
```

to provide a string name for a procedure that happens to be "trim". If you subsequently use p to invoke trim(), there's a lookup each time you use it.

That overhead can be avoided by getting the actual procedure value corresponding the name. The function proc(s) does this. If s is the name of a procedure, proc(s) produces the corresponding procedure value, but if s is not the name of a procedure, proc() fails. For example,

```
p := proc("trim")
```

assigns the procedure value for trim() to p. In the example above, proc() might be used as

```
p := proc(read()) |
  stop("procedure name expected")
```

While the notion that procedures are values is generally well known to experienced Icon programmers, many aren't aware that there also are values for operators. Unlike procedures, there are no global variables corresponding to operators. An expression like

```
i + j
```

is a syntactic form; you can't use

```
+(i, j)
```

although you can use

```
"+"(i, j)
```

Nevertheless, operators are values and you can use proc() to get them.

In addition to the string name for the operator, a second argument to proc() is needed to specify how many operands the operator has. Thus,

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

is the unary size operator, while

```
proc("*", 2)
```

is the binary multiplication operator.

The default for the second argument to proc() is 1, so

```
proc("*")
```

produces the unary size operator.

But what *is* the value produced by proc() for an operator? It's a procedure value. For example,

```
type(proc("*"))
```

produces "procedure".

This may seem a bit strange, but there's nothing particularly subtle about it. In the implementation of Icon, procedures and operators are much the same. The difference is that procedures are the initial values of global variables, while operators are distinguished syntactically.

The values for operators are not just curiosities; they allow operators to be invoked using the same syntax as is used for procedures. For example, you can do this:

```
size := proc("*")
```

and then

```
write(size(x))
```

There are a few things about proc() for operators that you need to know. One is that proc(s, n) fails if s not the name of an nary operator. A bit of care is needed in specifying the second argument. For example,

```
proc("...", 2)
```

fails, since to-by is a ternary operator given in distributed syntactic form, and proc() makes no allowance for the fact that the by clause can be omitted.

If you want to know if s is the name of any operator, you can use

```
proc(s, 1 to 3)
```

If s is the name of a unary operator, you get that operator value. If it isn't, proc() fails and 1 to 3 is resumed to produce 2. If s is name of a binary operator, you then get that value, and so on.

In some situations, you might want to reorder the alternatives, as in

```
proc(s, 2 | 1 | 3)
```

Incidentally, if s is the name of a procedure, proc() produces the corresponding procedure value and the second argument is ignored; the forms above can be used for either procedure names or operator names.

There is one subtle aspect of procedure names and values. Suppose the value of a variable that corresponds to a built-in procedure (function) has been changed. This happens if there is a declared procedure by the same name, as in

```
procedure tab(i)
   ...
end
```

This also happens if a value other than a procedure has been assigned to the global variable, as in

```
tab := 8
```

In the first case, proc("tab") gives the variable for the declared procedure, which certainly is reasonable. In the second case, proc("tab") fails, which also is reasonable, since attempting to use tab to invoke the function would not work

Suppose, however, you want to use the built-in function even if the corresponding global variable no longer has this function as value. Prior to Version 8.10 of Icon, there was no way to do this; in fact, if the value of a global variable corresponding to a built-in function was changed without being saved in another variable, there was no way to get to the built-in function.

Starting with Version 8.10, proc(s, 0) produces the built-in function for s, if there is one. This feature allows a built-in function to be retrieved if its global variable has been changed, and it also allows a procedure to "overload" a built-in function, while still allowing the built-in function to be used.

The latter case is illustrated by a procedure to trim both the beginning and end of a string:

```
procedure trim(s, c)
  static trim_end
  initial trim_end := proc("trim", 0)
  /c := ' '
  trim_end(s, c) ? {
    tab(many(c))
    return tab(0)
    }
end
```

Of course, you could use a name other than trim for this procedure, but if you want to change the functionality of trim() in a program, this is an elegant way to do it. This technique also can be used in library procedures to overload built-in functions without having to change the programs that link them.

There is another function that sometimes is useful in conjunction with proc()—args(p), which returns the number of arguments expected by the procedure or operator p.

For built-in functions like write() that accept an arbitrary number of arguments, args(p) produces 0. (There is no built-in function that expects no arguments.) For a declared procedure, args(p) produces the negative of the number of arguments given in the procedure's declaration. There is no way to tell if a procedure has been declared with an arbitrary number of arguments.

Finally, a word of caution; args(p) requires a procedure-valued argument; it does not automatically convert string names.

Applications of String Invocation

String invocation may seem a bit ethereal or even fraught with dangers. You may not even see why it's needed.

String invocation is one of those features that isn't needed often, but when it's needed, it can be very useful indeed. In some situations, string invocation may not be necessary, but it may simplify program design and provide generality that can't be provided by other means. Using string invocation may, in fact, suggest useful approaches to programming. We'll illustrate these points with a few examples.

Filtering Files

Many programming tasks involve "filtering" the lines of a file, applying the same operation to each line to produce another file. The operation may transform the line, eliminate it, or produce several lines from one.

Many "one-shot" filters are written in Icon simply by writing a short program that applies the operation, as in

```
procedure main()
  while write(trim(read()))
end
```

which trims trailing blanks from the lines of input. Suppose this program is named trim.icn. Then in UNIX, for example,

```
trim <input >output
```

writes the trimmed lines of input to output.

A more general approach is to write a program, ifilter.icn, that takes the name of the filtering operation on the command line, as in

```
ifilter trim <input >output
```

Since the name of the operation is given as a string on the command line, string invocation can be used to get the corresponding procedure, as in

```
procedure main(args)
  local p
  p := args[1] | stop("*** no operation")
  while line := read() do
    every write(p(line))
end
```

The every loop is used in case the operation is a generator.

This formulation has two problems. If the command-line argument is not the name of an operation, a run-time error occurs when it is applied and the message may be mystifying to a user. In addition, the procedure is looked up by its string name for every line that is read in. Both of these problems can be solved by converting the name to a procedure, as described in the preceding article:

```
procedure main(args)
  local p
  p := proc(args[1] ) |
    stop("*** invalid operation")
  while line := read() do
    every write(p(line))
end
```

Although this program doesn't use string invocation explicitly, what it does amounts to the same thing; whether string invocation is used directly or proc() is used, the same underlying feature is at work—getting a procedure value from its string name.

In either of the formulations, operators can be used for filtering, as in

```
ifilter '*' <input >output
```

which writes the lengths of the lines in input to output.

Procedures in libraries also can be used with these formulations, but they must be linked with ifilter.icn when it is translated, as in

```
icont ifilter wordform.u
```

which adds the procedures in wordform to ifilter. It's also necessary to add

```
invocable all
```

to ifilter.icn, as described in the article on string invocation in the last issue of the Analyst.

To make ifilter really useful, we need to be able to handle arguments as well as the operation to apply. For example, to use a function like right(), it's necessary to supply the field width and the padding string. With this addition, for example,

```
ifilter right 10 0 <input >output
```

could be used to produce lines of width 10 filled with zeros at the left.

Here's a version of ifilter that handles arguments:

```
invocable all

procedure main(args)
  local p

  p := proc(args[1], 1 to 3) |
    stop("*** invalid operation")

  while args[1] := read() do
    every write(p ! args)

end
```

This program requires some explanation. The second argument to proc() is used as described in the preceding article to cover unary, binary, and ternary operators (in that order — without an additional feature, the program uses the one it finds first).

The lines

```
while args[1] := read() do
  every write(p ! args)
```

may be less clear. The value of args for

```
ifilter right 10 0 <input >output
```

is

```
["right", "10", "0"]
```

while what is needed for a line of input is

```
p(read(), "10", "0")
```

where p is the result of proc(args[1], 1 to 3). By overwriting the first element of args, once it has been used to get p, args is equivalent to

```
[read(), "10", "0"]
```

This allows list invocation to be used in

```
p ! args
```

Note that the version of ifilter above takes advantage of the fact that many Icon functions have a string of interest as their first argument, while trailing arguments provide parameters. This also allows trailing arguments to be omitted when defaults apply, as in

```
ifilter right 10
```

which produces lines filled with blanks on the left and

```
ifilter map
```

which converts uppercase characters to lowercase ones.

Interactive Expression Evaluation

If you are learning or testing Icon, it's handy to be able to keyboard an Icon expression and immediately get the results of evaluating it without having to write and run a separate program. A program that reads in an Icon expression and evaluates it is what's needed.

Such a program needs to parse the expression, read in as a string, and use string evaluation to evaluate it. Here's such a program:

```
invocable all

link ivalue

procedure main()
  local line

  while line := read() do
    every write(eval(line))

end

procedure eval(expr)
  local p, args, tok

  &error := -1

  expr ? {
    p := trim(tab(upto('(')), '\t ') | {
      write(&errout, "** syntax error")
      fail
      }
    p := proc(p, 2 | 1 | 3) | {
      write(&errout, "** invalid operation")
```

```
      fail
      }
    move(1)

    args := []

    repeat {
      tab(many(' \t'))
      tok := trim(tab(upto(',)'))) | break
      put(args, ivalue(tok)) | fail
      move(1)
      }

    suspend p ! args
    }

end
```

This program assumes functional form for input and it can handle "expressions" like +(2, 3), but it can't handle infix expressions like (2 + 3) or expressions involving control structures.

Error conversion is used so that syntax errors in the input do not cause run-time errors. The name for the operators is found and converted to a procedure. The procedure ivalue() from the Icon program library is used to convert arguments, which are placed on a list for subsequent invocation. It can handle literals and constants. ivalue() is a story in itself; see the Icon program library if you're interested.

As you'll note, this program does not handle nested expressions; we'll leave that as an "exercise". It's not easy to handle nested generators. For this, "think recursive generators" [1].

A Suffix Calculator

Perhaps the most common use of string invocation is to carry out "commands" entered by a user of an application. We described a suffix calculator for Icon in an earlier Analyst article [2]. Since we explained this program in some detail in that article, we'll just list the program here. Look at it in terms of the material described in this article.

```
invocable all

link ivalue
link usage

global stack

procedure main()
  local line

  stack := [ ]

  while line := read() do
    (operation | value | command)(line) |
      Error("erroneous input ", image(line))

end

procedure command(line)

  case line of {
    "clear":    stack := [ ]
    "dump":     every write(image(!stack))
    "quit":     exit()
    default:    fail
    }

  return

end

procedure operation(line)
  local p, n, arglist

  if p := proc(line, 2 | 1 | 3) then {
    n := abs(args(p))
    arglist := stack[–n : *stack + 1] | {
      Error("too few arguments")
      fail
      }
    stack := stack[1 : –n]
    &error := 1     # anticipate possible error
    put(stack, p ! arglist) | {
      if &error = 0 then
        Error("error ", &errornumber,
          " evaluating ", image(line))
      else
        Error("failure evaluating ",
          image(line))
      stack |||:= arglist
      }
    &error := 0
    return
    }

  else fail

end

procedure value(line)

  put(stack,ivalue(line)) | fail

  return

end
```

Conclusions

You may not use string invocation often in our Icon programs, but at least keep it in mind. It

not only makes writing some programs much easier than they would be using other techniques, but it may suggest ways of designing programs that provide generality and flexibility that aren't feasible to provide in other ways.

References

1. "Recursive Generators", Icon Analyst 13, pp. 10-12.

2. "Anatomy of a Program – A Suffix Calculator", Icon Analyst 12, pp. 2-4.

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: (520) 621-6613

fax: (520) 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

Bright Forest Publishers

Tucson Arizona

Curiosity or Problem?

Bob Alexander noticed something apparently unusual about Icon's random number generator. Consider this program, in which the seed of the random number generator is set to successive even integers:

```
procedure main()
  i := 20
  every &random := 0 to 30 by 2 do
    write(
      right(?i, 3), right(?i, 5), right(?i, 5),
      right(?i, 5), right(?i, 5), right(?i, 5),
      right(?i, 5), right(?i, 5), right(?i, 5),
      )
end
```

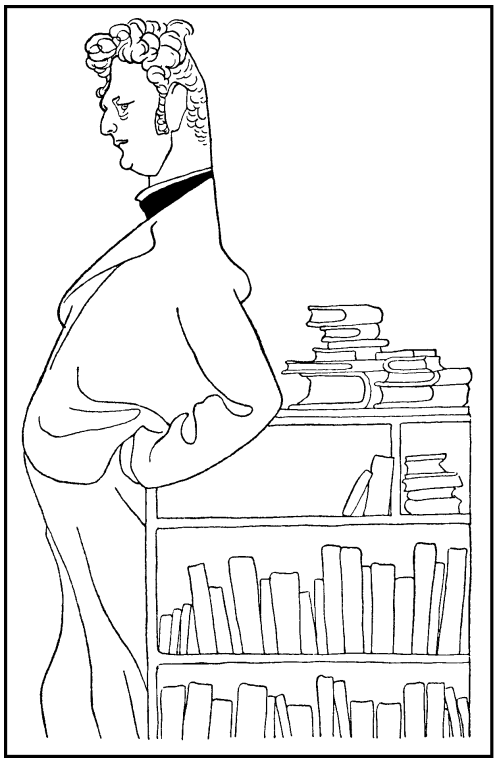
The output is:

```
  5    9    7   11    9    7    2   15    2
  5   10    7    5    3   13   11   20    7
  6   10    7   19   16   19   19    5   11
  6   11    7   14   10    5    8   10   16
  7   12    7    8    3   11   16   15    1
  7   13    8    3   17   18    5   20    6
  8   14    8   17   11    4   13    5   11
  9   15    8   11    4   10    2   10   16
  9   15    8    6   18   16   10   15    1
 10   16    8   20   12    2   19   20    6
 10   17    8   15    5    9    7    5   11
 11   18    9    9   19   15   16   10   16
 11   19    9    3   13    1    4   15    1
 12   19    9   18    6    7   13   20    6
 12   20    9   12   20   14    1    5   11
 13    1    9    6   13   20   10   10   16
```

The regularities in the columns are less apparent for larger values of i, but they're definitely there.

Some persons are amazed at the results and think something is terribly wrong with Icon's random number generator. Other persons think the output is interesting but not surprising or worrisome, and common to all linear congruential random number generators.

Perhaps, if you're mathematically inclined, you can shed some light on this that we can pass along to readers of the Analyst.



From the Library

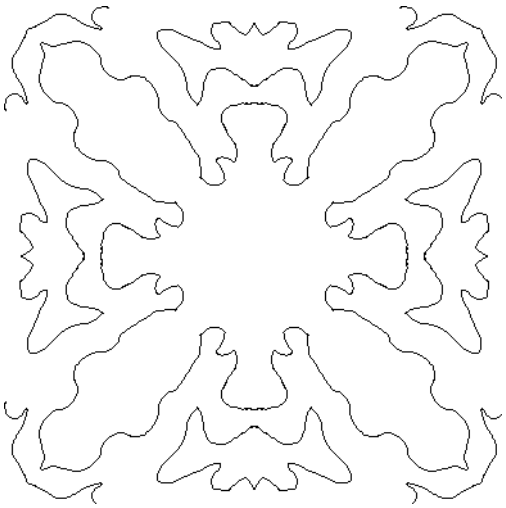
When we choose material from the Icon program library for these articles, we usually look for things that are the most useful. The Icon programming library also has many entertainments. These programs take several forms — the library has games and puzzles, and also a few visual amusements. We've picked one of these for this article.

The program we've chosen produces symmetrical drawings. Symmetry is fascinating and has been the subject of much study and writing. See References 1-18 for some of the more accessible literature.

There's something about symmetrical designs that human beings find attractive. The reasons for this are not at all clear and lead to deep water (such as, does attraction to symmetric features have a survival value that might have had an evolutionary effect?).

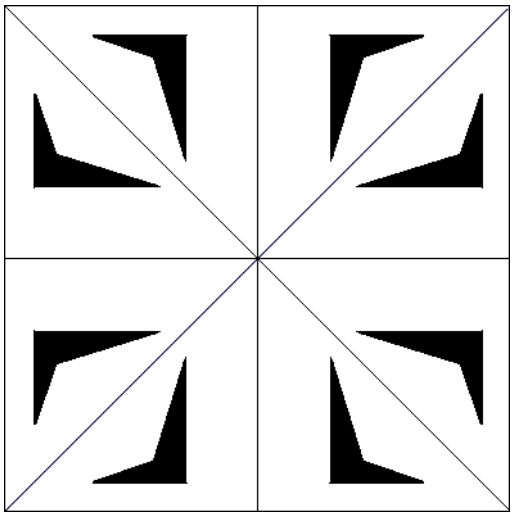
To appreciate the power symmetry has on our perception, consider this relatively meaningless scribble:

This scribble looks, perhaps, like a sketch of a piece of coastline, but there's nothing particularly attractive about it. But now consider the design below, which was constructed by mirroring the scribble above in a symmetric fashion:

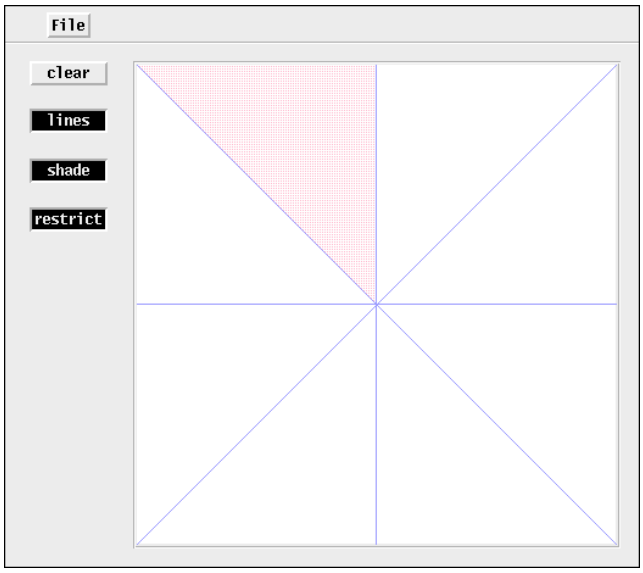


This isn't art, but it's certainly more interesting than the scribble from which it was created.

The symmetry used in this figure, one of the 17 plane symmetries, is called the sunflower symmetry in quilting [19] and carries the technical name p4m (or sometimes p4mm) in crystallography. The sunflower symmetry is produced by reflecting a drawing using two sets of mirrors, as indicated below:



The library program symdraw, whose visual interface is shown on at the top of the next page, lets a user create drawings with the sunflower symmetry.



The symdraw **Interface**

The drawing area at the right shows the axes of reflection as lines and one shaded octant. The shaded region is called the generating region, since anything drawn in it is reflected in the other octants.

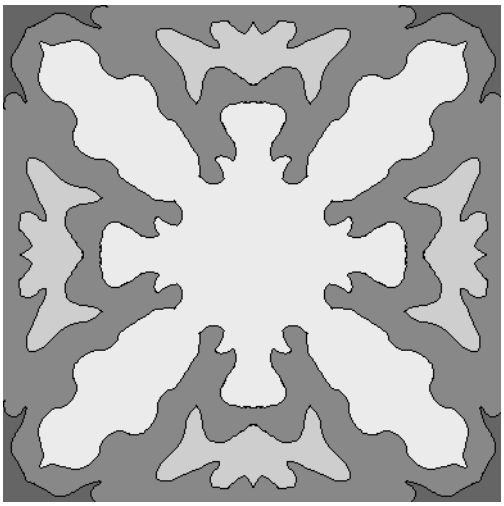
When a user presses and drags with the left mouse button with the mouse cursor in the generating region, a line is drawn following the mouse cursor, and this line is reflected in the other octants. Drawing stops if the mouse cursor moves outside the generating region.

Lines can be erased by using the right mouse button in a similar fashion. The middle mouse button is used to draw straight lines. A line begins where the middle mouse button is pressed and ends where it is released.

The File menu provides for saving a snapshot of the drawing (without the mirror lines and shading) and for quitting the application. The buttons at the left provide for clearing the drawing area, turning the lines and shading on and off, and restricting the drawing to the generating region. If drawing is not restricted to the generating region, drawing anywhere in the drawing area is reflected in all octants. Unrestricted drawing is easier and more fun, but it tends to produce less attractive results than restricted drawing.

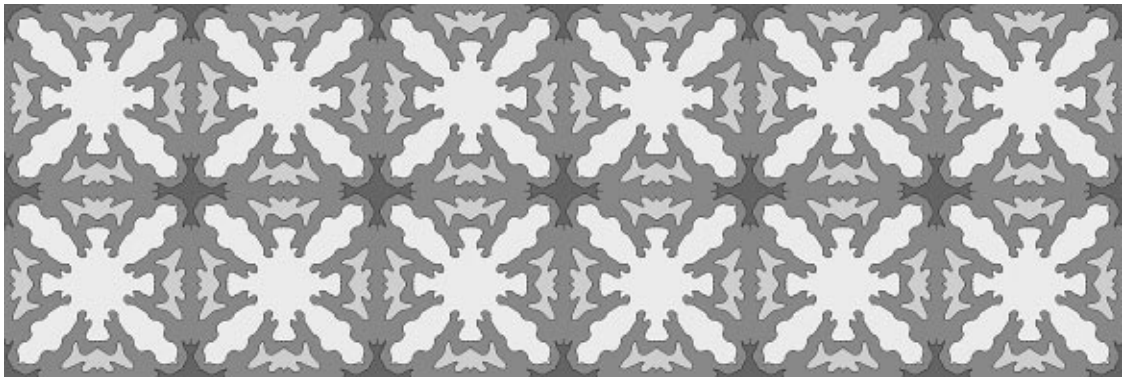
The images produced by symdraw are black-and-white line drawings, but they easily can be colored in any painting application that provides a "paint bucket" or similar tool for filling areas. The best results for color usually are obtained if symmetrically placed areas are filled with the same color.

We can't show you a color design here, but the grayscale image below may give you an idea of the possibilities.

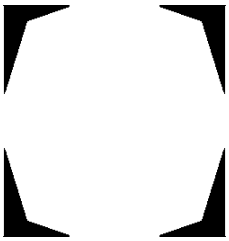


Images produced by symdraw tile seamlessly to produce larger repeat patterns such as the one shown in reduced form at the bottom of this page.

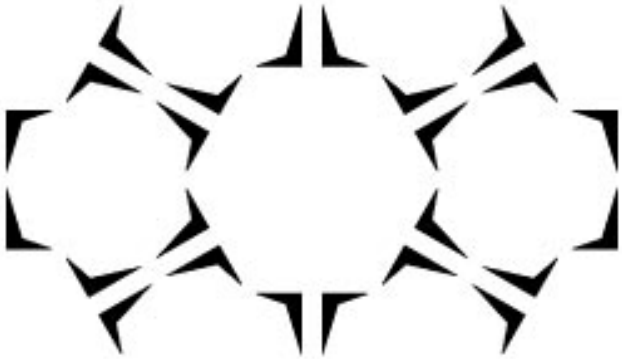
symdraw was included in a recent update of the Icon program library that was sent to update subscribers. It will be included in the next general release of the library, which is scheduled for this summer.



At present, symdraw only supports the sunflower symmetry. This symmetry is relatively easy to implement, since drawing symmetric points involves only sign changes and exchanges of the x,y coordinates. There are other symmetries that would be even easier to implement, such as the "prickly pear" symmetry shown below. Other symmetries, such as the "turnstile" symmetry, require more complicated computations.



prickly pear (pmm)



turnstile (p6m)

References

1. D'Arcy Wentworth Thompson, *On Growth and Form*, Cambridge University Press, 1942.

2. Hermann Weyl, *Symmetry*, Princeton University Press, 1952.

3. L. Fejes Tóth, *Regular Figures*, Pergamon Press, 1964.

4. Gyorgy Kepes, ed., *Module, Proportion, Symmetry, Rhythm*, George Braziller, 1966.

5. F. J. Budden, *The Fascination of Groups*, Cambridge University Press, 1972.

6. A. V. Shubnikov and V. A. Koptsik, *Symmetry in Science and Art*, Plenum Press, 1974.

7. Peter S. Stevens, *Patterns in Nature*, Little, Brown, and Company, 1974.

8. Joe Rosen, *Symmetry Discovered*, Cambridge University Press, 1975.

9. E. H. Lockwood and R. H. Macmillan, *Geometric Symmetry*, Cambridge University Press, 1978.

10. Krome Barratt, *Logic and Design in Art, Science, and Mathematics*, Design Books, 1980.

11. Branko Grünbaum and G. C. Shephard, *Tilings and Patterns*, W. H. Freeman and Company, 1987.

12. Dorothy K. Washburn and Donald W. Crowe, *Symmetries of Culture; Theory and Practice of Plane Pattern Analysis*, University of Washington Press, 1988.

13. Doris Schattschneider, *M. C. Escher; Visions of Symmetry*, W. H. Freemen and Company, 1990.

14. Jay Kappraff, *Connections; The Geometric Bridge Between Art and Science*, McGraw-Hill, Inc., 1991.

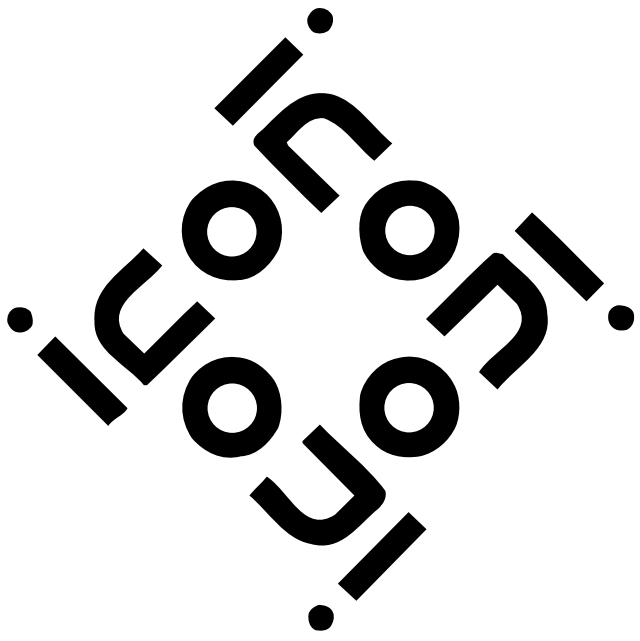
15. Peters S. Stevens, *Handbook of Regular Patterns*, The MIT Press, 1991.

16. Ian Stewart and Martin Golubitsky, *Fearful Symmetry; Is God a Geometer?*, Penguin Books, 1992.

17. Michael Field and Martin Golubitsky, *Symmetry in Chaos*, Oxford University Press, 1992.

18. Michelle Emmer, ed., *The Visual Mind; Art and Mathematics*, The MIT Press, 1993.

19. Xaos Tools, *Terrazzo; User's Guide, Macintosh Version 1.0*, 1994.



Dynamic Analysis of Icon Programs (Continued)

In the last issue of the Analyst, we started a series of articles on the dynamic analysis of Icon programs — what goes on during program execution. In this and subsequent articles, we'll explore various aspects of program execution in Icon. Before going on, we need to explain how we've chosen programs for analysis.

Analysis Test Bed

Since static analysis only depends on the text of programs, it can be applied to any program and it's not particularly difficult to select programs to analyze. In earlier static analyses, we used the entire Icon program library.

Choosing programs for dynamic analysis is more difficult. Dynamic analysis, unlike static analysis, is very time consuming. It's impractical to apply the same dynamic analysis to a large number of programs. We decided to pick about a dozen programs. Selecting even this few was a problem.

In order to perform dynamic analysis on a program, the program needs to perform enough computation to produce meaningful results. In order for analysis to be practical, a program also must run in batch mode. Appropriate test data also is needed. In order to make comparisons between programs, they also need to run for approximately the same amount of time.

We chose our programs for dynamic analysis from the Icon program library so that persons interested in dynamic analysis would have easy access to the programs. There are 285 programs in the Version 9 program library. You'd think it would be easy to find a dozen that are suitable, but it wasn't. Most of the graphics programs and many others are unsuitable because they can't be run in batch mode. We also lack test data for most of the remaining programs. In some cases we were able to create test data, but for others it wasn't possible to do this without investing more time and effort than we could afford. Among the remaining candidates, many were unsuitable because they weren't designed to perform the extensive computation needed to make the results of dynamic analysis useful.

We finally managed to find 11 programs that met our needs. These programs are by no means representative of Icon programming — if such a concept even is meaningful. Here are the ones we've chosen:

program	*functionality*
csgen.icn	sentences from context-free grammars
deal.icn	randomly dealt bridge hands
fileprnt.icn	character display of files
genqueen.icn	solutions to the *n*-queens problem
iiencode.icn	text encoding for files
ipxref.icn	cross references for Icon programs
kwic.icn	keyword-in-context listings
press.icn	file compression
queens.icn	solutions to the *n*-queens problem
rsg.icn	sentences from context-free grammars
turing.icn	Turing machine simulation

The choice of two programs for the *n*-queens problem was deliberate; the methods used in the two programs are different, and we thought it would be interesting to compare them.

In retrospect, after performing extensive dynamic analyses on these programs, we're not particularly satisfied with our choices. Some of the programs use specific features of Icon that aren't used in many programs, and it's easy to draw unwarranted conclusions if these programs are taken to be representative. For now, we'll present the result we have with a caution about drawing such conclusions.

The Evaluation of Functions and Operators

In the last issue of the Analyst, we showed a summary report of function events for one program (iiencode.icn), which we repeat here for reference:

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

cs.arizona.edu (cd /icon)

event	count
function call	197393
function failure	441
function return	190679
function suspension	6272
function resumption	0
function suspension removal	6272

This summary only provides information for functions, and for them, only the aggregate activity.

With the instrumentation provided by MT Icon and support procedures that have been developed for dynamic analysis, it's relatively easy to get information not only for individual functions but for operations as well. The listing at the bottom of this page shows the results for iiencode.icn, ordered by decreasing number of calls.

From this listing alone, we can determine many things about the program. The extensive use of iand(), ior(), ishift(), ord(), and char() suggests the kind of computation the program is doing, which is atypical; few Icon programs use these functions. The large number of uses of e1[e2] is interesting, and we'll have more to say about this later. The use of move(), pos(), and find() shows the program uses string scanning. In the column for suspensions, there are only two nonzero entries. move() always suspends so that it can restore the cursor position if it is resumed. Note, however, that move() never is resumed, indicating there is no backtracking in string scanning in this program. The only generator used in this program is e1 to e2 by e3, which almost certainly occurs in every-do, not goal-directed evaluation. In fact, Icon's expression-evaluation mechanism probably plays no significant role in this program. (To be sure of that, we'd have to have information on procedure activity. This is easy to get but is confusing when given in combination with the activity of built-in expressions.)

A plausible conclusion is that this program could have been written in a lower-level language like C without much of a change in structure.

Now look at the corresponding listing for csgen.icn on the next page. This program shows extensive use of generators. Most of the resumptions, however, are for !e, which probably is used in every-do, not in goal-directed evaluation. The

name	calls	returns	suspends	failures	resumps	removals
iand()	56867	56867	0	0	0	0
e1[e2]	37634	37634	0	0	0	0
ord()	37632	37632	0	0	0	0
ishift()	31360	31360	0	0	0	0
writes()	25929	25929	0	0	0	0
e1 := e2	25514	25514	0	0	0	0
e1 + e2	25509	25509	0	0	0	0
char()	25507	25507	0	0	0	0
e1 ~== e2	25507	24971	0	536	0	0
–e	18816	18816	0	0	0	0
+e	12544	12544	0	0	0	0
ior()	12544	12544	0	0	0	0
move()	6691	0	6272	419	0	6272
*e	421	421	0	0	0	0
reads()	420	419	0	1	0	0
pos()	419	419	0	0	0	0
find()	21	0	0	21	0	0
/e	5	4	0	1	0	0
\e	3	0	0	3	0	0
e1 === e2	2	0	0	2	0	0
close()	2	2	0	0	0	0
e1 <= e2	2	1	0	1	0	0
e1 to e2 by e3	1	0	21	1	21	0
exit()	1	0	0	0	0	0
e1 :=: e2	1	1	0	0	0	0
total	343352	336094	6293	985	21	6272

Expression Activity for iiencode.icn

name	calls	returns	suspends	failures	resumps	removals
e1[e2]	81343	81343	0	0	0	0
find()	70771	0	5303	65468	0	5303
?e	47869	47869	0	0	0	0
tab()	10608	0	10608	1	1	10607
e1 || e2	10568	10568	0	0	0	0
move()	5307	0	5306	1	0	5306
*e	5284	5284	0	0	0	0
e1 := e2	2864	2864	0	0	0	0
upto()	2812	0	2643	170	1	2642
\e	2643	0	0	2643	0	0
!e	2642	0	22882	0	20240	2642
write()	150	150	0	0	0	0
e1 == e2	24	2	0	22	0	0
read()	22	22	0	0	0	0
[...]	21	21	0	0	0	0
e1 to e2 by e3	20	0	169	20	169	0
put()	19	19	0	0	0	0
/e	3	1	0	2	0	0
integer()	2	1	0	1	0	0
table()	2	2	0	0	0	0
get()	1	0	0	1	0	0
any()	1	0	0	1	0	0
e1 < e2	1	1	0	0	0	0
pull()	1	0	0	1	0	0
total	242978	148147	46911	68331	20411	26500

Expression Activity for csgen.icn

use of string scanning clearly is much more extensive in this program than in iiencode.icn. In other words, csgen.icn is more "Icon-ish" than iiencode.icn.

Next Time

We've run out of space. In the next article in this series, we'll show a composite for expression activity in all 11 test programs and then go on to other aspects of program execution.

Subscription Renewal

For many of you, the next issue is the last in your present subscription to the Analyst and you'll find a subscription renewal form in the center of this issue. Renew now so that you won't miss an issue.

Your prompt renewal also helps us in planning and by reducing the number of follow-up notices we have to send.

What's Coming Up

We have several articles in the works — several on dynamic analysis, material in the Icon program library, one on integers and the design problems related to large-integer arithmetic, and the first of a series on building applications with visual interfaces. We also are planning a series of articles that go into some depth on the way things are implemented in Icon, especially how lists, sets, and tables are implemented.

We not yet sure what will appear in the next issue of the Analyst, but more on dynamic analysis is a good bet.