The Icon Analyst

In-Depth Coverage of the Icon Programming Language

April 1993
Number 17

In this issue …

Lost Languages — SL5

Background

As you probably know, Icon is only the latest in a series of programming languages. It all started in 1962 at Bell Labs with the development of a language called SNOBOL [1], which was designed to make it easier to manipulate symbolic information represented by strings of characters. There were three successive SNOBOL languages, culminating in 1968 with SNOBOL4 [2], which added sophisticated data structures.

Further work on this kind of programming language moved to The University of Arizona in 1971. This work first used SNOBOL4 as a basis for a number of experiments [3-5]. Eventually, the structure of SNOBOL4 became too limiting, which led to the design of an entirely new programming language called SL5 ("SNOBOL Language 5"). The name, although barely disguising its origins, was chosen to avoid the impression that the new language was just a variant of earlier SNOBOL languages. SL5 was, in fact, very different from its SNOBOL ancestors, although it contained many ideas derived from them.

If you're familiar with SNOBOL4, you'll see both differences and similarities between SNOBOL4 and SL5 in what follows. Whether or not you know SNOBOL4, you'll see what preceded and shaped Icon.

SL5 eventually became a large language in the sense of having many features and a large computational repertoire [6-10]. We won't attempt to cover all of SL5 here. Instead we'll focus on the highlights and those features that are most relevant to Icon.

Basic Features

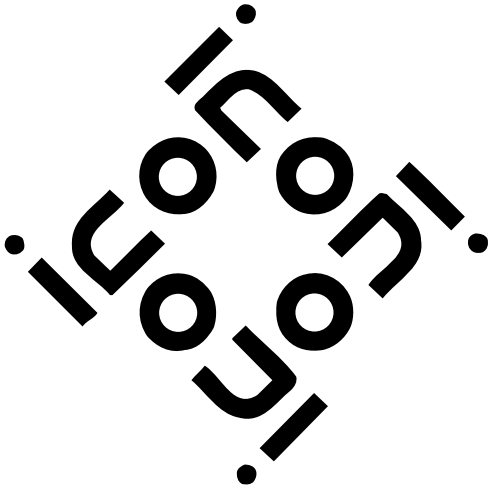
Most imperative programming languages (including SNOBOL4) have both expressions (that produce values) and statements (that do something but don't produce values). SL5, on the other hand, has only expressions, although some of them look like statements. Thus, in SL5

if *expr1* then *expr2* else *expr3*

is an expression and it produces a value (the value of *expr2* or *expr3*, depending on which is selected). Icon inherited this way of casting computation.

Like SNOBOL4, SL5 stresses run-time flexibility. Structures are created at run-time, the meaning of operations can be changed at run-time, and so on. Not surprisingly, SL5 also has automatic storage management with garbage collection to reclaim space when needed.

None of these features is unusual. The unusual features of SL5 are its method of controlling program flow, the way it deals with various aspects of procedures, and its approach to pattern matching.



Control Structures

In order to understand control structures in SL5, it is helpful to know how flow of control is handled in its predecessor. In SNOBOL4, the execution of a statement may succeed or fail, an idea that originated in COMIT [11]. SNOBOL4, however, does not have conventional control structures like if-then-else and while-do. Instead it has conditional gotos that allow program execution to be directed to different statements depending on the success or failure of the current one, as in

```
IDENT(X, Y)                :S(YES)F(NO)
```

In SNOBOL4, success and failure are termed *signals*.

These days gotos are deprecated because of the unstructured, confusing, and error-prone styles of programming that they allow. Furthermore, the absence of conventional control structures in SNOBOL4 makes it necessary to construct even simple loops using gotos and labeled statements.

One of the goals of SL5 was to support conventional control structures while preserving the useful concept of signals. In SL5, an expression produces a *result* that contains both a value and a signal. A result is constructed by the expression v&s, where v is the value and s is the signal.

Signals are nonnegative integers and are used in control structures to affect the flow of control. The signal 0 corresponds to failure and any positive signal (normally 1) corresponds to success. For example, the operation i > j produces the signal 1 if i is greater than j but the signal 0 otherwise. In

```
if i > j then expr1 else expr2
```

the signal produced by i > j determines whether *expr1* or *expr2* is evaluated. Thus, SL5 avoids Boolean values and allows any kind of expression to be a control expression — an idea that has been further refined in Icon.

SL5 allows a result to be composed from a value and any nonnegative integer signal. The signal component of a result also can be extracted to become a value.

Procedures

The most interesting aspect of SL5 is the way that it deals with programmer-defined procedures. Procedures are created at run-time, as illustrated by

```
gcd := procedure(i, j)
  while i ~= j do
    if i > j then i := i – j else j := j – i;
  return i
end;
```

Note that a procedure value is assigned to gcd. Such a procedure then can be called in the conventional manner, as in

```
k := gcd(47, 25);
```

Procedure invocation in most programming languages is an atomic operation as illustrated in this example. In SL5, however, procedure invocation can be decomposed into three separate operations:

- creation of an environment for the procedure
- binding of arguments to the environment
- "resuming" the environment to execute the code for the procedure

The expression

```
e := create p
```

creates an environment for the procedure p and assigns it to e. (The create operation here is not directly related to Icon's create for co-expressions.)

Arguments are bound (transmitted) to an environment by

```
e with (expr1, expr2, ..., exprn)
```

where *expr1*, *expr2*, …, *exprn* are evaluated and their values are bound to e. That is, they become the values of the formal parameters of p in e.

Finally, the execution of p is initiated with

```
resume e
```

All this is not necessary if all you want to do is call a procedure and get its result back. However, the decomposition of procedure invocation allows more sophisticated use of procedures, as illustrated by the following procedure:

```
genlab := procedure(prefix, i)
  repeat {
    return prefix || i;
    i := i + 1
    }
end;
```

This procedure might be used as follows:

```
label := create genlab with ("L", 1);
    ...
top := resume label;
    ...
bottom := resume label;
    ...
```

which assigns L1 to top and L2 to bottom, respectively, assuming there are no intermediate resumptions of label. The crucial point here is that when a procedure returns, its environment remains intact and it can be resumed to continue execution. New arguments can be passed to the environment, as in

```
label with ("P", 100);
```

to change the values produced on subsequent resumptions.

It's easy to see how SL5's resumption mechanism accomplishes what Icon does with suspend. SL5 is, however, more general than Icon — any suspended environment can be resumed at any time. SL5's return mechanism also is more general than Icon's. In SL5, it is not necessary for a procedure to return control to the procedure that resumed it. The general form of return in SL5 is

```
return r to e
```

where *r* is the result (value and signal) to be returned and control is transferred to e, which can be any environment. If you're familiar with coroutines, you'll recognize that SL5 provides a very general coroutine mechanism in which many procedure environments can be in existence at any one time and in which control can be transferred among them in any order.

A few more aspects of procedures in SL5 need to be mentioned before we go on to other parts of the language. As you've probably guessed from the examples above, if no signal is specified in return, 1 (for success) is supplied by default. There also are two abbreviations that make programs easier to read:

```
succeed v
```

is a synonym for

```
return v&1
```

and

```
fail v
```

is a synonym for

```
return v&0
```

In both cases, if v is omitted, the empty string is returned. (In SL5, the empty string serves a role similar to that of the null value in Icon.)

Identifiers in SL5 can be declared to be global, private, or public. The declaration global has the same meaning as it does in Icon; such identifiers are available throughout a program. Similarly, private is equivalent to local in Icon; such identifiers are available only within a single environment. The public declaration is more interesting. It provides the interpretation of undeclared identifiers. The meaning of an undeclared identifier is determined when an environment is created. This is done by searching for public identifiers in the ancestors of the environment in which the creation is done. An example that illustrates the usefulness of this kind of scoping is given in the section on data structures.

Filters

SL5 built on the SNOBOL4 idea of attaching procedural components to variables. The underlying idea is that it's often useful to have something done automatically when a value is assigned to a variable or when a variable is dereferenced to get its value.

This is particularly useful in reading and writing. When a value is assigned to out in SL5, that value also is written. For example,

```
out := "Hello world";
```

writes Hello world. Similarly, when the value of in is used, a new line of input is read and becomes the value of in. If there is no more input, the reference to in fails. Thus,

```
while out := in;
```

copies input to output.

In addition to such built-in associations, SL5 allows filters to be associated with variables in two ways: assignment and dereferencing. Filters are simply environments that are resumed automatically when a variable is referenced. A filter for dereferencing is associated with a variable by

```
v :- - e
```

and a filter is associated with a variable for assignment by

```
v :- e
```

As many filters as are needed can be attached

to a variable; they are processed in the order that they are attached.

Since filters are environments, filtering actions can be written using procedures. The arguments of a filtering procedure are the value and signal of the result being filtered. For example, to restrict the values assigned to count to positive integers, the following procedure could be used:

```
procedure posint(v, s)
   repeat
      if ident(datatype(v), "integer") and
         v > 0 then return v&s
      else return v&0
end;
      ...
count :-- create posint;
```

Positive integers pass through this filter unchanged. Other values are passed through with a failure signal. If the failure signal reaches the assignment, the assignment is not performed and the assignment expression fails. The procedure above reveals some other aspects of SL5; you should be able to figure out their meanings.

Filters also can be used in passing values to environments. The default argument transmission method is by value, as in Icon. Values also can be passed by reference. The method of transmission is specified in the procedure heading, as in

```
procedure p(i, x:ref)
```

in which i is passed by value and x is passed by reference.

Programmer-defined filters also can be used in argument transmission. For example, to assure that the procedure gcd given earlier is called only with positive integer values, all that's needed is

```
gcd := procedure(i:posint, j:posint)
   ...
```

String Scanning

String scanning in SL5 resembles both pattern matching in SNOBOL4 and string scanning in Icon. It is closer to SNOBOL4 in that SL5 environments, like SNOBOL4 patterns, are data objects that embody scanning operations, while in Icon, scanning operations are applied directly.

SL5 has a number of functions and operators that build scanning environments. For example, in

```
pet := ="dog" | ="cat"
```

the alternation operator creates an environment that matches either "cat" or "dog". Subsequently,

```
animal ? pet
```

applies this environment to animal.

Other environment constructors include

e1 -- e2	concatenation of e1 and e2
move(i)	move i characters
tab(i)	move to character i
break(s)	move up to character in s
span(s)	move past characters in s

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

The SNOBOL4 and Icon analogies should be obvious. The difference between SL5 and Icon is that in SL5, these expressions create environments; they do not perform any scanning. Scanning occurs when an environment is resumed in a scanning expression. Like patterns in SNOBOL4, however, environments can be combined to form more complex environments.

SL5 distinguishes between accepting strings during scanning and synthesizing strings as a result of scanning. Thus, move(i) creates an environment that accepts i characters, but it does not contribute to the result of scanning. The environment created by =move(i), on the other hand, contributes the characters accepted to the result of scanning.

SL5 has four public scanning variables:

subject	string being scanned
cursor	position in subject
scanlength	length of subject
scanvalue	current synthesized result

Since SL5 uses environments in string scanning, programmer-defined procedures can be used in scanning. An example is a procedure to pad the result of synthesis to a fixed number of columns:

```
synpad := procedure(i, c) private s;
  s := scanvalue;
  scanvalue := rpad(scanvalue, i, c);
  succeed;
  scanvalue := s;
  fail
end;
```

Note that the previous value of scanvalue is saved so that it can be restored if synpad() is resumed (as the result of the failure of a subsequent scanning environment).

Data Structures

It may not surprise you at this point to learn that SL5 also uses environments for programmer-defined data structures.

Records are easy, since the identifiers in an environment can be accessed as "fields" of the environment. For example, a procedure for use with employee records might be

```
employee := procedure(name, age, salary)
  end;
```

Then

```
hire := create employee;
```

creates an environment, and values can be assigned to its fields as in

```
hire.name := "John Smith";
hire.age := 35;
hire.salary := 32500.0;
```

It would be syntactically neater if this could be written as a procedure call:

```
hire := employee( "John Smith", 35, 325000.0)
```

However, since hire needs to be an environment, it is necessary for employee() to return its own environment. This is done using the keyword &self:

```
employee := procedure(name, age, salary)
  return &self
end;
```

To use environments for more complicated data structures, it usually is necessary to perform some initialization before returning &self. For the example above, an additional field might be added for the tax rate:

```
employee := procedure(name, age, salary)
  private taxrate;
  taxrate := rate(salary);
  return &self
end;
```

Stacks provide a more interesting example of a programmer-defined data structure:

```
stack := procedure private push, pop;
  public stk;
  push := create procedure(x)
    repeat {
      stk := node(x, stk);
      return x
      }
  end;
  pop := create procedure private t;
    repeat
      if ident(stk) then fail
      else {
        t := stk.value;
        stk := stk.link;
        return t
        }
  end;
  return &self
end;
```

Here a stack is composed of a linked list of nodes, which are records with two fields, value and link.

A stack is created by

```
parse := stack();
```

The fields push and pop are procedures, so that

```
parse.push(x)
```

pushes x onto parse, and

```
x := parse.pop();
```

pops a value off of parse and assigns that value to x unless parse is empty, in which case parse.pop() fails. Note that the public identifier stk in stack() provides the interpretation for the undeclared identifiers stk in pop and push, so that they apply to the stack for parse.

Conclusions

Writing this article has been an interesting experience for us. It's been years since we've given a serious thought to SL5, and we had to forage through old documents to produce the description we've given here.

In retrospect, SL5 has many interesting features. (We've only described the main ones here; there's a lot more to SL5). Not surprisingly, on rediscovering some of the features of SL5, we thought "Gee, those would be neat things to have in Icon." But Icon has grown large. If there's anything we think language designers should do is avoid the temptation to "kitchen sink" (to coin an abominable verb from a noun).

Not everything in SL5 is good. It's very general procedure mechanism is appealing, but it's also somewhat overbearing and, because of its generality, inefficient.

For better or worse, SL5 was abandoned in favor of Icon, simply because we did not have the resources to support two languages. And SL5 really is dead — or so we think. It was implemented only for the DEC-10 and CDC 6000 and we no longer have the code. But who knows what's on a dusty tape in some forgotten cabinet?

Although SL5 is gone, it served a useful purpose. It led to Icon and has influenced the design of other programming languages. The most notable of these is EZ [12], a very high-level language with a persistent memory that permits it to double as a programming environment.

Acknowledgments

SL5 was a substantial project, and many persons contributed to its design and implementation. The principal contributors were Diane Britton, Fred Druseikis, Ralph Griswold, Dave Hanson, and Tim Korb.

References

1. "SNOBOL, A String Manipulation Language", *Journal of the ACM*, David J. Farber, Ralph E. Griswold, and Ivan P. Polonsky, Vol. 11, No. 1 (January, 1964), pp. 21-30.

2. *The SNOBOL4 Programming Language*, second edition, Ralph E. Griswold, James F. Poage, and Ivan P. Polonsky, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1971.

3. *Advanced Data Structure Manipulation Facilities for the SNOBOL4 Programming Language*, John C. Hallyburton, Jr., Ph.D. dissertation, Department of Computer Science, The University of Arizona, Tucson, Arizona, 1974.

4. *An Extended Function Definition Facility for SNOBOL4*, Frederick C. Druseikis and Ralph E. Griswold, technical report S4D36, Department of Computer Science, The University of Arizona, Tucson, Arizona, 1973.

5. A Generalized Facility for the Analysis and Synthesis of Strings, and a Procedure-Based Model of an Implementation, John N. Doyle, Master's thesis, Department of Computer Science, The University of Arizona, Tucson, Arizona. 1975.

6. "String Analysis and Synthesis in SL5", Ralph E. Griswold, *Proceedings of the ACM Annual Conference*, 1976, pp. 410-414.

7. *Procedure-Based Linguistic Mechanisms in Programming Languages*, David R. Hanson, Ph.D. dissertation, Department of Computer Science, The University of Arizona, Tucson, Arizona. 1976.

8. "The SL5 Procedure Mechanism", David R. Hanson and Ralph E. Griswold, *Communications of the ACM*, Vol. 21, No. 5 (May, 1978), pp. 392-400.

9. "Filters in SL5", David R. Hanson, *The Computer Journal*, Vol. 21, No. 2 (May, 1978), pp. 134-143.

10. "Data Structures in SL5", David R. Hanson, *Computer Languages*, Vol. 3, No. 3 (October, 1978), pp. 181-192.

11. "A Programming Language for Mechanical Translation", V. H. Yngve, *Mechanical Translation*, Vol. 5, No. 1 (1958), pp. 25-41.

12. "*EZ* Processes", David R. Hanson and Makoto Kobayashi, *International Conference on Computer Languages*, 1990, pp. 90-97.

Drawing in X-Icon

We covered the basic aspects of X-Icon in an earlier article in the Analyst [1]. Since that time, X-Icon has matured and there are now implementations for VMS and OS/2 under Presentation Manager in addition to the former ones for UNIX.

As interest in X-Icon has increased, we've received requests to say more about it in the Analyst. This is the first of a series of articles designed to cover all the features of X-Icon. These articles are descriptive and tutorial in nature. If there's enough interest, we'll have articles of a more technical nature later. On the other hand, we realize that many of you can't run X-Icon, so we'll be careful not to let X-Icon dominate the Analyst. (This is the longest of the articles we have planned.)

There's a new technical report on X-Icon [2]. As a subscriber to the Analyst, you can get a free copy of this report for the asking. Be sure to identify yourself as a Analyst subscriber, because we can't make this offer to everyone.

A word to OS/2 users: Although X-Icon has essentially the same basic functionality under Presentation Manager as it does on X platforms, there are some differences. In this and subsequent articles, the description is based on X. See Reference 2 for information about OS/2 differences.

The Subject Window

One addition we've made to X-Icon since the earlier article is support for a *subject window*, inspired by the way that the way the subject of string scanning is used to simplify string analysis operations. If the value of &window is a window, as in

```
&window := open("game board", "x") |
   stop("*** cannot open window")
```

then the window argument in many X-Icon functions can be omitted, in which case &window is assumed to be the window for the operations. For example,

```
XFg("red")
```

changes the foreground color for &window to red.

Omission of the window argument is optional:

```
XFg(canvas, "red")
```

sets the foreground color for the window canvas.

Now on to a description of drawing in X-Icon.

Points

The function XDrawPoint() exemplifies the drawing functions. It has the form

```
XDrawPoint(window, x1, y1, ..., xn, yn)
```

where window is the window in which points at x,y coordinates x1, y1, ... xn, yn are drawn.

As mentioned above, the window argument can be omitted, in which case it defaults to &window. In the description that follows, we'll omit this argument, since the use of the &window default simplifies programming and is good practice in most situations.

As indicated above, XDrawPoint() allows an arbitrary number of arguments, in this case x,y coordinate pairs. Other drawing functions also allow multiple arguments, so that multiple drawings of the same type can be done with a single function call. To simplify the presentation that follows, we won't show multiple sets of arguments unless they are needed in examples, but it's worth keeping the possibility in mind.

Lines

The function XDrawLine(x1, y1, x2, y2) draws a line from the first x,y coordinate to the second. As indicated above, multiple lines can be drawn in one call by adding extra argument pairs, in which case the lines are connected. For example,

```
XDrawLine(
  100, 75,
  100, 150,
  200, 150,
  200, 75
  )
```

produces the result shown in Figure 1.

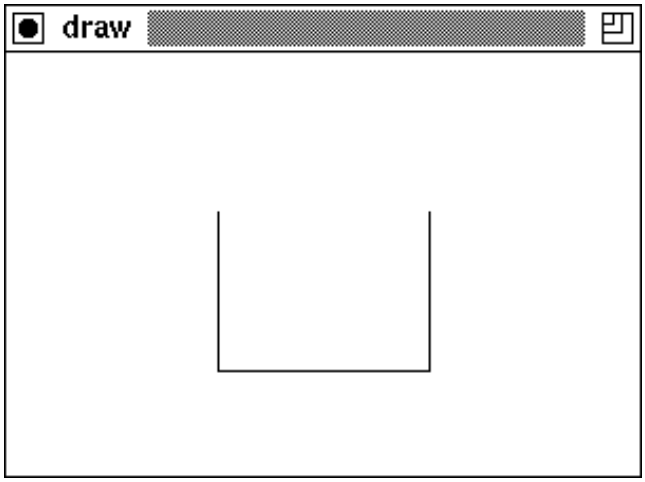


Figure 1. XDrawLine()

XDrawSegment() is similar to XDrawLIne(), but separate lines are drawn between each set of coordinate pairs. Thus,

```
XDrawSegment(
  100, 75,
  100, 150,
  200, 150,
  200, 75
  )
```

produces the result shown in Figure 2.

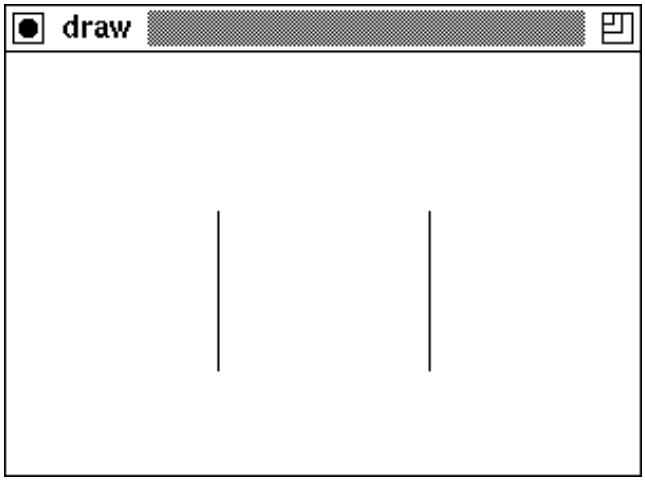


Figure 2. XDrawSegment()

XDrawCurve() draws a smooth curve connecting every point specified in its argument list. For example,

```
XDrawCurve(
  100, 75,
  100, 150,
  200, 150,
  200, 75,
  150, 50
  )
```

produces the result shown in Figure 3.

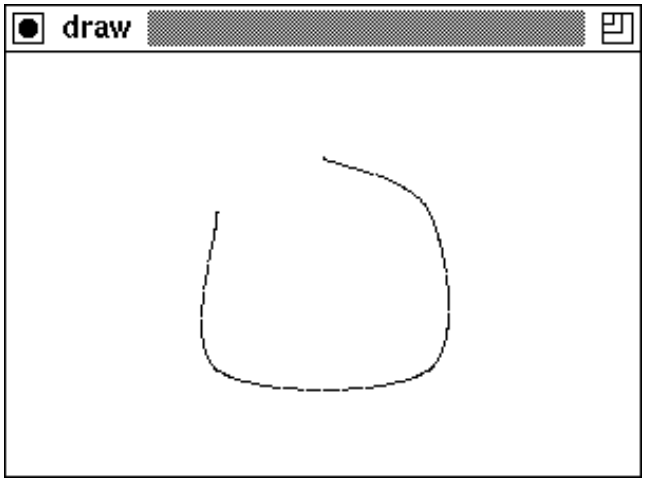


Figure 3. XDrawCurve()

If the first and last points are the same, the curve is smooth and connected through that point. For example,

```
XDrawCurve(
  100, 75,
  100, 150,
  200, 150,
  200, 75,
  150, 50,
  100, 75
  )
```

produces the result shown in Figure 4.

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

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

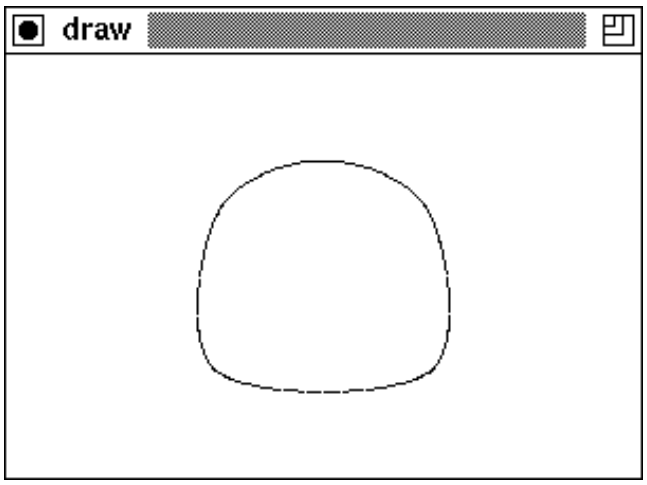


Figure 4. XDrawCurve()

X-Icon uses Catmull-Rom splines for drawing curves. If you're interested in the method, see Reference 3.

Rectangles

The function XDrawRectangle(x, y, w, h) draws a rectangle in the foreground color whose upper-left corner is at x and y and whose width and height are w and h, respectively. The width and height determine the *perceived* size of the rectangle; the actual dimensions are w + 1 and h + 1. For example,

```
XDrawRectangle(100, 75, 100, 50)
```

produces the result shown in Figure 5.

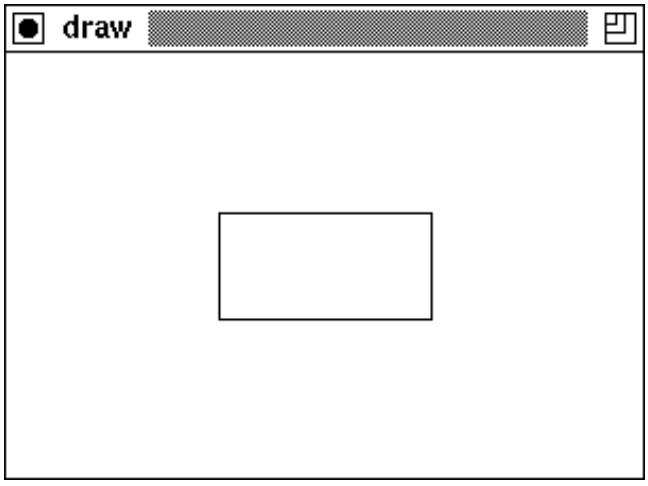


Figure 5. XDrawRectangle()

The function XFillRectangle() is similar to XDrawRectangle(), except that the interior of the rectangle is filled with the foreground color. For example,

```
XFillRectangle(100, 75, 100, 50)
```

produces the result shown in Figure 6.

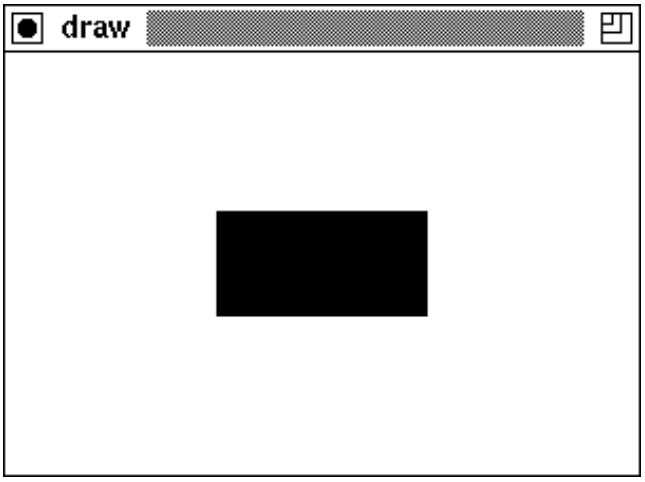


Figure 6. XFillRectangle()

As with other drawing functions, multiple rectangles can be drawn by providing four additional arguments for each additional rectangle.

Polygons

Polygons can be drawn with the function XDrawLine(), being sure that the first and last points are the same. For example,

```
XDrawLine(
  100, 75,
  100, 150,
  200, 150,
  100, 75
  )
```

produces the result shown in Figure 7.

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.

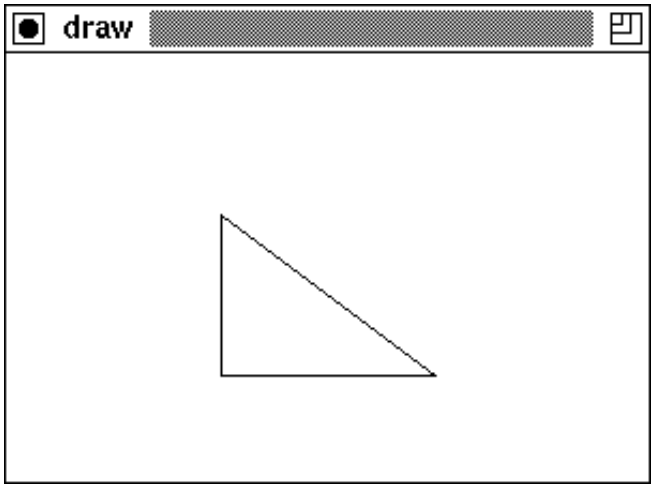


Figure 7. Polygon with XDrawLine()

The function XFillPolygon() produces a filled polygon. The beginning and ending points are connected if they are not the same. For example,

```
XFillPolygon(
  100, 75,
  100, 150,
  200, 150
  )
```

produces the result shown in Figure 8.

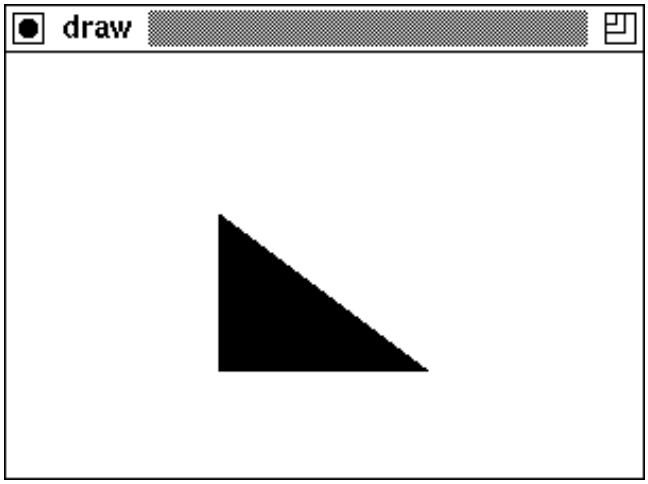


Figure 8. XFillPolygon()

Arcs and Circles

XDrawArc(x, y, w, h, start, extent) draws an arc bounded by the rectangle specified in the first four arguments. The center point of the bounding rectangle is x + w / 2, y + h / 2. The argument start is the starting angle of the arc. The starting angle 0 is at 3 o'clock. The argument extent is the extent of the arc (not the ending angle). Positive values for angles are in the counter-clockwise direction. Angles are specified in 64ths of a degree.

For example,

```
XDrawArc(100, 75, 50, 50,
  90 * 64,
  270 * 64
  )
```

produces the result shown in Figure 9.

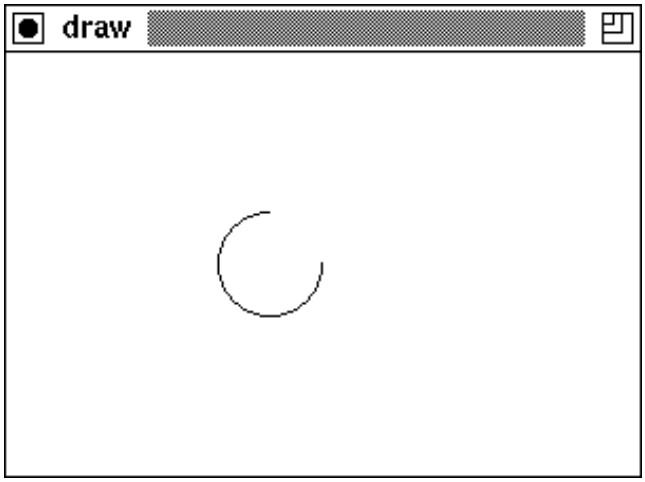


Figure 9. XDrawArc()

If h is omitted, it defaults to w, producing a circular arc. If h and w are different, the result is not a true elliptical arc, but rather a "squashed" circular arc (at least on our X servers).

If start is omitted, it defaults to 0. If extent is omitted, it defaults to 64 * 360, producing a complete circle.

XFillArc() is like XDrawArc(), except the arc is filled with the foreground color.

Line Attributes

The default line width is one pixel. A different line width can be set when a window is opened or subsequently changed using the linewidth attribute. For example,

```
XAttrib("linewidth=3")
```

sets the line width for subsequent drawing to three pixels.

The attribute linestyle determines the style of

lines. The default line style is "solid", as shown in preceding figures. The line style "doubledash" produces a dashed line. For example,

```
XAttrib("linewidth=2")
XAttrib("linestyle=doubledash")
XDrawRectangle(100, 75, 100, 50)
```

produces the result shown in Figure 10.

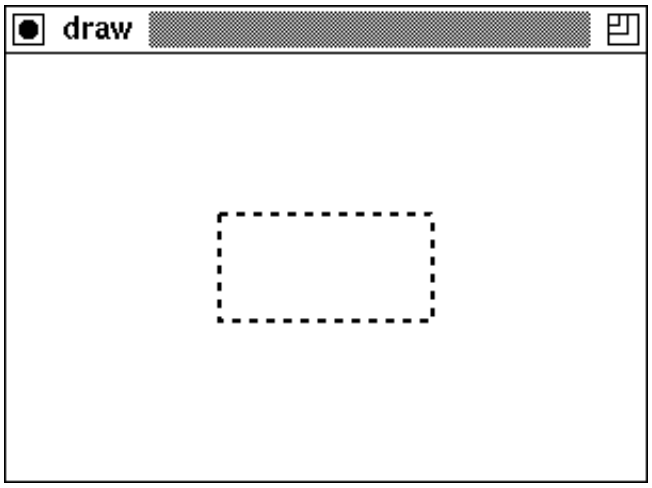


Figure 10. A Dashed Rectangle

The line style "onoff" is similar to "doubledash", except that with "onoff" the background part of the line is not drawn.

Note: Line attributes do not apply to XDrawCurve(), which always draws a one-pixel solid line.

Fill Attributes

The attribute fillstyle determines the way that shapes are filled. The default fill style is "solid", as illustrated in Figure 6. There are two other fill styles, "stippled" and "opaquestippled".

A stipple is a repeating bit pattern used as a mask when filling shapes. The function

```
XSetStipple(w, i1, i2, ... in)
```

defines a stipple pattern of width w, which must be between 1 and 32, inclusive. The least significant w bits of each subsequent integer argument are interpreted as a row in the pattern. For example,

```
XSetStipple(7, 0, 1, 3, 7, 15, 31, 63)
XAttrib("fillstyle=stippled")
XFillRectangle(100, 75, 100, 50)
```

produces the result shown in Figure 11.

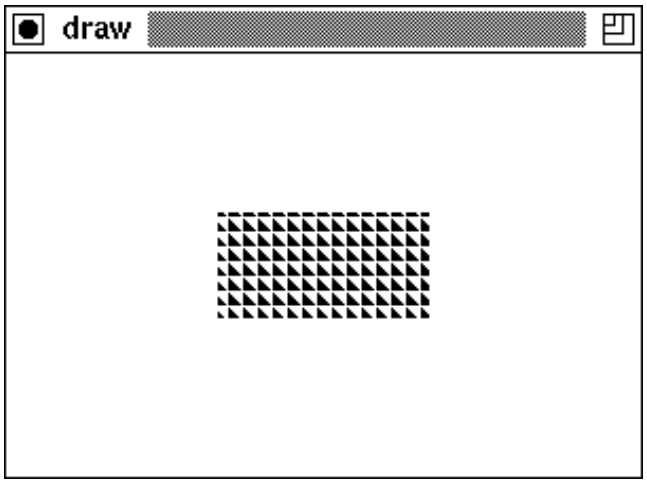


Figure 11. A Stippled Fill

With the fill attribute "stippled", fills are done only for those bits in the stipple pattern that are set to 1. With the fill attribute "opaquestippled", stipple pattern bits that are one are set in the foreground color, while those that are zero are set in the background color.

Drawing Operations

At the pixel level, all drawing operations combine some *source bits* (bits to be drawn) with some *destination bits* (bits presently in the window). By default, source bits overwrite destination bits.

Other logical combinations of source and destination bits are possible — 16 in all. See Reference 2. These combinations are specified by name as values of the drawop attribute. The default value of drawop is "copy", in which case source bits replace destination bits. Drawing operations other than "copy" are potentially unportable or even undefined and should be used only with a clear understanding of the X color model.

In addition to the standard 16 drawing operations, there is a special one, "reverse", that allows reversible drawing. If the drawop attribute is set to "reverse", drawing changes the pixels that are in the foreground color to the background color, and vice-versa. The color drawn on destination pixels that are neither the foreground or the background color is undefined, but in any event, drawing a pixel the second time restores the pixel to its original color.

For example,

```
XAttrib("drawop=reverse")
every x := 1 to 100 do {
   XFillRectangle(x, 100, 10, 20)
   XFlush()
   delay(1)
   XFillRectangle(x, 100, 10, 20)
   }
XFillRectangle(x, 100, 10, 20)
```

moves a small rectangle horizontally across the screen, leaving an image only at the end.

Argument Lists

Sometimes the number of arguments for a drawing function is not known when a program is written. This might happen, for example, when plotting a sequence of points or drawing a polygon with an arbitrary number of sides.

Icon's list-invocation facility can be useful in this case. For example, instead of using

```
XFillPolygon(x1, y1, ...)
```

the arguments could be put onto a list, as in

```
poly := []
every put(poly, xygen())
```

and the polygon drawn using

```
XFillPolygon ! poly
```

Next Time

The next article on X-Icon will deal with text — its file model of windows that allows text to be written to a window as if the window were a terminal, a few things about fonts, and how you can write text to a window so that you can erase it and leave the rest of the window contents unchanged.

Reference

1. "An Introduction to X-Icon", The Icon Analyst 13, pp. 5-10.

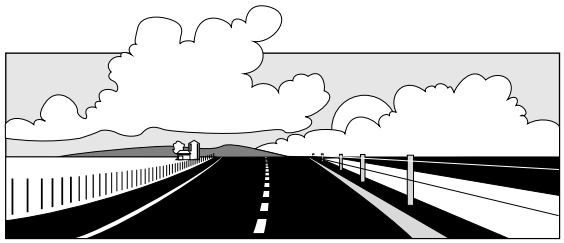
2. *X-Icon: An Icon Window Interface; Version 2*, Clinton L. Jeffery, technical report TR 92-26, Department of Computer Science, The University of Arizona, 1992.

3. "A Recursive Evaluation Algorithm for a Class of Catmull-Rom Splines", *Computer Graphics*, Vol. 22, No. 4 (August, 1988), pp. 199-204.

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 manage our resources.



What's Coming Up

In the next issue of the Analyst, we'll have an article on Rebus, another "lost language" and continue with the series on X-Icon with the article on dealing with text that we mentioned earlier.

We'll also have the first of two articles on the anatomy of a program for timing expression evaluation in Icon.