The Icon Analyst

In-Depth Coverage of the Icon Programming Language

October 1995
Number 32

In this issue …

Equivalent Versum Sequences

As we mentioned in the last article on versum sequences, different seeds may produce the same sequence. Seeds that produce the same sequences are said to be equivalent. Equivalence is important if you're exploring versum sequences, since some computations, such as producing many terms in a sequence, are time-consuming. Unnecessary computation for equivalent seeds is well worth avoiding.

In determining equivalent seeds, it is useful to deal with *n*-digit seeds for different values of *n*, rather than, say, dealing with all seeds in a numerical range, such as 1 to 999. There are seeds with different numbers of digits that have equivalent sequences. An example is 209 and 1100. There are, however, few such equivalences and we'll restrict ourselves to equivalent *n*-digit seeds. This allows a clean characterization, as we'll show.

It's useful to designate a representative seed from a class of equivalent ones. The obvious choice is the smallest one, which we call the *primary seed* for the class.

We can find primary seeds easily enough with a brute-force approach. All we have to do is run though all *n*-digit seeds in numerical order and keep track of the first terms in their versum sequences. If the first term is a new one, the seed is a primary one; otherwise it's not.

In the spirit of Icon, here's a procedure that generates the primary seeds for a given *n*:

```
procedure pvseeds(n)
   local first, last, first_terms, i, j

   first := 10 ^ (n – 1)
   last := (10 ^ n) – 1

   first_terms := set()

   every i := first to last do {
      j := i + reverse(i)
      if not member(first_terms, j) then {
         insert(first_terms, j)
         suspend i
         }
      }

end
```

It doesn't take much more to keep track of equivalent seeds and generate lists of them:

```
procedure eqvseeds(n)
   local first, last, first_terms, primaries, i, j

   first := 10 ^ (n – 1)
   last := (10 ^ n) – 1

   first_terms := set()
   primaries := table()

   every i := first to last do {
      j := i + reverse(i)
      if not member(first_terms, j) then {
         insert(first_terms, j)
         primaries[j] := [i]
         }
      else put(primaries[j], i)
      }

   primaries := sort(primaries, 3)

   while get(primaries) do         # discard keys
      suspend get(primaries)       # produce list

end
```

Notice that this procedure does not generate any lists until all have been computed.

Here's what the lists look like for *n* = 2.

```
[10]
[11,20]
[12,21,30]
[13,22,31,40]
[14,23,32,41,50]
[15,24,33,42,51,60]
[16,25,34,43,52,61,70]
[17,26,35,44,53,62,71,80]
[18,27,36,45,54,63,72,81,90]
[19,28,37,46,55,64,73,82,91]
[29,38,47,56,65,74,83,92]
[39,48,57,66,75,84,93]
[49,58,67,76,85,94]
[59,68,77,86,95]
[69,78,87,96]
[79,88,97]
[89,98]
[99]
```

If the nature of equivalent seeds wasn't already evident, this example should be suggestive. If we can understand why seeds are equivalent, we can surely do much better than using the brute-force approach, which quickly becomes impractical as *n* gets large.

For *n* = 1, it's obvious by inspection that all the seeds produce different sequences, so all 1-digit seeds are primary.

For *n* = 2, we've already shown the primary seeds and their equivalents. Consider the seed 19. Adding one to its first digit and subtracting one from its last produces a equivalent seed: 19 + 91 = 100 and 28 + 82 = 100. We can continue this process to get the equivalent seeds 19, 28, 37, 46, 55, 64, 73, 82, and 91. We have to stop there because another step would produce a 3-digit number. If we start with 18, on the other hand, we get the equivalent seeds 18, 27, 36, 45, 54, 63, 72, 81, and 90, stopping there because the next step would produce a negative number for the last digit. This approach gives a systematic procedure for producing the 2-digit seeds and their equivalents.

Icon on the Web

Information about Icon is available on the World Wide Web at

http://www.cs.arizona.edu/icon/www/

You can treat this as a filter, if you like. Write down all 2-digit seeds in numerical order, start with the first, compute its equivalents (there aren't any for the first seed), cross them out, and continue with the next seed that hasn't been crossed out, and so on until you run out of seeds that haven't been crossed out. This process ensures that you'll get all the primary seeds and their equivalents and that there's nothing missing.

The result is 18 primary 2-digit seeds, as shown by the first terms in the lists shown in the left column. There are 90 2-digit seeds altogether, but it's only necessary to perform computations on 20% of them. But what about 3-digit seeds, 4-digit seeds, and *n*-digit seeds in general?

We can compute the primary *n*-digit seeds recursively ("when in doubt, recurse"). The method depends on whether *n* is odd or even.

If *n* is odd, we take the primary seeds for *n*–1 (which is even), split them in the middle, and insert 0 and all the 1-digit primary seeds in each. For example, for *n* = 3, we can take all the primary seeds for n = 2

```
10
11
12
…
19
…
89
99
```

and insert in the middle of each 0 and all the 1-digit primary seeds:

```
0
1
…
9
```

Thus, the seed 10 produces

```
100
110
120
…
190
```

and so on.

Note that every (*n*–1)-digit primary seed produces 10 *n*-digit primary seeds. Thus, there are 180 3-digit primary seeds.

If *n* is even, on the other hand, we take the primary seeds for *n*–2 (which also is even), split them in half, and insert 00 and all the 2-digit primary seeds:

```
00
10
11
…
99
```

Thus we get 19 n-digit seeds for every n–2 digit primary seed.

Here's a recursive generator that produces the n-digit primary seeds:

```
procedure pvseeds(n)
  local i, lpart, rpart, h

  if n = 1 then suspend 1 to 9
  else if n = 2 then
    suspend (10 to 19) | (29 to 99 by 10)
  else if n % 2 = 0 then {                # even
    h := (n –2) / 2
    every i := pvseeds(n – 2) do {
      i ? {
        lpart := move(h)
        rpart := tab(0)
        }
      suspend integer(lpart || ("00" | pvseeds(2)) ||
        rpart)
      }
    }
  else {                                  # odd
    h := (n –1) / 2
    every i := pvseeds(n – 1) do {
      i ? {
        lpart := move(h)
        rpart := tab(0)
        }
     suspend integer(lpart || (0 | pvseeds(1)) || rpart)
      }
    }

end
```

This procedure illustrates the combination of string processing and arithmetic that is needed in dealing with versum sequences. Since pvseeds() calls itself recursively, most results are used as strings. We've chosen, for the sake of clarity, to use integer values rather than string values and let automatic type conversion produce strings as needed. An alternative would have been to cast values as strings. For example,

```
suspend 1 to 9
```

could have been written

```
suspend !"123456789"
```

or to use mixed integer and string values to minimize type conversion.

Note that we didn't carry the focus on integer values to the point of replacing "00" by (0 || 0). We were tempted, though.

It does make a difference what pvseeds() returns at the top level. A program that calls pvseeds() may expect an integer value and certainly not an integer in one case and a string in another. That's why the results of concatenation are converted to integers before returning them — even though it means more type conversion in intermediate results. Another way of dealing with this kind of situation is shown later in this article.

From the arguments above, we easily can calculate the number of n-digit primary seeds, which we designate by $P(n)$:

$$P(1) = 9$$
$$P(2) = 18$$
$$P(n) = 10 \times P(n\text{-}1) \qquad n > 2,\ \text{odd}$$
$$P(n) = 19 \times P(n–2) \qquad n > 2,\ \text{even}$$

which has the closed form

$$P(1) = 9$$
$$P(2) = 18$$
$$P(n) = 180 \times 19^{(n-3)/2} \qquad n > 2,\ \text{odd}$$
$$P(n) = 18 \times 19^{(n/2)-1} \qquad n > 2,\ \text{even}$$

We prefer the recursive formulation, which reveals the structure of the construction procedure; the closed form only hints at it.

A simpler recursive formulation is

$$P(1) = 9$$
$$P(2) = 18$$
$$P(3) = 180$$
$$P(n) = 19 \times P(n–2) \qquad n > 3$$

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

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

Can you find a method for producing primary seeds that directly relates to this formulation?

Here's an enumeration of the number of primary *n*-digit seeds and their ratio to the number of all *n*-digit seeds ($9 \times 10^{n-1}$) for the first few values of *n*:

n	P(*n*)	*ratio*
1	9	1.00000
2	18	0.20000
3	180	0.20000
4	342	0.03800
5	3420	0.03800
6	6498	0.00722
7	64980	0.00722
8	123426	0.00137

It's obviously very worthwhile to compute versum sequences only for primary seeds.

In order to deal with an arbitrary seed, we need procedures to produce the primary seed given any seed in its equivalence class:

```
procedure vprimary(s)

  return integer(vprimary_(s, 1))

end

procedure vprimary_(s, low)
  local h, mpart, lpart, rpart

  if *s < 2 then return s
  else if s = 0 then return s
  else {
    s ? {
      lpart := tab(2)
      mpart := tab(–1)
      rpart := tab(0)
      until (lpart = low) | (rpart = 9) do {
        lpart –:= 1
        rpart +:= 1
        }
      return lpart || vprimary_(mpart, 0) || rpart
      }
    }
end
```

The procedure vprimary() is a wrapper to assure the values returned are integers. The procedure vprimary_() does the actual computation, which deals with strings. The wrapper avoids unnecessary type conversion during recursive computation.

Note that although 0 is not a versum seed, it may appear in the interior of seeds in the form of strings like "0", "00", "000", …. That's the reason for

```
if s = 0 then return s
```

before going to the general case. The numeric comparison automatically converts strings of zeros to the integer 0.

The second argument to vprimary_() determines whether the left digit goes to 1 (for the final result) or to zero (for intermediate results that

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

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

and

Bright Forest Publishers
Tucson Arizona

eventually appear in the interior of the final result).

Before going on, here are procedures to generate all the seeds in the equivalence class for a primary seed (including the primary seed itself):

```
procedure eqvseeds(i)

   suspend integer(eqvseeds_(vprimary(i)))

end


procedure eqvseeds_(s)
   local mpart, lpart, rpart

   if *s < 2 then return s
   else if s = 0 then return s
   else {
      s ? {
         lpart := tab(2)
         mpart := tab(–1)
         rpart := tab(0)
         until (lpart > 9) | (rpart < 0) do {
            suspend lpart || eqvseeds_(mpart) || rpart
            lpart +:= 1
            rpart –:= 1
            }
         }
      }

end
```

Again, the procedure `eqvseeds()` is a wrapper to assure the values returned are integers.

In the last article we showed a program for plotting the palindromes for seeds 1 through 999. Here's that procedure again for reference:

```
link wopen

procedure main()
   local i, x, input, line

   WOpen("canvas=hidden", "size=300,999") |
      stop("*** cannot open window")

   every i := 1 to 999 do {
      input := open(i || ".pal") |
         stop("*** cannot open file for seed ", i)
      while line := read(input) do {
         line ? {
            x := tab(many(&digits)) – 1
            }
         DrawPoint(x, i – 1)
         }
      close(input)
      }

   WriteImage("palimage.gif")

end
```

For comparison, here's a program that does the plot using equivalent seeds:

```
link    eqvseeds
link    pvseeds
link    wopen

procedure main()
   local primary, equiv, input, x, line

   WOpen("canvas=hidden", "size=300,999") |
       stop("*** cannot open window")

   every primary := pvseeds(1 to 3) do {
      input := open(name := primary || ".pal") |
         stop("*** cannot open ", name)

      while line := read(input) do {
         line ? {
            x := tab(many(&digits)) – 1
            }
         every DrawPoint(x, eqvseeds(primary))
         }
      close(input)
      }

   WriteImage("palimage.gif")

end
```

This program only requires palindrome files for primary seeds and is considerably faster than the former one. The plot develops in a different order from the former one, but the final result is the same.

Until now, when dealing with versum sequences, it was sufficient to read a file whose name corresponded to the seeds, as in

```
 input := open(i || ".vsq") | …
 while term:= read() do
   process(term)
```

This is a situation in which a generator simplifies the code:

```
 input := open(i || ".vsq") | …
 every process(!input)
```

Now, however, we don't have a file for every seed. It's simple enough to fix that:

```
 input := open(vprimary(i) || ".vsq") | …
```

An alternative approach, which will prove useful later, is to provide an envelope to hide the details:

```
 link vprimary

 procedure vsterm(i)
   static input

   close(\input)

   input := open(vprimary(i) || ".vsq") |
     stop("*** cannot find versum sequence for ", i)

   suspend !input

   close(input)

 end
```

You may wonder about

```
 close(\input)
```

at the beginning, since vsterm() closes input at the end. The problem is that there is no guarantee that vsterm() will get to the end, as is illustrated by

```
 every write(vsterm(1001) \ 100)
```

It's for this reason that vsterm() closes input before opening it. (The nonnull test takes care of the first time vsterm() is called.) This assures that at most one file will be left open by the use of vsterm(). Incidentally, closing a closed file is not an error.

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.

Next Time

When we started this article, we thought we could finish up versum sequences in this issue of the Analyst. Instead, we kept finding new and interesting problems and results.

Rather than fill up this issue with versum sequences, we'll continue in subsequent issues.

Vidgets

In the last issue of the Analyst, we described Icon's interface tools. These tools are what the user sees and uses to communicate with a program that has a visual interface.

When the user presses a button, selects an item from a menu, or activates some other interface tool, a corresponding procedure in the program is called. Such procedures are called callbacks — they are called as a result of user actions, not by the program itself.

Icon's interface tools are called vidgets (virtual interface devices). The word is a pun on "widget" and is used to avoid confusion with Athena widgets [1].

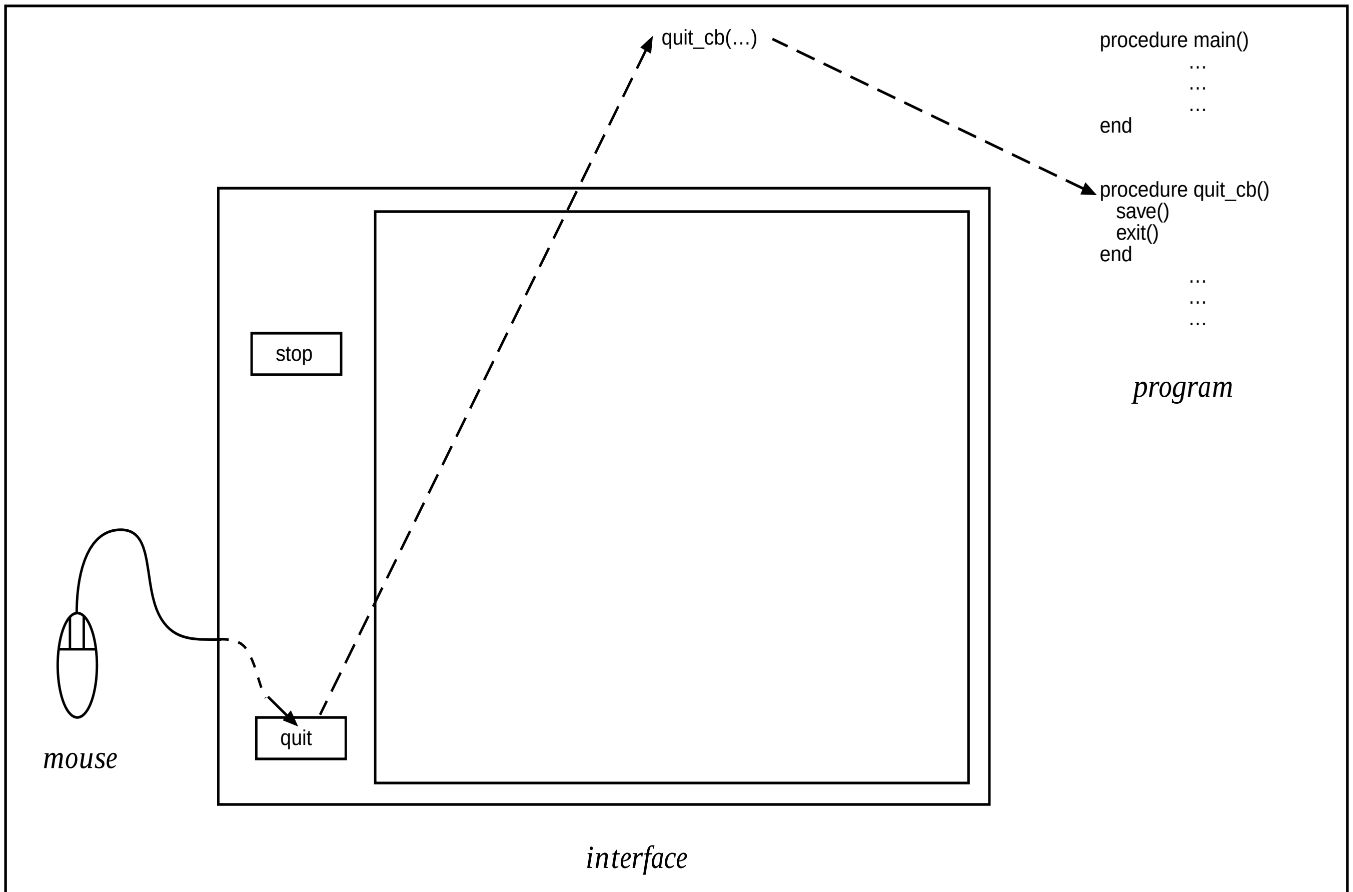
Vidgets are implemented by Icon records. A vidget record contains information about the vidget: an identifying name, its type (such as button), its location and size, and so on.

All vidgets on an interface are enclosed within a "root" vidget. The root vidget accepts user events (such as mouse presses) and identifies the vidget, if any, on which the mouse cursor is positioned. If the mouse cursor is on a vidget when an event occurs, that vidget is activated. For example, if a mouse button is pressed with the cursor on a slider vidget, the callback for that vidget is called. If the event is not appropriate for that vidget (such as a keypress on a button), it is rejected by the vidget. See the diagram at the top of the next page.

Callbacks from vidgets have the form:

```
 cb(vidget, value)
```

The first argument identifies the vidget that produced the callback and the second argument gives a value. The vidget argument is not always needed, but it can be used to distinguish among different vidgets that share the same callback procedure. The value often is important, since in many cases it



User Interaction with an Application Through a Visual Interface

indicates the nature of the user action.

For a toggle button, the value is null when the toggle is turned off and 1 (nonnull) when it is turned on. This makes testing of the state of a toggle easy, as in

```
procedure pause_cb(vidget, value)

   if \value then …                # stop display
   else …                          # continue display

   return

end
```

The callback value for a radio button is the (string) label of the selected button, as in

```
procedure shape_cb(vidget, value)

   case value of {
      "disks":    …                # filled circle
      "rings":    …                # outline circle
     }

    return

end
```

Since menus can have submenus, their callbacks are lists whose first element is the text for the item selected from the main menu, whose second element is the text for the item selected from the first submenu, and so on. If there are no submenus, the callback value for a menu is a one-element list, as illustrated by

```
procedure file_cb(vidget, value)

   case value[1] of {
      "snapshot @S" :  …       # take snapshot
      "quit        @Q":  …     # shut down
      }

   return

end
```

Notice that the list element is the complete text for the item selected.

The callback value for a text-entry field is the text in the field at the time the user types return with the I-beam cursor in the field. There is no callback until the user types return.

The callback value for a slider or scroll bar is the numerical value in the given range, as determined by the position of the thumb. A slider or scroll bar can be configured in two ways: to provide callbacks as the user moves the thumb, or "filtered" to provide a callback only when the user releases the thumb. Filtering is appropriate when only the final value is important, as in

```
procedure density_cb(vidget, value)
   density := value      # set global variable
   return
end
```

Unfiltered callbacks may be needed when the application needs to respond as the user moves the thumb, as in scrolling an image.

The callback for a region is somewhat different from the callbacks for other vidgets. The second argument is the event produced by the user and there are two additional arguments that indicate where on the application canvas (not the region) the event occurred:

```
cb(vidget, e, x, y)
```

Note in particular that e is not a value associated with the region vidget, as it is for other kinds of vidgets; it is the actual event, such as a mouse press or a character from the keyboard.

Labels and lines do not produce callbacks; they provide decoration only.

Vidget States

Toggle buttons, radio buttons, text-entry fields, sliders, and scroll bars maintain internal states. The state of such a vidget is the same as the last callback value it produced.

Since the callback values and the states are the same, it usually is not necessary to ascertain the state of a vidget. If it is necessary, the procedure

```
VGetState(vidget)
```

produces the state.

The procedure

```
VSetState(vidget, value)
```

sets the state of the vidget to the given value. It does this by producing a callback, as if the user had produced it through the interface. For example, VSetState() can be used to set the state of a slider and move its thumb to the corresponding position.

Vidget Fields

Vidgets have a number of fields that contain attributes. Most of these fields are used for internal purposes, but some provide useful information, such as the location and size of a vidget on the interface canvas. Except for lines, vidgets occupy a rectangular area and have these fields:

vidget.ax	x coordinate of the upper-left corner of the vidget
vidget.ay	y coordinate of the upper-left corner of the vidget
vidget.aw	width of the vidget
vidget.ah	height of the vidget

Regions also have fields that give the "usable" area that can be drawn on without overwriting borders used for three-dimensional effects:

vidget.ux	x coordinate of the upper-left corner of the usable area
vidget.uy	y coordinate of the upper-left corner of the usable area
vidget.uw	width of the usable area
vidget.uh	height of the usable area

Next Time

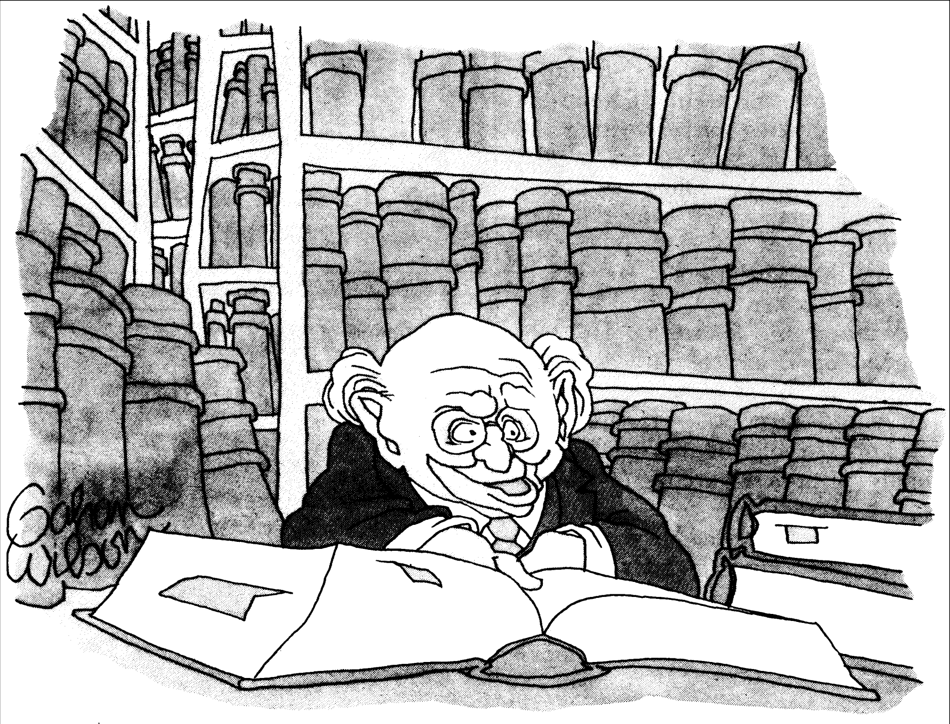
At this point, the concepts of interface tools and their implementation by vidgets must seem somewhat abstract. We haven't even told you how to place a vidget on an interface.

In the next article on visual interfaces, we'll start to describe how to build interfaces, using the kaleidoscope application from the previous article as an example.

The key to building interfaces easily is a visual interface builder that lets you create, place, configure, move, and delete vidgets easily. And, of course, it allows you to do it in a visual manner, directly manipulating images of the interface tools you want.

Reference

1. *Volume 4: X Toolkit Intrinsics Programming Manual: Standard Edition*, third edition, Adrian Nye and Tim O'Reilly, O'Reilly & Associates, Inc., Sebastopol, California, 1993.


"By God, for a minute there it suddenly all made sense!'

Drawing by Gahan Wilson; © 1986
The New Yorker Magazine, Inc.

From the Library

Editors' notes: The cartoon above reminds us of the state of the Icon program library. We hope you enjoy it as much as we do.

We've chosen what we consider to be the most useful procedure in the library to celebrate the cartoon.

Command-Line Arguments

When Icon is run from the command line, arguments are passed to the main procedure in the form of a list of strings, one string for each argument. This is the main way in which information is passed to a program that is run from the command line. For example, if a program that is named plot begins with

```
procedure main(args)

   shape := args[1]
   bound := args[2]
   points := args[3]
      …
```

and plot is called as

```
plot lemniscate 10.0 1000
```

shape is set to "lemniscate" and bound is set to "10.0", and points is set to "1000". A more sophisticated program might issue an error message for an inappropriate value, convert the second and third arguments to real and integer, respectively, and provide defaults for omitted arguments.

Of course, command-line arguments can be used in any way you like. The use above has the disadvantages that the arguments must be in a fixed order and there's only a hint in a call of what they mean.

The standard format that is used by the Icon program library identifies options by name, with a prefix – and follows the name by a value, if any. The program plot then might be called as

```
plot –s lemniscate –b 10.0 –p 1000
```

In this form, the options can be given in any order and carry identification. (Multi-character option names can be used also; we'll come to that, but we'll stick to one-character names for now.)

It's not that hard to write a preamble to a program to handle named options. That's not necessary, however — the procedure options() in the Icon program library takes care of almost everything you might want.

Using options()

options(args, opts) processes command-line options in the list args according to the specifications given in the string opts. It returns a table with the option names as keys and with corresponding values from the command line.

Using options(), the program plot might start as follows:

```
link options

procedure main(args)

   opt_tbl := options(args, "s:b.p+")
   shape := opt_tbl["s"]
   bound := opt_tbl["b"]
```

```
    points := opt_tbl["p"]
       …
```

The option string consists of letters for the option names followed by a type flag. The flag ":" indicates the option value must be a string, "." indicates a real number, and "+" indicates an integer.

If an option appears on the command line, its value in the table is the result of converting to the specified type. Otherwise, it's the null value.

An option that does not take a value can be specified also. In this case, no type flag is specified. If such an option is given on the command line, its value in the table returned by options() is 1 (and hence nonnull); otherwise it's null. An example is

```
  link options

  procedure main(args)

    opt_tbl := options(args, "s:b.p+t")
    shape := opt_tbl["s"]
    bound := opt_tbl["b"]
    points := opt_tbl["p"]
    if \opt_tbl["t"] then &trace := –1
       …
```

Here, the command line option –t turns on tracing in plot.

A test for a table value being null can be used to set defaults, as in

```
  link options

  procedure main(args)

    opt_tbl := options(args, "s:b.p+t")
    shape := \opt_tbl["s"] | "circle"
    bound := \opt_tbl["b"] | 1.0
    points := \opt_tbl["p"] | 100
    if \opt_tbl["t"] then &trace := –1
       …
```

Multi-character option names are supported. They must be preceded in the option string by a – to distinguish them from single-character option names.

For the example we've been using, this might take the form

```
  link options

  procedure main(args)

    opt_tbl := options(args,
      "–shape:–bound.–points+–ttrace")
    shape := opt_tbl["shape"] | "circle"
    bound := opt_tbl["bound"] | 1.0
    points := opt_tbl["points"] | 100
    if \opt_tbl["t"] then &trace := –1
       …
```

where a command-line call might be

```
  plot –shape lemniscate –bound 10.0 –point 1000
```

Many other features are supported by options(). The most important ones are:

• Options can appear in any order in the options string and on the command line.

• Blanks between single-character option names and the corresponding values are optional on the command line.

• If a command-line argument begins with an @, the subsequent string is taken to be the name of a file that contains options, one per line.

• options() removes option names and their values from the argument list, leaving anything else for subsequent processing by the program.

• The special argument –– terminates option processing, leaving the remaining values in the argument list.

• options() normally terminates with a run-time error if an option value cannot be converted to the specified type or if there is an unrecognized option on the command line.

• If a third procedure-valued argument is supplied in a call of options(), that procedure is called in case of an error instead of terminating execution.

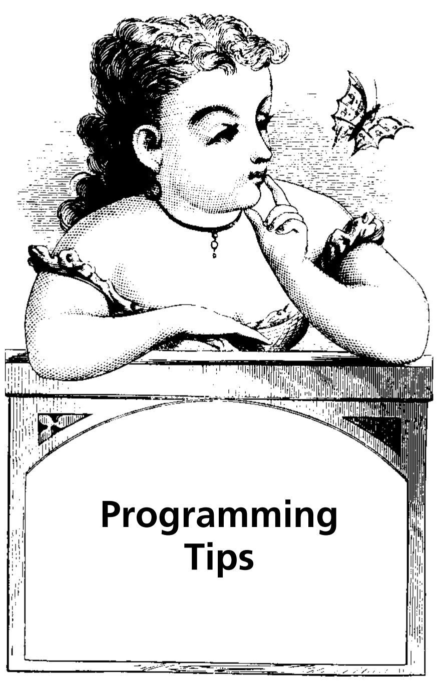
Conclusion

Some effort is required to become familiar with options(), and some effort is required to incorporate it in a program. The results are well worth it. Not only will your programs conform to the standard used in the Icon program library, but all the details of processing and error checking will be taken care of for you.

A beneficial side-effect is that once the option mechanism is part of a program, it's easy to add new options and increase program functionality. In fact, the use of the options mechanism often suggests useful functionality.

Acknowledgments

The procedure options() was conceived and implemented by Bob Alexander. Gregg Townsend contributed additional features.



Keeping Track of Structures

Icon's data structures provide convenient ways for organizing and accessing collections of values. Since Icon data structures are created at run-time in Icon and are first-class values, there is more flexibility during program execution than in most programming languages.

More programmers probably use Icon for its data structures than for any other reason. Nonetheless, data structures often cause problems in program design and development: which kind of structure to use for a particular task (such as manipulating graphs), how much space they require, how fast access is, and so on.

Questions related to these matters are among the most frequent ones that we get. Unfortunately, there rarely are simple answers, although we've made some suggestions and given some size and speed comparisons among alternatives in previous issues of the Analyst. We'll continue to do this in future issues.

The ideal way to understand structures would be a visualization of program execution that showed how many structures there are, how big they are, how they are related, and so forth. This is a difficult problem. An Icon program may create thousands of data structures, they may be large in size, and the number of interconnections between them can be enormous. Finding ways to represent such information visually in a manner that is useful is in itself a daunting task. The design and implementation of such a tool remains in our "job jar".

In this programming tip, we'll describe something much simpler, but which provides information about structure usage that may help you decide among alternatives and get a better understanding of what's going on in programs that use a lot of structures.

Suppose, for example, that you suspect a program is creating more lists than you think it should be. Or suppose you want to know how many sets are being created for representing a large directed graph.

If you have a handle on the most recently created structure, this information is easy to get. Every structure has a serial number. Each kind of structure and each different record type has a separate sequence of serial numbers that start with 1 and is incremented as new structures are created.

The serial number of a structure can be obtained from its string image. For example, if the value of coordinates is a list,

```
write(image(coordinates))
```

might produce something like

```
list_34(507)
```

The type appears at the beginning, in this case indicating a list. The serial number follows a separating colon. In this case, the value of coordinates is the thirty-fourth list created since the beginning of program execution. The value in parentheses is the number of list elements.

That's fine if you can identify the last list created, but if lists are created by procedures (even library procedures), you may have no way of knowing, at a particular point in program execution, what the most recently created list is.

If you're willing to create another list to find out how many have been created so far, all you have to do is

```
write(image([ ]))
```

The newly created list shows a serial number one greater than the last list created. Such expressions can be used for other kinds of structures.

Getting the serial number is easy, but a "helper" procedure is worth having:

```
procedure serial(x)

   image(x) ? {
      tab(integer(upto('_')) + 1) | fail
      return integer(tab(many(&digits)))
      }
   end
```

Note that this procedure may give incorrect results if given an inappropriate value like the string "whatever_3". There are ways of making this procedure more robust, which we'll leave to you.

You might wonder why there isn't a built-in function to do this. Whenever there's a possibility for a new function, we have to weigh its usefulness against the increase in the size of Icon's computational repertoire, the size of Icon's run-time system, the additional documentation, and so forth. In this case, we decided that a function serial() would not be used often enough to justify its being built-in — especially since it's so easy to write in Icon. However, we do have a built-in version that we use for work on dynamic program analysis and visualization.

Now for a bit of wizardry. Here's a procedure that gives the serial number of the last structure of a specified type:

```
   procedure created(type)
      return serial(proc(type)()) – 1
   end
```

For example, created("table") gives the serial number of the most recently created table.

proc(type) converts the type name to a function that creates structures of that type. For example, proc("list") produces the function list. This function is then applied to an empty argument list to create a structure. (If type is not the name of a structure type, "bad things" may happen. Again, we'll leave it to you to patch things up. Take care, though: It's not trivial.) The resulting serial number is decremented to give the number of the last previously created structure of that type. Since the newly created structure is not assigned to a variable, its existence is transient and it can be garbage collected.

Cautions:

• created(type) increments the serial number of that type as a side effect.

• The serial number of the most recently created structure tells you nothing about how many structures of that type still exist. Some, even all, may have been garbage collected.

There's also a small wrinkle in the serial number for lists. If the main procedure has an argument, as in

```
   procedure main(args)
```

a list for args is automatically created for command-line arguments and has the serial number 1. If you want to take this into account,

```
   args(main)
```

returns the number of arguments for the procedure main. If it's greater than zero, a list was created before program execution began.

What's Coming Up

We'd planned to have another article on the dynamic analysis of Icon programs in this issue of the Analyst. Between moving to a new platform and some unexpected difficulties, we didn't make it. We'll try once again for the next issue.

In the next issue of the Analyst, we'll continue with versum sequences, this time looking at versum sequences that are not equivalent but merge at some point. Using mergers, we'll show that the amount of data needed to represent versum sequences can be reduced dramatically.

We'll also continue our series of articles on programs with visual interfaces, describing how to design and build interfaces.