The Icon Analyst

In-Depth Coverage of the Icon Programming Language

February 1997
Number 40

In this issue ...

Dialogs

Dialogs are temporary windows that provide a way for an application to notify its user of situations that require attention or action. They also provide a way to request information from its user, such as the name for a file in which to save data.

Icon provides two kinds of dialogs: standard ones, which handle common situations, and custom dialogs built by VIB, which can be tailored for specific uses. In this article, we'll cover the standard dialogs, which are produced by calling procedures in Icon's graphics repertoire.

Notification Dialogs

Situations often occur in which a user needs to be alerted to a condition before a program continues.

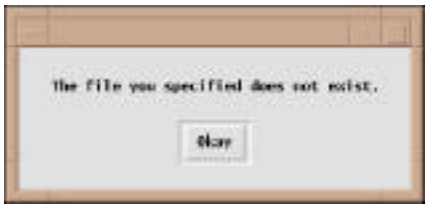
The procedure

```
Notice(line1, line2, ... )
```

produces a dialog with the strings line1, line2 ,.... For example,

```
Notice("The file you specified does not exist.")
```

produces the following dialog:



The application then waits for the user to respond. When the user clicks on the Okay button to dismiss the dialog, it disappears, and program execution continues. Typing a return character also dismisses the dialog.

File Name Dialogs

Opening files and saving data are such common operations that dialog procedures are provided for querying for file names. The procedure

```
OpenDialog(caption, filename, length)
```

produces a dialog that allows the user to specify the name of a file to be opened. The argument caption, which defaults to "Open:" if not given, appears at the top of the dialog. A text-entry field appears below where the user can enter the name of the file to open. The argument filename provides the string used to initialize the text-entry field. It defaults to the empty string for the common case where there is no meaningful file name. The argument length specifies the the number of characters that the text-entry field can accomodate and has a default value of 50. For example,

```
OpenDialog()
```

produces the following dialog:

The user can type in the text-entry field. An "I-beam" text cursor shows the current location in the field where typed text is inserted. This cursor can be positioned in the text by clicking with the mouse pointer at the desired location. Dragging over the characters in the text field selects them for editing and highlights them (reversing the foreground and background colors). Characters that are typed then replace the selected ones. A backspace character deletes the character immediately to the left of the text cursor, if there is one. All this sounds complicated, but as in many interactive operations, it becomes natural in practice, and it is easier to do than it is to describe.

The following window shows the dialog after a file name has been entered in the text-entry field.

The procedure OpenDialog(), like all dialog procedures, returns the string name of the button selected. Before returning, it assigns the string in the text-entry field to the global variable dialog_value. A typical use of OpenDialog() is

```
repeat {
   case OpenDialog() of {
      "Okay":    {
         if input := open(dialog_value) then {
            current_file := dialog_value       # save name
            data_list := []
            while put(data_list, read(input))  # get data
            close(input)
            return data_list
            }
         else Notice("Cannot open file.")
         }
      "Cancel":  fail
      }
   }
```

If the user selects Okay (or types a return character), the specified file is opened and the data in it is read into a list. If the file cannot be opened, however, the user is notified and a new dialog is presented for the user to try again. The procedure fails without trying to open a file if the user selects Cancel.

The procedure

```
SaveDialog(caption, filename)
```

is used to provide a dialog for saving data to a file. The argument caption, if omitted, defaults to "Save:". Providing a file name often saves typing when the user has opened a file, modified data within the application, and wants to update the file with the modified information. An example is

```
SaveDialog(, current_file)
```

which might produce the dialog shown below.

One use of SaveDialog() is to check whether the user wants to save modified data before quitting an application. Typical code to do this is

```
repeat {
   case SaveDialog("Save before quitting?",
      current_file) of {
         "Yes":       {
            if output := open(dialog_value, "w") then {
               every write(output, !data_list)
               exit()
               }
            else Notice("Cannot open file for writing")
            }
         "No":        exit()
         "Cancel":    fail
      }
   }
```

If the user elects to save the current data, it is written to the specified file and program execution is terminated. If the file cannot be opened for writing, however, the user is notified and the process is repeated with a new dialog box. If the user selects No, program execution is terminated without saving any data. If the user selects Cancel, perhaps because of second thoughts about quitting the application, the procedure fails and program execution continues.

Text Dialogs

Notice(), OpenDialog(), and SaveDialog() are special cases of a more general procedure, TextDialog(), which allows customized dialogs for text entry. TextDialog(), in its most general usage, is rather complicated because dialogs can be complicated. Defaults are provided, however, to make TextDialog() easy to use if all its generality is not needed. The general form is:

```
TextDialog(captions, labels, defaults, widths,
    buttons, index)
```

The argument captions is a list of caption lines that serve the same purpose as the multiple arguments in Notice(). The arguments labels, defaults, and widths are lists that give the details of a sequence of text-entry fields. The argument buttons is a list of buttons and index is the index in buttons of the default button.

TextDialog() returns the name of the button that was selected to dismiss the dialog. The global variable dialog_value is assigned a list containing the values of the text fields at the time the dialog was dismissed.

Unlike the dialogs that were described previously, TextDialog() provides for labels that appear before text-entry fields to identify them. Each field can have a default value and a width to accommodate a specified number of characters.

Here's an example of the most general kind of usage:

```
TextDialog(
  ["To open a window, fill in the values",
    "and click on the Okay button:"],
  ["xpos", "ypos", "width", "height"],
  [10, 10, 500, 300], [4, 4, 4, 4], ["Okay", "No"], 1
  )
```

The dialog produced by this call is shown below.

If there is only one caption line, it can be given as a string instead of a list. If there is only one text-entry field, the specifications for it also can be given as single values instead of lists. In the case where there are several fields that all have the same value, a single value can be given for that argument in place of a list. If there are no labels or defaults for fields, these arguments can be omitted altogether. The default field width, if its argument is omitted, is 10.

If the buttons argument is omitted, Okay and Cancel buttons are provided. If no button index is given, the first button is the default button. An index of 0 indicates that there is no default button.

An example of the use of defaults is:

```
TextDialog("Open window:", ["x", "y", "w", "h"])
```

which produces the dialog shown below.

In a dialog that has more than one text-entry field, the text cursor indicates the field in which text can be entered and edited. The text cursor initially is in the first field. Typing a tab moves the text cursor to the next field. From the last field, it moves to the first. A specific field also can be selected by clicking on it. Pressing return or clicking on a button dismisses the dialog.

Selection Dialogs

The procedure SelectDialog() allows the user to pick one of several choices. Its general form is

```
SelectDialog(captions, choices, dflt, buttons, index)
```

The arguments captions, buttons, and index serve the same purpose that they do in TextDialog(). The argument choices is a list of choices from which the user can select. The argument dflt is the default choice, which is highlighted in the dialog. The defaults for omitted arguments follow the same rules as the defaults for TextDialog(). The user's choice is returned as a string in dialog_value.

The following procedure call illustrates the use of SelectDialog().

```
SelectDialog(
  "Pick a suit:",
  ["spades", "hearts", "diamonds", "clubs"],
  "hearts"
  )
```

The dialog produced by this call is shown below.

Toggle Dialogs

The procedure ToggleDialog() allows the user to set several toggles (on/off states) at the same time in one dialog. Its general form is

```
ToggleDialog(captions, toggles, states, buttons,
  index)
```

The arguments captions, buttons, and index serve the same purpose as they do in TextDialog() and SelectDialog(). The argument toggles is a list of toggle names and the argument states is a list of their respective states: 1 for on, null for off. The defaults for omitted arguments follow the same rules as for TextDialog() and SelectDialog(). A list of the states of the toggles that the user chooses is returned in dialog_value.

The following procedure call illustrates the use of ToggleDialog().

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.

```
ToggleDialog(
  "Controls:",
  ["generate report", "stop on bad data", "trace"],
  [, , 1]
  )
```

The dialog produced by this call is shown below.

Color Dialogs

The procedure ColorDialog() allows the user pick a color interactively using either the RGB or HSV color model. Its general form is

```
ColorDialog(captions, color, callback, value)
```

The argument captions serves the same purpose as it does in preceding dialog procedures. The optional argument color is a reference color that is displayed at the bottom of a rectangular area where color is displayed. The initial color for the rest of the rectangle is color, if provided, otherwise the current foreground color. The optional argument callback is a procedure that is called whenever the user adjusts the color setting. The procedure is called as

```
callback(value, setting)
```

where setting is the current color setting and value is the final argument, otherwise unused, from the ColorValue() call. Thus, an application can track changes in the color setting, and value can be used to pass along an arbitrary value to the caller of ColorValue(). The final setting is returned as a string in dialog_value.

The following procedure call illustrates the use of ColorDialog().

```
ColorDialog("Pick a color, any color", "black")
```

The dialog produced by this call is shown on the next page.

Next Time

In the next article on dailogs, we'll describe how to build dialogs that contain other vidgets and that can be arranged to suit your needs.

Debugging: Error Messages

Most programmers find debugging to be unpleasant. It is frustrating, it's generally not a creative activity, and often the only reward comes at the end in the form of relief. But unlike other important but unpleasant aspects of programming, such as documentation and thorough testing, debugging can't be ignored.

In looking over past issues of the Analyst, we found a few articles that touched on debugging, but only one short article directly related to debugging [1]. And it appeared in the first year of publication. Well, we don't enjoy debugging either. It is, however, important, so we're starting a series of articles on debugging in Icon.

Our plan is to discuss error messages first, then the built-in diagnostic facilities Icon provides, followed by debugging support available in the Icon program library, and ending with a description of itweak, Håkan Söderström's interactive Icon debugger. Of course, other things may come up in the meantime.

Errors can be divided into two classes: those that Icon detects (something "illegal") and those that Icon doesn't ("legal" but wrong). The latter kind of error usually is more difficult to deal with, since the only symptom may be incorrect results.

In this article, we'll start with the errors Icon detects: the messages, how to interpret them, and some suggestions for avoiding and correcting mistakes detected by Icon.

General Rule: Believe an error detected by Icon, even if the source of the problem is not obvious. Don't jump to the conclusion that the error is the result of a bug in Icon. Although there certainly are bugs in Icon, Icon has been used for many years by many programmers, and the bugs likely to produce error messages have been systematically eliminated. (It's the bugs that don't produce error messages that cause the most trouble.) Possible exceptions are noted in the discussion that follows.

Icon detects four kinds of errors, corresponding to the phases in processing an Icon program:

errors in preprocessor directives
syntax errors
linking errors
run-time errors

All error messages are written to standard error output.

Preprocessor Error Messages

When the preprocessor detects an error in a preprocessor directive, it produces a message giving the file name, the line number in that file, and then a # followed by a message describing the error. An error in a preprocessor directive does not terminate preprocessing, but it prevents compilation.

The error messages produced by the preprocessor generally are self-explanatory:

"*string*": cannot open
"*string*": circular include
"*string*": explicit $error
"*string*": extraneous arguments on $else/$endif
"*string*": invalid preprocessing directive
"*string*": value redefined
$define: "(" after name requires preceding space
$define: missing name
$define: unterminated literal
$ifdef/$ifndef: missing name
$ifdef/$ifndef: too many arguments
$include: invalid file name
$include: too many arguments
$line: invalid file name
$line: no line number
$line: too many arguments

```
$undef: missing name
$undef: too many arguments
unexpected $else
unexpected $endif
unterminated $if
```

In the first six messages, *string* denotes the offending string.

Two items deserve note:

• It is possible to produce a syntax error using a valid symbol definition in a way that otherwise would be syntactically correct. An example is

```
$define J  j+

procedure main()
        ...
  write(J)
        ...
```

It is, of course, generally bad practice to define incomplete expressions.

• File names must be enclosed in quotation marks unless they satisfy the syntax for Icon identifiers. Thus,

```
$include symbols
```

is legal, but

```
$include symbols.icn
```

is not. The latter directive produces the somewhat confusing error message


```
$include: too many arguments
```

but the source of the problem should be clear.

It is good practice to enclose all file names in quotation marks even if they are not necessary.

Messages for Syntactic Errors

When the compiler detects a syntactic error, it produces a message in the same form as the preprocessor. A syntactic error does not terminate compilation, but it prevents linking.

Icon has a rich syntax and there are many possible ways to make syntactic errors in a program. Some messages are self-explanatory, but others may not be. There are two main reasons why the messages for syntactic errors may confusing:

• The error messages reflect the grammar of Icon, not necessarily the way Icon programmers think.

• A mistake may not produce an actual syntax error until considerably farther on. Syntax errors are detected by the end of the procedure (or other declaration) in which they occur, but the distance from the mistake and the actual error may be significant.

The messages for syntax errors are:

```
end of file expected
global, record, or procedure declaration expected
invalid argument list
invalid by clause
invalid case clause
invalid case control expression
invalid create expression
invalid declaration
invalid default clause
invalid do clause
invalid else clause
invalid every control expression
invalid field name
invalid global declaration
invalid if control expression
invalid initial expression
invalid keyword construction
invalid local declaration
invalid argument
invalid argument for unary operator
invalid argument in alternation
invalid argument in assignment
invalid argument in augmented assignment
invalid repeat expression
invalid section
```

```
invalid then clause
invalid to clause
invalid until control expression
invalid while control expression
link list expected
missing colon
missing colon or ampersand
missing end
missing field list in record declaration
missing identifier
missing left brace
missing link file name
missing of
missing parameter list in procedure declaration
missing procedure name
missing record name
missing right brace
missing right brace or semicolon
missing right bracket
missing right bracket or ampersand
missing right parenthesis
missing semicolon
missing semicolon or operator
missing then
syntax error
```

One of the most common syntactic errors results from a missing left (opening) brace or parenthesis. Consider the following procedure:

```
procedure parse()                                  # 1
   every 1 to count do                             # 2
      s := goal                                    # 3
      repeat {                                     # 4
         if not upto(&ucase,s) then break          # 5
         if not(s ? replace(!xlist)) then break next  # 6
         until s ?:= replace(?xlist)               # 7
         }                                         # 8
      write(s)                                     # 9
      }                                            # 10
end                                                # 11
```

Here the error is a missing left (opening) brace following the do on line 2. The subsequent code is syntatically correct until the end of the procedure is reached. At that point, the message is

```
… Line 10 # "}": missing semicolon or operator
```

This certainly isn't the most informative message you can imagine for this situation, but there often is not a good choice for an error that can occur for a variety of reasons. Nonetheless, with a little experience you will recognize the likely cause by looking at line 10 and realizing an earlier left brace is missing.

A missing right (closing) brace may not be evident until the end of the procedure either. Here a right brace should appear after line 7:

```
procedure parse()                                  # 1
   every 1 to count do {                           # 2
      s := goal                                    # 3
      repeat {                                     # 4
         if not upto(&ucase,s) then break          # 5
         if not(s ? replace(!xlist)) then break next  # 6
         until s ?:= replace(?xlist)               # 7
      write(s)                                     # 8
      }                                            # 9
end                                                # 10
```

As in the previous example, a syntax error does not occur until the end of the procedure, although the message is different:

```
… Line 10 # "}": "end": invalid expression
```

Again, the message might be phrased better, but the cause of the error, if not its exact location, should be clear. Using a consistent indentation style makes this kind of error easier to locate.

Error Message during Linking

There is only one kind of error that can occur during linking:

```
inconsistent redeclaration
```

Such an error is the result of more than one procedure or record declaration with the same name (duplicate global declarations are allowed). The source of such an error usually is easy to find and fix, but note that the problem may occur because a declaration name in a linked library module duplicates one in the program that links the module.

Run-Time Error Messages

Errors occur during program execution generally are the most difficult to diagnose and correct.

An error during program execution produces an error message, a traceback of the procedure calls leading to the place of the error, and the offending expression. Run-time error messages are numbered and divided into categories, depending on the nature of the error:

Category 1: Invalid Type or Form

101 integer expected or out of range
102 numeric expected
103 string expected
104 cset expected
105 file expected
106 procedure or integer expected
107 record expected
108 list expected
109 string or file expected
110 string or list expected
111 variable expected
112 invalid type to size operation
113 invalid type to random operation
114 invalid type to subscript operation
115 structure expected
116 invalid type to element generator
117 missing main procedure
118 co-expression expected
119 set expected
120 two csets or two sets expected
121 function not supported
122 set or table expected
123 invalid type
124 table expected
125 list, record, or set expected
126 list or record expected
140 window expected
141 program terminated by window manager
142 attempt to read/write on closed window
143 malformed event queue
144 window system error
145 bad window attribute
146 incorrect number of arguments to drawing function

Category 2: Invalid Value or Computation

201 division by zero
202 remaindering by zero
203 integer overflow
204 real overflow, underflow, or division by zero
205 invalid value
206 negative first argument to real exponentiation
207 invalid field name
208 second and third arguments to map of unequal length
209 invalid second argument to open
210 non-ascending arguments to detab/entab
211 by value equal to zero
212 attempt to read file not open for reading
213 attempt to write file not open for writing
214 input/output error
215 attempt to refresh &main
216 external function not found

Category 3: Capacity Exceeded

301 evaluation stack overflow
302 memory violation
303 inadequate space for evaluation stack
304 inadequate space in qualifier list
305 inadequate space for static allocation
306 inadequate space in string region
307 inadequate space in block region
308 system stack overflow in co-expression
316 interpreter stack too large
318 co–expression stack too large

Category 4: Feature Not Implemented

401 co–expressions not implemented

Category 5: Programmer-Specified Error

500 program malfunction

Errors 140 through 146 relate to Icon's graphics facilities. While the rule "believe the error message" still applies, graphics facilities are the latest addition to Icon and are the most likely to have bugs that could manifest themselves as spurious run-time errors. But start by believing.

Most of the run-time error messages accurately describe the corresponding errors, but in some cases you may need to have more than a superficial knowledge of Icon to interpret them.

The following error messages deserve note:

- 301 evaluation stack overflow

Evaluation stack overflow usually occurs because of runaway recursion.

- 302 memory violation

Although this message accurately reflects what happened, the cause is not at all obvious. It sounds like Icon tried to access memory illegally, but the most probable cause is system (C) stack overflow. When this happens, an attempt is made to write an error message, which causes a memory violation because the stack already had overflowed. The text of this message has been left as it is, in case there really is a situation in which Icon itself accesses memory illegally. Incidentally, system stack overflow is most likely to occur during garbage collection when the routine

that marks blocks recursively encounters an unusually long chain of pointers.

- 500 program malfunction

This error message does not mean a malfunction in Icon. Rather it is provided for the use of programmers to force a run-time error through runerr() in case an Icon program detects a malfunction in its own code.

Next Time

In the next article on debugging, we'll discuss Icon's built-in diagnostic facilities: error traceback, procedure and co-expression tracing, display(), and termination dumps.

Reference

1. "Gedanken Debugging", Icon Analyst 5, pp. 10-11.

Factors of Versum Numbers

Oh no! Not another article on versum numbers!

It probably seems like this series of articles will never end. But one (versum) thing leads to another, and here we are again, this time looking at the factors of versum numbers.

We're trying a somewhat different format for the presentation in the hope it will make the material more interesting.

Recall that a versum number results from the addition of a number and its digit reversal. For example, for 100, 100 + 001 = 101. A versum sequence results from starting with a *seed* and continuing the reverse-addition process. Thus the seed 100 produces the sequence 101, 202, 404, 808, 1616, … .

I remember that; get on with it.

Okay. In getting information for this article, we looked at a lot of versum numbers (or rather, our programs did). For reference in the following discussion, we looked at two sets of versum numbers:

B: the first 1,000 terms in the versum sequences for 1- through 8-digit base seeds [1]. There are 8,760 of these seeds, so B contains 8,760,000 versum numbers.

N: All 1- through 10-digit versum numbers. There are 2,541,196 of these.

Isn't there a lot of duplication between the two sets?

Actually, no. Of course, all 1- through 8-digit numbers in N are in B also. However, only 42,552 of the 9- and 10-digit numbers in N are also in B. There are only 133,729 1- through 8-digit versum numbers, so the overlap between the two sets is only 176,281 out of a total of 11,301,196 in the two sets — less than 1.6%.

But why not just lump the two together?

The two sets have different characteristics. N is "shallow". The maximum number of steps from a seed to any member of N is only 24.

B is "narrow" but much deeper. It contains versum numbers with as many as 430 digits.

I understand what you're saying, but what does it mean?

That remains to be seen.

Okay, be inscrutable.

We're not tying to be inscrutable. We just don't know. But on to the chase.

Things Eleven

In the last article on versum numbers [2], we showed that the reverse sum of a (base-10) number with an even number of digits is divisible by 11. This raises several questions regarding the factors of versum numbers.

Can the reverse sum of a number with an odd number of digits produce a versum number that is divisible by 11?

Only if the number itself is divisible by 11. Since the odd- and even-numbered digits "line up" when the number is reversed, the difference of the sums of the even- and odd-numbered digits is the same in the reversal as it is in the original number, and the result upon addition can't be divisible by 11 unless the original number is.

Granted that any versum sequence eventually reaches a term with an even number of digits, and its successor is divisible by 11, how many terms can there be before one that's divisible by 11?

Five. All seeds of the form 10_{2n} ($n \geq 1$) have this property. (The notation 0_{2n} stands for a string of $2n$ 0s.) For example, for n=1, the sequence for the seed 100 is

101
202
404
808
1616
7777 = 707 × 11

For $n \geq 1$, 10_{2n} goes to $11 \times (70_{2n-1}7)$ in six steps.

It's clear that no larger 3-digit number could have more terms before reaching one that has an even number of digits. Although 100 is the only 3-digit base seed with this property, for $n = 2$, there are seven base 5-digit base seeds in addition to 10000 that have five odd-numbered initial terms. They all begin with the digit 1, of course.

Is the reverse sum of a number that is divisible by 11 also divisible by 11?

Yes. Since divisibility by 11 depends on the difference of the sums of the even- and odd-numbered digits, the reversal of a number that is divisible by 11 also is divisible by 11. Thus, if x is divisible by 11, then $x + \underline{x} = (i \times 11) + (j \times 11) = (i + j) \times 11$. As a consequence, once a versum sequence reaches a term with an even number of digits, all subsequent terms are divisible by 11.

Hey, I've got an idea. Maybe 11^j is a versum number for all $j > 0$.

Interesting idea; 11 is very special. Let's see:

$11^1 = 11$: yes

$11^2 = 121$: yes

$11^3 = 1331$: yes

$11^4 = 14641$: yes

Maybe you're onto something; let's continue:

$11^5 = 161051$: yes

$11^6 = 1771561$: no …

Oh, well. Moreover, for $j = 6$ through 34, 11^j is not a versum number. We stopped there because it seemed unlikely that a higher power would yield a versum number, and the time it takes to test a number with a large number of digits for versumness becomes a limiting factor.

There's an easy way to disprove your conjecture without testing for versumness. Recall that if a versum number begins with a digit greater than one, its last digit must be equal to or one less than its first. Here's a little procedure that fails for numbers that cannot possibly be versum:

```
procedure mayvers(i)
  local first, last
  first := i[1] | fail                    # remember vpred()
  if first == "1" then return i
  last := i[–1]
  if first == (last | (last + 1)) then return i
  else fail
end
```

Using this procedure, it's easy and fast to find powers of 11 that cannot possibly be versum numbers. The smallest is $11^{12} = 3138428376721$.

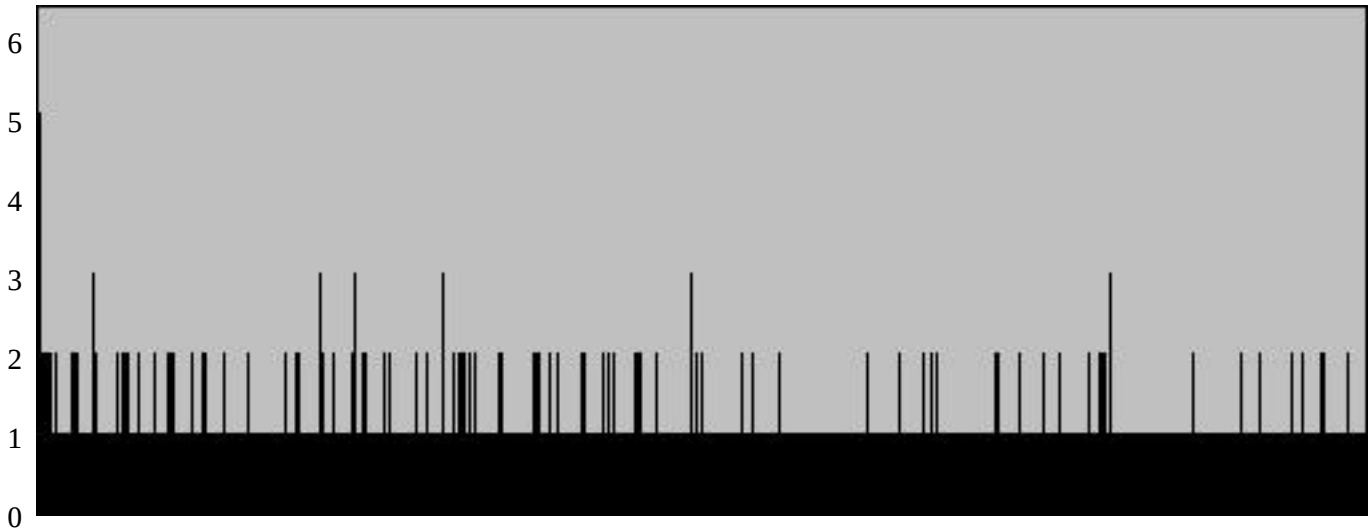


Figure 1. Powers of 11 at the Beginning of the Versum Sequence for Seed 101950

But that doesn't prove that there aren't arbitrarily large powers of 11 that are versum numbers.

Right. We haven't found any, but we haven't a clue as to how to prove or disprove such a conjecture.

By the way, what sequence contains 11^5?

It's the first term in the sequence for base seed 101950. Figure 1 below shows the number of factors of 11 in the initial part of that versum sequence. Incidentally, 101950 is not divisible by 11.

Do factors of 11 build up in successive terms in a versum sequence that's "gone eleven"?

No. Consider again $x + \underline{x} = (i \times 11) + (j \times 11) = (i + j) \times 11$. There's nothing that dictates that $(i + j)$ be divisible by 11. In fact, the reverse sum of a number that is divisible by 11^j, $j > 1$, may not be divisible by 11^k, $k > 1$. The "killing of elevenses" is not restricted to a single power per reverse sum; it can drop the number of powers of 11, however large, to one. See Figure 2 below, in which the number of factors of 11 drops from 6 to 1 in one step.

Granted that the number of factors of 11 in a versum sequence may increase and decrease as the sequence goes along, is there a limit to how many factors of 11 there can be in a versum number?

Probably not, although more than three factors of 11 are uncommon. In B, the maximum number of factors of 11 is only 7 and there are only 10 of these. There 78 (other) versum numbers that are divisible by 11^6.

In N there is only one versum number that is divisible by 11^7 and 9 (others) that are divisible by 11^6.

Here are the details:

Versum numbers divisible by 11^7:

	base seed	*term*	*digits*
B	10103	258	111
	1001285	867	372
	1001586	85	43
	1003319	28	19
	1003319	29	19
	10004338	151	71
	10007107	566	242
	10089905	870	372
	10206945	224	99
	15006096	577	24
N	180929913	1	10

Versum numbers divisible by 11^6:

	base seed	*term*	*digits*
B	5	882	364
	10278	521	218
	10426	745	311
	70749	36	19
	140496	478	199
	1000070	9	9
	1000204	191	86
	1000540	6	9
	1000578	73	39
	1001003	20	14
	1001465	236	108
	1001674	925	390
	1001738	767	336
	1002101	57	31
	1002208	11	12
	1002371	397	174
	1002748	46	25
	1003077	335	153

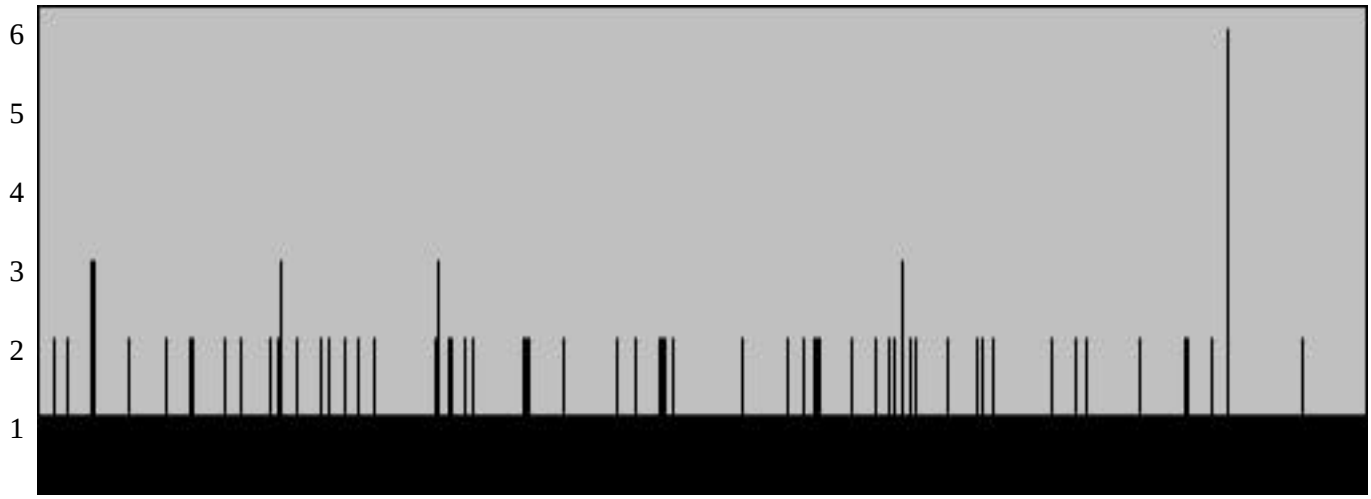


Figure 2. Powers of 11 in a Segment of the Versum Sequence for Seed 5

1003471	904	368
1003628	980	420
1003872	127	57
1004341	41	25
1004785	163	75
1005486	733	312
1005946	451	193
1005963	142	62
1006370	782	328
1006751	93	43
1007503	508	208
1007630	993	412
1007769	362	159
1008790	773	327
1017977	710	296
1029962	388	165
1045976	831	358
1052986	186	88
1067947	347	157
1206598	40	23
1223996	483	209
1303494	921	387
1409996	15	14
1422996	93	44
1705498	13	12
3007929	22	17
3905099	88	41
7005929	988	434
8900399	516	221
10000096	8	11
10000300	545	238
10000523	413	169
10000677	815	346
10001922	199	87
10002433	198	96
10004335	535	233
10006767	32	22
10006916	35	22
10007155	6	10
10007377	605	266
10007713	172	80
10007955	208	91
10008855	917	388
10009006	864	370
10009405	577	238
10009446	542	240
10099453	427	192
10459973	15	14
10459973	16	15
10509954	35	21
10589915	473	203
11106996	605	253
15009396	866	375
18099994	24	16
29008599	274	118
69003999	456	196
70000899	982	409
80099939	855	363
82008999	488	217
89007399	326	149

N

10000540	6	9
20000779	2	9
69099199	1	9
200036769	2	10
300011479	2	10
990055099	1	10
1008029353	2	10
1023089920	2	10
1070690982	2	10

That's a lot more details than I needed … .

With all the trouble we went to get these numbers, there as no way we weren't going to show them. Do you see any patterns or other interesting things about these numbers?

Not really. Do you?

The only really interesting thing we see is that there are two consecutive versum numbers that are divisible by 11^7 — terms 28 and 29 in the sequence for base seed 1003319.

That is surprising.

For what it's worth, Figure 3 shows a plot of the number of terms to number in B and N that are divisible by 11^6 (dots) and 11^7 (circles).

Hmm … ; I wouldn't have expected it to look like that. It's obvious how you got the base seeds for B*, but I'm surprised you know the ones for* N*. I thought you had to construct base seeds from scratch, seed by seed, and you only gave them through 8-digits in an earlier article.*

It's possible (but not easy) to work backward to find the base seed for a given versum number.

Figure 3 Number of Terms in Versum Sequences before Reaching Factors of 11^6 and 11^7

seed digits	*number of factors of 11*							
	0	1	2	3	4	5	6	7
1	9	4372	575	38	5	1	0	0
2	0	0	0	0	0	0	0	0
3	70	35347	5037	493	51	2	0	0
4	0	13877	1921	187	14	1	0	0
5	890	401354	55781	5431	491	49	3	1
6	0	190851	26312	2586	235	15	1	0
7	11090	4291627	588837	55820	5110	470	42	4
8	0	2659272	363816	34462	3130	284	31	5
total	12059	7596700	1042279	99017	9036	822	77	10

Figure 4. Distribution of Factors of 11 in B

We'll come back to that in another article.

I'm having enough trouble with this article. But I have another question. What's the distribution of factors of 11 in versum numbers?

The distribution for B is shown in Figure 4 at the top of the next page.

So almost all versum numbers are divisible by 11.

It might seem that way, but it's not. Less than half of all versum numbers are divisible by 11. Granted, once a versum sequence "goes eleven", it stays that way. However, the number of digits in the terms in a versum sequence increases by one on the average of about 2.45 reverse additions. Any one base seed doesn't produce many *n*-digit versum numbers.

Would you care to estimate the percentage of all versum numbers that are divisible by 11?

50%?

We claimed above that it isn't this large. What motivates you to guess 50%? Can you prove it?

Well, 50% would be a satisfying result. I pass on trying to prove it. I never was comfortable with infinity.

To be politically correct, you should say you are infinity-challenged. Perhaps you have a friend who isn't.

Maybe. I'll let you know.

In the meantime look at Figure 5. It shows the running percentage of versum numbers that are divisible by 11. That is, for each versum number on the horizontal axis, the height of the line is the percentage of versum numbers to that point that are divisible by 11. The values of *n* show the number of digits in the seed. The plot ends at the beginning of 11-digit versum numbers. From there, the running percentage starts to decline because a smaller percentage of 11-digit versum numbers are divisible by 11 (nothing mysterious; just because 11 is odd).

How does this compare with the percentages of all numbers?

We don't have a plot for this, but these figures

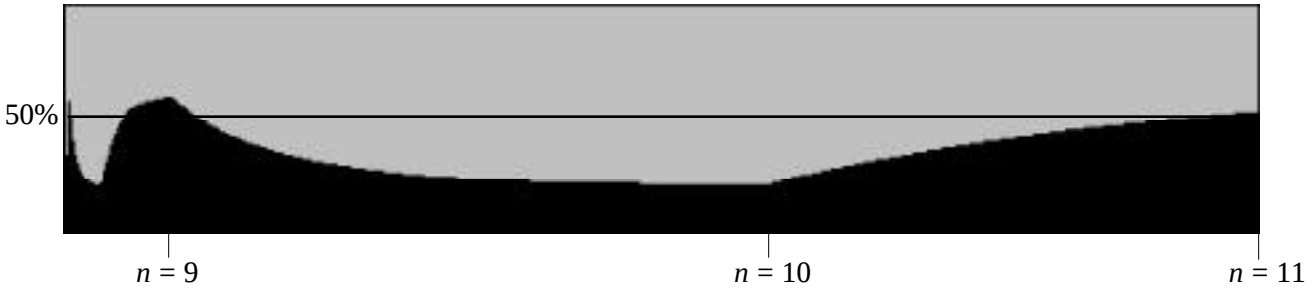


Figure 5. Running Percentage of Versum Numbers Divisible by 11

for all *n*-digit numbers should give you an idea:

n	*percentage*
1	0.00%
2	10.00%
3	9.00%
4	9.10%
5	9.09%
6	9.09%
7	9.09%
8	9.09%

Incidentally, it's comparatively easy to compute the exact number of *n*-digit numbers that are divisible by 11 — just think about what can be multiplied by 11 to give an *n*-digit number.

By the way, you haven't said anything about versum primes.

We're saving versum primes until the next article. We have more to say about composite versum numbers first. Meanwhile we'll leave you with the remark that 11 is a prime versum number with a unique property.

There's 11 again! I'm going to have nightmares about that number. But I'll be back, if only to hear about prime versum numbers. It had better be good.

Of course.

Next Time

We expect to wrap up the discussion of the factors of versum numbers in the next article. We'll start by discussing other special kinds of factors, then composite numbers in general, and end with a description of versum primes.

References

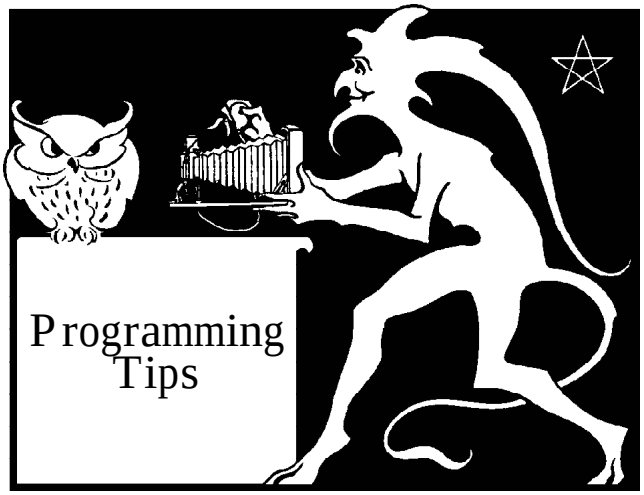
1. "Versum Sequence Mergers", Icon Analyst 33, pp. 6-12.

2. "Versum Bimorphs", Icon Analyst 39, pp. 10-15.

Icon on the Web

Information about Icon is available on the World Wide Web at

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



Tables

One of the things we try to do in the Analyst is provide more detailed coverage of Icon's language features than is available in other documentation. Here we are at Issue 40 and still working on this. We also try to give priority to features that are the most important to programmers. This article is the result of these concerns.

We covered table access briefly in the first issue of the Analyst [1]. That was over six years ago, and although table access in Icon has not changed since then, many present Analyst subscriptions don't date back that far. We also think the topic deserves more discussion than appeared in the old article.

Tables have an interesting history. As far as we know, tables first appeared in a high-level programming language in SNOBOL4 developed at Bell Telephone Laboratories in 1968 [2]. And they almost didn't make it. Only Doug McIlroy's stubborn insistence made them a part of SNOBOL4. He believed that any programming language that was good for string processing should have the facilities needed to write assemblers and compilers — namely, tables with associative look up for strings. In fact, the first distribution of SNOBOL4 was put together before tables were added. They were implemented crudely and added to the distribution as a hexadecimal patch.

Tables proved very popular in SNOBOL4 and, not surprisingly, were used for many things in addition to the ones Doug envisioned. Since that time, tables have been added to several high-level languages, either as a result of the influence of SNOBOL4 or reinvented by persons who were

unaware of their existence in other languages.

Icon inherited tables from SNOBOL4 with only minor changes.

Ironically, no one thought of adding sets to Icon until late in Icon's evolution, despite the fact that sets are more fundamental than tables, and one might expect tables to have developed as an extension of sets.

Sets in Icon were the result of a graduate course in "language internals" that over a period of time studied the implementations of different programming languages in depth. Sets were chosen as a design and implementation project for an instance of this course devoted to Icon. The design was a common effort, but each student did his or her own implementation. Only two of the implementations passed muster — those by Rob McConeghy and Gregg Townsend. Rob went on to add his implementation to Icon, and sets first appeared in Version 6. As we'll see, sets had an effect on table access.

The design of tables reflects the original motivation for them — associative look up to extend the numerical subscripting of arrays to strings and other types of data. Constructions like

```
T[k] := x
```

and

```
y:= T[k]
```

have a certain elegance and make it easy to understand what is going on.

Some aspects of the original design of tables were rather unusual, but they were not accidents or the result of arbitrary decisions.

For example, subscripting a table with a key to which no value has been assigned produces a default value rather than, say, failing. As a consequence of this, there originally was no way to tell for sure if a value had been assigned to a key. All that could be done was to pick a default value that was not used for assigning values to keys in the table and check for it.

There also was no way to delete a key to which a value had been assigned. All that could be done was to assign the default value to the key. This, however, did not reduce the size of the table.

When sets were designed, functions were included to insert members, delete members, and test membership. It was only natural to extend these functions to apply to tables. Thus, the following operations now apply to tables:

```
insert(T, k, x)
delete(T, k)
member(T, k)
```

The first function accomplishes the same thing as

```
T[k] := y
```

The second function deletes the key k and its corresponding value from T if k is a key to which a value has been assigned (it does nothing otherwise).

The third function succeeds if k is a key in T to which a value has been assigned but fails otherwise.

The functions delete() and member() add facilities for tables that were not available formerly. Both should be used when the respective facilities are needed in place of relying on the use of the default table value.

The function insert() doesn't add any functionality. It may, however, be somewhat faster than T[], depending on how big T is, what's in it, and whether garbage collections are necessary to reclaim transient blocks that T[] produces but insert() doesn't.

Although the use of member() and delete() clearly show what is intended, the use of insert() generally does not:

```
T[k] := x
```

stands out clearly, and its syntactic similarity with to

```
L[i] := x
```

for lists provides a useful conceptual link.

Incidentally, T[] is about twice as fast as it was when Reference 1 was written. The improvement is due to Frank Lhota — one of his many contributions to the implementation of Icon.

References

1. "Programming Tips", Icon Analyst 1, p. 6.

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.

Analyst on Disk?

We're thinking about distributing the Analyst on diskette as an alternative to (but not a replacement for) paper copies. The PostScript file for a typical issue of the Analyst is too large to fit on a single diskette, and we'd need to support different kinds of compression for different platforms. An Acrobat Reader file for an Analyst, however, easily fits on one diskette and can be read on most platforms.

The Icon Analyst

Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend
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
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
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

© 1997 by Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend

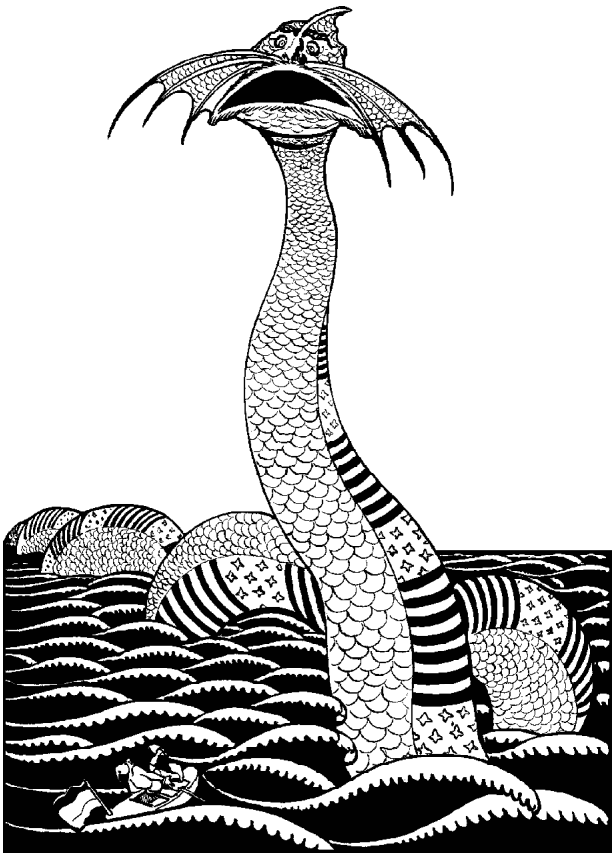
All rights reserved.

You could read your Analyst on-line, as it were, see color images in color, extract text and images, make a printed copy if you have a PostScript printer, and avoid the accumulation of paper.

The potential disadvantages are that you need Acrobat Reader (freely available for MS-DOS, Windows, the Macintosh, and several UNIX platforms) and a platform that can handle it.

We're just thinking about this, and in any event we would continue to make the Analyst available in printed form.

If you have any thoughts on this subject, let us know.



What's Coming Up

We'll be plugging away with more articles in the series that are still incomplete, with articles on custom dialogs, debugging, and factors of versum numbers. We haven't forgotten program visualization either.

We have a couple of articles in our **From the Library** series waiting for a place, and there also are more **Programming Tips** to come.