The Icon Analyst

In-Depth Coverage of the Icon Programming Language

April 1996
Number 35

In this issue …

Building a Visual Interface 1
Corrections 5
Versum Numbers 5
Icon Glossary 11
Subscription Renewal 12
What's Coming Up 12

Building a Visual Interface

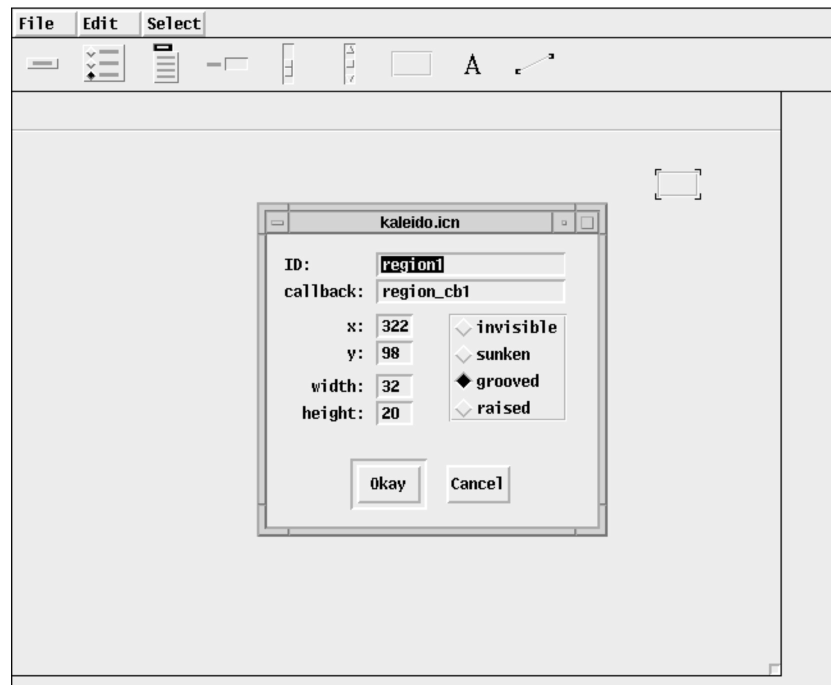
In the article on building visual interfaces in the last issue of the Analyst, we started to show how to use VIB. We got as far as sizing the canvas for the kaleidoscope application and adding a line for the menu bar. In this article, we'll continue to add vidgets.

The size and placement of the display region are among the most important features of the layout. The image at the right shows a newly created region vidget and a dialog for configuring it.

We chose to use a dialog to configure the region, since we wanted to specify a precise size. For approximate sizing, we could have dragged on the corners of the selected region vidget.

The suggested ID is almost what we want, but since there's just one region, we'll remove the number.

The dialog also suggests the name of a callback for the region. Since the region is used only for the display, there's no functionality associated with user events in it, and we don't need a callback. The callback can be eliminated by deleting the text in the field, leaving it empty. When there is no callback for a vidget, events that occur on it are ignored.



Dialog for Configuring a Region

We know what the width and height of the region should be, and we can make a guess as to where the upper-left corner should be. If we're wrong, we can move the region later.

The four radio buttons at the right of the region dialog provide alternatives for the visual appearance of the region's border. We decided on "raised".

Icon on the Web

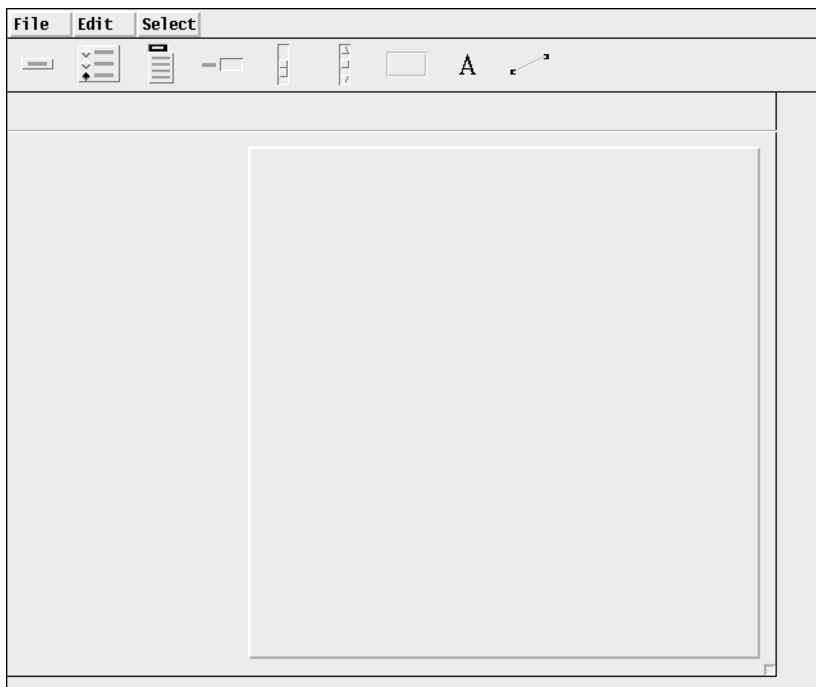
Information about Icon is available on the World Wide Web at

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

If we don't like the effect of a raised region, we can change it later. In fact, we may not know if the effect is what we want until we are able to run the kaleidoscope application. As we'll explain a subsequent article, it's always possible to go back to VIB to modify an existing interface.

Once we've edited the dialog and dismissed it, we have a region of the correct size, but it's not quite where we want it. A selected vidget can be moved one pixel at a time by using the arrow keys on the keyboard. The image at the right shows the final result of this adjustment.

Now we're ready to create the other vidgets. We'll start with the menu, which is the only vidget on the top part of the canvas.



The Configured Region

The image at the bottom of the page shows the result of creating a menu vidget and bringing up a dialog for it.

The menu label needs to be changed to File, since that's what should appear on the menu button on the interface.

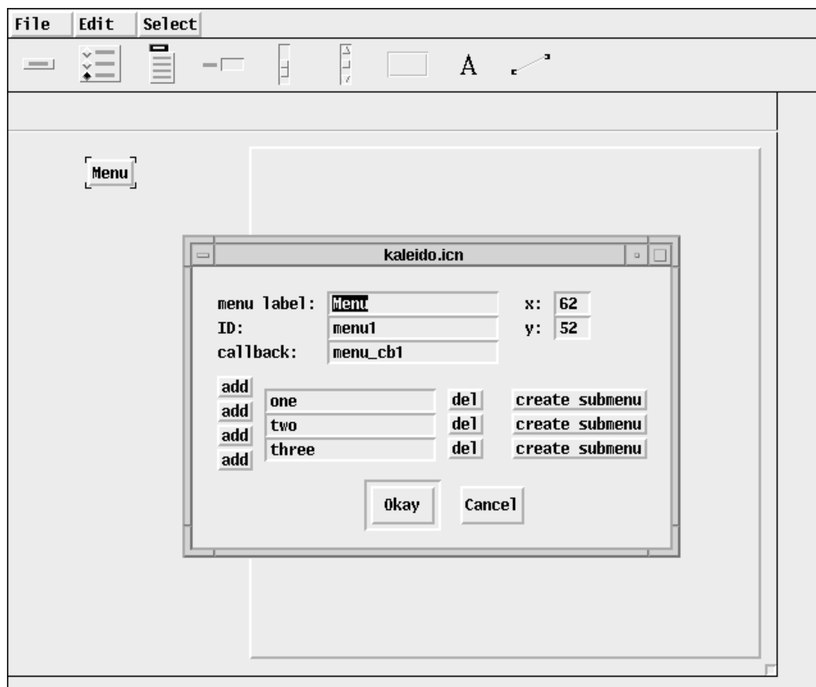
The kaleidoscope application needs only two items; one of the three items can be deleted by clicking on the del button beside it

The File menu needs no submenus, so we can ignore the create submenu buttons. If a menu

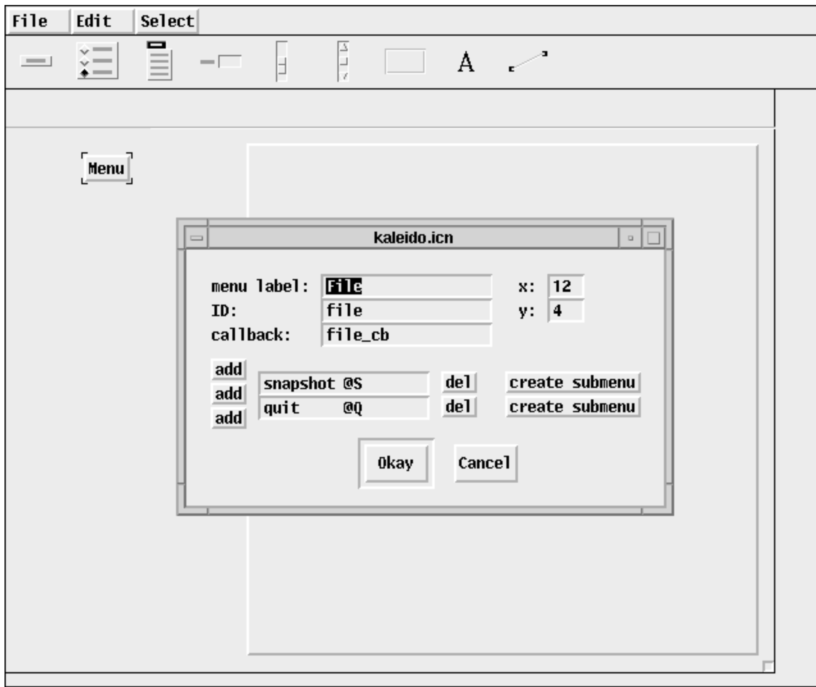
Since there's only one menu in the kaleidoscope application, we could leave the ID as it is, but an ID that corresponds to the name of the menu makes it easier to identify.

The callback should be changed to identify the functionality of the menu. We use the suffix _cb to distinguish callbacks from other procedures in the application, but this is only a convention.

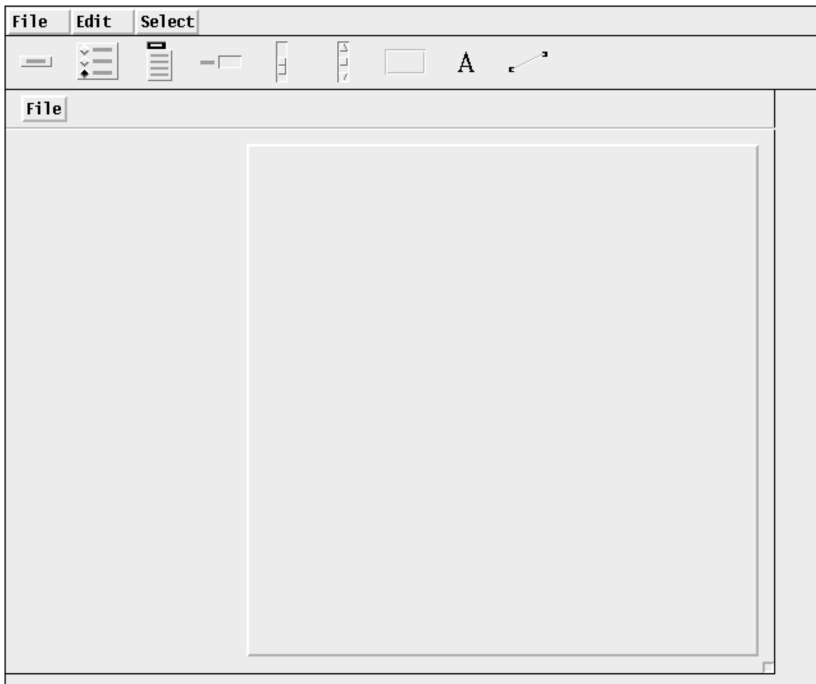
A newly created menu vidget provides three items. If we needed more, we could add them; clicking on an add button between two items adds an item there. There's no specific limit to the number of items a menu may have, but if when pulled down the menu is too long to fit in the application window, not all the items will be available.



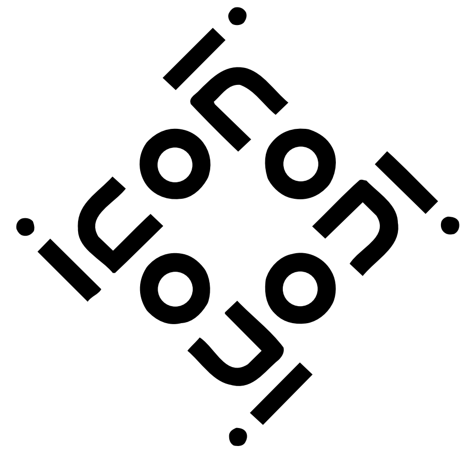
A Menu Dialog



The Edited Menu Dialog



The Menu in Place



item needs a submenu, clicking on the button opposite the corresponding item produces a dialog for the submenu.

The edited dialog is shown at the left. The result after positioning the menu vidget is shown below it.

When a menu is pulled down, it may obscure other parts of the interface. This can be tested in VIB by pressing the middle mouse button on the menu vidget. Obscuring part of an interface temporarily usually isn't a problem, but sometimes the interface looks better if the menu and other vidgets are placed with this in mind.

Next we'll start creating the vidgets to the left of the display region, working from top to bottom, starting with a button for pausing the kaleidoscope display.

The image at the top of the next page shows the result of creating a button vidget and bringing up a dialog for it.

Buttons have more attributes than their apparent simplicity might suggest. Most of the attributes are devoted to how the button looks, not its functionality.

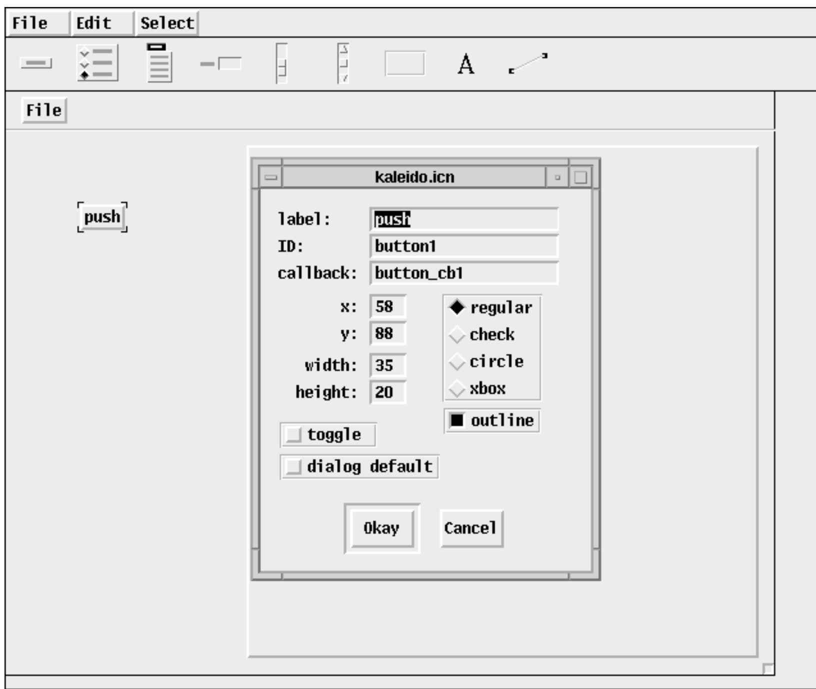
We need to change the label for the button to pause and to choose an appropriate callback name.

Since we're configuring the button for temporarily stopping the display, it's a toggle button, and we need to check that box.

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

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



A Button Dialog

Ordinarily, we'd pick a style that clearly shows it's a toggle when displayed on the interface, but since we have only one other button, and it's not a toggle, we decided to use the same appearance for both of them. The default style is our preference.

The dialog default option doesn't concern us here — we'll cover that in a later article.

We don't need to change the size of the button, since the size adapts to the length of the text for the label, but we can make it larger if we want. This usually is best done after seeing what the automatically sized button looks like.

The result of editing the button dialog is shown at the right.

We also need a reset button, but we won't go through all the details here. The process is similar to that for creating the pause button, except that the reset button is not a toggle. To make the buttons look balanced, we set the same width for both buttons, enlarging the reset button to match the automatic sizing of the pause button.

The image at the top of the next page shows the canvas with the two buttons after positioning them where we thought they looked best.

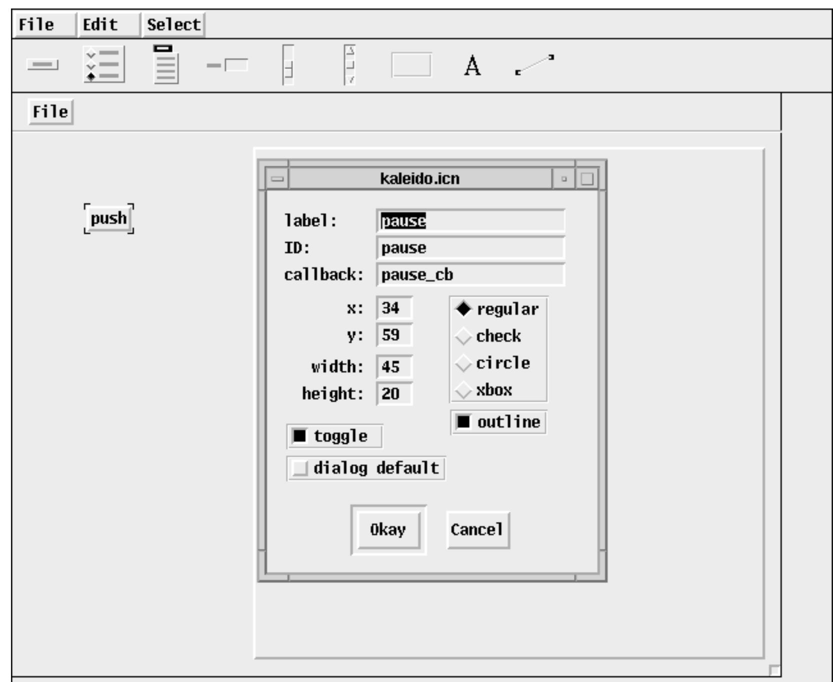
Next Time

In this article, we've added three vidgets and the interface is beginning to take shape. In the next article on building visual interfaces, we'll add the remaining vidgets: sliders, radio buttons, and labels. This will complete the layout of the interface itself.

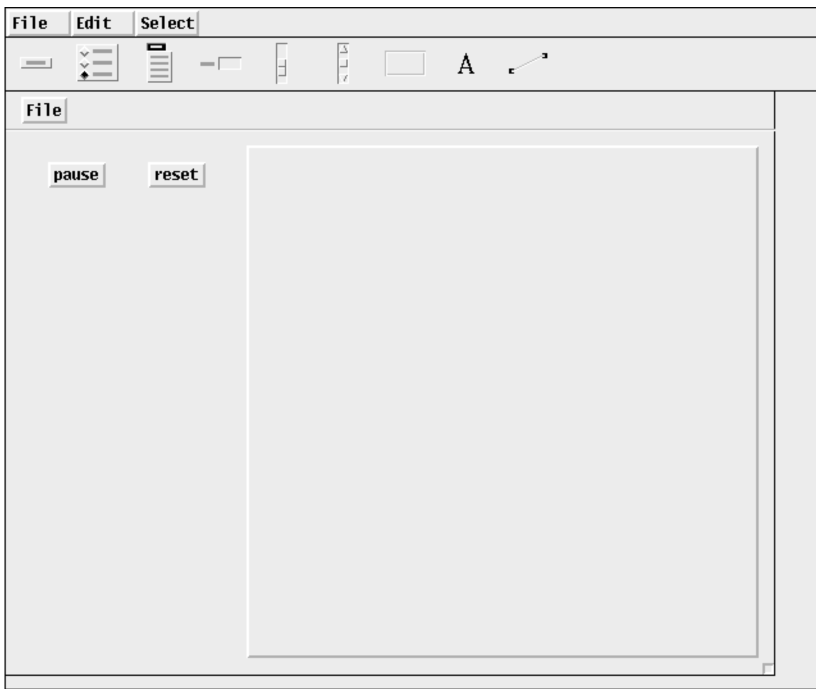
In subsequent articles, we'll show how to prototype an application within VIB to see how it responds to events on vidgets. We'll also discuss other aspects of VIB, including its menus.

Then we'll be in a position to consider the other part of the kaleidoscope application — the code itself. There are essentially two distinct parts to the code. One part is devoted to the basic functionality of the application: the kaleidoscope display. The other, and for our purposes, more interesting, part is how the code relates to the visual interface and handles user events on vidgets.

It's going to take several more articles to get through all this, but we plan to have an article in every issue of the Analyst until the series is complete.



The Edited Button Dialog



The Interface with Five Vidgets

Corrections

In the article on versum palindromes in the last issue of the Analyst, the formula for the number of n-digit numeric palindromes was given incorrectly. The correct formula is

$$9 \times 10^{(n-1)/2} \qquad n \text{ odd}$$
$$9 \times 10^{(n/2)-1} \qquad n \text{ even}$$

The number of versum palindromes with an odd number of digits also was given incorrectly. The correct formula is

$$4 \qquad n = 1$$
$$4.5 \times 10^{(n-1)/2} \qquad n > 1$$

In addition, the procedures for generating palindromes should have checks for a specification of 0 digits; otherwise the procedures plunge recursively. Failure is a reasonable choice in this case, as it is for negative specifications.

Versum Numbers

In preceding articles [1-6], we've explored many aspects of versum sequences resulting from the addition of a positive integer and its reversal. In this article, we turn to the question of versum numbers, those numbers that can be formed by reverse addition and hence are found in versum sequences.

We'll look at three topics:

- the number of n-digit versum numbers
- the nature of versum numbers
- how to determine if a number is a versum number

Tabulating Versum Numbers

It's obvious that not all positive integers are versum numbers: 1, 3, 5, 7, 9, and 11 are examples. It's easy to find all n-digit versum numbers by accumulating the numbers in versum sequences. But there's an easier and much faster way. All n-digit versum numbers must come from the reverse sum of an (n–1)- or n-digit primary seed. Here's all that's needed:

```
link pvseeds

procedure main(args)

   i := (0 < integer(args[1])) |
      stop("*** invalid argument")

   versums := set()

   every k := (i – 1 | i) do
      every j := pvseeds(k) do {
         j +:= reverse(j)
         if *j > i then break
         if *j = i then insert(versums, j)
         }

   every write(!sort(versums))

end
```

Using a set to accumulate versum numbers and then sorting the set before writing them out works well as long as there aren't too many versum numbers. The problem is that not only must all versum numbers be kept in the set in memory, but at the end, the set must be sorted into a list. Both structures must be in memory at that point. (Icon does not modify a structure that it sorts.)

A more practical approach is to write out versum numbers as they are found, and then use a sort utility on the result, keeping only unique values. It's simple to modify the last program to do

this:

```
                …

        every k := (i – 1 | i) do
           every j := pvseeds(k) do {
              j +:= reverse(j)
              if *j > i then break
              if *j = i then write(j)
              }
                …
```

If this program is named vernums.icn, in UNIX the following will do:

```
vernums n | sort –u > versums.n
```

where *n* is the number of digits.

Incidentally, there are very few duplicates using this method — only 21 for $n = 10$.

Here are counts of versum numbers though $n = 10$, for example:

n	*count*	*% of all*
1	4	44.44
2	14	15.56
3	93	10.33
4	256	2.84
5	1793	1.99
6	4872	0.54
7	34107	0.38
8	92590	0.10
9	648154	0.07
10	1759313	0.02

Counts of Versum Numbers

The last column shows the percentage of versum numbers among all positive *n*-digit numbers. The percentage clearly becomes vanishingly small for large *n*.

Although the trend in the counts of versum numbers is clear, the counts do not appear to fit a simple formula. Here's where carries enter the picture.

A versum number whose initial digit is 2 through 9 cannot come from a carry on reverse addition, while a versum number with an initial 1 may or may not come from a carry. For example, the reverse sum of 100 is 101, while the reverse sum of 55 is 110.

Here's a breakdown of the tabulation into these two cases:

n	*1*	*2-9*	*total*
1	0	4	4
2	6	8	14
3	13	80	93
4	104	152	256
5	273	1520	1793
6	1984	2888	4872
7	5227	28880	34107
8	37718	54872	92590
9	99434	548720	648154
10	716745	1042568	1759313

Breakdown of Versum Numbers

It's easy to see that the counts for initial digits 2 through 9 can be described by a simple recurrence and — except for the initial terms — the same one as for counting primary *n*-digit seeds [3]:

$$\mathfrak{V}^{2-9}(1) = 4$$
$$\mathfrak{V}^{2-9}(2) = 8$$
$$\mathfrak{V}^{2-9}(n) = 10 \times \mathfrak{V}^{2-9}(n-1) \qquad n > 2, \text{ odd}$$
$$\mathfrak{V}^{2-9}(n) = 19 \times \mathfrak{V}^{2-9}(n-2) \qquad n > 2, \text{ even}$$

And as for the earlier recurrence, there is a simpler if less revealing, form:

$$\mathfrak{V}^{2-9}(1) = 4$$
$$\mathfrak{V}^{2-9}(2) = 8$$
$$\mathfrak{V}^{2-9}(3) = 80$$
$$\mathfrak{V}^{2-9}(n) = 19 \times \mathfrak{V}^{2-9}(n-2) \qquad n > 3$$

These recurrences are, of course, conjectures. But it's hard to imagine they don't hold in general — although there have been bigger surprises in number theory.

The counts for versum numbers with an initial 1 do not appear to follow a simple rule. We might suppose that these versum numbers are composed of two classes: those that come from reverse addition with a carry and those that come from reverse addition without a carry. It's apparently not that simple. The counts of versum numbers with an initial 1 do appear to follow a recur-

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.

rence for *approximating* the number of versum numbers with an initial 1:

$$\mathcal{V}^1(1) = 0$$
$$\mathcal{V}^1(2) = 6$$
$$\mathcal{V}^1(3) = 13$$
$$\mathcal{V}^1(n) = 19 \times \mathcal{V}^1(n–2) \qquad n > 3$$

With more initial terms from the actual data, the recurrence gives progressively better agreement with the actual data:

	initial terms	3	4	5	6
n	*actual*		*recurrence approximations*		
1	**0**	**0**	**0**	**0**	**0**
2	**6**	**6**	**6**	**6**	**6**
3	**13**	**13**	**13**	**13**	**13**
4	**104**	114	**104**	**104**	**104**
5	**273**	247	247	**273**	**273**
6	**1984**	2166	1976	1976	**1984**
7	**5227**	4693	4693	5187	5187
8	**37718**	41154	37544	37544	37696
9	**99434**	89167	89167	98553	98553
10	**716745**	781926	713336	713336	716224

Results of the Recurrence Approximation

Bold type shows actual values, while regular type shows values predicted by the recurrence. Even with only six initial terms taken from actual data, the approximation for $n = 10$ is only off by 521 — about 0.07%.

Since the recurrence is the same for versum numbers regardless of their initial digit, it serves as an approximation for all versum numbers:

$$\mathcal{V}(n) = 19 \times \mathcal{V}(n–2) \qquad n > i$$

with the quality of the approximation depending on the number of initial terms, *i*, taken from actual data.

The Nature of Versum Numbers

The recurrences shown in the last section, especially the exact one for initial digits 2 through 9, suggest that a regularity or pattern may be found in versum numbers. We'll skip those that begin with 1 for the moment, and look at the others for which there is a (conjectured) exact recurrence.

Here are listings of the *n*-digit versum numbers for initial digits 2 through 9 for $n = 1$ through 4 (for larger *n*, there are too many to show):

n=1:	2							
n=2:	22	33	44	55	66	77	88	99
n=3:	201	302	403	504	605	706	807	908
	202	303	404	505	606	707	808	909
	221	322	423	524	625	726	827	928
	222	323	424	525	626	727	828	929
	241	342	443	544	645	746	847	948
	242	343	444	545	646	747	848	949
	261	362	463	564	665	766	867	968
	262	363	464	565	666	767	868	969
	281	382	483	584	685	786	887	988
	282	383	484	585	686	787	888	989
n=4:	2002	3003	4004	5005	6006	7007	8008	9009
	2101	3102	4103	5104	6105	7106	8107	9108
	2112	3113	4114	5115	6116	7117	8118	9119
	2211	3212	4213	5214	6215	7216	8217	9218
	2222	3223	4224	5225	6226	7227	8228	9229
	2321	3322	4323	5324	6325	7326	8327	9328
	2332	3333	4334	5335	6336	7337	8338	9339
	2431	3432	4433	5434	6435	7436	8437	9438
	2442	3443	4444	5445	6446	7447	8448	9449
	2541	3542	4543	5544	6545	7546	8547	9548
	2552	3553	4554	5555	6556	7557	8558	9559
	2651	3652	4653	5654	6655	7656	8657	9658
	2662	3663	4664	5665	6666	7667	8668	9669
	2761	3762	4763	5764	6765	7766	8767	9768
	2772	3773	4774	5775	6776	7777	8778	9779
	2871	3872	4873	5874	6875	7876	8877	9878
	2882	3883	4884	5885	6886	7887	8888	9889
	2981	3982	4983	5984	6985	7986	8987	9988
	2992	3993	4994	5995	6996	7997	8998	9999

Note that the last digit of all numbers is equal to the first digit or one less than it. This must be the case, as shown by the following argument.

We'll call a number whose reverse addition produces a versum number a *predecessor* of that number. Note that a versum predecessor is not necessarily a versum number.

Let *x* stand for a string of digits and $\underline{x}$ stand for its reverse. Consider an *n*-digit versum number, $n > 1$, of the form 2*x*2. This versum number can only have predecessors of the form 1*y*1 (and 2*y*0, which is equivalent), where $y+\underline{y} = x$ (with no carry on addition to produce a longer string of digits). For example 2992 has the predecessor 1091; $y = 09$ and $y+\underline{y} = 99$ (the initial zero is not suppressed because *y* is internal to a longer string).

Similarly, a versum number of the form 2*x*1 can only have a predecessor of the form 1*y*0 where $y+\underline{y}$=1*x* (with a carry to produce a string one digit

longer than y). For example, 2981 has the predecessor 1990; $y = 99$ and $y+\underline{y} = 198$.

Any other combinations of initial and final digits would produce an initial digit other than 2 (0 is precluded as an initial digit, since it is suppressed and would produce a shorter number in the reverse addition process).

The same argument applies to versum numbers with initial digits 3 through 9.

Now look at the versum numbers in the list above. All the n-digit versum numbers with an initial 3 can be obtained by adding $10_{n-2}1$ to a corresponding versum number with an initial 2, where 0_{n-2} stands for $(n–2)$ 0 digits. A similar observation applies to versum numbers with initial digits 4 through 9.

To see why this is true, consider the possible predecessors of a versum number of the form $2x2$. It has a predecessor of the form $1y1$ as shown above. Add 1 to this predecessor to get $1y2$. Then $1y2+2\underline{y}1 = 3x3$. That is, if $2x2$ is a versum number, so is $3x3$. (We call this process *promoting* a predecessor.) A similar argument applies to versum numbers of the form $2x1$ and for those with initial digits 3 through 8.

Furthermore, the versum numbers obtained by successively promoting n-digit versum numbers with an initial 2 are the only n-digit versum numbers with initial digits 3 through 9. Suppose, for example, there is a versum number of the form $3x3$ that does not come from promoting a versum number of the form $2x2$. This $3x3$ versum number must have a predecessor for the form $2y1$ (and the equivalents $1y2$ and $3y0$). Now demote the predecessor $2y1$, giving $2y0$. But $2y0+0\underline{y}2 = 2x2$, a versum number that contradicts the assumption. Similar arguments apply to versum numbers of other forms.

But this is getting tedious. The really interesting problem is to find a way of generating all n-digit numbers with an initial 2. The strong evidence that the count of such numbers follows a precise recurrence suggests that there should be a (recursive) rule for generating such n-digit versum numbers from $(n–1)$- and $(n–2)$-digit versum numbers. It seems like this or something similar *must* be possible, but it's eluded us. Perhaps we're just looking at the problem in the wrong way. If you come up with a method, please let us know.

There still is the question of versum numbers with an initial 1. We can apply the demotion process to versum numbers of the form $2x2$. For example, 2992 has a predecessor 1091, which when demoted becomes 1090, a predecessor of the versum number 1991. As shown earlier, however, a versum number of the form $2x1$ has a predecessor of the form $1y0$. Subtracting 1 from this number changes nature of the calculation. For example, 2981 has the predecessor 1990, which would be demoted to 1989, the predecessor of 11880 — a versum number with an additional digit. Promoting the predecessor of an n-digit number with an initial 9 also produces an $(n+1)$-digit versum number with an initial 1. These two methods, however, only produce a small percentage of $(n+1)$-digit versum numbers with an initial 1.

Here are the versum numbers with an initial 1 for $n = 1$ through 4. See if you can find a pattern:

n=1:	1							
n=2:	10	11	12	14	16	18		
n=3:	101	121	141	154	165	181	198	
	110	132	143	161	176	187		
n=4:	1001	1130	1251	1372	1493	1615	1736	1857
	1009	1131	1252	1373	1494	1616	1737	1858
	1010	1150	1271	1392	1514	1635	1756	1877
	1029	1151	1272	1393	1515	1636	1757	1878
	1030	1170	1291	1413	1534	1655	1771	1881
	1049	1171	1292	1414	1535	1656	1776	1897
	1050	1190	1312	1433	1551	1661	1777	1898
	1069	1191	1313	1434	1554	1675	1796	1918
	1070	1211	1331	1441	1555	1676	1797	1938
	1089	1212	1332	1453	1574	1695	1817	1958
	1090	1221	1333	1454	1575	1696	1818	1978
	1110	1231	1352	1473	1594	1716	1837	1991
	1111	1232	1353	1474	1595	1717	1838	1998

Identifying Versum Numbers

Despite the discussion above, we don't really know much about what versum numbers are. We know how they are formed, but it's not possible, in general, to tell if a number is a versum number just by looking at it, especially if the number is large.

This leads to the problem of testing numbers for "versumness". We can, of course, do this exhaustively, but that approach is both unattractive and impractical. So we started looking at the reverse addition process and the nature of versum predecessors.

We thought it would be fairly easy to write a procedure to find the predecessors of a versum number (a number with no predecessors is, of course, not a versum number). It's not so hard to do

this for numbers with an initial 2 through 9, but for numbers with an initial 1, which may come from a carry on reverse addition, special cases seem to proliferate.

The package of procedures we developed is far larger and uglier than we think it should be, but we decided to show it anyway. Perhaps it will inspire a better solution. Commentary follows the code.

```
link eqvseeds
link vprimary

procedure vpred(i)              # generate predecessors
   local s, preds

   if i < 1 then fail

   preds := set()

   every s := integer(vpred_(i)) do {
      if s + reverse(s) = i then
         insert(preds, vprimary(s))
      }

   suspend !sort(preds)

end

procedure vpred_(s)             # break down into cases

   if (s == "") | (s = 0) then return s
   else suspend case *s of {
      1:          vpred_1(s)
      2:          vpred_2(s)
      3:          vpred_3(s)
      default:    case integer(s[1]) of {
         0:          vpred_i0(s)
         1:          vpred_i1(s)
         default:    vpred_in(s)
         }
      }

end

procedure vpred_1(s)            # one–digit number

   return (s % 2 = 0) & (s / 2)

end

procedure vpred_2(s)            # two–digit number
   local first, last

   first := integer(s[1])
   last := integer(s[2])

   return ((first = last) & (first || 0)) |
      ((first = 1) & (last % 2 = 0) & (s / 2))

end

procedure vpred_3(s)            # three–digit number
   local first, middle, last

   first := integer(s[1])
   middle := integer(s[2])
   last := integer(s[3])

   suspend case first of {
      0:          vpred_i0(s)
      1:          {
         ((middle = last + 1) & (middle || 9)) |
         ((last = 1) & (1 || vpred_1(middle) || 0)) |
         ((last = 0) & right(s / 2, *s, 0))
         }
      default:  {
         1 || case first of {
            last:          vpred_1(middle)
            (last + 1):    (middle % 2 = 0) & (middle / 2) + 5
            } || (last – 1)
         }
      }

end
```


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

© 1996 by Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend

All rights reserved.

```
procedure vpred_i0(s)          # initial 0
  local lzeros, middle

  suspend (s[–1] = 0) & (0 || vpred_noinc(s[2:–1]) || 0)

end

procedure vpred_i1(s)          # initial 1
  local last, middle, first, mtail, zmid

  last := integer(s[–1])
  middle := s[2:–1]
  first := integer(s[2])

  mtail := middle[2:0]

  if mtail > 0 then zmid := right(mtail – 1, *mtail, 0)

  if last = 1 then suspend {
    (1 || vpred_noinc(middle) || 0) |
    (2 || vpred_noinc(\zmid) || 9) |
    (2 || vpred_noinc((1 || mtail) – 1) || 9) |
    (2 || vpred_inc(1 || \zmid) || 9)
    }

  else if last = 0 then suspend {
    (1 || vpred_noinc(\zmid) || 9) |
    (1 || vpred_inc(1 || \zmid) || 9) |
    (1 || (middle > 0, vpred_noinc(middle – 1)) || 9) |
    (0 || vpred_inc(1 || middle) || last) |
    (right(s / 2, *s, 0))
    }

  else suspend {               # last digit > 1
    ((last + 1) || vpred_noinc(\zmid) || 9) |
    ((last + 1) || vpred_noinc((1 || mtail) – 1) || 9) |
    ((last + 1) || vpred_inc(1 || \zmid) || 9) |
    (last || vpred_inc(1 || mtail) || 0)
    }

end

procedure vpred_in(s)          # initial > 1
  local first, middle, last, t

  first := integer(s[1])
  middle := s[2:–1]
  last := integer(s[–1])

  if first = last then         # no internal carry
    suspend 1 || vpred_noinc(middle) || (last – 1)

  else if first = (last + 1) then    # internal carry
    suspend last || vpred_inc(1 || middle) || 0

  else fail

end

procedure vpred_inc(s)         # predecessor carry
  local t

  suspend 1(t := vpred_(s), *s = (*t + 1))

end

procedure vpred_noinc(s)       # no predecessor carry
  local t

  suspend 1(t := vpred_(s), *t = *s)

end
```

The procedure vpred(i) generates the versum predecessors, if any, of i. It's written as a generator, since some versum numbers have more than one predecessor. It calls vpred_() to start the process. Unfortunately, not all the numbers returned by vpred_() are versum predecessors, so vpred() checks this, adding the primary of a number to a set if it qualifies. Using the primary assures that the same predecessor is not represented in more than one form (as, for example, 33 and 42). All the equivalents of a primary can be generated if needed [3]. Finally the set is sorted and the predecessors are generated. Of course, if there are no predecessors, nothing is generated, and the call of vpred() fails.

The procedure vpred_() separates numbers by size first and then by initial digit. Although vpred_() is not called by vpred() with a 0 argument, it may be by internal calls. That or an empty argument ends the recursion. One, two, and three-digit numbers are handled as special cases. Longer numbers are handled differently according to their first digit.

The procedure vpred_0() handles cases with an initial 0, which may occur during the recursive process.

The procedure vpred_1() is the ugly one, because the number may or may not have been produced by a carry on reverse addition. Various possible cases are tried, distinguishing predecessors that do and not result in a carry.

The procedure vpred_in(), which handles numbers with initial digits 2 through 9, is fairly straightforward, since no carry on reverse addition is possible.

As we mentioned earlier, this package of procedures seems larger and uglier than it should be. However, every case included in these procedures is needed to produce all versum predecessors, at least within the framework we used.

The correctness of these procedures is, of course, conjectural, but we've tested them successfully for a very large number of cases.

Incidentally, we tried putting a "guard" procedure around the code for the individual cases, hoping to improve performance by pruning parts of the search tree that were not producing valid versum predecessors. This didn't help at all for numbers with even a moderately large number of digits; the tests were taking as much time as they were saving — and they made the code much uglier than it already was. Presumably, for very large numbers, pruning would offer a substantial improvement in performance.

Next Time

Every time we think we're at the end of the articles on versum numbers, the "last" article produces material for another.

In working on the article here, we looked at versum numbers with more than one distinct predecessor. So that's next. (Questions: What's the largest number of distinct predecessors a versum number can have? How frequently is it more than one?)

We do, however, think we're getting near the end.

References

1. "The Versum Problem", Icon Analyst 30, pp. 1-4.

2. "The Versum Problem", Icon Analyst 31, pp. 5-12.

3. "Equivalent Versum Sequences", Icon Analyst 32, p. 1-6.

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

5. "Versum Base Seeds", Icon Analyst 34, p. 6.

6. "Versum Palindromes", Icon Analyst 34, p. 6-9.

Icon Glossary

Icon documentation uses some terms in a technical way. This article is the beginning of a glossary of such terms. This is just a start; there's more to come. The part here also is not closed — there are references to terms that are not yet included.

Icon terminology developed over time. Some terms have been used differently in different documents. What follows reflects current usage.

This glossary assumes familiarity with computer terminology. Words in boldface within entries refer to glossary terms, although word forms are different in some cases.

assignment: associating a value with a **variable.**

backtracking: control backtracking or **data backtracking;** usually used as a synonym for the former.

bounded expression: an expression that is limited to at most one **result** because of the syntactic context in which it appears. See also **limitation**.

compilation: the process of converting an Icon source program into directly executable code. Compilation is an alternative to **translation** and **linking**.

control backtracking: returning control to previously evaluated but **suspended generators**. Control backtracking is the underlying mechanism for accomplishing **goal-directed evaluation**.

data backtracking: restoring previous values to **variables** during **control backtracking**. Data backtracking occurs only for a few specific operations.

data structure: a collection of values. Different kinds of data structures are organized and accessed in different ways. See also **records**, **lists**, **sets**, and **tables**.

dereferencing: producing the value of a **variable**. Dereferencing is done automatically when the value of a **variable** is needed in a computation. Dereferencing also may be done explicitly using an **operator**.

element: a value in a **record**, **list**, **or set**; or a **key/value pair** in a **table**.

execution: the process of running an Icon program that results from **translation** and **linking** or from **compilation**.

failure: the lack of a **result**; expression evaluation that does not produce a **result**. Failure is the opposite of **success**.

generation: the production of more than one **result** in sequence.

generator: an expression that is capable of producing more than one **result**.

goal-directed evaluation: the attempt to produce a successful **outcome** by **resuming suspended generators** to get **alternative** values when an expression otherwise would **fail**. Goal-directed evaluation is implicit in expression evaluation. See also **iteration** and **control backtracking**.

icode: the result of **linking ucode** files. icode files are in a binary format.

identifier: a string of characters that names a **variable**.

interpretation: processing an **icode** file to effect

the execution of an Icon program. See also **compilation**.

interpreter: the program that processes **icode** files.

iteration: producing all the **results** of a **generator**. Iteration can be accomplished by a **control structure** or by **conjunction** with an expression that always **fails**. See also **goal-directed evaluation**.

limitation: restricting the number of times a **generator** is **resumed**. Limitation can be specified by a **control structure** or because of the syntactic context in which the **generator** appears. See also **bounded expression**.

linker: the program that converts **ucode** to **icode**.

linking: the process of converting one or more pairs of **ucode** files into an **icode** file suitable for **interpretation**.

list: a **data structure** that consists of a sequence of values called **elements**. Lists can be accessed by position (**subscripted**) and as stacks and queues. Positional accesses produce **variables**.

member: a value in a **set**; also called **element**.

outcome: a **result** or **failure** resulting from the evaluation of an expression.

record: a **data structure** consisting of a fixed number of values that are referenced by **field names**. The **fields** of a record are **variables**.

reserved word: a string of letters that has syntactic meaning and cannot be used as an **identifier**.

result: a value or a **variable** as a consequence of evaluating an expression.

result sequence: the sequence of **results** that a **generator** is *capable* of producing. This is an abstract concept used for characterizing **generators**, not a program construct.

run-time: the time during program **execution**.

resumption: continuing the evaluation of a **suspended generator**. See also **suspension**.

set: a **data structure** consisting of distinct values upon which set operations can be performed. A value in a set is called a **member** and sometimes by the more general term **element**.

success: evaluation of an expression that produces a **result**; the opposite of **failure**.

suspension: stopping the evaluation of a **generator** when a **result** is produced. See also **resumption**.

table: a **data structure** composed of **key/value** pairs, in which **keys** are distinct. Tables can be **subscripted** by **keys** to assign corresponding values. Table **subscripting** produces **variables**.

termination: the end of **execution**.

translation: the process of converting Icon source code to code for an imaginary machine (**virtual machine**). The result of translation of a source code file is a pair of **ucode** files. See also **compilation**.

translator: the program that translates Icon source code into **ucode**.

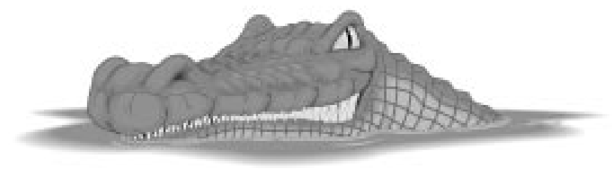
ucode: the result of **translating** Icon source code into code for a **virtual machine**. Ucode files are readable text.

variable: a reference to a value and to which assignment can be made. There are several kinds of variables, including **identifiers**, some **keywords**, and the **elements** of **records**, **lists**, and **table subscripts**. See also **dereferencing**.

Subscription Renewal

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

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



What's Coming Up

Don't let the crocodile alarm you; we don't plan anything sinister or vicious for the next issue of the Analyst. Surprises, however, always are possible.

We're working on additional articles on building visual interfaces, versum numbers, and the glossary. As usual, we have other things in the works, including dynamic analysis, programming tips, and articles on material in the Icon program library.