The Icon Analyst

In-Depth Coverage of the Icon Programming Language

February 1996
Number 34

In this issue …

Welcome to a New Editor

We're pleased to welcome Gregg Townsend as an editor for the Analyst.

From the Analyst's inception, Gregg has participated in its production and has made numerous suggestions as well as providing material.

We are, in fact, simply recognizing the role Gregg has played. The only problem is that it no longer will be appropriate to explicitly acknowledge his contributions.

Icon Newsletter Subscriptions

As your read in the last *Newsletter*, it now is available on the Web. The *Newsletter* is sent by postal mail only to subscribers who pay a one-time fee. That fee is waived for subscribers to the Analyst.

We plan to time publication of a *Newsletter* to coincide with the publication of an Analyst, and we'll mail them together. Since the Analyst is published twice as often as the *Newsletter*, expect to get a *Newsletter* with every other Analyst.

Icon on the Web

Information about Icon is available on the World Wide Web at

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

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

e-mail: icon-project@cs.arizona.edu

THE UNIVERSITY OF
ARIZONA®
TUCSON ARIZONA

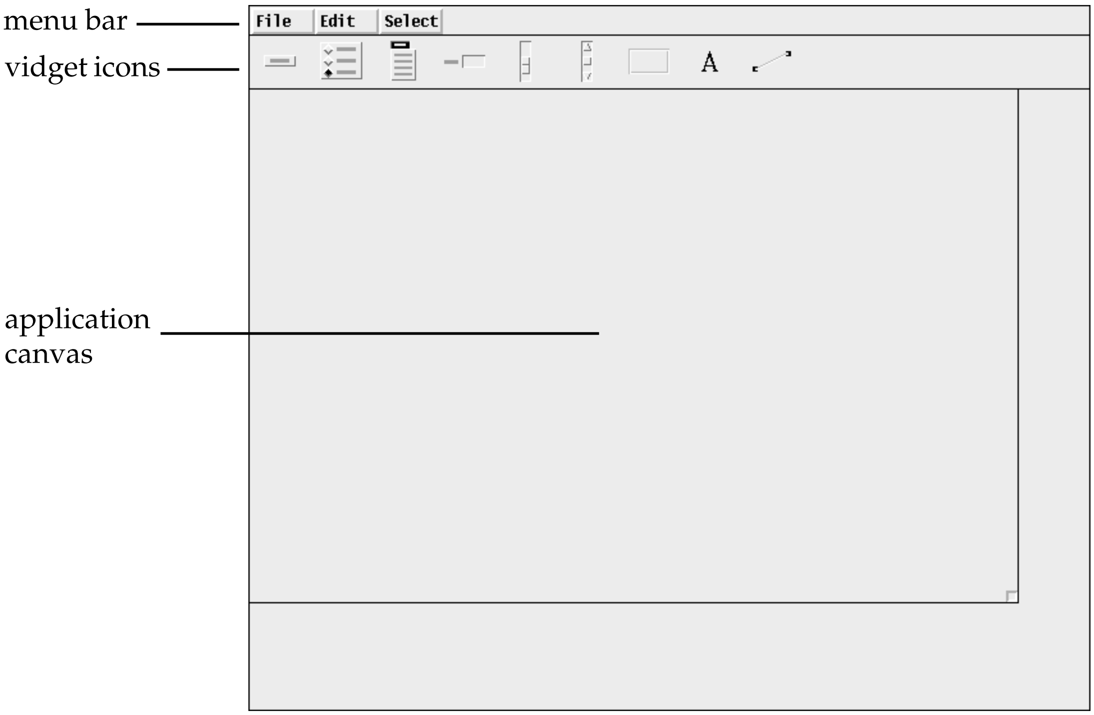
and

Bright Forest Publishers
Tucson Arizona

Building a Visual Interface

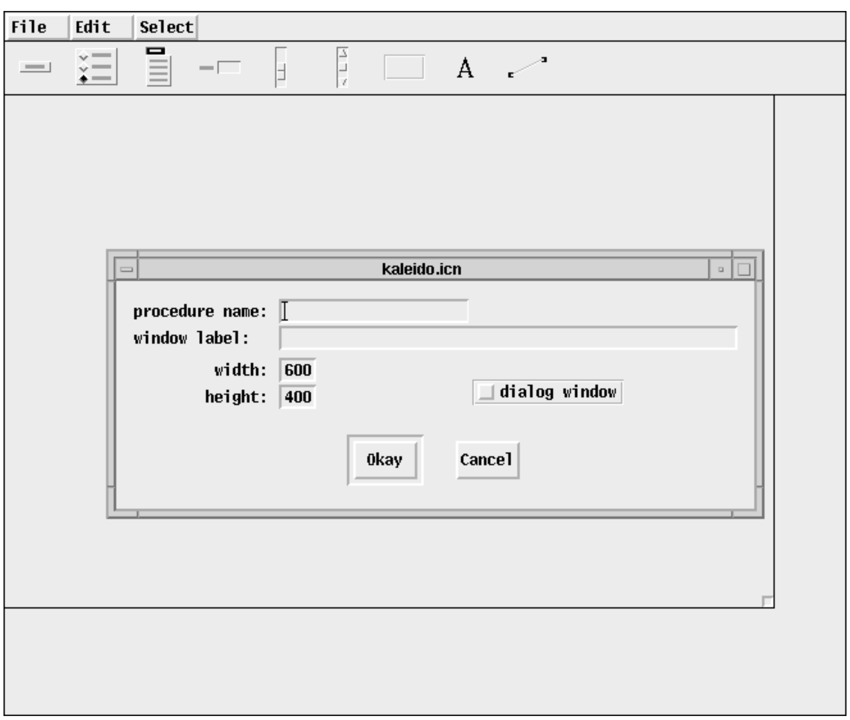
In the last article on building visual interfaces [1], we sketched the layout of the tools for the kaleidoscope interface. In this article we'll start building the interface using VIB.

The VIB application for a new interface is shown below. The menus at the top provide operations needed to use VIB. The icons below the VIB menu bar from left to right represent buttons, radio buttons, menus, text-entry fields, sliders, scroll bars, regions, labels, and lines. The inner rectangle represents the canvas of the interface being developed.



The VIB Application

It's generally a good idea, before creating any vidgets, to set the desired size of the application canvas. This can be done by dragging with the left mouse button on the lower-right corner of the rectangle representing the application canvas. Alternatively, clicking the right mouse button on the lower-right corner of the canvas area brings up a dialog, which is shown at the left.

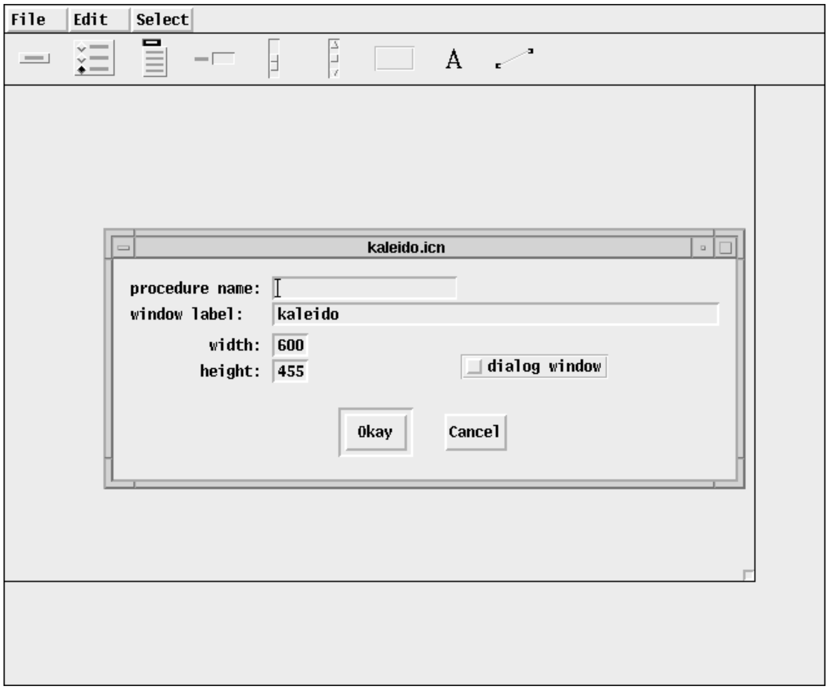


Application Canvas Dialog

To build the interface for the kaleidoscope, we don't need the procedure name field or the dialog window toggle. We'll explain these in a later article.

The window label refers to the label for the application, which we can set now. The default width is reasonable for our design; the critical dimension is the height, which needs to be increased to accommodate the display region and menu bar, with some space for a visual border around the display region. The image at the top of the next page shows the edited canvas dialog. The new canvas size is reflected in subsequent images.

The question is what to do next. There are quite a few vidgets to create, configure, and position. We can't be sure (unless we have a detailed draw-



Specifications for the Kaleidoscope Application

ing of the interface and are sure it's the way we want it) that the canvas size is correct. A good approach at this point is to start laying out the portions of the interface that depend most on the canvas size. One approach is to start by subdividing the canvas into its main areas; first the menu bar that divides the canvas vertically, and then the display region, which is the most crucial part of the area below the menu bar.

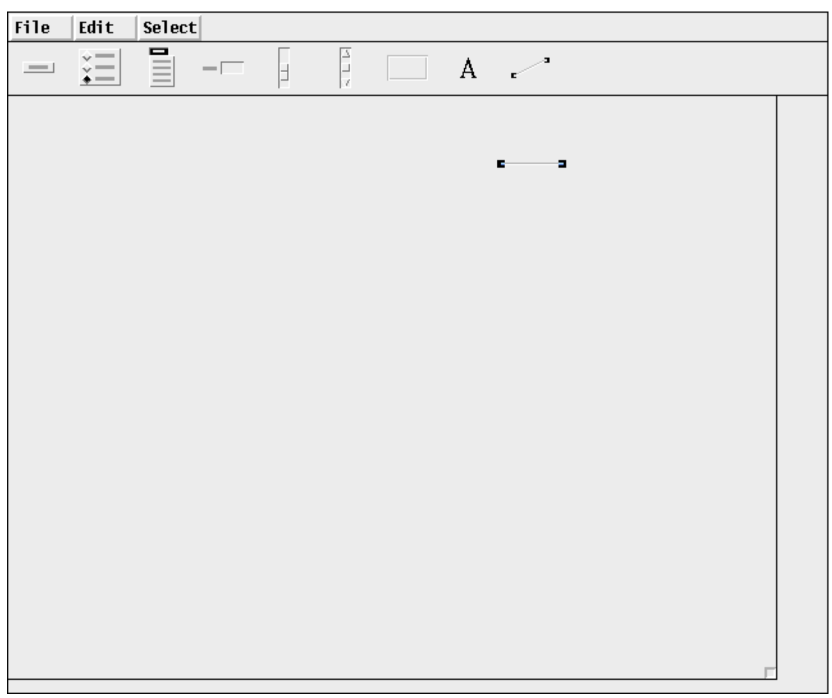
Lines provide visual cues for the user (and also for the interface designer). Therefore, the first vidget we'll create is a line to separate the menu bar from the rest of the canvas.

A vidget is created by pressing the left mouse button on its icon and dragging it onto the canvas. For a line vidget, the result is a short horizontal line as shown at the bottom of this page.

The end points of the line are highlighted to indicate that the vidget is "selected". Operations are performed on the currently selected vidget. A vidget is selected when it is created. A vidget that is not selected can be selected by clicking on it with the left mouse button. Only one vidget can be selected at any one time.

When a vidget is created, it's almost always necessary to change its configuration. Here, the line needs to be longer and moved up.

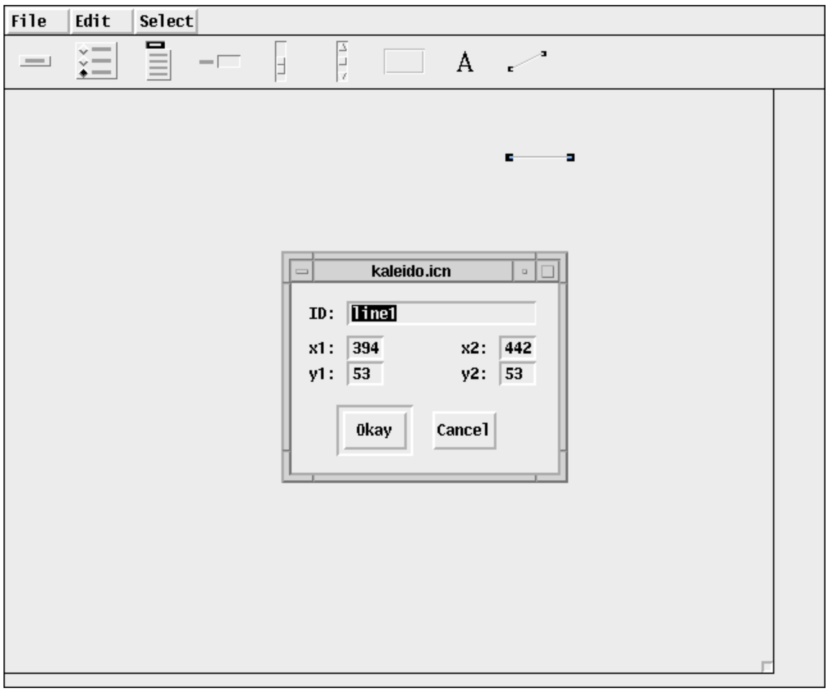
There are several ways we can adjust the length and position of the line. We can press the left mouse button on the line and drag it to a new position. And we can press and drag on an end point to move it, changing the position of that end point (the other remains anchored) to change the length and orientation of the line. Alternatively we can press the right mouse button to bring up a dialog that allows us to specify the length and positions of the end points.



A Line Vidget

For a long line like we want, it's usually easier to start with a dialog, which allows the length and end points to be specified precisely. Once the line is the right length and positioned approximately, its position can be adjusted using the mouse as described above.

The dialog for the newly created line vidget is shown at the left. Different kinds of vidgets have somewhat different dialogs, but all of them have an ID field for a string used to identify the vidget.



Dialog for a Line Vidget

For a newly created vidget, a suggested ID is provided. It's generally a good idea to change the suggested ID to something more mnemonic. In this case it might be menu line.

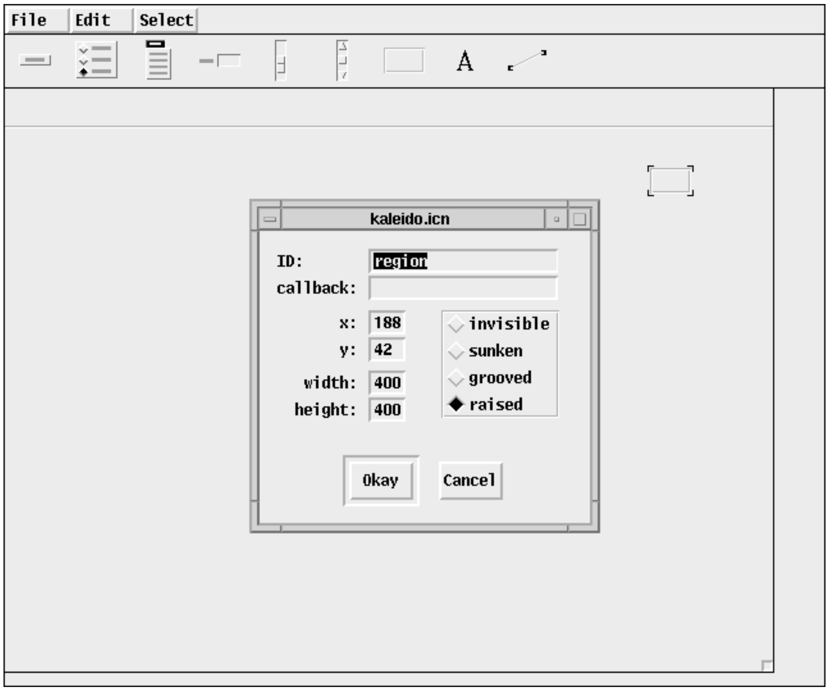
The x1 coordinate should be set 0 and the x2 coordinate to 599 to fit the width of the canvas. (If a line is a little too long to fit on the canvas, that doesn't matter, since nothing appears beyond the edge of the canvas when the application is run.) The values of y1 and y2 need to be the same to produce a horizontal line. We chose 35 for the vertical position, with the results shown at the bottom of this page.

The canvas now is divided vertically into the menu bar and the part that will contain the display, buttons, and sliders. We could add the menu at this point, but we prefer to continue with our strategy of dividing areas. This gives us a view of the canvas that is not cluttered by interface tools. Consequently, the display region is the next order of business.

The approach to creating a region vidget is similar to that for creating a line vidget, although a region has more attributes. To save space, we'll skip images of the newly created region vidget and the initial values in its dialog and go directly to the situation after the region vidget has been created and its dialog edited, which is shown at the top of the next page.

We've set the region's width and height to the size of the kaleidoscope display. The x and y coordinates that specify the upper-left corner of the region are only approximate; they are difficult to specify numerically without a detailed layout, and one of the advantages of VIB is that you can manipulate the vidgets directly. We'll do this after dismissing the dialog.

The Menu Bar Line



Region Dialog after Editing

There's an easy way to move a selected vidget in small increments: Pressing an arrow key moves the vidget one pixel in the direction specified by the key.

As indicated by the second field in the dialog above, a region can have a callback. Since this region is only for the display and there's no functionality associated with user events on the region, we don't need a callback. The callback can be eliminated by deleting the text in the field, leaving it empty, as we have done. When there is no callback for a vidget, events that occur on it are ignored.

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". 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. As we'll show in a later article, it's always possible to go back to VIB to modify the interface.

The result, after moving the region to where we wanted it, is shown at the bottom of this page. Although there are only two vidgets so far, the interface is beginning to take shape.

Next Time

We've run out of space for this article. It may not seem like we've accomplished much, but it doesn't take much time to do what we've described.

The Configured Region

We'll continue with the other vidgets in the next article.

Reference

1. "Designing a Visual Interface", *Icon Analyst* 33, pp. 1-3.

Versum Base Seeds

In the last issue of the Analyst, we introduced the concept of a base seed for versum sequences — the smallest seed whose sequence does not merge to another sequence.

In that article, we commented that all observed 1- through 8-digit base seeds that started with a digit greater than 1 ended with the digit 9, and we asked if anyone could prove or disprove that this is generally true.

Here's a proof by contradiction:

Suppose $a\underline{x}b$ is a base seed, where

a is a digit > 1

$\underline{x}$ is some sequence of digits

b is a digit < 9

Then let

$a' = a - 1$ (still a single digit > 0)

$b' = b + 1$ (still a single digit ≤ 9)

Because

$$a' + b' = (a - 1) + (b + 1) = a + b$$

then $rs(a\underline{x}b) = rs(a'\underline{x}b')$, where $rs(i) = i + reverse(i)$; that is, $a\underline{x}b$ and $a'\underline{x}b'$ merge.

But, by definition, a base seed is the smallest seed whose sequence does not merge to another sequence. Since $a\underline{x}b$ is larger than $a'\underline{x}b'$, $a\underline{x}b$ cannot be a base seed if $a > 1$ and $b < 9$.

Versum Palinromes

Interest in the versum problem stems from the high frequency of palindromes in versum sequences and the puzzle of whether all versum sequences contain at least one palindrome [1].

In the last article on versum sequences [2], we showed a method by which the amount of data that is needed to study versum sequences can be reduced dramatically. This makes it practical to study versum sequences, at least for seeds that have only a modest number of digits.

In this article we go back to the original issue of palindromes and look at two subjects: (1) where palindromes occur in versum sequences and (2) the nature of versum palindromes.

Versum Palindrome Extremes

In the first article on versum sequences [1], we showed empirical evidence that at least for small seeds, the first palindrome in a versum sequence occurs after only a few terms and that the last palindrome apparently is not that far out either. We also mentioned a probability argument that supported these observations.

Here's a table showing where the "farthest first" palindromes occur for 1- through 8-digit seeds, going out to 500 terms (these are, of course, conjectures, indicated by † in headings that follow):

farthest first palindromes†

n	*term*	*primary seed*	*base seed*	*approx. size*	*palindrome*
1	2	5	5	1.1×10^{1}	P1
2	24	89	7	8.8×10^{12}	P3
3	23	187	7	8.8×10^{12}	P3
4	21	1297	7	8.8×10^{12}	P3
5	55	10911	10137	4.7×10^{27}	P8
6	64	150296	150296	6.8×10^{32}	P11
7	96	9008299	1003346	5.5×10^{47}	P12
8	96	15059593	1003346	5.5×10^{47}	P12

The labels in the last column identify the specific palindromes, which are listed at the bottom of the next page.

Here are the "farthest last" palindromes:

farthest last palindromes†

n	*term*	*primary seed*	*base seed*	*approx. size*	*palindrome*
1	35	5	5	6.8×10^{14}	P5
2	34	10	5	6.8×10^{14}	P5
3	39	739	739	1.7×10^{15}	P6
4	39	1792	739	1.7×10^{15}	P6
5	81	10151	10058	1.3×10^{31}	P9
6	79	103946	10058	1.3×10^{31}	P9
7	101	1702190	1003346	5.5×10^{47}	P12
8	98	10300930	1003346	5.5×10^{47}	P12

We were surprised that as the number of digits in seeds increased that the extremes moved out as far as they did. This certainly indicates that the probability argument, which places a vanishingly small value on these, is strained, to say the least. Clearly versum numbers — numbers that occur in versum sequences — have characteristics that are far from those of "ordinary numbers".

For some more trivia, here are the largest palindromes found:

largest first palindromes†

n	*term*	*primary seed*	*base seed*	*approx. size*	*palindrome*
1	2	9	9	9.9×10^{1}	P2
2	24	89	7	8.8×10^{12}	P3
3	23	187	7	8.8×10^{12}	P3
4	20	6999	6999	1.7×10^{13}	P4
5	55	10911	10137	4.7×10^{27}	P8
6	64	150296	150296	6.8×10^{32}	P11
7	96	9008299	1003346	5.5×10^{47}	P12
8	95	10309988	1003346	5.5×10^{47}	P12

largest last palindromes†

n	*term*	*primary seed*	*base seed*	*approx. size*	*palindrome*
1	35	5	5	6.8×10^{14}	P5
2	34	10	5	6.8×10^{14}	P5
3	36	166	166	6.9×10^{16}	P7
4	71	1052	166	6.9×10^{16}	P7
5	64	10911	10137	1.5×10^{31}	P10
6	64	150296	150296	6.8×10^{32}	P11
7	99	9008299	1003346	5.5×10^{47}	P12
8	95	10000748	1003346	5.5×10^{47}	P12

We find it interesting that there are only 12 different palindromes in all these tabulations.

Incidentally, all of there palindromes were found by simple Icon programs. For example, the "farthest first" palindromes were found using this program, which takes *n* as a command-line argument:

```
link pvseeds
link vsterm

procedure main(args)
   local i, ndist, idist, term, iterm, pterm

   ndist := 0

   every i := pvseeds(args[1]) do {
      idist := 0
      every term := vsterm(i) do {
         idist +:= 1
         if term == reverse(term) then {
            if ndist <:= idist then {
               iterm := i
               pterm := term
               }
            break
            }
         }
      }

   write("seed=", iterm, " term=", ndist,
      " palindrome=", pterm)

end
```

In our journeys through versum sequences, we found that the seeds whose sequences have the most palindromes follow a simple pattern:

sequences with most palindromes†

n	*seed*	*number*
1	1	10
2	10	9

Rogue's Gallery of Palindromes

P1	11
P2	99
P3	8813200023188
P4	16668488486661
P5	678736545637876
P6	1685872332785861
P6	69567677677676596
P8	4668731596684224866951378664
P9	1337865254228921129822452568731
P10	1475872457859888888958754278574 1
P11	682049569465550121055564965940286
P12	555458774083726674580862268085476627380477854555

3	100	8
4	1000	9
5	10000	12
6	100000	10
7	1000000	11
8	10000000	11

It's easy to show that for $n > 6$, the seed 10^{n-1} has at least 11 palindromes. Empirical evidence strongly suggests that there are no more palindromes in the sequences for such seeds, but a proof, like a proof for the 196 conjecture, is unlikely to be found.

This again brings up the question of how many versum sequences have no palindromes. Here are empirical results for *n*-digit primary seeds:

sequences with no palindromes†

n	*number*
1	0
2	0
3	3
4	12
5	248
6	939
7	14405
8	43160

The Nature of Versum Palindromes

Leaving the question of where palindromes occur in versum sequences, a more basic question is the nature of versum palindromes.

Do all numeric palindromes (palindromes composed of digits but without a leading 0) occur in versum sequences? Clearly not: 131 is an example of a numeric palindrome that does not occur in a versum sequence.

If we look at versum palindromes, there's an evident regularity. Here are the ones for the first few values of *n*.

n=1:	*n*=2:	*n*=3:				
	11	101	121	141	161	181
2	22	202	222	242	262	282
	33	303	323	343	363	383
4	44	404	424	444	464	484
	55	505	525	545	565	585
6	66	606	626	646	666	686
	77	707	727	747	767	787
8	88	808	828	848	868	888
	99	909	929	949	969	989

n=4:

1001	1111	1221	1331	1441	1551	1661	1771	1881	1991
2002	2112	2222	2332	2442	2552	2662	2772	2882	2992
3003	3113	3223	3333	3443	3553	3663	3773	3883	3993
4004	4114	4224	4334	4444	4554	4664	4774	4884	4994
5005	5115	5225	5335	5445	5555	5665	5775	5885	5995
6006	6116	6226	6336	6446	6556	6666	6776	6886	6996
7007	7117	7227	7337	7447	7557	7667	7777	7887	7997
8008	8118	8228	8338	8448	8558	8668	8778	8888	8998
9009	9119	9229	9339	9449	9559	9669	9779	9889	9999

From the patterns in these listings, we can derive a recursive procedure that generates *n*-digit versum palindromes:

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

   if n = 1 then suspend 2 to 8 by 2
   else if n = 2 then {
      every i := 1 to 9 do
         suspend i || i
      }
   else if n % 2 = 0 then {         # even
      h := (n – 2) / 2
      every i := vspalins(n – 2) do {
         i ? {
            lpart := move(h)
            rpart := tab(0)
            }
         suspend lpart || ("00" | vspalins(2)) || rpart
         }
      }
   else {                           # odd
      h := (n – 1) / 2
      every i := vspals(n – 1) do {
         i ? {
            lpart := move(h)
            rpart := tab(0)
            }
         suspend lpart || ("0" | vspalins(1)) || rpart
         }
      }

end
```

By construction, the numbers that this procedure generates are versum palindromes. It remains to be shown that there are no others.

One way to approach this is see what kinds of palindromic numbers are not versum palindromes. Here's a procedure to generate all the *n*-character palindromes for a specified set of characters:

```
procedure palins(c, n)
  local s, lpart, mpart, rpart, h, p

  s := string(c)

  if n = 1 then suspend !s
  else if n = 2 then
    every c := !s do suspend c || c
  else if n % 2 = 0 then {                    # even
    h := (n – 2) / 2
    every p := palins(s, n – 2) do {
      p ? {
        lpart := move(h)
        rpart := tab(0)
        }
      every c := !s do {
        mpart := c || c
        suspend lpart || mpart || rpart
        }
      }
    }
  else {                                      # odd
    h := (n – 1) / 2
    every p := palins(s, n – 1) do {
      p ? {
        lpart := move(h)
        rpart := tab(0)
        }
      every suspend lpart || !s || rpart
      }
    }
end
```

A simple filter produces the numerical palindromes:

```
procedure npalins(n)
  local i

  every i := palins(&digits, n) do
    if i[1] ~== "0" then suspend i

end
```

If we compare the results of vspalins() and npalins(), we find a simple answer to the question of versum palindromes: All numeric palindromes are versum palindromes, except those that have an odd number of digits and an odd middle digit. For example, 12421 is a versum palindrome, but 12321 is not.

It's easy to prove that a numeric palindrome with an odd middle digit cannot be a versum number. To start with, the middle digit of the reverse sum of an n-digit number can be odd only if there is a carry into it. We'll leave you to work out the rest.

For what it's worth, there are $90 \times 10^{n-2}$ numeric n-digit palindromes. The number of versum palindromes with an odd number of digits is 4 for $n = 1$ and $45 \times 10^{n-2}$ for $n > 1$.

Next Time

We're not finished with versum numbers. In the next article on the subject, we'll explore the question of what numbers are versum numbers — what "versumness" is.

This turns out to be a much more difficult problem than characterizing versum palindromes. The problem has to do with carries on addition; something about which you may have painful memories of childhood learning experiences. We do.

References

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

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

From the Library

Encoding Icon Values

Many programs start with an initialization phase in which data structures such as lists and tables are built. Sometimes these structures are large, complicated, and time-consuming to build but are the same from run to run.

For example, the Icon program we use to process orders for Icon material builds a database for all the items that can be ordered, all the material that needs to be assembled to fill an order, and so on. Several structures are involved and some are quite large. The information that this program uses changes only occasionally; for the most part, it's the same from run to run.

In such situations, it is useful to have a way by which the structures can be built and saved to a file once and then reconstructed from the file whenever they are needed. Such a scheme offers not only the potential for less initialization time, but it also may move a large block of code out of the program.

Another situation in which being able to save program data to a file and use it later occurs in interactive applications, in which a user constructs complex data during a run and wants to be able to reuse it in the application at a later time.

There are several difficult issues in encoding Icon values as strings so they can be saved in a file:

• The encoding should be able to handle any kind of value, although there are limitations on what is possible, as we'll discuss later.

• Pointers to structures, and in particular, loops, should be handled properly.

• The encoding should be reasonably compact.

• The encoding should be portable across different platforms.

The Icon program library contains several procedure packages that encode Icon values as strings that can be written to a file and later decoded to restore the values. The best of these packages is xcode.icn, which is the subject of this article.

Encoding and Decoding Procedures

The file xcode.icn contains two procedures,

```
xencode(x, f)
```

which encodes an arbitrary Icon value x and writes it to file f, and

```
xdecode(f)
```

which reads an encoded value from the file f and reconstructs it.

Using xencode() and xdecode() is simple. In a program that encodes a value, it might amount to something like this:

```
output := open("store.xcd", "w") | …
xencode(x, output)
close(output)
```

which saves the encoded representation of x in file store.xcd.

To reconstruct an encoded value, something like this is all that's needed:

```
input := open("store.xcd") | …
x := xdecode(input)
close(input)
```

It is important to understand that x can be an arbitrarily complex value, such as a table, list, or set that itself points to other structures.

If you want to save several values in one encoding, you can put them in a record or list, as in

```
xencode(
  [
    color_table,
    attrib_set,
    name_list
    ],
  output
  )
```

In the case of encoding a list of structures, the decoding might be

```
value := xdecode(input)
color_table := value[1]
attrib_set := value[2]
name_list := value[3]
```

Multiple encodings also can be written to the same file by calling xencode() several times and then decoded by successive calls of xdecode(), as in

```
xencode(color_table, output)
xencode(attrib_set, output)
xencode(name_list, output)
    …
color_table := xdecode(input)
attrib_set := xdecode(input)
name_list := xdecode(input)
```

Encoding and Decoding Details

Suppose x is a value that has been encoded and y is the result of decoding it. The relationship between x and y depends on the type.

For "scalar" types — the null value, integers, real numbers, csets, and strings — x and y are identical. In Icon terms, this means that

```
x === y
```

succeeds.

The encoding of strings and csets handles all characters in a way that they are correct when decoded.

For structured types — records, lists, sets, and tables — x and y are, of course, not identical, but they have the same shape and their elements bear the same relationship to each other. In other words, x and y are indistinguishable. In Icon terms,

```
equiv(x, y)
```

succeeds, where equiv() is a library procedure in structs.icn. Studying this procedure may help in understanding the meaning of equivalence for structures. (The Icon program library currently is being reorganized; you may find equiv() in a different file in the future.)

There is no way to encode files, co-expressions, and windows so that they are identical when decoded. Values of these types are encoded as empty lists so that when they are decoded they are (a) unique, and (b) likely to produce run-time errors if they are used (probably erroneously). The special files &input, &output, and &errout are, however, preserved in the encoding/decoding process. Notice that if these types occur in structures, the structure and its decoding may not be equivalent.

There isn't much that can be done with function and procedure values, but their type and identification are preserved. If a record is declared differently in the encoding and decoding programs, the results of using the decoded record may be incorrect.

xdecode() fails if given a file in the wrong format or if the file encodes a record or procedure for which there is no declaration in the decoding program.

Complete Calling Sequences

xencode(x, f, p) returns f where

- x is the value to encode.
- f is the file to write (default &output).
- p is an optional procedure that writes a line to f using the same interface as write(). The first argument of p is f. The remaining arguments of p are string encodings. The default for p is write.

xdecode(f, p) returns the restored value where

- f is the file to read (default &input).
- p is an optional procedure that reads a line from f using the same interface as read(). The argument of p is f. The default for p is read.

The parameter p normally is not used for storage in files, but it provides the flexibility to store the data in other ways, such as a string in memory. If p is provided, f need not be a file.

For example, the encoding of x can be "written" to an Icon string by

```
code_string :=
  xencode(x, [""], encode_string)[1]
```

using

```
procedure encode_string(lstr, s[ ])

  every lstr[1] ||:= !s
  lstr[1] ||:= "\n"

  return

end
```

Notice that a list containing an initially empty string is used to capture the encoding. Since xencode() returns its second argument, the desired string is obtained by subscripting the returned list.

Similarly, the string can be decoded by

```
y := xdecode(code_string, decode_string)
```

using

```
procedure decode_string(lstr)
  local line
  static last_arg, code_string

  if lstr ~=== last_arg then {
    last_arg := lstr
    code_string := lstr[1]
    }

  code_string ?:= {
    if line := tab(upto('\n')) then {
      move(1)
      tab(0)
      }
    else fail
    }

  return line

end
```

The reason for passing the string as an element of a list is to allow decode_string() to detect different calls of xdecode(), since decode_string() may be called several times by one call of xdecode(). xencode() must be used in the expected way, of course.

Notes on the Encoding

Values are encoded as a sequence of one or more lines written to a plain text file. The first or only line of a value begins with a single character that unambiguously indicates its type. For some types, the remainder of the line contains additional value information. Then, for some types, there are additional lines of encoding. The null value is a special case consisting of an empty line.

All values except the null value are assigned an integer tag as they are encoded. The tag is not, however, written to the output file. On input, tags are assigned in the same order as values are decoded, so each restored value is associated with the same integer tag as it was when being written. In encoding, any recurrence of a value is represented by the original value's tag. Tag references are represented as integers, and are easily recognized since no value's representation begins with a digit.

The encodings of a structure's elements follow the structure's specification on subsequent lines. The form of the encoding contains the information needed to separate consecutive elements.

Here are some examples of values and their encodings:

x	xencode(x)
1	N1
2.0	N2.0
&null	
"abc"	"abc"
"\000\001"	"\x00\x01"
'abc'	'abc'
main	p "main"
[]	L N0
set()	S N0
table("")	T N0 ""
["hi", "there"]	L N2 "hi" "there"

A loop is illustrated by

```
L := []
put(L, L)
```

for which the encoding is

x	xencode(x)
L2	L N1 2

The 2 on the third line is a tag referring to the list L2. The tag ordering specifies that a value is tagged "after" its describing values. Thus, the list L2 has the tag 2 (the integer 1, the size of L, has tag 1).

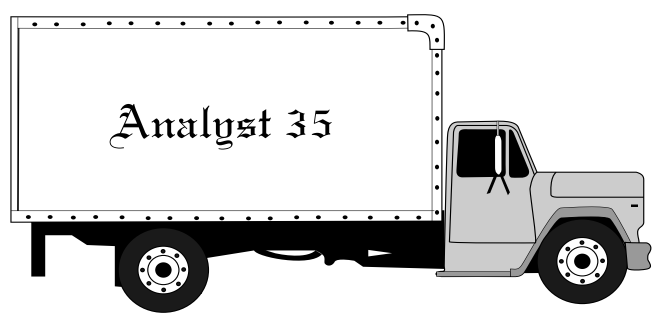
Of course, you don't need to know all this to use xencode() and xdecode().

Getting xcode.icn

xcode.icn is included in the Icon program library. The most recent version, with corrrections and enhancements, is available by anonymous FTP to ftp.cs.arizona.edu; cd /icon/library and get xcode.icn.

Acknowledgment

Bob Alexander designed and implemented xencode() and xdecode(). Some of the material in this article comes from his documentation.



What's Coming Up

In the next issue of the Analyst, we'll continue the series on building visual interfaces and the series on the versum problem.

We have a number of other things in the works, including an article on loading C functions dynamically in Icon and a glossary of Icon terms.

What actually appears in the next issue will depend in large part on how things fit — the images related to visual interfaces make layout tricky.