The Icon Analyst

In-Depth Coverage of the Icon Programming Language

February 1998
Number 46

In this issue …

The Icon Analyst **on the Web**

The idea of providing supplementary information related to the Analyst started with the thought that it would be helpful to readers to have hot links to Web pages mentioned in the Analyst.

Given that start, we decided to add images from the Analyst, primarily to give our readers a way to see the colors, which often are important but impossibly expensive to produce for a printed publication with a small circulation. And it then seemed natural to add related images for which there wasn't room in the Analyst.

We started the on-line supplement to the Analyst with Issue 43 to show the results of program visualization, where color is a key element [1]. In Issue 44 we added VRML worlds that could be downloaded and explored [2].

The article on numerical carpets in Issue 45 [3] raised another possibility — providing programs described in the Analyst (these programs are not yet in the Icon program library). We've now done that; you'll find them in the in the supplement to that issue, not this one. See the article in this Analyst that starts on page 4 to see some of the consequences of putting programs on-line.

We plan to provide program material from the Analyst in future on-line supplements. If there are other things from the Analyst you'd like to see on-line, let us know. Realize, however, that we can't put everything on the Web — that would be unfair to our subscribers and we'd need the income from subscriptions to produce the Analyst even if it were entirely on-line.

The on-line supplements to the Analyst are not static. From time to time we make corrections, delete dead links, add new ones, and so forth. The date of the last update is shown at the top of the main page for each supplement.

Note: In recent Analysts, we listed incorrect URLs for supplementary material. They used to work, but were changed to clean up our Web site. See page 8 for the URL for this issue.

References

1."Kaleidoscopic Visualization" Icon Analyst 43, pp. 1-9.

2."Program Visualization in 3D", Icon Analyst 44, pp. 1-7.

3."Anatomy of a Program — Numerical Carpets", Icon Analyst 45, pp. 1-10.

Text-List Vidgets

With Version 9.3.1 of the Icon program library, there's a new kind of vidget that can be used when building visual interfaces with VIB: the text-list vidget.

The User's View

A text-list vidget consists of an area in which lines of text are displayed and a scrollbar that allows moving through the lines. See Figure 1 on the next page.

Figure 1. A Text-List Vidget

There are three types of text-list vidgets:

- "read-only" vidgets that allow only scrolling but not the selection of a line
- vidgets that allow the selection of a single line
- vidgets that allow the selection of more than one line

The type is indicated visually by grooves at the left side of the vidget: none if selection is not allowed (Figure 2), one if a single line can be selected (Figure 1), and two if more than one line can be selected (Figure 3).

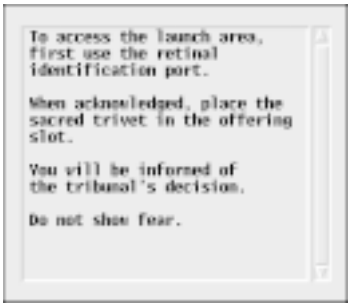


Figure 2. A Read-Only Text-List Vidget

Figure 3. A Multiple-Selection Text-List Vidget

When the user selects a line of text by clicking on it, the line is highlighted, as shown in Figure 4, and there is a callback that notifies the program of the selection. When the user clicks on a previously selected, highlighted line, there also is a callback to notify the program that the line is no longer selected.

Configuring Text-List Vidgets in VIB

The icon for a text-list vidget appears fourth from the left in VIB's icon bar. See Figure 4.

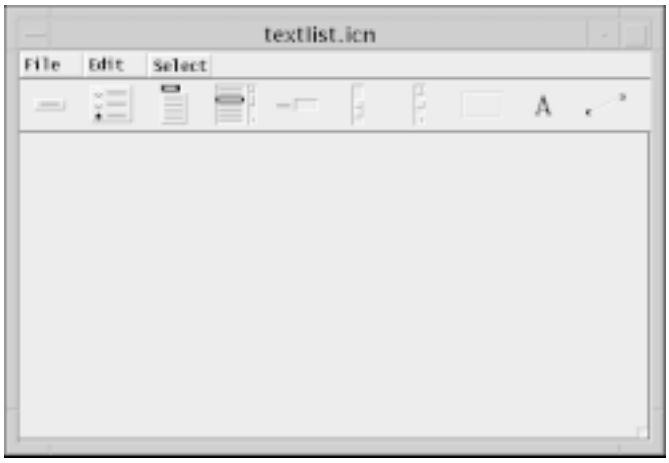


Figure 4. The VIB Window

Clicking on the text-list icon and dragging onto the application canvas area creates a new text-list vidget. See Figure 5.

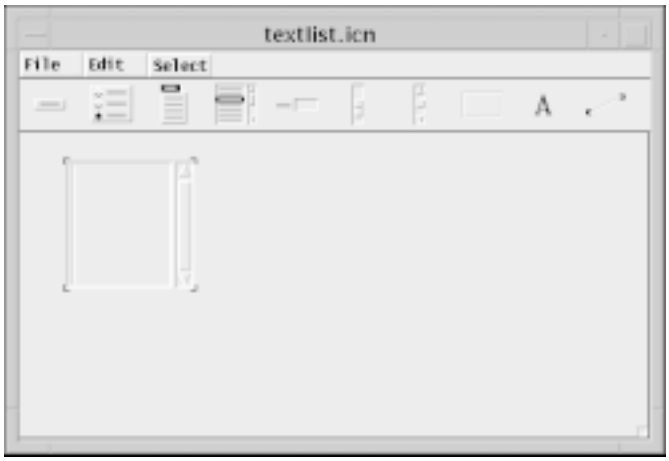


Figure 5. A Text-List Vidget in VIB

Text-list vidgets are positioned and sized in the same way as other vidgets [1]. The dialog for a text-list vidget, as shown in Figure 6, allows the specification of its selection type and other attributes.

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

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

Figure 6. Dialog for a Text-List Vidget

Using Text-List Vidgets in an Application

Text-list vidgets are appropriate when there are a large number of items for the user to view and from which selections may be made. Text-list vidgets in which no selection is allowed can be used to display data or instructions.

Text-list vidgets that allow only a single selection provide functionality similar to that for menus and radio buttons — one of a number of mutually exclusive choices. Multiple-selection text-list vidgets have functionality similar to multiple independent buttons and toggle dialogs.

Setting Up Text-List Vidgets

The lines of text for a text-list vidget are set up using

```
VSetItems(vidget, items)
```

where vidget is the text-list vidget and items is a list of strings. There is no limit to the number of items in such a list except the amount of memory available to form the list.

Callbacks

As is the case for most vidgets, the callback procedure for a text-list vidget has two arguments: the vidget itself and a value:

```
procedure tl_cb(vidget, value)
    ...
```

The nature of the callback for a text-list depends on its selection type. (There is, of course, no callback for read-only text-list vidgets.) For a single-selection vidget, the value is the line (string) selected. In the case of a multiple-selection vidget, the value is a list of the currently selected lines (strings).

When a callback occurs for a single-selection vidget because a line has been deselected, the value is null. It is important to handle deselection callbacks properly. In many applications, deselection callbacks require no action, but they must he handled, since the value is null, not a string as it is for selection callbacks. A typical way of handling deselection callbacks is

```
procedure tl_cb(vidget, value)
  if /value then return
      ...       # processed selection callback
  return
end
```

Text-List Vidget States

A text-list vidget maintains a state that can be queried and set in the same manner as for other vidgets that maintain states [2].

For example, in some uses of a single-selection text-list vidget, it may be desirable for the callback procedure itself to deselect a selected line after it has been processed, so that the user has a visual indication that the selection has been processed (in analogy to a regular button, which is briefly highlighted, as opposed to a toggle button that remains "selected" until the user pushes it again). Such a callback might look like this:

```
procedure tl_cb(vidget, value)
  if /value then return
      ...       # processed selection callback
  VSetVidget(vidget, value)
  return
end
```

The call of VSetVidget() produces a callback to tl_cb() itself, but this does not cause a problem, since the value is null and the procedure immediately returns.

VGetState() can be used to get the state of a text-list vidget. Unlike other vidgets that maintain state, the state of a text-list vidget is not the same as its last callback value. Instead, it is a list of integers. The first integer is the index in the list of the top line of the scrollable area. The remaining integers are the indices of selected lines. Unlike a value in a callback, which only gives the strings selected, the state provides information about their location. This can be useful if the same string occurs more than once in a list and the position of the selection is significant.

Limitations

Like other vidgets used on VIB interfaces, text-list vidgets cannot be resized by the program using the interface. Since lines of text scroll vertically, the height of the vidget usually does not present a problem. There is, however, no horizontal scrolling, and lines that are too long to fit in the area are truncated at the right. Consequently, the width must be pre-chosen to provide adequate space, even though the lengths of lines to be displayed may not be known when the interface is constructed.

Text-list vidgets can only be used on VIB interfaces — there is no text-list dialog. Such a dialog would be very useful, but it is not available because of technical problems in the implementation (in other words, we haven't figured out how to make it work).

References

1."Building a Visual Interface", Icon Analyst 34, pp. 2-3.

2."The Kaleidoscope", Icon Analyst 39, pp. 5-10.

Numerical Carpets Update

When we were finishing the article on numerical carpets in the last issue of the Analyst [1], we inserted a last-minute note to the effect that we hoped to make the programs available on-line. We did this with some trepidation — at the time the programs definitely were "works in progress". They worked, but they had rough edges and parts of them had been hastily put together; they were not in shape to make public, even with disclaimers.

As has often been the case, we set out to put the programs in acceptable (if imperfect) form. But it wasn't that simple: Fixing problems exposed others and also led to the addition of facilities that were useful but not essential. This in turn led to conceptual problems, major revisions to the programs, and so on around the loop.

The programs are better and more capable than the were when the original article was written, but at a penalty: More features have added complexity and the programs are considerably larger.

Then documentation was needed. It has long been our experience that the way to do research is to write about it. This doesn't mean starting with nothing and writing up significant results. Rather, it means documenting existing results and taking very seriously the problems in exposition that come up — discovering gaps, pieces that don't fit together, and giving special attention to things that are hard to explain.

When we first started documenting our research results, we thought that if we couldn't explain something clearly, it was because of our lack of skill and writing experience. That was part of it, but later we came to understand that such difficulties often indicate something wrong with what we were trying to describe or a lack of sufficiently deep understanding on our part. Now when we have a problem explaining a concept, we question our grasp of the material and suspect a flaw in the concept itself. That often turns out to be the case, and although it's not always easy to fix the problems, fixing them often produces new and better results.

The same is true of program documentation. We started to document the numerical carpets programs, stumbled over an explanation, realized the problem was in a concept or a feature, and stopped to fix the problem. Even when we gave up and documented a weakness, we found that a few days later, we had an idea for improvement. Program development was driven by documentation — something that's not evident in commercial software products ... .

Out of all this came better and more capable programs, but they still are "works in progress". (We don't recall ever declaring a program finished, although many have wound up that way.)

New concepts and features included in the current version of the numerical carpets programs are:

- support for databases containing multiple carpet specifications
- a definition facility that allows a name to be associated with a string and used in place of the string
- improved facilities for specifying colors
- the ability to specify the library modules that are needed to generate carpets
- improved control over the process of generating carpets

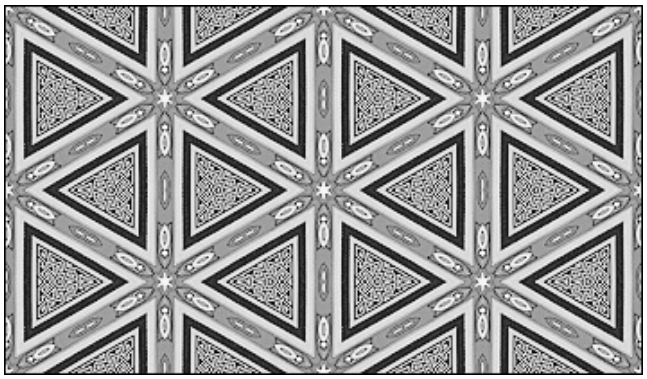
The interface for the new version of the carpet-specification program uses scrollable text lists, a recent addition to VIB's repertoire that is described in an article that begins on page 1. Figure 1 shows what the new interface looks like.

Figure 1. The New Carpet-Specification Interface

There's documentation and example data. Drop by and tell us what you think.

Reference

1."Anatomy of a Program — Numerical Carpets", Icon Analyst 45, pp. 1-10.



Graphics Corner — Gamma

In the RGB color model, all colors are composed from three components of light: red, green, and blue. The intensity of light for each component ranges from 0 to a maximum value that corresponds to maximum intensity. The color model is linear — a specification of 50% of a color corresponds to a midway value in intensity

Unfortunately, monitor hardware is not linear. Skipping the details, it takes disproportionately more voltage for electron guns to produce the expected brightness as specified intensity increases. A good approximation to this behavior is given by

$$B = I^{\gamma}$$

where I is the specified intensity and B is the monitor brightness, both normalized to the range from 0.0 to 1.0. γ is a constant that depends on the particular monitor and can have any value greater than 0.

Adjusting the specified intensities in this manner is known as gamma correction. The correction is applied separately to each color component. Maximum (1.0) and minimum (0.0) intensities are unaffected; mid-range values are affected the most.

For most monitors, the value of γ is between 2.0 and 3.0, with 2.2 being typical. (γ = 2.2 is standard for color television in North America.) For example, if γ is 2.2, then a mid-range intensity of 0.5 translates into a brightness of about 0.218. In order for a specified intensity of 0.5 to produce the expected brightness, it must be changed to $0.5^{1/\gamma}$, which is about 0.730.

Figure 1 shows a plot for a few different values of γ. See the side-bar **Plotting Curves in Icon** on the next page to see how it was done.

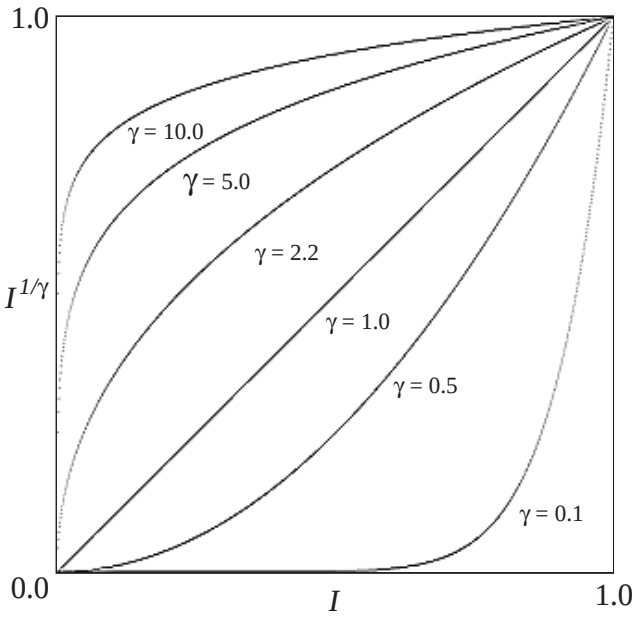


Figure 1. I Versus $I^{1/\gamma}$

Note that low values of γ produce very dark images, while large values produce very light images. Monitors do not have the extreme values of γ shown in these plots, but we'll explain later why applying gamma adjustment independent of the monitor's γ can be useful.

If it were just a matter of converting color specifications to what's needed to display them properly, gamma correction could be done without even knowing about the underlying problem.

Gamma correction *is* done automatically by the graphics systems for the Macintosh and Windows. For example, reading in an image file that was saved with linear intensity values looks like it should (or at least approximately so). Not all graphics systems do automatic gamma correction however; the X Window System, which is commonly used on UNIX platforms, does not. So a direct monitor display of a linear-intensity image tends to look dark and murky. It is left to applications to do gamma correction. Icon does this, using the best information available on the probable value of γ for monitors used with various UNIX platforms. So if you use Icon, image files you read into a window are automatically corrected. There is more too it, unfortunately.

The assumption in applying gamma correction automatically is that images files represent intensities linearly. Not all do; some are designed for direct display without gamma correction and their intensity values have been adjusted for a monitor's γ (2.2 is typical). There generally is no way of knowing whether an image file has been saved with linear intensities or gamma-corrected intensities, and if so, what value of γ was used.

Gamma Values in Icon

In Icon, the attribute gamma has the current value of γ. For example,

```
WAttrib("gamma")
```

returns the value of γ, and

```
WAttrib("gamma=1.0")
```

sets γ to 1.0.

Changing gamma does not change existing pixels in the window; it only affects pixels that are subsequently drawn. Specifically, it affects the interpretation of the foreground and background colors. Put simply, all operations that read and write pixels are interpreted with the current value of γ.

Gamma correction normally is used to produce proper appearance on a specific monitor. It also can be used to change the appearance of images, such as lightening an image for a Web page background.

Plotting Curves in Icon

It's quite easy to plot two-dimensional curves in Icon. Since windows are composed of discrete pixels, you need to decide how many pixels (points) are needed to produce an acceptable result. Then it's just a matter of computing and drawing each point on the curve.

Here's the program we used to plot the curves in Figure 1.

```
link graphics

$define Dimension 1000

procedure main()

   WOpen("size=" || Dimension || "," || Dimension) |
      stop("*** cannot open window")

   every plot_gamma( 0.1 | 0.5 | 1.0 | 2.2 | 5.0 | 10.0)

   WriteImage("gamma_plot.gif")

end

procedure plot_gamma(gamma)
   local x, y

   every x := 0 to Dimension – 1 do {
      y := (real(b) / Dimension) ^ (1.0 / gamma)
      DrawPoint(x, Dimension * (1 – y))
      }

   return

end
```

Dimension gives the number of pixels (points) in the x and y directions (the same in this case). The pixel values in the x direction are normalized for the plotting range of 0.0 to 1.0 and the y value is computed for each.

Since y increases downward in Icon's coordinate system, it is necessary to invert the y coordinate with respect to the window height.

This program does not label the axes or the curves. This is possible, but it's much easier to add these by eye using a drawing or painting program.

We could make the program shorter at the expense of clarity. Or we could make it much more general, but since such programs are so easy to write, it hardly seems worth it.

For some curves, connected line segments produce better results than points. We leave this as an exercise.

Increasing the value of γ above the standard generally lightens the appearance of images that are read (except for fully saturated and fully unsaturated colors). Figure 2 shows an image read with the normal value of γ, while Figure 3 shows the same image read with γ set to 5.0.

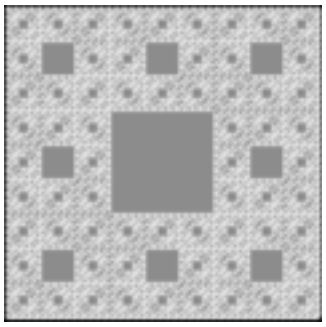


Figure 2. Image Read with Normal γ Value

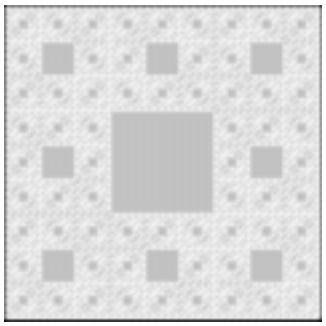


Figure 3. Image Read with γ = 5.0

If, however, an image is read with, say, γ = 5.0 and then written out, the process of writing the image applies the inverse gamma correction and the resulting image file is the same as the one read in.

The way to change the appearance of the image is to change the value of γ back to the normal one before writing:

```
gamma := WAttrib("gamma")      # normal gamma
WAttrib("gamma=5.0")           # new value
ReadImage( … )
WAttrib("gamma=" || gamma)     # restore normal
WriteImage( … )
```

Additional Information

We have simplified some aspects of the visual appearance of images on computer monitors. Hardware is not as simple as we've assumed, and the situation is complicated by ambient light and the way the human visual system perceives light. The complete and correct handling of all the issues is very difficult.

You can find more information about the subject in text books on computer graphics [1,2] and on the Web {1}.

More to Come

This article provides the groundwork for a subsequent article on an application that allows the value of γ to be changed interactively.

References

1. *Computer Graphics*, Donald Hearn and M. Pauline Baker, 2nd ed., Prentice Hall, 1994, pp. 513-515.

2. *Computer Graphics: Principles and Practice*, James A. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, 2nd ed., Addison-Wesley, 1990, pp. 564-568.

Link

1. Poynton's Gamma FAQ:

http://www.inforamp.net/~poynton/notes/colour_and_gamma/GammaFAQ.html

itweak — An Icon Debugger

itweak is an interactive debugger for Icon, developed by Håkan Söderström. itweak allows you to

- set and clear breakpoints
- attach conditions to breakpoints
- evaluate Icon expressions
- define macros
- examine the procedure call chain
- get a variety of information about variables
- print information

See also the side-bar **Debugger Commands** on the next page.

itweak is written in Icon. The idea behind itweak is to modify the ucode files for the program being debugged, first done in David Shartsis' debugify program. Ucode, as you may recall, is the assembly language for Icon's virtual machine [1-2].

itweak has two components, itweak.icn, which prepares the ucode files for the program to be debugged, and dbg_run.icn, which provides run-

time support. Ucode files for dbg_run.icn are linked with the tweaked ucode files for the program to be debugged.

Here's an example of using itweak, taken from the demonstration included with the itweak package. The program to be debugged is ipxref.icn from the Icon program library. It links options.icn from the Icon program library. In a command-line environment, ucode files for both are created by

```
icont –c ipxref.icn
icont –c options.icn
```

The next step is to run itweak on the resulting ucode files to prepare them for debugging:

```
itweak ipxref  options
```

This generates a file dbg_init.icn. Ucode files for it are created by

```
icont –c dbg_init.icn
```

Finally, an icode ("executable") file named sample is made by linking all the ucode files:

```
icont –o sample ipxref.u1 options.u1
```

The ucode files for dbg_init.icn are linked automatically as a result of modifications to ucode files made by itweak. Running sample starts the debugging session in which commands can be entered. For the demonstration, this is

```
sample ipxref.icn
```

which takes ipxref.icn itself as input.

While this may seem complicated, it's entirely straightforward and can be encapsulated in a Makefile, as is done in the itweak package.

Figure 1 on pages 9 through 12 shows the demonstration debugging session. Text entered by the user is underlined, while program output is not. To shorten the listing, we've omitted portions of the program output, as indicated by ellipses.

Reading through the session will tell you more about itweak and its capabilities than we can describe in prose. We encourage you to spend some time with it — and to get itweak and try it yourself. itweak is in the Icon program library and comes with extensive documentation. You can read the documentation on-line {1}.

Debugger Commands

break	set breakpoint
clear	clear breakpoint or debugger parameter
condition	attach condition to breakpoint
do	attach macro to breakpoint
end	terminate macro definition
eprint	print every value from expression
fprint	formatted print
frame	inspect procedure call chain
goon	resume execution
help	print explanatory text
ignore	set ignore counter on breakpoint
info	print information
macro	define new command
next	resume execution, break on every line
print	print expressions
set	set a debugger parameter
source	read debugging commands from file
stop	terminate debugging session
trace	set value of &trace
where	print procedure call chain

References

1."An Imaginary Icon Computer", Icon Analyst 8, pp. 2-6.

2. *The Implementation of the Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Princeton University Press, Princeton, New Jersey, 1986.

Link

1. *itweak*: An Interactive Debugging Facility for the Icon Programming Language:

http://www.cs.arizona.edu/icon/docs/itweak.htm

Supplementary Material

Supplementary material for this issue of the Analyst, including color images and Web links, is available on the Web. The URL is

http://www.cs.arizona.edu/icon/analyst/iasub/ia46/ia46sub.htm

```
# Annotated debugging commands for the demo debugging session.
# $Id: demo.cmd,v 2.21 1996/10/04 03:45:37 hs Rel $
# After seeing the 'automatic' debugging session you may want to repeat some of the commands manually in a new interactive session.
# The following commands use a liberal amount of 'fprint' to make the output more readable. The first few commands are spelled
# out fully. Then we start  using abbreviations.
# When you get the first prompt you are somewhere in anonymous initialization code. Enter 'next' to step into a real source file. This
#  is not necessary, but may allow you to omit the file name in 'breakpoint' commands.
next
[0] main (ipxref.icn:62)
# What source files do we have?
info files
Tweaked source files in this program:
   ipxref.icn
   options.icn
# Let's find out what globals the program contains...
fprint "––– Globals:\n"
–––Globals:
info global
alphas
buffer
……
var
xflag
# ...and the locals of the current procedure:
fprint "––– Locals in %1:\n"; &proc
––– Locals in main:
info locals
Local identifiers in the current procedure:
   L
   args
……
   w2
   word
# Set a breakpoint in the main loop.
break 88
[1]
goon
[1] main (ipxref.icn:88)
# Got the first break.
print word
{word} "link"
goon
[1] main (ipxref.icn:88)
# Next break.
pr word
{word} "global"
# Boring to 'print word' every time. Add this command to the breakpoint.  Note that when a breakpoint has commands the usual
# prelude is not printed when a breakpoint is reached. Thus add some extra printing. Note that 'fprint' does not automatically output
# a newline.
do .
fprint "––– Break in %1 line %2: "; &proc; &line
print word
end
go
––– Break in main line 88: {word} "global"
go
```

Figure 1. Demonstration Debugging Session

```
––– Break in main line 88: {word} "record"
go
––– Break in main line 88: {word} "("
# Attach a condition to the breakpoint. This time we use the explicit breakpoint id (1).
cond 1 word == "buffer"
go
––– Break in main line 88: {word} "buffer"
# Let's examine a compound variable.
fprint "––– Examining 'resword'.\n"
––– Examining 'resword'.
pr resword
{resword} list_429(28)
# It's a list. Try 'eprint' to see all elements.
eprint !resword
{!resword}
  1: "break"
  2: "by"
……
 27: "until"
 28: "while"
# 'eprint' prints 'every' value generated by an expression.
# Try another one.
pr prec
{prec} list_430(1)
# A list again. Prints its elements,
epr !prec
{!prec}
  1: record procrec_1(3)
# Only one element which is a record.
pr prec[1].pname
{prec[1].pname} "main"
epr !prec[1]
{!pred[1]}
  1: "main"
  2: 62
  3: 0
# We may even invoke one of the target program's procedures. Here we invoke 'addword' to add a bogus entry in the cross reference.
# We use global 'linenum' to provide the line number.
pr addword("ZORRO", "nowhere", linenum)
{addword("ZORRO", "nowhere", linenum)} &fail
# Examine globals again.
fprint "––– Globals one more time:\n"
––– Globals one more time:
inf gl
alphas
buffer
……
var
xflag
[debug NOTE] The following global(s) no longer hold their usual Icon functions:
  proc
fprint "––– WHAT??!!! The program has modified 'proc' — bad manners!\n"
––– WHAT??!!! The program has modified 'proc' –– bad manners!
# It's good to have a robust debugger. Let's examine the new value.
pr proc; type(proc)
{proc} "main"
{type(proc)} "string"
```

Figure 1 (continued). Demonstration Debugging Session

```
# Examine the current breakpoint.
fprint "––– The current breakpoint:\n"
––– The current breakpoint:
info br .
[1] ipxref.icn 88:88  DO defined
   CONDITION: word == "buffer"
# Let's set a breakpoint in procedure 'addword'...
br 150
[2]
# ...and delete the first breakpoint.
clear br 1
go
[2] addword (ipxref.icn:150)
# This is the way to find out where we are (the procedure call chain):
where
Current call stack in co–expression_1(1):
(2) addword
(1) main
# It is possible to examine any of the frames in the call chain.
frame 1
(1) main local identifiers:
   args = list_1 = ["ipxref.icn"]
   word = "buffer"
……
   switches = table_11(0)
   nfile = &null
# Let the program work along for a while. Ignore the 280 next breaks.
fprint "––– Ignoring the next 280 breaks...\n"
––– Ignoring the next 280 breaks...
ign . 280
go
[2] addword (ipxref.icn:150)
# Find out about the word "word":
pr var["word"]
{var["word"]} table_50(2)
# It's a table. Examine its keys and entries.
epr key(var["word"])
{var["word"]} table_50(2)
{key(var["word"])}
  1: "addword"
  2: "main"
epr !var["word"]
{!var["word"]}
  1: list_2472(13)
  2: list_802(24)
# The entries are lists. Let's look at the "addword" entry.
epr !var["word"]["addword"]
{!var["word"]["addword"]}
  1: "word"
  2: "addword"
……
 12: 156
 13: 157
# That's a lot of typing. Let's try a macro.
mac var
```

Figure 1 (continued). Demonstration Debugging Session

```
eprint !var["word"]["addword"]
fprint "That was %1 items.\n"; *var["word"]["addword"]
end
# Try the macro (which has now become a new command):
var
{!var["word"]["addword"]}
  1: "word"
  2: "addword"
……
 12: 156
 13: 157
That was 13 items.
# Now we've tried the most common commands. Let the program run to completion undisturbed. The following is an abbreviation
# of 'goon nobreak'
fpr "––– Now let the program produce its normal output...\n\n"
––– Now let the program produce its normal output...
# We will see the normal output of the program: a cross reference listing (in this case applied to its own source code). Note the
# bogus 'ZORRO' variable we entered by calling 'addword'.
go no
variable    procedure   line numbers

L           format      209 215 228 231 231 232 233
L           main        64 142 144 145
……
ZORRO       nowhere     74
addword *   main        103 107 114 119 138
……
```

Figure 1 (continued). Demonstration Debugging Session

Versum Primes

Most persons interested in numbers consider the primes to be the most fascinating of all. We've touched on versum primes in previous articles; now it's time to take a direct look at them.

We've seen versum primes as factors, and it's clear that there are a lot of them, but how many and what kinds of properties do they have?

The smallest versum prime is the infamous 11 we encountered earlier [1]. It has the distinction of being the only palindromic prime with an even number of digits. We'll explore palindromic primes later in this article, but what about versum primes in general? They exist; the smallest non-palindromic versum prime is 241. The "big" question is are there an infinite number of versum primes? That seems very likely, but there are a number of conjectures about the infinitude of various kinds of numbers among primes that have defied the efforts of the best mathematicians. We're certainly not going to try to prove there are (or are not) an infinite number of versum primes.

Palindromic Primes

Let's start with an easier (?) question: "Are there an infinite number of palindromic versum primes?". Or, to start even more simply, "Are there an infinite number of palindromic primes?" Palindromes (of odd length) are relatively plentiful among primes. Very large palindromic primes are known. As of this writing, the largest recorded one has 11811 digits (11811 itself is a versum palin-

drome but not a prime). This palindromic prime can be represented by the digit pattern

$$10_{5901}1465641 0_{5901}1$$

where x_n stands for n repetitions of the string x. Since the middle digit is odd, this is not a versum number. The largest known versum palindromic prime has 11011 digits (11011 also is a nonprime versum palindrome) and the digit pattern

$$10_{5501}32424230_{5501}1$$

It's generally believed that there are an infinite number of palindromic primes, but it's usually listed as an unproved conjecture. We encountered, however, the following remark in a news group [2]:

> The number of palindromic primes is infinite. This follows from the fact that the class of automorphic quadratic forms is also infinite.

That sounds impressive, but we have no idea what it means, and we're inclined to believe authoritative references that say the conjecture is unproved.

You'll recall that all palindromes with an even number of digits are versum numbers, while palindromes with an odd number of digits are versum only if the middle digit is even [3].

A survey of small palindromic primes and all the known large ones shows about half have an even middle digit. So it's reasonable to conjecture that *if* there are an infinite number of palindromic primes, then there are an infinite number of versum palindromic primes, and hence an infinite number of versum primes.

Nonpalindromic Versum Primes

But what about nonpalindromic versum primes? As suggested earlier, there probably are infinitely many of these, but we have no hope of a proof, and we lack a viable way of testing large nonpalindromic primes for versumness. The limit for testing numbers for versumness with the best tool we have, isversum(), is about 25 digits [4]. In the world of primes, numbers of this size are like dust mites.

Figure 1 is an attempt to put primes and versum numbers in perspective. The numbers in the left column are the numbers of digits. Thus the scale is logarithmic for the magnitude of integers. At the beginning are tiny numbers in the world of primes. Hash marks separate gaps — gaps so vast as to be incomprehensible.

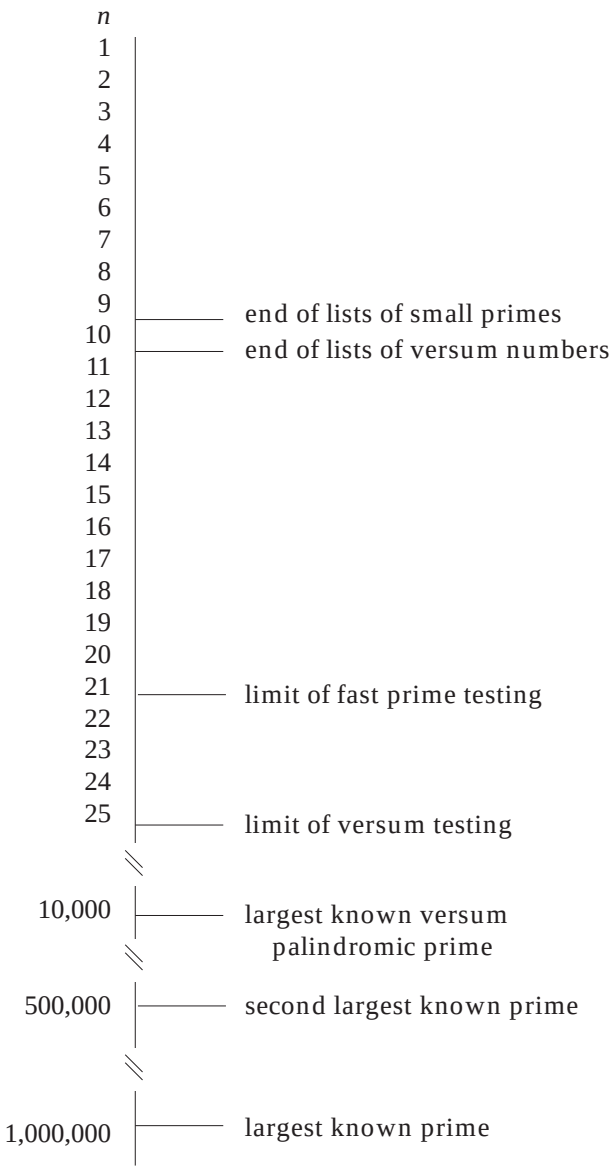


Figure 1. Primes and Versum Numbers

In finding versum primes, we can test primes for versumness or we can test versum numbers for primality. We have lists of versum numbers up to $n = 10$ [5], and it's easy to test these for primality. But for $n > 10$ we're stuck on the versum side. Readily available lists of small primes are more limited, but more can be computed without difficulty. There's a problem with testing primes for versumness; there are far too many primes. The following listing shows the number of primes, versum numbers, and versum primes (obtained by testing versum numbers for primality):

n	*primes*	*versum numbers*	*versum primes*
1	4	4	1
2	21	14	1
3	143	93	12

4	1061	256	13
5	8363	1793	157
6	68906	4872	147
7	586081	34107	2047
8	5096876	92590	1862
9	45086079	648154	30018
10	404204977	1759313	28280
11	3663002302	~12314926	?
12	33489857205	~33426947	?
13	308457624821	~233983594	?
14	2858876213963	~635111993	?
15	26639628671757	~4445688286	?

The numbers of versum numbers for $n > 10$ are approximations obtained from the recurrence given in the article cited above.

To tabulate the versum primes for $n = 11$, we'd have to test more than 3.6 billion primes. Forget that.

Nonetheless, there are still tractable problems related to nonpalindromic versum primes. For example, we easily can divide numbers into two categories: non-versum and possibly versum [1]. We also might find patterns of digits other than palindromes for which we can test for versumness. But to do anything like this, we need the digits.

Digits, Digits

We might as well start with the biggest known primes. Primes with 10,000 and more digits are dubbed "gigantic", and are available on the Net {2}. As of this writing, there are 5401 recorded gigantic primes (the number increases frequently).

That number of primes is reasonable to subject to the "maybe versum" test. There's a hitch, though. All recorded large primes are given by mathematical expressions. For versum testing, we need the digits for numbers.

Most mathematical expressions for primes involve only arithmetic operations and are comparatively short. It may seem surprising that enormous primes can be represented in this way. In some cases, this is the result of *a priori* knowledge about the mathematical structure, such as for the Mersenne primes. See the side-bar **Mersenne Primes** on the next page.

Even if there is no *a priori* knowledge about the mathematical structure of a prime, it's generally easy to get to a short mathematical expression for a prime p: $p \pm 1$ are composite and often yield prime factorizations with only a few terms. (Mersenne primes are extreme examples.)

Many of the mathematical expressions for gigantic primes are given in a form that can be directly evaluated in Icon. For example, the largest known prime is given as

```
2^2976221–1
```

It is trivially easy to wrap such an expression in a program that evaluates and writes it:

```
procedure main()
  write(2^2976221–1)
end
```

Such computations take a while, but they are feasible.

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

Mersenne Primes

For many centuries it was thought that 2^p-1 was a prime for all prime p. In 1536, Hudalricus Regius showed that $2^{11}-1 = 2047$ was composite (23×89).

Others added to the knowledge of numbers of this form, but it was the French monk Marin Mersenne who attracted attention by making the audacious assertion in 1644 that 2^p-1 is prime for p = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, and 257 but composite for all other primes < 257. He was wrong, but his name is now associated with primes of the form 2^p-1.

At that time, no one, including Mersenne, was able to test his assertion, which required hand calculation. It wasn't until 1947 that the range in Mersenne's assertion was completely checked. The primes in this range occur for p = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, and 127.

With more sophisticated mathematical tools and fast computers, many more Mersenne primes have been found. As of this writing, there are 36 known Mersenne primes:

number	*prime*	*digits*	*number*	*prime*	*digits*
1	2	1	19	4253	1281
2	3	1	20	4423	1332
3	5	2	21	9689	2917
4	7	3	22	9941	2993
5	13	4	23	11213	3376
6	17	6	24	19937	6002
7	19	6	25	21701	6533
8	31	10	26	23209	6987
9	61	19	27	44497	13395
10	89	27	28	86243	25962
11	107	33	29	110503	33265
12	127	39	30	132049	39751
13	521	157	31	216091	65050
14	607	183	32	756839	227832
15	1279	386	33	859433	258716
16	2203	664	34	1257787	378632
17	2281	687	35	1398269	420921
18	3217	969	??	2976221	895932

It is not known if the last Mersenne prime is the 36th, since the region between it and Mersenne prime 35 has not been completely explored.

Mersenne primes are prominent in the quest for ever larger primes. The primary reason is the Lucas-Lehmer test {2}, which makes testing for them much faster than for other numbers of comparable size. As of this writing, the five largest recorded primes are Mersenne and the smallest of them has more than three times as many digits as the largest known non-Mersenne prime.

The search for ever larger Mersenne primes continues with GIMPS (the Great Internet Mersenne Prime Search). This project uses, among other things, the resources of thousands of PCs scattered around the world, working on pieces of the project when they otherwise would be idle. The necessary software is free, and anyone with an appropriate PC can participate {3}.

Not all primes can be represented in a compact form using simple arithmetic operations. And in some cases, the mathematical expressions given for primes involve notation and computations that are not commonly seen. A simple example is

```
8*R(12600)*10^3705+1
```

where R(12600) stands for a number consisting of 12,600 1s. Numbers like this are called *repunits*. In base 10, they can be expressed as

$$(10^n - 1) / 9$$

It's easy to handle repunits expressed as R(n) — simply add a library module containing a procedure R(n):

```
procedure R(n)
   return repl("1", n)
end
```

We used repl() rather than an arithmetic computation because n can be huge. This module then can be linked as part of the evaluation wrapper.

Other unfamiliar functions and operations can be added to this module as needed. For example, one commonly used function in the expressions for gigantic primes is the "primorial" function, *primorial*(n), which is the product of all primes less than or equal to n (note the analogy to factorials).

The primorial function can be cast as an Icon procedure as follows:

```
procedure primorial(n)
   local k, m

   m := 1

   every k := prime() do {          # prime generator
      if k <= n then m *:= k
      else return m
      }

end
```

Incidently, in expressions for gigantic primes, the primorial function is not given in functional notation, but rather as a suffix operation, *n*# in analogy to *n*!. Fortunately, it's easy to convert this notation to functional notation:

```
expr ? {
   while writes(tab(upto(&digits))) do {
      span := tab(many(&digits))
      if ="#" then
         writes("primorial(", span, ")")
      else writes(span)
      }
   write(tab(0))
   }
```

This simple approach works because of the regularity of the expression notation in the list of gigantic primes.

Not all the functions used in the expressions for gigantic primes are easily implemented. One that occurs several times is the cyclotomic polynomial of order *n* in *x*, which is given by

$$C_n(x) = \prod_k (x - e^{2\pi ik/n})$$

where *k* runs over all positive integers less than *n* that are relatively prime to *n*.

For this, we went to *Mathematica*, where it's a part of the standard computational repertoire. (If you'd like to implement this in Icon, we'll be happy to put it in the Icon program library.)

We've now computed the digit expressions for all but a few gigantic primes. They turn out to be interesting for reasons unrelated to versum numbers. We'll have something to say about these in a subsequent article.

More Information

The literature about primes is vast. In addition to the links listed below, we'll put references and additional links that we don't have room for here on the Web page for this issue of the Analyst.

References

1."Factors of Versum Factors", Icon Analyst 40, p. 9-14.

2."Palindromic Primes", Robert D. Silverman, net.math, June 24, 1983.

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

4."Versum Numbers as Factors", Icon Analyst 45, pp. 12-16.

5.."Versum Numbers", Icon Analyst 35, pp. 5-11.

Links

1.The Largest Known Primes:
http://www.utm.edu/research/primes/largest.html#contents

2.Mersenne Primes: History, Theorems and Lists:
http://www.utm.edu/research/primes/mersenne.shtml#test

3.Mersenne Prime Search:
http://www.mersenne.org/prime.htm

What's Coming Up

Our plans for the next issue of the Analyst include articles on exploring numerical carpet space and an application for interactively changing the gamma value of images. We also expect to describe changes we've made to the versum-testing procedure to make it more efficient.

We have lots of other things on the deck, including material for **Graphics Corner** and **From the Library**.