The Icon Analyst

In-Depth Coverage of the Icon Programming Language

December 1996
Number 39

In this issue …

Glossary 1
A Framework for Monitoring 1
The Kaleidoscope 5
Versum Bimorphs 10
Program Size 13
From the Library 15
What's Coming Up 16

Icon Glossary

A copy of the completed glossary is enclosed with this mailing of the Analyst. This glossary is the same one that appears in the third edition of the *Icon Programming Language* [1].

One thing we've done for this glossary is to use the word "compiler" for both forms of the implementation (formerly "interpreter" and "optimizing compiler"). The distinction is that one form of the implementation produces code for a virtual machine and the other produces native code for a specific computer. The virtual machine code is then interpreted, while the native code is executed.

We recognize that there still is potential for confusion, but we believe the new terminology conforms more closely to the general understanding of implementation techniques than the old terminology.

It helps that Java has popularized the concept of using a virtual machine as an implementation technique [2]. The idea, however, is quite old — it goes back to the 1960s [3].

References

1. *The Icon Programming Language*, 3rd edition, Ralph E. Griswold and Madge T. Griswold, Peer-to-Peer Communications, Inc., 1996.

2. *The Java Virtual Machine Specification (The Java Series)*, Tim Lindholm and Frank Yellin, Addison-Wesley, 1996.

3. "The Mechanical Evaluation of Expressions", P. J. Landin, *Computer Journal*, Vol. 6, 1964, pp. 308-320.

A Framework for Monitoring Program Execution

In past articles in the Analyst, we've covered various aspects of monitoring program execution in Icon. We have more articles on the subject planned, but before going on, we want to present a framework for monitoring program execution. The framework is sufficiently general to include programs written in most programming languages. It may be helpful, however, to consider how it applies to Icon.

Events

This framework views program execution as a sequence of events [1,2]. Within it, the concept of an event is quite general. The event could be program output, the execution of a particular language construct, an implicit activity like garbage collection, or even the execution of instructions in the underlying hardware. We'll generally be interested in events related to the source language and its underlying implementation. In the case of Icon, we'll be interested in events like expression evaluation, storage allocation, and possibly the workings of the underlying virtual machine [3,4].

The information content of an event and the form it takes can be most anything: a number, a string, or even a complex structure. We generally will characterize events as *n*-tuples. For example, in monitoring string creation, the events might be just sizes, pairs that identify the operation that

produced the string as well as the size, triples that carry the source-code location of the operation as well, quadruples that carry a time stamp, or larger *n*-tuples with even more information.

For many monitoring purposes, pairs suffice, as illustrated in Analyst articles on event monitoring [5,6]. In fact, the instrumentation of the Icon run-time system is based on pairs, consisting of a type and a value. We'll present more on that later, but note that a *n*-tuple can always be represented by a sequence of pairs that carry an identification and one of the *n* values.

The Framework

Given the event model of program activity, the monitoring process can be viewed as an event source, ES, that sends a sequence of event reports to a report processor, RP, that interprets the events. See Figure 1.

This model is intentionally very general and can be instantiated in many ways. The separation of ES and RP into separate components is not necessary, but it helps in describing various possibilities. In an actual implementation, ES and RP might be parts of the same program. Feedback for RP to ES, as indicated by the dashed arrow, is a possibility but not usually present.

We've used two models for Icon event sources in earlier articles. One uses a program, P, and an MT-Icon monitor, M, running in the same execution space [6]. See Figure 2.

In this model, M requests event reports about P from the instrumentation in the run-time system. M determines what events are to be reported. When a requested event occurs in P, control is transferred to M.

A model that we've used more recently involves adding source-code instrumentation to the

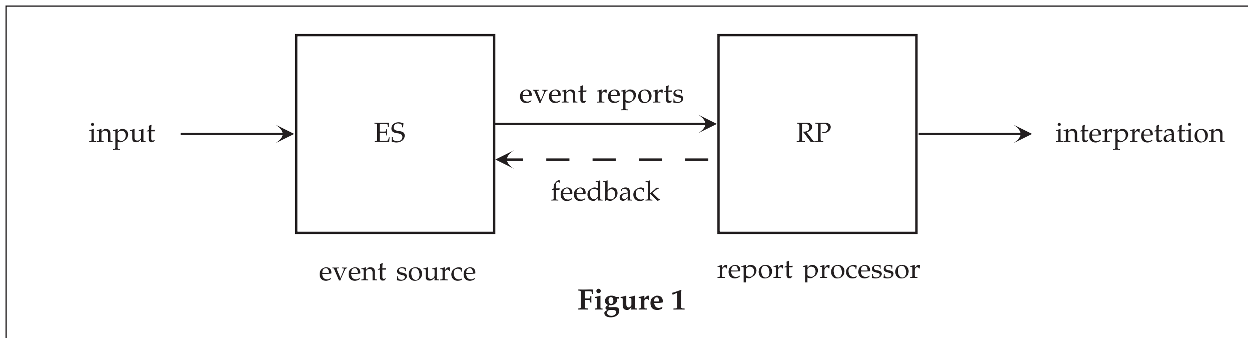


Figure 1

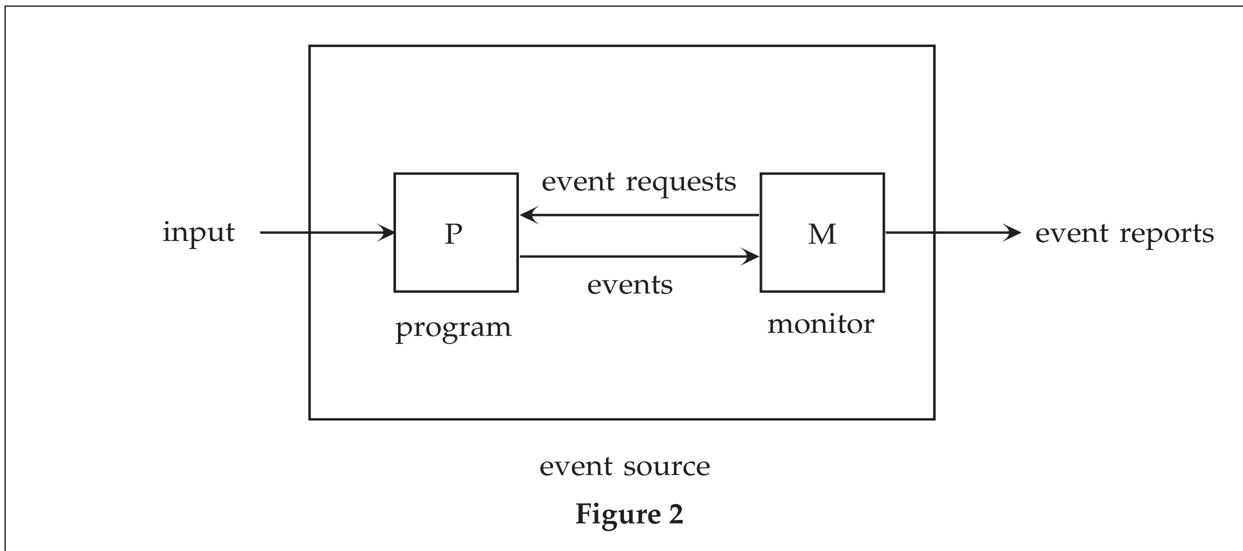


Figure 2

program being monitored [7]. This can be depicted as shown in Figure 3.

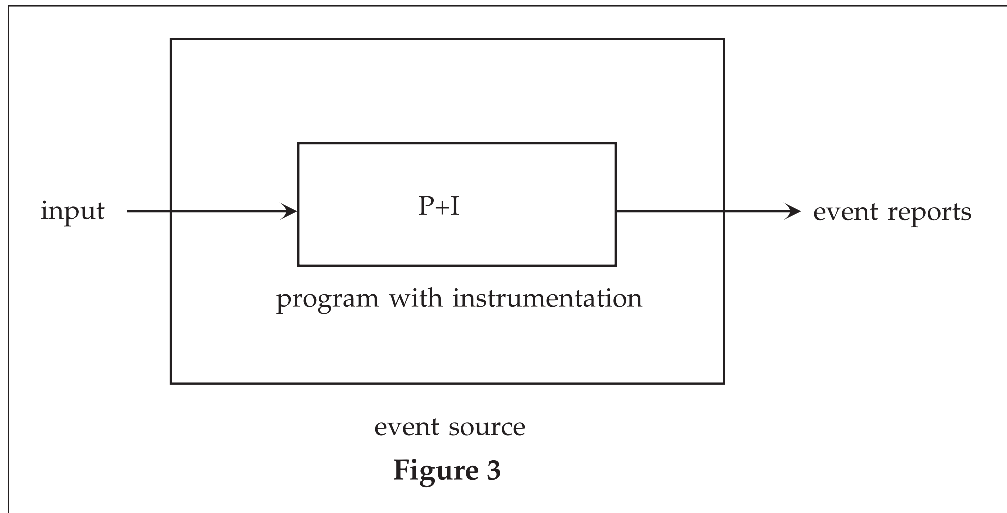


Figure 3

The event reporter also can take many forms. It can do anything from accumulating event reports and producing summary tabulations at the end of the execution of the monitored program to producing animated visualizations [6,8-11].

There are many other possibilities for RP, including simply recording reports in an "event history" for subsequent processing, as shown in Figure 4.

Using this scheme, the event history can be converted to events by a "history player", as shown in Figure 5 .

There are, of course, issues related to encoding events and how much information may be lost in the encoding.

We'll come back to other possibilities for this model in later articles, but note that event sources need not produce real events. For example, events can be fabricated to test ERs or for less serious purposes such as driving a visualization ER with random data.

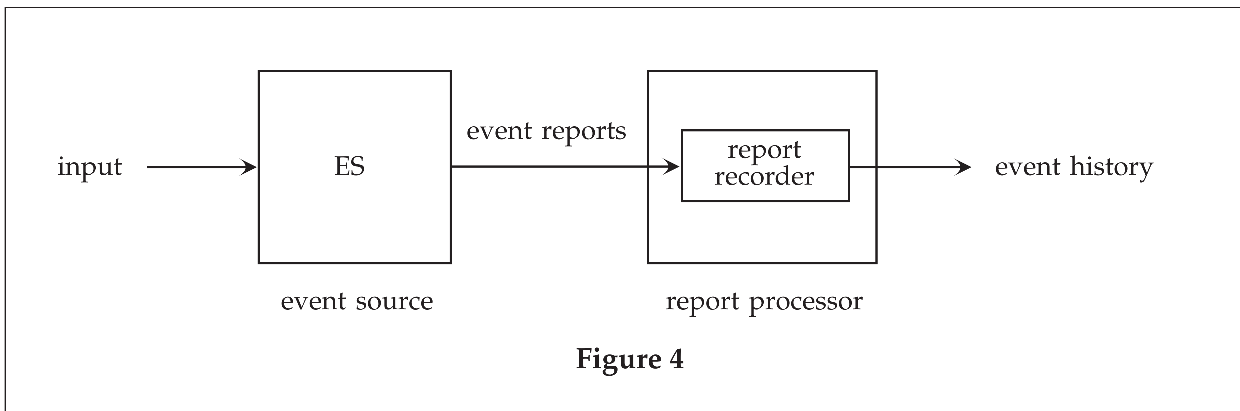


Figure 4

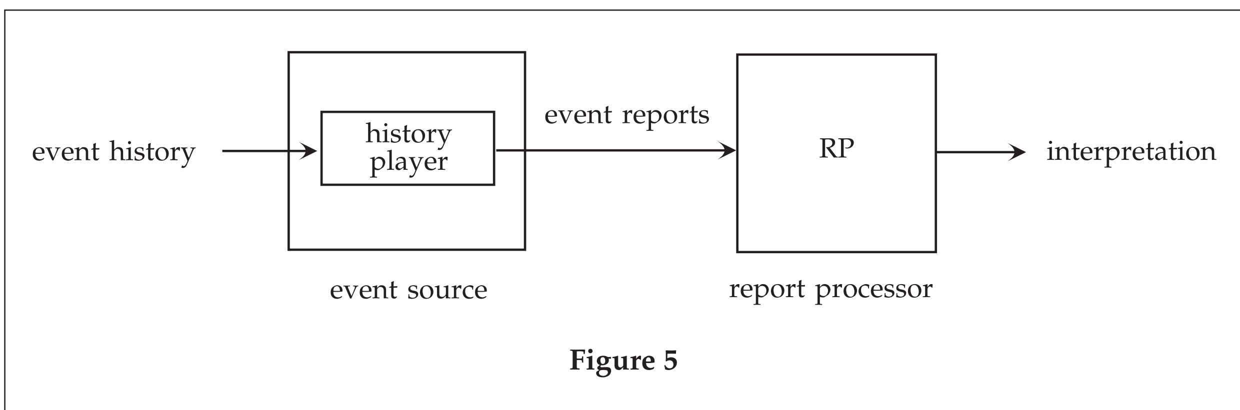


Figure 5

Event Spaces

An important concept in the framework we are developing is that of an *event space*. An event space characterizes the kinds of events that an ES produces and an RP interprets.

Events that consist of only a single value constitute an event space, pairs constitute another, triples another, and so on. Within any of these event spaces there are subspaces. For example, within the pair space there is a subspace in which the first value is a category and the second is a magnitude. Within this subspace, there is another in which the categories are the string-creation operations as described earlier. But the subspace of category-magnitude pairs also contains another subspace of allocation events within which the values are data types and the magnitudes are allocation sizes.

The reason this view is important is that RPs can be designed to operate on an event space and be used to, for example, visualize either string-creation events or allocation events. In an earlier Analyst article [11], we showed RPs that provided different abstract visualizations of storage allocation. These RPs can just as well be used to visualize string-creation events.

The question is interpretation: For the subspace of category-magnitude events, how are the categories and magnitudes to be interpreted? The RPs designed for abstract visualization of allocation events interpret categories (types of data) as colors. Magnitudes are interpreted in various ways such as width, length, area, and sometimes with nonlinear scaling. Some of the tools ignore magnitudes and project the category-magnitude event space onto a category event space.

To visualize string-creation events, categories (the creation operations) could be interpreted as colors. The interpretation of magnitudes used for allocation events generally suits string-creation events just as well.

Of course, the number of different categories and the range of magnitudes are serious considerations. The human visual system can distinguish among only a few colors at the same time, and hence only a few categories when interpreted in this way. And if the range of magnitudes is too large, some forms of visualizations are inappropriate.

The interpretation need not be built into the RP; it can be provided by another process in a pipeline or by a procedure called from the RP. One way to view this is shown in Figure 6 below.

These are only a few examples. There are many other ways of viewing event spaces and the interpretation of events. The important point is that event spaces provide a way of characterizing events and abstracting essential characteristics from them. This leads to the concept of RPs that can be designed for event spaces without *a priori* self-contained interpretation and hence be applicable to many different kinds of program activity.

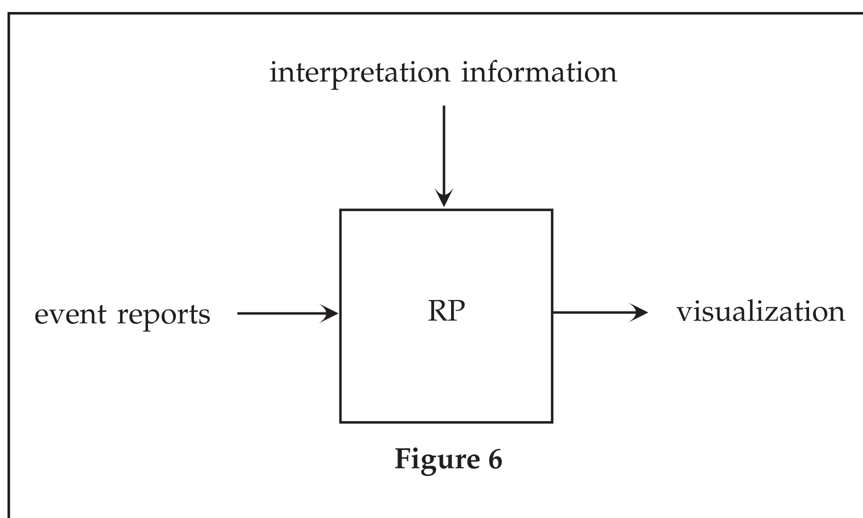


Figure 6

Conclusion

We've discussed the framework for monitoring in terms of Icon programs. It is much more general than that. The basic ideas apply to most programming languages. In a specific case, the instantiation, as well as what is possible and meaningful may be different. It also is possible to use the same RPs to visualize program activity in different programming languages. For example, the event source might be a Prolog program with the RP written in Java.

Note, however, that there many tools, especially visualization ones, already written in Icon. In many cases, they can be used to visualize events in programs written in C, Prolog, Java, or other programming languages. The catch is that it may not be so easy to get event reports from some languages. Adding instrumentation to a source program usually is the easiest route, but some languages have ready mechanisms for producing event

reports from the implementation. Hooks in Prolog are an example [12].

References

1. "Event Definition Language: An Aid to Monitoring Complex Software Systems", P. C. Bates and J. C. Wildeden, *Proceedings of the 5th Hawaii International Conferences on System Sciences*, 1982.

2. "A Framework for Dynamic Program Analysis", B. Bruegge, T. Gottschalk, and B. Luo, *Proceedings of the Eighth Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1993, 65-82.

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

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

5. "Monitoring Icon Programs", Icon Analyst 15, pp. 6-10.

6. "Dynamic Analysis of Icon Programs", Icon Analyst 28, pp. 9-12.

7. "Dynamic Analysis — A Different Approach", Icon Analyst 37, pp. 3-9.

8. "Dynamic Analysis of Icon Programs", Icon Analyst 29, pp. 10-12.

9. "Dynamic Analysis", Icon Analyst 30, pp. 6-11.

10. "Dynamic Analysis", Icon Analyst 33, pp. 3-6.

11. "Program Visualization", Icon Analyst 16, pp. 1-8.

12. *SICStus Prolog User's Manual*, Swedish Institute of Computer Sciences, 1993.

The Kaleidoscope

This has been a long series of articles, starting with Issue 31 of the Analyst [1]. We hope you're still with us — the end is in sight.

In the last article [2], we ran out of space while describing the callbacks. We'll finish that subject and then take a look at the VIB code for the application.

The Remaining Callbacks

The kaleidoscope application has only one menu, File. Here's its callback:

```
procedure file_cb(vidget, value)

   case value[1] of {
     "snapshot @S":  snapshot(pane,
       –half, –half, size, size)
     "quit          @Q":  exit()
     }

   return

end
```

The value in a menu callback always is a list. Its first element is the name of the selection. Its second value, if any, is the selection from the selected item's submenu, and so on. There are no submenus in the kaleidoscopes File menu, so value is a one-element list.

There are two menu items, as shown in the case expression. Note that the names of the items appear exactly as they do on the interface, with the keyboard shortcuts indicated. Showing the shortcuts in this way is, of course, an optional part of the interface design.

The snapshot item uses snapshot() from the library module interact. This procedure provides a dialog for the user to name a file in which to save the image, warns the user if there already is a file with that name, and so on. Notice that the origin of the drawing area and its extent are taken into account.

The quit item simply terminates program execution. (There is nothing to save in this application; in other applications there may be user work that has not been saved, and the user should be warned and given the opportunity to save the work.)

Although not a direct callback, the procedure that handles keyboard shortcuts is called as a result of user actions for events that are not handled by a vidget. The procedure itself is given as the third argument of ProcessEvent():

```
ProcessEvent(root, , shortcuts)
```

The only keyboard shortcuts are those shown in the File menu:

```
procedure shortcuts(e)

  if &meta then case map(e) of {          # fold case
    "q":      exit()
    "s":      snapshot(pane, –half, –half, size, size)
    }

  return

end
```

Note that the shortcuts only apply if the meta key is depressed when a letter is entered. This is a design decision and helps prevent accidental keystrokes from having unintended effects.

In a more complicated application, there might be many more shortcuts. Shortcuts also are handy for features that have no visible manifestation on the interface, such as undocumented "Easter eggs" and debugging facilities. Hidden features can be more difficult for an uninformed user to discover by requiring two modifier keys or a modifier key in combination with a mouse press instead of a character.

The remaining two callbacks control the maximum and minimum radii of circles:

```
procedure max_radius_cb(vidget, value)

  max_radius := value

  if max_radius < min_radius then {
    min_radius := max_radius
    VSetState(
      vidgets["sld_min_radius"], min_radius)
      )
    }

  reset := 1

  return

end

procedure min_radius_cb(vidget, value)

  min_radius := value

  if min_radius > max_radius then {
    max_radius := min_radius
    VSetState(
       vidgets["sld_max_radius"],
      max_radius
      )
    }

  reset := 1

  return

end
```

Here there is a need for communication between the two callbacks to prevent the possibility of a logical inconsistency in which the maximum radius is smaller than the minimum one. Each callback checks the two radii and, if necessary, sets the state of the other slider by calling VSetState() if the specified value would violate the constraints. This causes the two sliders to be set to the same value. From a user's viewpoint, trying the move the thumb of slider to a disallowed value merely moves the other slider along with it.

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.

The VIB Section

Until now, we've let the code produced by VIB remain mysterious. It's shown at the the right.

The comment lines at the beginning and end delimit the code produced by VIB and are used by VIB to find its code segment in a program when VIB is run again to modify the interface.

The procedure ui_atts() returns a list of attributes for the interface window. It can be used, for example, to find out how big the window is. We'll mention other uses for ui_atts() in a later article.

The procedure ui() initializes the interface by calling the library procedure vsetup(), which opens the interface window, draws the vidgets, and so on. The first two arguments of vsetup() are optional and are used in special situations. The remaining arguments to vsetup() describe the interface. The first is a list that describes the interface window. Following this, there is a list for each vidget.

It is not necessary to know what all the values in these lists are, although you probably can figure out most of them.

```
#===<<vib:begin>>===  modify using vib; do not remove this marker line
procedure ui_atts()
   return ["size=600,455", "bg=gray–white", "label=kaleido"]
end

procedure ui(win, cbk)
return vsetup(win, cbk,
   [":Sizer:::0,0,600,455:kaleido",],
   ["file:Menu:pull::12,3,36,21:File",file_cb,
      ["snapshot @S","quit     @Q"]],
   ["label01:Label:::13,180,21,13:min",],
   ["label02:Label:::152,180,21,13:max",],
   ["label03:Label:::13,240,21,13:min",],
   ["label04:Label:::152,240,21,13:max",],
   ["label05:Label:::13,300,21,13:min",],
   ["label06:Label:::152,300,21,13:max",],
   ["label07:Label:::7,120,28,13:slow",],
   ["label08:Label:::151,120,28,13:fast",],
   ["lbl_density:Label:::67,160,49,13:density",],
   ["lbl_max_radius:Label:::43,280,98,13:maximum radius",],
   ["lbl_min_radius:Label:::44,220,98,13:minimum radius",],
   ["lbl_speed:Label:::74,100,35,13:speed",],
   ["line:Line:::0,30,600,30:",],
   ["pause:Button:regular:1:33,55,45,20:pause",pause_cb],
   ["reset:Button:regular::111,55,45,20:reset",reset_cb],
   ["sld_density:Slider:h:1:42,180,100,15:1,100,50",density_cb],
   ["sld_max_radius:Slider:h:1:42,300,100,15:1,230,115",max_radius_cb],
   ["sld_min_radius:Slider:h:1:42,240,100,15:1,230,115",min_radius_cb],
   ["sld_shape:Choice::2:66,359,57,42:",shape_cb,
      ["discs","rings"]],
   ["sld_speed:Slider:h:1:42,120,100,15:500,0,250",speed_cb],
   ["region:Rect:raised::188,42,400,400:",],
   )
end
#===<<vib:end>>=== end of section maintained by vib
```

VIB Code for the Kaleidoscope Application

Note: It is, of course, possible to modify the code in the VIB section using a text editor. This is risky; if the results are not valid, VIB may be unable to process its code section. We confess, however, that we've edited VIB code on occasion to, for example, change the name of a callback procedure without using VIB.

The Complete Application

We've presented the code for the kaleidoscope application in pieces over two issues of the Analyst. Here's the complete program for reference (a luxury we afford now that we have more pages available). The program layout has been changed slightly to accommodate the constraints of the Analyst layout.

```
link interact
link random
link vsetup

# Interface globals

global vidgets          # table of vidgets
global root             # the root vidget
global size             # size of view area (width & height)
global half             # half size of view area
global pane             # graphics context for viewing

# Parameters that can be set from the interface

global delayval         # delay between drawing circles
```

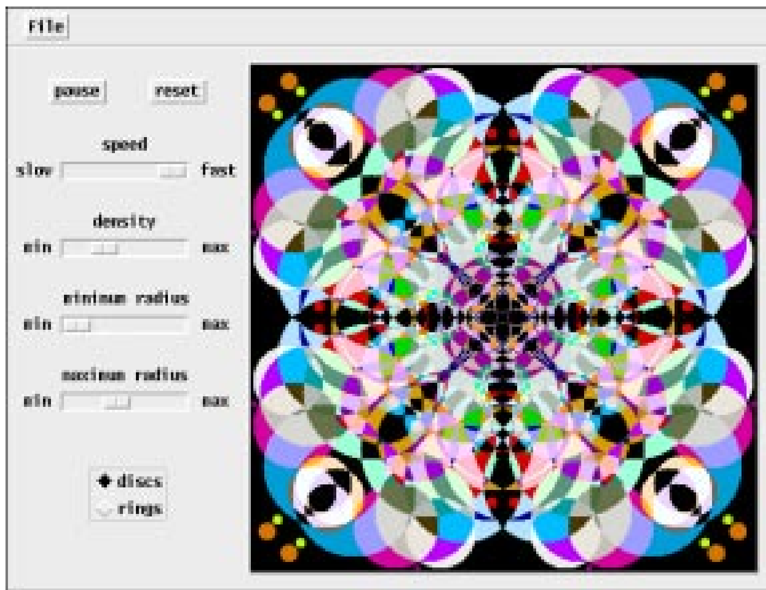
```
global density          # number of circles in steady state
global draw_proc        # drawing procedure
global max_off          # maximum offset of circle
global min_off          # minimum offset of circle
global max_radius       # maximum radius of circle
global min_radius       # minimum radius of circle

# State information

global draw_list        # list of pending drawing parameters
global reset            # nonnull to reset display
global state            # nonnull when display paused

# Main procedure

procedure main(args)

  init()

  kaleidoscope()

end

# initialization

procedure init()

  vidgets := ui()

  root := vidgets["root"]
  size := vidgets["region"].uw
  if vidgets["region"].uh ~= size then
    stop("*** improper interface layout")

  # produce different display on every execution

  randomize()

  # set initial values

  delayval := 0
  density := 30
  max_radius := size / 4
  min_radius := 1
  draw_proc := FillCircle

  state := &null

  # initialize vidget values

  VSetState(vidgets["sld_speed"], delayval)
  VSetState(vidgets["sld_density"], density)
  VSetState(vidgets["sld_min_radius"], min_radius)
  VSetState(vidgets["sld_max_radius"], max_radius)
  VSetState(vidgets["sld_shape"], "discs")

  # get graphics context for drawing

  half := size / 2

  pane := Clone("bg=black", "dx=" ||
    (vidgets["region"].ux + half),  "dy=" ||
      (vidgets["region"].uy + half), "drawop=reverse")

  return

end

# The kaleidoscope

procedure kaleidoscope()

  #  Each time through this loop, the display is cleared and a
  #  new drawing is started.

  repeat {
    EraseArea(pane, –half, –half, size, size)   # clear display
    draw_list := []                              # new  list
    reset := &null

   #  In this loop a new circle is drawn and an old one erased,
    #  once the specified density has been reached.  This
    #  maintains a steady state.

    repeat {
      while (*Pending() > 0) | \state do {
        ProcessEvent(root, , shortcuts)
        if \reset then break break next
        }
      putcircle()
      WDelay(delayval)

      #  Don't start clearing circles until the specified density
      # has  reached. (The drawing list has four elements for
      # each  circle.)

      if *draw_list > (4 * density) then clrcircle()
      }
    }
end

# Drawing procedures

procedure clrcircle()

  outcircle(
    get(draw_list),                 # off1
    get(draw_list),                 # off2
    get(draw_list),                 # radius
    get(draw_list)                  # color
    )

  return

end

procedure outcircle(off1, off2, radius, color)

  Fg(pane, color)

  # Draw in symmetric positions.

  draw_proc(pane, off1, off2, radius)
  draw_proc(pane, off1, –off2, radius)
  draw_proc(pane, –off1, off2, radius)
  draw_proc(pane, –off1, –off2, radius)
  draw_proc(pane, off2, off1, radius)
  draw_proc(pane, off2, –off1, radius)
  draw_proc(pane, –off2, off1, radius)
  draw_proc(pane, –off2, –off1, radius)

 return

end

procedure putcircle()
  local off1, off2, radius, color
```

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

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

```
   static colors

   initial colors := PaletteChars("c1")

   # get a random center point and radius

   off1 := ?size % half
   off2 := ?size % half
   radius := ((max_radius – min_radius) * ?0 + min_radius) %
      (half – ((off1 < off2) | off1))

   color := PaletteColor("c1", ?colors)

   put(draw_list, off1, off2, radius, color)

   outcircle(off1, off2, radius, color)

   return

end

# Callbacks

procedure density_cb(vidget, value)

   density := value

   reset := 1

end

procedure file_cb(vidget, value)

   case value[1] of {
      "snapshot @S":  snapshot(pane, –half, –half, size, size)
      "quit        @Q":  exit()
      }

   return

end

procedure max_radius_cb(vidget, value)

   max_radius := value

   if max_radius < min_radius then {
      min_radius := max_radius
      VSetState(vidgets["sld_min_radius"], min_radius)
      }

   reset := 1

   return

end

procedure min_radius_cb(vidget, value)

   min_radius := value

   if min_radius > max_radius then {
      max_radius := min_radius
      VSetState(vidgets["sld_max_radius"], max_radius )
      }

   reset := 1

   return

end

procedure pause_cb(vidget, value)

   state := value

   return

end

procedure reset_cb(vidget, value)

   reset := 1

   return

end

procedure shape_cb(vidget, value)

  draw_proc := case value of {
    "discs":  FillCircle
    "rings":  DrawCircle
    }

  reset := 1

  return

end

procedure speed_cb(vidget, value)

  delayval := sqrt(value)

  return

end

# Keyboard shortcuts

procedure shortcuts(e)

  if &meta then case map(e) of {       # fold case
     "q":    exit()
     "s":    snapshot(pane, –half, –half, size, size)
     }

  return

end

#===<<vib:begin>>===      modify using vib; do not remove …
procedure ui_atts()
  return ["size=600,455", "bg=gray–white", "label=kaleido"]
end

procedure ui(win, cbk)
return vsetup(win, cbk,
  [":Sizer:::0,0,600,455:kaleido",],
  ["file:Menu:pull::12,3,36,21:File",file_cb,
     ["snapshot @S","quit     @Q"]],
  ["label01:Label:::13,180,21,13:min",],
  ["label02:Label:::152,180,21,13:max",],
  ["label03:Label:::13,240,21,13:min",],
  ["label04:Label:::152,240,21,13:max",],
  ["label05:Label:::13,300,21,13:min",],
  ["label06:Label:::152,300,21,13:max",],
  ["label07:Label:::7,120,28,13:slow",],
  ["label08:Label:::151,120,28,13:fast",],
  ["lbl_density:Label:::67,160,49,13:density",],
  ["lbl_max_radius:Label:::43,280,98,13:maximum radius",],
  ["lbl_min_radius:Label:::44,220,98,13:minimum radius",],
  ["lbl_speed:Label:::74,100,35,13:speed",],
  ["line:Line:::0,30,600,30:",],
  ["pause:Button:regular:1:33,55,45,20:pause",pause_cb],
  ["reset:Button:regular::111,55,45,20:reset",reset_cb],
  ["sld_density:Slider:h:1:42,180,100,15:1,100,50",
     density_cb],
  ["sld_max_radius:Slider:h:1:42,300,100,15:1,230,115",
     max_radius_cb],
  ["sld_min_radius:Slider:h:1:42,240,100,15:1,230,115",
     min_radius_cb],
```




```
  ["sld_shape:Choice::2:66,359,64,42:",shape_cb,
     ["discs","rings"]],
  ["sld_speed:Slider:h:1:42,120,100,15:2000,0,250",
     speed_cb],
  ["region:Rect:raised::188,42,400,400:",],
  )
end
#===<<vib:end>>===        end of section maintained by vib
```

Next Time

That's the end of the kaleidoscope application. We have one other topic to cover before we leave the subject of interface design altogether: dialogs. We'll start with that subject in the next issue of the Analyst.

References

1. "Visual Interfaces", Icon Analyst 31, pp. 1-4.

2. "The Kaleidoscope", Icon Analyst 38, pp. 8-13.

Versum Bimorphs

In the last article on versum numbers [1], we explored predecessors of versum numbers — numbers whose reverse sums produce versum numbers. We showed that a versum number has at most two inequivalent predecessors and called versum numbers with two predecessors bimorphs. We also observed that there are very few bimorphs among versum numbers.

We have a few more things related to bimorphs before going on to other aspects of versum numbers.

Bimorph Correspondences

One aspect of bimorphs noted in the last article is that there are the same number of bimorphs for successive odd/even numbers of digits. For example, there are 200 13-digit bimorphs and 200 14-digit bimorphs. We've found this to hold through 16-digit bimorphs, but we haven't tried to prove it; take it as a conjecture.

We can easily compute the n-digit bimorphs, n even, from the (n-1)-digit bimorphs. The process involves working with 2x9 predecessors of the (n-1)-digit bimorphs.

A 2x9 predecessor of an (n-1)-digit bimorph has (n-2) digits. Since the number of digits is even, we can split the predecessor in half and insert either a 0 or 9 in the center, whichever makes the result divisible by 11 (and one or the other will). The result is a 2x9 predecessor of an n-digit bimorph. For example, the 2x9 predecessor of the 9-digit bimorph 119777801 is 20007799. Inserting a 9 in the middle of this predecessor produces 200097799, which is divisible by 11 and is the predecessor of the 10-digit bimorph 1197887801. (Incidentally, this only works for $n \geq 5$.)

Witchcraft? Not entirely. Note that divisibility by 11 raises its head again. We'll come back to that. A 2x9 predecessor of a bimorph with an even number of digits must be divisible by 11, as must a 1x0 predecessor of a bimorph with an odd number of digits.

Here's a procedure that "promotes" an (n-1)-digit bimorph, n even, to an n-digit bimorph:

```
link vpred

procedure promoter(j)
   local i, j, try, lh, rh, count

      if (*j % 2 = 0) | (*j < 5) then fail
      count := 0
      every i := vpred(j) do          # get 2x9 predecessor
         count +:= 1
      if count ~= 2 then fail         # not bimorph
      lh := left(i, *i / 2)
      rh := right(i, *i / 2)
      every try := lh || ("0" | "9") || rh do
         if try % 11 = 0 then
            return try + reverse(try)

end
```

The procedure vpred() is used to get the predecessor. The section of code

```
every i := vpred(j) do          # get 2x9 predecessor
   count +:= 1
if count ~= 2 then fail         # not bimorph
```

is explained by the fact that vpred() is designed to produce a $1x0$ predecessor first.

It's also possible to "demote" an n-digit bimorph, n even, to an (n-1)-digit bimorph by deleting the middle digit of its $2x9$ predecessor, which produces a $2x9$ predecessor of an (n-1)-digit bimorph. If we'd used demotion instead of promotion above, the mysterious divisibility by 11 wouldn't have appeared and the process would have seemed simpler. In any event, promotion and demotion produce a one-to-one correspondence between (n-1)-digit bimorphs, n even, and n-digit bimorphs.

Divisibility by 11

The divisibility of versum numbers by 11 has come up several times. There's nothing magic about 11 — it's a consequence of representing numbers in the base 10. In general, in base b, divisibility by b+1 applies in the same way.

Among various divisibility criteria, it's known that b+1 divides a number i represented in base b if b+1 divides the difference of the sums of the odd and even digits.

For example, consider the base-10 number 8827391431036289. The sum of the even-numbered digits is 42 and the sum of the odd-numbered digits is 31; the difference is 11, which of course is divisible by 11, so 8827391431036289 is divisible by 11.

Divisibility by 11 is relevant to versum numbers because if you compute the reverse sum of a number with an even number of digits, the odd and even digits line up and the difference of the sums is 0 and hence divisible by 11. In other words, the reverse sum of a number with an even number of digits is divisible by 11. Since all versum bimorphs have a predecessor with an even number of digits (either of the form $1x0$ or $2x9$), all versum bimorphs are divisible by 11.

Since all versum bimorphs are divisible by 11, we can further improve vpred():

```
procedure vpred(i)
   local s, firstp

    if (i[1] == "1" == i[–1]) & (i % 11 = 0)) then {
      every s := integer(vpred_(i)) do {
         if (s + reverse(s)) = i then {
            s := vprimary(s)
            (/firstp := s) | {
               if s ~= firstp then {
                  suspend firstp
                  return s
                  }
               }
            }
         }
      return \firstp                    # may be none
      }
   else {                               # not bimorph
                  …
```

The divisibility of bimorphs by 11 doesn't explain why the $2x9$ predecessors of n-digit bimorphs, n even, are divisible by 11 or why the $1x0$ predecessors of such bimorphs also are divisible by 11, but you probably get the feeling that there's a "connection".

Palindromic Bimorphs

The versum problem originated because of interest in palindromes [2], so it's appropriate to look at palindromic bimorphs.

Palindromic bimorphs seem quite remarkable. For example, the number of palindromic n-digit bimorphs is $2^{\lfloor (n-1)/2 \rfloor -1}$ ($n > 2$) where $\lfloor r \rfloor$ is the integer part of r. Consequently, there are the same numbers of palindromes for successive odd/even digit bimorphs, just as there are the same number of bimorphs. In fact, that should not be so surprising, since you'd expect them to pair up. Here are some numbers:

n	*palindromic bimorphs*
3	1
4	1
5	2
6	2
7	4
8	4
…	

Icon on the Web

Information about Icon is available on the World Wide Web at

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

The power of 2 suggests a doubling process in which an *n*-digit palindromic bimorph "produces" two (*n*-2)-digit ones. There's a doubling process, but it occurs in a different way.

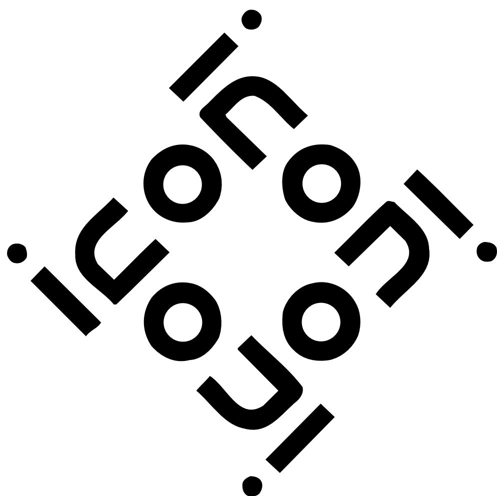
Let's start by looking at some palindromic bimorphs. Here are the 9-digit ones:

```
110000011
110121011
111101111
111222111
121000121
121121121
122101221
122222221
```

As in other cases with bimorphs, it's instructive to look at their 2*x*9 predecessors, since we know how to get corresponding 10-digit bimorphs from them:

```
20000009
20029009
20200909
20229909
22000099
22029099
22200999
22229999
```

These numbers have several interesting properties. They consist entirely of 2s, 9s, and 0s. All the 2s are in the left half and all the 9s are in the right half. There are as many 2s as 9s in each number (and hence all are divisible by 11). Furthermore, all combinations of 2s and 0s, as well as 9s and 0s, occur except for all 0s (which would not be of the form 2*x*9). And if we replace all the 2s and 9s by, say 4s, the resulting patterns are palindromic:



```
400040004
400444004
404040404
404444404
440040044
440444044
444444444
```

And although we haven't proved it, these regularities hold for the 2*x*9 predecessors of all *n*-digit bimorphs, *n* odd, at least through *n* = 15.

Here's a procedure to generates the *n*-digit palindromic bimorphs, *n* even:

```
procedure bipalgen(n)
  local s, mid

  if n < 2 then fail

  if n % 2 = 0 then {
    mid := "0"
    n –:= 1
    }
  else mid := ""

  every s := allpat("02", (n – 3) / 2) do {
    s := "2" || s || mid ||
      map(reverse(s), "2", "9") || "9"
    suspend s + reverse(s)
    }

end

procedure allpat(s, i)

  if i = 0 then return ""

  suspend !s || allpat(s, i – 1)

end
```

The procedure allpat() is a recursive generator that generates all patterns of the characters in its argument. See Reference 3.

One other observation: At least through *n* = 16, palindromic bimorphs contain only the digits 0, 1, and 2.

Back Issues

Back issues of *The Icon Analyst* are available for $5 each. This price includes shipping in the United States, Canada, and Mexico. Add $2 per order for airmail postage to other countries.

Next Time

Having brought up the issue of divisibility by 11, we'll go on to an exploration of the factors of versum numbers, and, of course, prime versum numbers.

References

1. "Versum Predecessors", Icon Analyst 37, pp. 11-15.

2. "The Versum Problem", Icon Analyst 37, pp. 1-4.

3. *The Icon Programming Language*, 3rd edition, Ralph E. Griswold and Madge T. Griswold, Peer-to-Peer Communications, Inc., 1996, p. 210.

Program Size

In our courses on graphics programming and string and list processing, each student does a large individual programming project in lieu of a final examination. The students are concerned, quite naturally, about what is expected for their projects — especially since the choice of topic and the project design, implementation, and documentation is the individual responsibility of each student.

Although we stress functionality, design, and robustness as the main concerns, these are qualitative measures and students want quantitative ones: "How many lines of code do I have to write?"

Although code bulk is not our major criterion in evaluating projects, it bears at least some relationship to effort and program functionality. We usually end up placating anxious students by saying something like "Well, in the past, the best projects have ranged from 1,500 to 3,000 lines of code". (If the projects were done in, say, C instead of Icon, the figure probably would be 5 to 10 times larger. One of the reasons for using Icon in an educational context is that much more can be done in the course of a semester than if a more conventional language were used.)

This brings us to the more fundamental question of how to measure program size. In the **From the Library** article in the last issue of the Analyst, we pointed out that functionality may depend to a large extent on linked procedures. Of course, where student effort is an issue, it's the code they write themselves, not the library code they link, that is of primary interest.

There's a program in the Icon program library, isrcline, that counts the lines of code in a file that contain actual code, as opposed to full-line comments and blank lines. We use this program as one of the preliminary parts of evaluating student projects, primarily to identify programs that are unusually small or large (it's not a part of the grading process). The program also counts non-code lines, which gives an indication of program layout and the amount of commentary.

Of course the number and percentage of code lines is only a rough measure. The same program can be laid out in different ways to give quite different results. For example, any Icon program can be written on a single line (there is no limit to the length of line that the Icon translator can handle). You'd never manage to write anything but a trivial program on a single line by hand, and if you try, you may discover some idiosyncracies of Icon's syntax, like situations in which whitespace but not semicolons must be used to separate declarations that normally would be written on separate lines (try putting a semicolon after end).

As an amusement, we wrote an Icon meta-translator [1] that converts any Icon program into a one-line equivalent. This translator only has joke value. We can, for example, advertise that an entire relational database program can be written in one line in Icon.

But what *is* a good measure of Icon program size? You might think that the size of the icode file produced by compiling a program would be a good measure. There are two problems with this. One is that it includes linked procedures, and hence can only measure total program size, not, for example, the amount of code a student writes for a project. Another problem is that there are idiosyncracies in the way icode files are laid out that can give misleading information. For example, having many records with many fields bloats icode files. (One of Clint Jeffery's students has provided modifications to Icon to mitigate this problem, but those changes currently are not enabled.)

The number of bytes in an Icon source-language file often is a better measure of program size than the number of source-code lines. But that size is easily distorted, for example, by many long string literals, as in built-in help information and string images for graphics. Or on a more twisted

note, there is no limit on the length of an Icon identifier, so the size of an otherwise "normal" program can be bloated by using very long identifiers.

But jokes and bogus programs aside, the measure of program size that we prefer is the number of syntactic tokens it contains. Tokens are things like identifiers, reserved words, literals, operators, and so on. To count tokens, we use the meta-translator described in the articles on static analysis [2]. This meta-translator's view of "token" is somewhat different from a typical lexical analyzer's — the meta-translator, for example, views an if-then-else control structure as one token, not three. For our purposes, that's preferable.

We applied these measures to the final projects in a string-and-list-processing course given in the spring of 1992 and described in *Icon Newsletter 39* [3]. The course co-convened an upper-division undergraduate section (450) and a graduate section (550). There were 14 graduate students and seven undergraduate students. All but one student completed the course.

The results are shown below, with student names converted to numbers to protect their identities. The results are ordered in terms of decreasing numbers of tokens.

The project grades were based on the qualitative factors mentioned earlier, as well as documentation. Notice that although some of the top grades are associated with large programs, the correlation is relatively weak. Note, for example, that programs 1 and 13 received the same grade, although the former is nearly four times as large as the latter. (We're not revealing anything embarrassing by telling you that Mary Cameron wrote the largest program, which was the precursor to the visual interface builder, VIB.)

We find number of bytes per token to be interesting. High values are suspect and usually are due to many long string literals. These may occur for good reasons, as mentioned earlier, but they usually do not contribute much to program functionality.

The number of tokens per code line usually is related to program layout. Programs with high values tend to be short on documentation and hard to read. Incidentally, program 19 was *very* easy to read — but it didn't compile.

As another example, we applied these measures to the kaleidoscope application as shown on pages 7 through 10 of this issue of the Analyst. We

number	*tokens*	*code lines*	*lines*	*bytes*	*bytes/token*	*tokens/code line*	*grade*	*class*
1	9670	1990	2682	73900	7.64	4.86	95	550
2	8659	1686	1896	54289	6.27	5.14	80	450
3	7645	1317	1903	50912	6.66	5.80	92	550
4	7586	761	1200	39063	5.15	9.97	92	550
5	7322	1547	2206	50476	6.89	4.73	85	550
6	6483	1450	2111	55776	8.06	4.47	90	550
7	5406	937	1574	44872	8.30	5.77	95	550
8	4455	931	1519	38897	8.73	4.79	90	550
9	4072	872	1740	54669	13.43	4.67	82	550
10	3364	751	1063	32586	9.69	4.48	90	550
11	2785	613	1065	25447	9.14	4.54	85	550
12	2540	695	1362	33759	13.29	3.65	83	550
13	2500	398	492	13929	5.57	6.28	95	550
14	2447	714	1476	41923	17.13	3.43	85	450
15	1958	265	472	12171	6.22	7.39	84	450
16	1815	402	686	24963	13.75	4.51	90	550
17	1205	375	590	10671	8.86	3.21	85	450
18	1098	223	486	14676	13.37	4.92	80	550
19	398	363	521	9871	24.80	1.10	30	450
20	209	62	79	1658	7.93	3.37	50	450

did three measurements: the body of the program, the VIB segment, and the whole program combining the two. Here are the results, where we've abbreviated the labels on the last two columns so that they would fit in the space:

	tokens	*code lines*	*lines*	*bytes*	*b/t*	*t/cl*
body	500	151	293	6065	8.41	3.31
VIB	74	32	35	1487	18.43	2.31
all	574	183	329	7553	9.70	3.13

We explained earlier why we didn't use icode file sizes to measure the size of student projects. For what it's worth, here they are, built with Version 9 of Icon on an Alpha with 64-bit words (for which icode file sizes are considerably larger than for platforms with 32-bit words):

1	220664	11	53710
2	201150	12	65587
3	176577	13	60735
4	97429	14	65074
5	77226	15	24857
6	118225	16	38672
7	121031	17	46384
8	119840	18	30073
9	68096	19	—
10	90606	20	7253

Are you prepared for this? The size of the icode file for the kaleidoscope application is 563479 bytes. The explanation for this much-larger size lies in the functionality of the application: a visual interface and all that goes with it. With Icon's run-time system and storage regions, the kaleidoscope application requires more than 1MB of RAM.

Not many years ago, we would have been horrified by an application of this size. When we developed ProIcon for the Macinoth in 1989, we worried that the 400KB of RAM required and the 1MB recommended would exclude many potential users. It's hard to comprehend how quickly things related to computers change. Now many applications require 10 to 20MB.

References

1. "Meta-Variant Translators", The Icon Analyst 23, pp. 8-10.

2. "Static Analysis of Icon Programs", Icon Analyst 27, pp. 5-11.

3. "Icon Class Projects", *Icon Newsletter 39*, pp. 11-12.

In an earlier article [1], we described a very general and powerful library procedure for encoding values as strings so that they could be saved in files and restored later. In the cases where the values are not structures or other complicated values, there are simpler methods.

An example occurred when we were monitoring string concatenation [2]. In one experiment, we wanted to know the actual strings produced so that we could determine how many duplicates there were, the distributions of individual characters, and so forth. In the monitoring model we were using, the strings were written out and processed later in various ways.

Since Icon allows any character to occur in a string, the result of a concatenation might include line terminators and hence split a single result into more than one, producing erroneous results.

Writing and reading in "binary" mode without terminators wouldn't help, since it was important to distinguish each string resulting from concatenation and there was no separator that might not appear in the data itself.

The solution to such a problem is to use image(), which provides escape sequences for characters that are "unprintable". The function image() also produces surrounding quotes, double for strings and single for csets. This allows, for example, the values 1, "1", and '1' to be distinguished.

That takes care of encoding the data. The problem then is the decoding. This is rather messy, since there are many escape sequences. Although image() uses hexadecimal escapes for most characters that need encoding, it uses the more mnemonic escapes, like \l for the linefeed character, where they are available.

Once again, the Icon program library can come to the rescue (if asked). It contains a procedure escape() that interprets escape sequences and produces appropriate characters for them. We won't show escape() here; it has to handle a lot of possibilities and the code is not very illuminating, but if you're curious, check the library.

With escape(), it's easy to decode an imaged string:

```
decode := escape(encode[2:–1]))
```

In some situations, you may want to encode other data like numbers, the null value, keywords, and so forth on a line-per-value basis. The library procedure ivalue() expands on the capabilities of escape() to handle numbers, keywords, non-local variables, functions, procedures, and, in a limited way, structures.

Instances of non-local variables, functions, and procedures must, of course, exist in the context in which ivalue() is evaluated. This generally is the case for functions. For example, image(upto) produces "function upto" and ivalue("function upto") produces the function upto.

For the imaged value of a structure, ivalue() produces a value of the right types and sizes, but the contents, of course, generally are not correct, since that information is not provided by image(). For example, if vector is a list with 1000 elements, image(vector) produces a string such as "list_21(1000)" (where the 21 is the serial number for the list), and ivalue("list_21(1000)") produces a list of 1000 elements (all of which happen to be null).

A few facts about keywords deserve mention. For some keywords, image() produces the name of the keyword, not the value. For example, image(&cset) produces "&cset". (Pop quiz: For what keywords does image() produce the keyword name?)

The procedure ivalue() handles these cases. And, for keywords that are variables, ivalue() produces a variable. You can do things like

```
ivalue("&subject") := "Hello world."
```

Although we motivated this article by encoding and decoding arbitrary strings, there are other uses for ivalue(). See the description of icalc.icn in Reference 3.

We'll show the code for ivalue() in a later article.

References

1. "From the Library", Icon Analyst 34, pp. 9-12.

2. "Dynamic Analysis — A Different Approach", Icon Analyst 37, pp. 3-9.

3. "Anatomy of a Program", Icon Analyst 12, pp. 2-4.

In the next issue of the Analyst, we plan to have the first of a short series of articles on dialog windows. We also have an article on the factors of versum numbers waiting in the wings, as it were. And we have material on dynamic analysis backed up.

It's time, however, to do something different from what we've had in recent issues. For this, we'll start a series of articles on debugging.