The Icon Analyst

*In-Depth Coverage of the Icon Programming Language*

**April 1997**
Number 41

In this issue …



## Custom Dialogs

If no standard dialog fits a particular need, a customized dialog can be built using VIB. The method for building a dialog using VIB is very similar to the one used for building an application interface. The few differences are noted in the following example.

### An Example — Dialog for Setting Attributes

There are several commonly used graphics attributes that a painting or drawing application might allow the user to change — attributes like foreground and background colors, the font, line width, line style, and so forth.

One approach to this problem is to provide a variety of standard dialogs such as text-entry dialogs for entering color names and the font, a selection dialog for picking a line style, and so forth. Another approach is to use a single custom dialog with which all attributes of interest can be set.

In order to create such a dialog using VIB, VIB must be informed that a dialog, rather than an application interface, is being created. This is done in the VIB canvas dialog by checking the dialog window box and entering the name of a procedure, as shown in Figure 1:

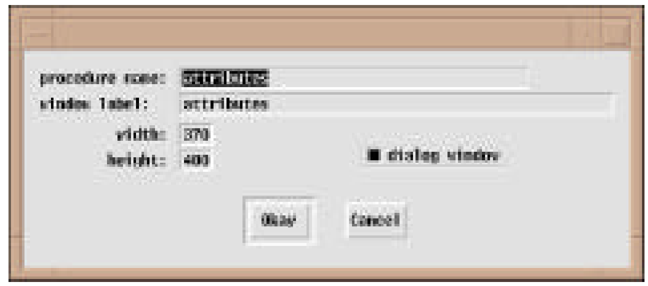



**Figure 1: Specifications for a Custom Dialog**

The window label is irrelevant for a dialog; the label for the dialog is inherited from the window of the application that invokes the dialog.

Next the vidgets for the custom dialog are created and placed as they are in building an application interface. See Figure 2:

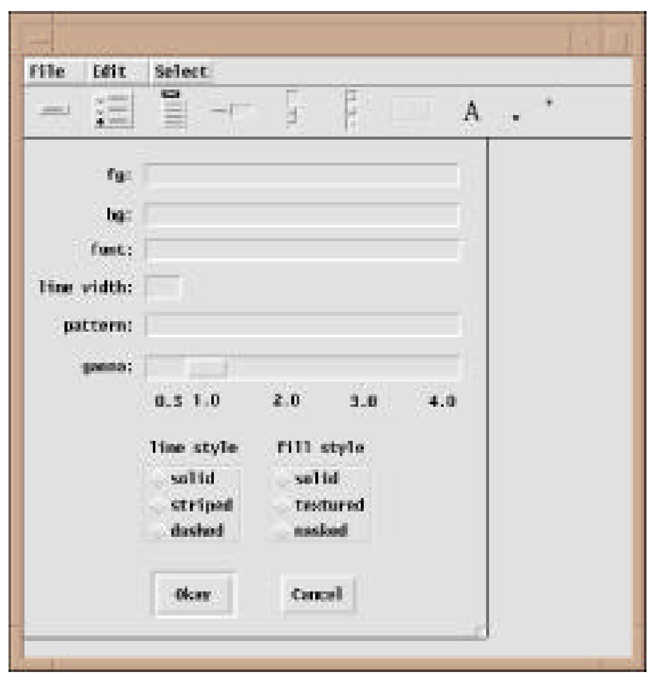



**Figure 2: The Layout for a Custom Dialog**

The order in which text-entry fields in a custom dialog are selected when the user presses the tab key is the lexical order of their IDs. In constructing this dialog we used IDs 1_bg, 2_fg, 3_font, and so on so that the fields would be selected in the order in which they are arranged in the dialog.

A dialog must have at least one regular button; otherwise there would be no way to dismiss it. VIB enforces this. A default button can be designated by selecting dialog default in the button dialog as shown in Figure 3:

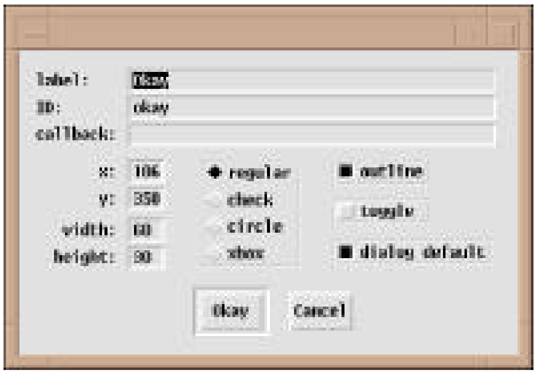



**Figure 3: The Default Button Dialog**

The code produced by VIB for a custom dialog is similar to that produced for an application. It is shown later at the end of the complete listing that follows later.

## Using a Custom Dialog

A custom dialog is invoked by calling the procedure named in VIB's canvas dialog. The argument of the procedure is a table whose keys are the IDs of the vidgets in the dialog and whose corresponding values are the states of these vidgets.

When a dialog is dismissed, it returns the text of the button used to dismiss it (as for standard dialogs). Before returning, it also changes the values in the table to correspond to the states of the vidgets when the dialog is dismissed.

In the case of the attribute dialog, a significant amount of work is needed to set up the table before invoking the dialog. After the dialog is dismissed, more work is needed to set the attributes. This code is encapsulated in the following procedure.

```
link dsetup                 # dialog setup

procedure attribs(win)
  static atts

  initial atts := table()   # table of vidget IDs

  /win := &window

  # Assign values from current attributes.

  atts["1_fg"] := Fg(win)
  atts["2_bg"] := Bg(win)
  atts["3_font"] := Font(win)
  atts["4_linewidth"] := WAttrib(win, "linewidth")
  atts["5_pattern"] := WAttrib(win, "pattern")
  atts["linestyle"] := WAttrib(win, "linestyle")
  atts["fillstyle"] := WAttrib(win, "fillstyle")
  atts["gamma"] := WAttrib(win, "gamma")

  repeat {

    # Call up the dialog.

    attributes(win, atts) == "Okay" | fail

    # Set attributes from table.

    Fg(win, atts["1_fg"]) | {
      Notice("Invalid foreground color.")
      next
      }
    Bg(win, atts["2_bg"]) | {
      Notice("Invalid background color.")
      next
      }
    Font(win, atts["3_font"]) | {
      Notice("Invalid font.")
      next
      }
    WAttrib(win, "linewidth=" ||
       integer(atts["4_linewidth"])) | {
        Notice("Invalid line width.")
        next
        }
    WAttrib(win, "pattern=" || atts["5_pattern"]) | {
        Notice("Invalid pattern.")
        next
        }
    WAttrib(win, "linestyle=" || atts["linestyle"])
    WAttrib(win, "fillstyle=" || atts["fillstyle"])
    WAttrib(win, "gamma=" || atts["gamma"])

    return

    }
end

#===<<vib:begin>>===
procedure attributes(win, deftbl)
static dstate
initial dstate := dsetup(win,
  ["attributes:Sizer::1:0,0,370,400:attributes",],
  ["0.5:Label:::105,204,21,13:0.5",],
  ["1.0:Label:::135,203,21,13:1.0",],
  ["1_fg:Text::35:10,20,339,19:       fg: \\=",],
```

```
   ["2.0:Label:::199,203,21,13:2.0",],
   ["2_bg:Text::35:10,52,339,19:       bg: \\=",],
   ["3.0:Label:::261,204,21,13:3.0",],
   ["3_font:Text::35:11,80,339,19:      font: \\=",],
   ["4.0:Label:::324,204,21,13:4.0",],
   ["4_linewidth:Text::3:11,110,115,19:line width: \\=",],
   ["5_pattern:Text::35:11,140,339,19:   pattern: \\=",],
   ["button1:Button:regular::206,350,60,30:Cancel",],
   ["fill label:Label:::202,241,70,13:fill style",],
   ["fillstyle:Choice::3:195,262,85,63:",,
     ["solid","textured","masked"]],
   ["gamma:Slider:h:1:97,174,253,20:0.5,4.0,1.0",],
   ["glabel:Label:::11,176,84,13:     gamma: ",],
   ["line label:Label:::100,241,70,13:line style",],
   ["linestyle:Choice::3:96,262,78,63:",,
     ["solid","striped","dashed"]],
   ["okay:Button:regular:-1:106,350,60,30:Okay",],
   ["tick1:Line:::117,196,117,201:",],
   ["tick2:Line:::146,195,146,200:",],
   ["tick3:Line:::209,195,209,200:",],
   ["tick4:Line:::272,195,272,200:",],
   ["tick5:Line:::335,195,335,200:",],
   )
return dpopup(win, deftbl, dstate)
end
#===<<vib:end>>===
```

Error checking is needed when the dialog is dismissed because the user may have entered inappropriate values in the text-entry fields. An invalid attribute value can be detected by the failure that occurs when attempting to set it. (The radio button choices are guaranteed to be valid by virtue of the button names used, and the gamma value is guaranteed to be a number in the range specified by its endpoint values.) In the case of an erroneous value, attributes() is called again in the repeat loop enclosing it. Only if all values are legal does attribs() return.

An example of the use of the attribute dialog is shown in Figure 4 at the top of the next column.

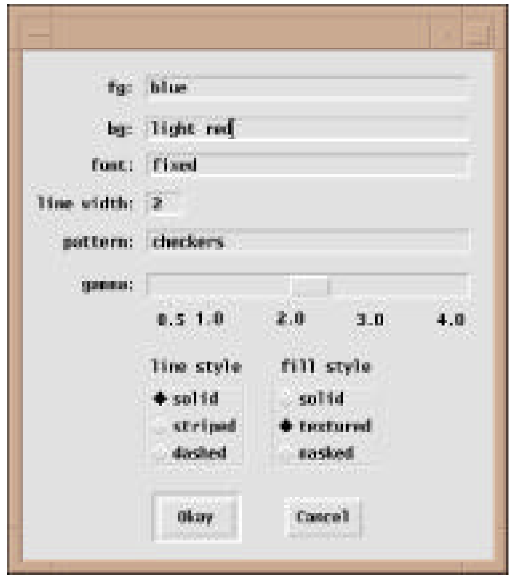



**Figure 4: The Custom Dialog**

## Standard Dialogs Versus Custom Dialogs

Standard dialogs generally are easier to use in a program than custom dialogs, and they have the virtue of providing a standard appearance. Standard dialogs also offer a facility that is easily overlooked. A standard dialog is constructed using the arguments given when the corresponding dialog procedure is invoked. These arguments can be lists that change depending on current data. For example, in an application that allows the user to create and delete items, standard dialogs can display the current list of items, which may change the number of items presented in the dialog.

Constructing custom dialogs requires time and effort. Custom dialogs, however, can be laid out for a particular situation, and slider, scroll bar, label, and line vidgets can be used in their construction. Unlike standard dialogs, however, the structure of a custom dialog is fixed when it is created. The states of the vidgets can be changed, but the vidgets themselves cannot.

Since VIB can handle only one VIB section in a file, custom dialogs must be kept in separate files if they are to be maintained using VIB. In this case, the applications that use them must link their ucode files. The need for multiple files causes organizational, packaging, and maintenance problems.

A general guideline is to use custom dialogs only when standard dialogs won't do or when a custom dialog can provide a substantially better interface.

### Icon on the Web

Information about Icon is available on the World Wide Web at

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

## Debugging: Built-In Facilities

This is the second in a series of articles on debugging in Icon and covers the facilities in Icon that can help find errors that occur during execution.

### Error Termination Information

A large percentage of bugs result in error termination. When a program terminates because of run-time error, the nature and location of the error is listed, followed by a traceback of procedure calls leading to the error. At the end, the offending expression is shown. Here's an example:

```
Run-time error 106
File recorder.icn; Line 32
procedure or integer expected
offending value: &null
Trace back:
  main()
  &null("Summary information …") from line
    32 in recorder.icn
```

We've modified the output slightly, as we will with other examples, so that long lines can be accommodated in our two-column format.

Such error termination messages generally are self-explanatory. For example, the error that resulted in the output above occurred in the main procedure. The offending expression was an invocation, but instead of a function, procedure, or integer, there was an attempt to apply the null value. This usually is the result of a misspelling. In the program in which this error occurred, two letters were transposed, resulting in wirte instead of write. As you'd expect, wirte has the null value, since no assignment was made to it. This kind of error is very common; it usually can be detected before program execution by using Icon's option for reporting undeclared identifiers, as in:

```
icont –u recorder
```

Note that the file name and line number information is duplicated at the beginning and end of the error termination message. This is handy when the traceback is long.

The way the offending expression appears varies. In the example above, the ellipses indicate that the string was long and part of it was omitted so that the line would not be very long. (It's not possible, in general, to tell whether the ellipses are actually in the string or whether they indicate that the string has been truncated.

The form in which the offending expression appears depends on what it is. Here's another example, in which the subject of string scanning is null:

```
Run-time error 103
File summary.icn; Line 402
string expected
offending value: &null
Trace back:
  main()
  {&null ? ...} from line 402 in summary.icn
```

Notice that the expression to be applied to the subject is not shown; it has not been evaluated at the time the error is detected.

Sometimes the offending expression may appear puzzling:

```
Run-time error 111
File vitamin.icn; Line 377
variable expected
offending value: "A"
Trace back:
  main()
  {"A" := ""} from line 377 in vitamin.icn
```

Certainly no Icon programmer would write such an expression in the expectation that it would work. The actual expression is

```
&letters[1] := ""
```

The reason the offending expression appears as it does is that &letters is not a variable. As a result, &letters[1] produces the string "A". By the time the error occurs, the expression that produced "A" is no longer available to show in the offending expression.

Here's example that shows a different format for the offending expression:

```
Run-time error 114
File format.icn; Line 222
invalid type to subscript operation
offending value: set_1(14)
Trace back:
  main()
  {(variable = set_1(14))[3]} from line
    222 in format.icn
```

This error results from attempting to subscript a set as if it were a table. The variable = indicates that a set was the value of a variable. This information is provided if it is available. The reason it isn't shown in some cases is because the variable has been dereferenced before the detection of the error. This is the case for the null value of wirte shown earlier.

Here's another example, which resulted in attempting to subscript a file as if it were a string:

```
Run-time error 103
File rotor.icn; Line 118
string expected
offending value: &output
Trace back:
  main()
  {&output[2] := "1"} from line 118 in rotor.icn
```

When interpreting tracebacks, it's important to remember that the information shown is what's current at the time of the error. Consider this procedure, called as process("Type A", 2):

```
procedure process(category, number, scale)

  number –:= 1          # adjust

  counter := number    scale
       …

end
```

Because the third argument of process() is omitted in the call, and hence null, a run-time error occurs in the expression number   scale. The traceback is:

```
Run-time error 102
File change.icn; Line 121
numeric expected
offending value: &null
Trace back:
  main()
  process("Type A",1) from line 311 in change.icn
  {1    &null} from line 11 in change.icn
```

It appears as if the call was process("Type A", 1), rather than process("Type A", 2). This is because the value of the parameter number was decremented before the error occurred. (Changing the value of a parameter of a procedure may be unwise, but it's legal.)

## Problems with Error Traceback

One problem with error termination messages is that they may be very voluminous. There are two common causes for this: stack overflow and use of graphics procedures.

### Stack Overflow

If a run-time error results from excessive depth of procedure calls (usually runaway recursion), the traceback may look like this:

```
Run–time error 301
File treeview.icn; Line 28
evaluation stack overflow
  Trace back:
    main()
    parse() from line 22 in treeview.icn
    parse() from line 28 in treeview.icn
    parse() from line 28 in treeview.icn
    parse() from line 28 in treeview.icn
    parse() from line 28 in treeview.icn
    parse() from line 28 in treeview.icn
         ...
```

and so on for many lines.

Here it's clear that parse() is the offending procedure, although if there is mutual recursion among several procedures, the cause of the problem may not be so obvious. In such a situation, it's often helpful to look at the beginning of the traceback to locate the source of the problem.

### Graphics Procedures

A substantial portion of Icon's graphics facilities, including the vidgets provided by VIB, are written in terms of Icon procedures. Because of this, error traceback may be truly overwhelming. Here's a rather tame example (it's hopeless to try to show a really unruly one):

```
Run–time error 102
File kaleido.icn; Line 256
numeric expected
offending value: &null
Trace back:
  main(list_1 = [])
  kaleidoscope() from line 66 in kaleido.icn
  ProcessEvent(record Vframe_rec_1(window_1, …
  event_Vtoggle(record Vbutton_rec_13(window_ …
  toggle_Vbool_coupler(record Vcoupler_rec_1(1,
  set_Vcoupler(record Vcoupler_rec_1(1,list_86(1 …
  call_clients_Vcoupler(record Vcoupler_rec_1(1 …
  pause_cb(record Vbutton_rec_13(window_1,1 …
  {&null = 1} from line 256 in kaleido.icn
```

This may not look so bad, but the ellipses are ours; the lengths of individual lines go up to 341 characters. We've even had cases in which we thought the traceback was in a loop because the output was so huge. About all you can do in cases like this is look at the beginning and end of the error termination message. To do that, you may have to save it in a file. (All error messages are written to standard error output, which complicates the process of saving it in a file.)

There's a lesson in language design here, but that's the way it is. In a later article, we'll show library procedures that can help with these kinds of problems.

## Termination Dumps and Displays of Variables

If the value of &dump is nonzero (its initial default value is zero) when a program terminates, a dump of variables and their values is produced.

The dump starts with an image of the current co-expression. Following this, there are listings of the local identifiers and their values in procedure calls back to the original invocation of the main procedures. Finally, there is a listing of global variables and their values. Here's an example:

```
Run–time error 103
File csgen.icn; Line 137
string expected
offending value: &null
Trace back:
  main(list_1 = [])
  subst(list_4 = ["X","abc"]) from line 127 in csgen.icn
  find(&null,"X",&null,&null) from line 137 in csgen.icn

Termination dump:

co–expression_1 (1)
subst local identifiers:
  a = list_4 = ["X","abc"]
main local identifiers:
  args = list_1 = []
  line = "X:10"
  goal = "X"
  count = 10
  s = "X"
  opts = table_1(0)

global identifiers:
  any = function any
  close = function close
  find = function find
  get = function get
  integer = function integer
  main = procedure main
  many = function many
  map = function map
  move = function move
  open = function open
  options = procedure options
  pos = function pos
  pull = function pull
  push = function push
  put = function put
  randomize = procedure randomize
  read = function read
  real = function real
  stop = function stop
  string = function string
  subst = procedure subst
  tab = function tab
  table = function table
  upto = function upto
  write = function write
  xlist = list_3 = [list_4(2),list_5(2),list_6(2),
    ...,list_8(2),list_9(2),list_10(2)]
  xpairs = procedure xpairs
```

Notice that the global variables include the functions that the program uses.

Termination dumps occur regardless of whether a program terminates normally or as the result of an error. In a program in which a dump is not wanted if a program terminates normally, this kind of usage is typical:

```
procedure main()
  &dump := 1          # just in case
      ...
  &dump := 0          # disable before completion
end
```

If the program terminates by some other means than flowing off the end of the main procedure, such as a run-time error or a call of stop() or exit(), a dump is produced.

The keyword &dump was added to Icon in Version 9. Programmers who started using Icon before Version 9 may have overlooked &dump or not thought about its usefulness. If you are in this group, think about adding &dump to the tools you use on a regular basis.

### Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

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

Termination dumps actually use a facility that has been in Icon for a long time: display(i, f). This function writes a dump in the same format as &dump, but only going back i levels of procedure call. The default for i is &level, giving the local identifiers for all procedure calls back to the invocation of the main procedure. The argument f allows a file to be specified, &errout being the default.

Because display() writes to a file, it is not particularly useful, especially now that &dump is available. We'll have more to say on this subject when we get to library support for debugging.

## Next Time

In the next article on diagnostic facilities and debugging, we'll tackle procedure and co-expression tracing.

# Records

Many programming languages support records — structures that are fixed in size and have fields that are referenced by name. In Icon, records are useful, but they are somewhat mundane compared to Icon's other structures; so much so that records are not covered in the Icon portion of our Comparative Programming Languages course [1, 2]. And why should they be covered? Surely records in Icon are simple enough that any programmer can figure them out.

While that's probably true, there are aspects of records and uses for them that sometimes are overlooked.

## The Properties of Records

By way of review, here are the essential characteristics of records in Icon.

• Records are declared, as in

```
record rational(numer, denom)
```

which declares a record named rational with field names numer and denom.

• Instances of records are created during program execution by using a *record-constructor* function whose name corresponds to the record name and whose arguments correspond to the fields, as in

```
portion := rational(2, 3)
```

• A record declaration adds a type corresponding to the record name to Icon's built-in type repertoire. For example, type(portion) produces "rational".

• There is no limit to the numbers of record types that can be declared except the memory available for representing them. (It's unlikely for memory to be a limitation for any "real" program.)

• A record can be declared with no fields, as in

```
record marker()
```

• There is no limit to the number of fields in a record declaration, except for memory.

• The same field name can be used in different record declarations, as in

```
record particle(charge, mass, spin)
record anti_particle(charge, mass, spin)
              …
```

• The duplicate field names need not be in the same position in different record declarations, as in

```
record employee(name, ssn, position, salary)
record tax_return(ssn, name, address)
```

• Record fields are accessed by name using the binary "dot" operator, as in

```
portion.numer := 1
```

• Attempting to reference a record by a field name the record does not have causes a run-time error.

• Record fields also can be accessed by position, as in

```
portion[2] := 5
```

which changes the denom field of portion to 5. As with lists, subscripts can be expressed in positive or non-positive form, and out-of-bounds subscripts result in failure. For example,

```
portion[–2] := 1
```

assigns 1 to the numer field of portion, but

```
write(portion[3])
```

fails.

• Record fields can be accessed by their string names, as in

```
write(portion["numer"])
```

Subscripting a record by a field name it does not have fails. Thus,

```
portion.ratio
```

causes a run-time error, but

```
portion["ratio"]
```

fails.

• Field names and identifiers have separate “name spaces” and do not conflict with each other. For example, in

```
record complex(real, imaginary)
```

the field name real does not conflict with the identifier real and hence the function real(). In fact, a record name and a field name can be the same, even in the same declaration, as in

```
record word(word, part_of_speech)
```

which might lead to code like

```
item9 := word("world", noun)
   …
write(item9.word)
```

Such usage is likely to be confusing if not actually obfuscating. (Now that we’ve said that, someone probably will come up with a valid and helpful use for this feature.)

• A record name can be the same as the name of a built-in type. An example is

```
record file(name, position, status)
```

Internally, Icon distinguishes between record types and built-in types of the same name, but type() does not. Consequently in

```
log := file("standard.log", 0, "closed")
```

type(log) and type(&input) both produce "file".

If, however, a record name is the same as the name of a function, the record constructor replaces the function as the value of the corresponding identifier. For example, in

```
real(exponent, mantissa)
```

real() creates a record of this type instead of attempting to convert its argument to a real number. The function still is available, however;

```
float := proc("real", 0)
```

assigns the function to float.

• The operation R produces the number of fields in the record R. For example, portion produces 2.

• The operation !R generates the fields of R as variables. This can be used to assign a value to all fields in a record, as in

```
every !portion := 1
```

• The operation ?R produces a randomly selected field of R as a variable. For example,

```
?portion := 3
```

assigns 3 to either the numer or denom field of portion. ?R fails if R has no fields.

• Records have serial numbers. Each record type has a separate serial sequence, starting at 1.

• The function serial() produces the serial number of a record. For example, if portion is the first-created record of type rational, serial(portion) produces 1.

• The function image() gives the record type followed by an underscore, its serial number, and its size in parentheses. For example, if portion is the first-created record of type rational, image(portion) produces

```
"record rational_1(2)"
```

• The function name(), when applied to a string corresponding to a record field reference, produces the record type and field name, as in

```
name("portion.numer")
```

which produces "rational.numer".

• In sorting, records sort after all other data types. Record types are lexically ordered. Records of the same type are ordered by their serial numbers.

• A record can be sorted to produce a list with the field values in sorted order. Like other obscure operations on records, it’s hard to imagine a use for this, but there probably are some. In any event, it puts records on par with other structures.

• Records can be used to provide arguments in invocation in the same manner as lists, as in

```
atan ! portion
```

which is equivalent to

```
atan(portion.numer, portion.denom)
```

and produces the arctangent of portion.numer divided by portion.denom.

• List invocation can be used to create records, as in

```
ratio := rational ! args
```

where args is a list whose (first two) elements provide the values of the numer and denom fields of a rational record. Omitted and extra arguments are treated as they are in any function or procedure call; null if omitted, ignored if extra.

• "Record invocation" can be used to create a new record from another. For example, as a result of

```
neutron := particle(0, 1.6749286e–27, "1/2")

antineutron := anti_particle ! neutron
```

assigns to antineutron an anti_particle with the same field values as neutron.

## Examples

### Rational Arithmetic

Here's an example of a conventional use of records to manipulate rational numbers. In this package, the sign is carried in a separate field to simplify some computations.

The procedure str2rat() converts a string representing a rational number in the form "n/d" to a record representing that rational number. Conversely, the procedure rat2str() converts the record representation to the string representation.

There are a variety of special cases to handle, including rejecting a denominator of 0 and reducing rational numbers to their lowest form. The code itself is mostly self-explanatory.

```
link numbers

record rational(numer, denom, sign)

procedure str2rat(s)
   local div, numer, denom, sign

   s ? {
      numer := integer(tab(upto('/'))) &
      move(1) &
      denom := integer(tab(0))
      } | fail

   div := gcd(numer, denom) | fail
   numer /:= div
   denom /:= div
   if numer    denom >= 0 then sign := 1
      else sign := –1

   return rational(abs(numer), abs(denom), sign)

end

procedure rat2str(r)

   return  r.numer    r.sign || "/" || r.denom

end

procedure mpyrat(r1, r2)
   local numer, denom, div

   numer := r1.numer    r2.numer
   denom := r1.denom    r2.denom
   div := gcd(numer, denom)
   return rational(numer / div, denom / div,
      r1.sign    r2.sign)

end

procedure divrat(r1, r2)

   return mpyrat(r1, reciprat(r2))          # may fail

end

procedure reciprat(r)

   if r.numer = 0 then fail
   else return rational(r.denom, r.numer, r.sign)

end

procedure negrat(r)

   return rational(r.numer, r.denom, –r.sign)

end
```

```
procedure addrat(r1, r2)
   local denom, numer, div, sign

   denom := r1.denom    r2.denom
   numer := r1.sign    r1.numer    r2.denom +
      r2.sign    r2.numer    r1.denom
   sign :=  if numer >= 0 then 1 else –1
   div := gcd(numer, denom) | fail

   return rational(abs(numer / div),
      abs(denom / div),sign)

end

procedure subrat(r1, r2)

   return addrat(r1, negrat(r2))

end
```

#### Generating Field Names

It's sometimes useful to be able to find out the names of the fields of an arbitrary record. This is used in Bob Alexander's ximage() procedure [3] and in other programs that display the details of Icon's structures.

We've suggested how this might be done in the section **The Properties of Records**. Here's a procedure that takes a record as its argument and generates the names of its fields:

```
procedure field(R)
   local i

   every i := 1 to   R do
      name(R[i]) ? {
         tab(upto('.') + 1)
         suspend tab(0)
        }

end
```

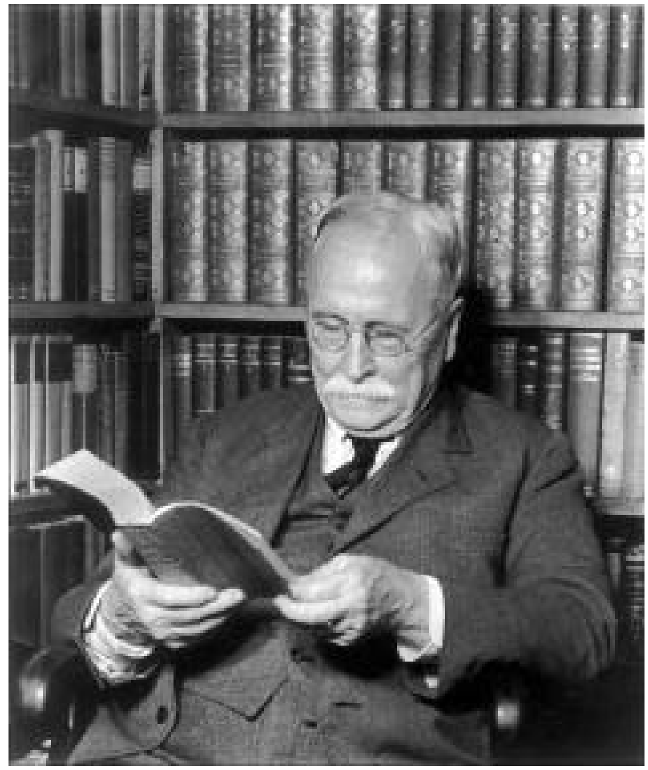
### References


1. "Teaching Icon", *Icon Newsletter* 51, pp. 2-6.

2. "Teaching Icon", *Icon Newsletter* 52, pp. 2-3.

3. "From the Library", *Icon Analyst* 25, pp. 1-5.



## From the Library

### Interactive Expression Evaluation

In his *Newsletter* article on teaching Icon [1], Bill Mitchell described a program that he wrote to allow students to formulate and evaluate Icon expressions interactively.

The central idea is simple: Take an expression the user enters, wrap it in a main procedure, save it in a file, and then compile and execute it.

This idea has been around for a long time and has been used in rudimentary ways in the Icon program library.

On the face of it, such a "gross" approach, which requires creating, compiling, and executing a program for every expression the user enters in interactive mode, seems impractical if not ludicrous. And indeed it was, not that many years ago, when working on a 16-bit 286 PC. Granted, workstations at the current high end have been able to handle this adequately for some time, but the user hardly had the feel of immediate response until recently. Every month or so now there are faster platforms, and the former gap between workstations and PCs is no longer so evident. In fact, with a Pentium Pro PC or a modest workstation, this method works quite nicely.

Bill's original program has been reworked, spruced up, increased in functionality, and made more portable. The current version is called qei (from the Latin *quod erat inveniendum*, meaning "which was to be found out").

## Using qei

When qei is launched, it presents the character > as a prompt, after which the user can type an Icon expression to be evaluated, as in

```
> integer(&pi);
```

A semicolon is necessary to terminate the expression; otherwise qei prompts for more to add to what's already been entered.

The response to this expression is

```
r1_ := 3 (integer)
```

r1_ is a variable created by qei; subsequent ones are named r2_, r3_, and so on. As shown, the type of the result is given in parentheses.

The variables qei that creates can be used in subsequent expressions, as in

```
> r1_ + r1_;
  r2_ := 6  (integer)
```

The user can provide variables also, as in

```
> pi := &pi;
  r3_ := 3.141592654  (real)
> pi ^ 4;
  r4_ := 97.40909103  (real)
```

Any kind of expression can be entered, as indicated by the following sequence:

```
> list(100, 0);
  r5_ := list_1(100)  (list)
> r5_[2] := &e;
  r6_ := 2.718281828  (real)
```

## Commands

qei provides several commands that allow the user to control the program.

For example, a generator only produces its first result unless the command :every is used, as in

```
> :every !r6_;
  "2"  (string)
  "."  (string)
  "7"  (string)
  "1"  (string)
  "8"  (string)
  "2"  (string)
  "8"  (string)
  "1"  (string)
  "8"  (string)
  "2"  (string)
  "8"  (string)
```

The command :list lists all expressions that have been entered so far, as in

```
> :list
r1_ := (integer(&pi))
r2_ := (r1_ + r1_)
r3_ := (pi := &pi)
```


# 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

```
r4_ := (pi ^ 4)
r5_ := (list(100, 0))
r6_ := (r5_[2] := &e)
r7_ := (!r6_)
```

The command :type is a toggle that turns the display of the type name off and on; it is on initially. The command :? lists all the available commands. Finally, :quit terminates a qei session.

## Failure and Errors

If an expression fails, that is noted, as in

```
> integer("a");
Failure
```

A syntactic error in an expression produces an error message, as in

```
> &bad;
File qei_.icn; Line 22 # "bad": invalid keyword
1 error
```

This message also reveals some things about how qei works.

## A Peek Inside

As you'd expect, qei uses the system() function to compile and execute the programs that it creates. This facility is supported by most operating systems, including MS-DOS, Windows, and UNIX.

A key idea in qei is to maintain a list of all the expressions the user enters. Every new expression is appended to the list and all the expressions are included with each new program that is created. Consequently, all previous expressions are evaluated when a new expression is entered.

For the most part, this works nicely. It makes all previous results available in a newly entered expression and the performance impact is not noticeable in most cases. It is, of course, possible to enter an expression that takes a long time to execute. When this happens, all subsequent expression are affected. For this reason, the command :clear is provided to remove all previously entered expressions. In this case, all previous results are lost also.

Here's what qei produces for the expressions shown earlier, up through pi ^ 4:

```
procedure main()

   r1_ := (integer(&pi))
   r2_ := (r1_+r1_)
   r3_ := (pi := &pi)

   showtype := 1

   if (r4_ := (pi ^ 4)) then WR("r4_ := ",r4_)
   else write("Failure")

end

procedure WR(tag, e)

   writes("   ",tag, image(e))
   write(if \showtype then "  (" || type(e) || ")" else "")

end
```

## Conclusion

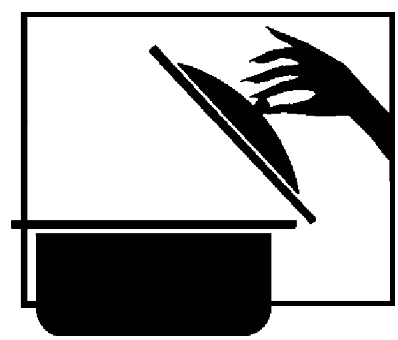
We encourage you to try qei. It's an excellent way to get started with Icon, and for an experienced Icon programer, it can provide valuable insights into the darker corners of Icon.

## Note

This article describes Version 1.2 of qei, which was sent to subscribers to the Icon program library update service in March. An earlier version is in the 9.3 release of the library. More versions are sure to come.

## Reference


1. "Teaching Icon", *Icon Newsletter* 51, pp. 2-6.

## What's Coming Up

We had planned another article on versum numbers for this Analyst, but we didn't get it done in time — due in part to loud croaking sounds from the pond in which our computers live. We should have the article ready for the next issue.

We'll continue our series on debugging with an article on tracing, and we expect to have an article related to program visualization.