
The Icon Analyst

In-Depth Coverage of the Icon Programming Language

December 1992
Number 15

In this issue ...

Idiomatic Programming ... 1
Monitoring Icon Programs ... 6
Programming Tips ... 11
What's Coming Up ... 12

Idiomatic Icon

This is the second of two articles on idiomatic programming in Icon. The numbering of idioms continues from the first article.

Factoring Expressions

Because of the way that generators and goal-directed evaluation work, it's often possible to "factor out" operations that are applied to several alternatives. For example,

```
find(1) | find(s2) | find(s3)
```

can be written more compactly as

```
find(s1 | s2 | s3) (22)
```

This kind of idiomatic usage has the nice property of being not only more concise but also easier to understand than the expanded form.

This idiom is a reminder that where alternation can be used, other generators can also. For example,

```
find(!words) (23)
```

applies `find()` to all the elements of `words`, which might, for example, be a set or list.

Another useful factoring idiom is illustrated by

```
every close(\(f1 | f2 | f3)) (24)
```

which closes any of `f1`, `f2`, or `f3` whose values are nonnull.

Factoring also is possible where different operators are applied in alternation to the same arguments. Thus

```
parse(text) | error(text)
```

can be written as

```
(parse | error)(text) (25)
```

and

```
integer(x) | string(x) | proc(x)
```

can be written as

```
(integer | string | proc)(x)
```

And, as above, it's worth remembering that the same idea applies to other generators, as in

```
(!fnclist)(x) (26)
```

See [1] for a more detailed discussion of the uses of this kind of idiom.

Other kinds of factoring are possible because control structures in Icon are expressions, not statements. For example,

```
if count > 0 then state := 1 else state := 0
```

can be written as

```
state := if count > 0 then 1 else 0 (27)
```

Similarly,

```
case state of {  
  0: return i  
  1: return j  
  2: return k  
}
```

can be written more compactly as

```
return case state of { (28)
```

```
  0: i  
  1: j  
  2: k  
}
```

In a similar vein, a construction like

```

if x := expr1 then return x
else if x := expr2 then return x
else return y

```

can be written as

```

return expr1 | expr2 | y           (29)

```

Some care is needed in the use of constructions like this. For example

```

return expr1 | expr2

```

causes the procedure call to fail if both *expr1* and *expr2* fail, since *return* causes return from a procedure call with the *outcome* of its argument. This can lead to mistakes, since

```

suspend expr1 | expr2

```

does not cause the procedure call to fail if both *expr1* and *expr2* fail — it simply does nothing and execution continues with the expression after the *suspend*.

Odds and Ends

As we warned in the first article on idioms, there are many idioms, including some of the most important and commonly used ones, that don't fall into any specific class except "miscellaneous". Here they are.

Scanning a list of items: Strings frequently consist of items separated by markers such as commas or blanks, but without a marker after the last item. Extracting and processing the items one by one can be done using string scanning to find successive markers. The problem is with the last item, which is not followed by a marker.

An unattractive solution is to append a marker to the end of the string before scanning it, as in

```

text ll:= marker
text ? {
  while process(tab(find(marker))) do
    =marker
}

```

Another unattractive solution is to handle the last item separately after all the other items have been processed, as in

```

text ? {
  while process(tab(find(marker))) do
    =marker
  process(tab(0))
}

```

An idiomatic approach for handling the special case of the last item is to put the alternative in the argument of *tab()*, as in

```

text ? {                               (30)
  while process(tab(find(marker) | 0)) do
    =marker | break
}

```

Here, *tab(0)* finds the last item when there is not another instance of *marker*. In this case, *=marker* in the *do* clause fails and *break* terminates the loop.

Quoting strings: Sometimes it's necessary to construct strings with quotation marks around them (as, for example, in a program that writes another program). You can do this by concatenating literal quotation marks, as in

```

qs := "\"" || s || "\""

```

but it's easy to get lost between the quotes for the literals and the escaped quotes. An easier method is

```

qs := image(s)                          (31)

```

This method has the additional advantage of providing escape sequences for any characters in *s*, like quotes, that require escape sequences in program text.

Limitation: Sometimes it's necessary to limit the number of times an expression can be resumed. This often happens when testing a generator that can produce a large or even infinite number of results. The way to do this is to use the limitation control structure, as in

```

every write(primeseq()) \ 100           (32)

```

You might well argue that this is hardly an idiom. That's true, but we are listing it here because novice programmers often do something like this:

```

i := 1
every write(primeseq()) do {
  i += 1
  if i > 100 then break
}

```

Note that this doesn't work correctly if the limit is 0, while the limitation control structure does work correctly for this case.

Repeated Alternation: This leads us to another topic. Icon only has a few generators. Most operations in Icon are cast in the conventional way and produce at most a single result. For example, *read()*

produces (at most) one value every time it is evaluated — it doesn't generate lines from the input file by suspending and being resumed.

There's a reason why only a few operations in Icon are generators. Although generators are very useful in the appropriate context, they can be dangerous and difficult to control. For example, if `read()` were a generator,

```
flag == read()
```

would read the entire input file if it didn't contain a line equal to `flag`.

It's for this reason that Icon's built-in operations are cast as generators only when it's necessary to preserve an internal state between the production of successive values.

For example, `find(s)` needs to "remember" the value of `s` and where it is in the subject between the production of successive positions. The suspension/resumption mechanism allows this. In the case of `read()`, however, no such "memory" is needed; that's handled by the input mechanism itself.

Another reason why there are only a few generators in Icon is that it's easy to make a generator out of a non-generator by using repeated alternation. For example,

```
|read() (33)
```

generates all the lines of the input file. It stops at the end of the file because repeated alternation stops if its argument doesn't produce a value. Similarly,

```
|?x (34)
```

generates an infinite sequence of values selected at random from `x`.

Another interesting idiom is

```
!|x (35)
```

which generates the elements from `x` repeatedly as if `x` were circular.

Incidentally, if you want to produce the results of evaluating an expression a fixed number of times, you can do this using conjunction with an expression that generates the desired number, as in

```
(1 to 5) & !x (36)
```

which generates the elements from `x` five times.

Note that repeated alternation can be used in combination with limitation, as in

```
every write(|read()) \ 100 (37)
```

It's important to remember that limitation limits the total number of values, not the number of times the repeated alternation control structure is applied. In Idiom 37 these are the same, but in

```
!|x \ i
```

they generally are not.

Another idiomatic use of repeated alternation arises in the use of co-expressions. The activation operator,

```
@C
```

produces at most one result from the expression associated with the co-expression `C`. If it generated all results instead, it would defeat the purpose of co-expressions, which allows the controlled production of values from an expression.

On the other hand, it's sometimes necessary to produce a sequence of values from a co-expression. This can be done in a loop, as in

```
while x := @C do  
  suspend x
```

Repeated alternation provides a neater idiomatic formulation:

```
suspend |@C (38)
```

Empty lists: Suppose you want to determine if a list `L` is empty. The obvious way is

```
if *L > 0 then ...
```

Another (idiomatic?) way is

```
if L[1] then ... (39)
```

Yet another is

```
if !L then ... (40)
```

Of the three methods, the second two are certainly less obvious in their intentions, and the third more so than the second. The third method has the somewhat dubious virtue of requiring the fewest keystrokes.

Although such tests are not likely to be used often enough for efficiency to be a consideration, for whatever it's worth, the first is the slowest, the second is about 20% faster. The speed of the third depends significantly on whether or not the list is empty; it is about twice as fast as the first if it is, but about the same as the first if it isn't. These are just quirks of the implementation.

Generating elements: If you have a structure L that is a list of lists, it's worth noting that

```
!!L (41)
```

generates all the elements of the lists in L. For example,

```
!!L === x (42)
```

can be used to determine if any element is the same as x. The same idea works for records whose fields contain lists, and so on.

We had an occasion recently to use multiple element generation when using records whose fields were lists of integers. The problem got even more interesting when such records were passed to a procedure declared with a variable number of arguments. A section of the code looked like this:

```
procedure plot(points[ ]
  :
  if \debug then
    every write(!!!points)
  :
  :
```

Multiple generation also can be useful when dealing with the results of sorting a table. The default for sort(T) produces a list of two-element lists containing the key/value pairs. Therefore, to write out successive keys and their values

```
every write(!sort(T)) (43)
```

can be used. Nevertheless, sort(T, 3), which produces a "flat" list of successive keys and values is better, since it requires much less memory. The code equivalent to Idiom 43 is

```
every write(!sort(T, 3))
```

Disabling functions: Sometimes it's useful to be able to disable functions. For example, when measuring the performance of a program, it may be useful to get rid of output. There's an easy way to do this for most programs: Just add

```
write := writes := 1 (44)
```

at the beginning of the main procedure. This has the effect of making expressions like

```
write(s1, s2)
```

operate as if they had been written

```
1(s1, s2)
```

The argument selection operation simply returns the value of s1. This has no effect on the program

other than preventing output, unless the program uses the value produced by write() in a significant way. This possibility can be handled by the slightly more subtle

```
write := writes := -1 (45)
```

which causes the last argument of write() and writes() to be produced (which is what these functions do after writing). Of course, it's possible to contrive programs where this idiom won't work. We'll leave it to you to find examples.

The Icon Analyst

Madge T. Griswold and Ralph E. Griswold
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
Gould-Simpson Building
The University of Arizona
Tucson, Arizona 85721
U.S.A.

voice: (602) 621-8448

fax: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

...uunet!arizona!icon-project

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA
and



The Bright Forest Company
Tucson Arizona

© 1992 by Madge T. Griswold and Ralph E. Griswold
All rights reserved.

Another case where it's useful to replace a function is in a program that uses `system()`. This function is powerful, but it's also quite dangerous [2]. In testing a program that uses `system()`, a good start is to add

```
system := write (46)
```

at the beginning of the main procedure. Calls of `system()` then just write their arguments. Of course, this change is hardly as transparent as disabling output in the manner discussed above.

Avoiding negation: Alternation often can be used to replace the more conventional logical negation control structure. When opening a file, it is very important to catch a case in which `open()` fails [3]. The obvious method is:

```
if not (file := open(name)) then
  stop("*** cannot open file")
Negation, even in a simple case like this, tends
to be confusing. An idiomatic alternative is
file := open(name) |
stop("*** cannot open file") (47)
```

It's worth noting that, in general

```
if not expr1 then expr2
```

is *not* equivalent to

```
expr1 | expr2
```

since the latter expression generates all the values of *expr1* before those for *expr2*, while the former produces at most one value from *expr1* and that value is not included among the values generated by the if-then expression.

Case folding: We'll end with one final idiom that is not an idiom at all, but simply the use of deliberately designed default values.

Case differences among letters often are not significant. In fact, it's often easier to design a program that is insensitive to case, as in one that processes commands. For this,

```
map(line) (48)
```

is all that's needed, since if the second and third arguments of `map()` are omitted, uppercase letters are mapped to their lowercase equivalents.

Incidentally, the design decision to map uppercase letters to lowercase ones, rather than the other way around, was based on years of annoyance with computer output that was printed all in uppercase. Such output was prevalent partly be-

cause of the limitations of early computer printers, partly because of the limitations of computer architecture and software, and partly, would you believe, because some persons thought (and still do) that the use of all uppercase letters would (rightly) identify the printed material as having originated on a computer and (wrongly) that this would lend it more importance. Of course, these days, with sophisticated printers, it's often impossible to tell whether printed material originated from a computer or a person, and the distinction often is irrelevant. But we still like lowercase letters better than uppercase ones.

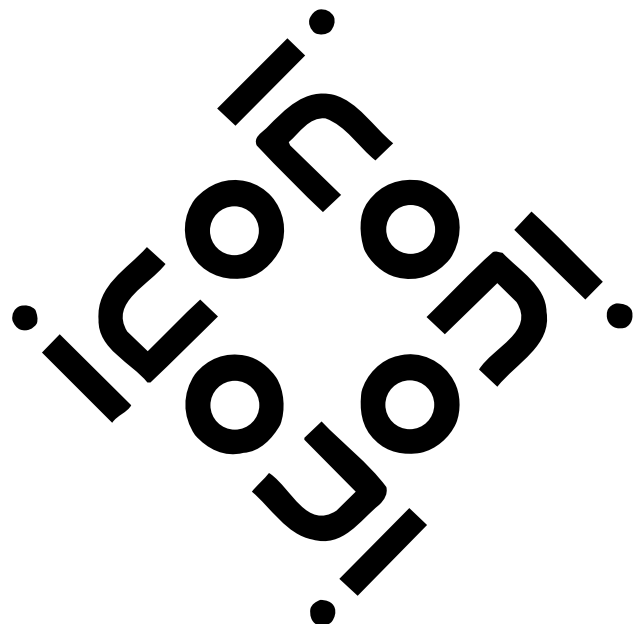
Conclusions

This list of idioms is by no means exhaustive. Almost by definition, it can't be.

We've already had some additional suggestions from readers. If you have favorite Icon idioms that are not listed here, please send them to us. When we get enough of them, we'll write another article for the *Analyst*.

References

1. "Result Sequences", *The Icon Analyst* 7, pp. 4-8.
2. "Getting to the System", *The Icon Analyst* 10, pp. 1-2.
3. "Writing Bullet-Proof Programs", *The Icon Analyst* 10, p. 10.



Monitoring Icon Programs

In the last issue of the *Analyst*, we described MT Icon [1], a version of Icon that allows several programs to run in the same execution environment and communicate with each other. While MT Icon has many uses, it was motivated by our research in program visualization and visual debugging, in which a *monitor* needs to obtain information about what's going on in a *monitored program*.

The key phrase in the preceding paragraph is “obtain information”. We need to explain both what we mean by information and how it is obtained.

All kinds of things go on “under the surface” when an Icon program runs. There are, of course, many things directly related to the computations the program carries out, such as performing arithmetic, making assignments to variables, calling procedures, and so forth. These kinds of activities are of potential interest to monitors that may, for example, be watching for assignment to a particular variable or be checking on the use of a particular procedure. But there are many other kinds of activity that are less directly related to the actual computations a program performs, such as the location in the source program where execution is taking place, storage allocation, garbage collection, type checking and conversion — in fact, a host of things may be of interest to a monitor.

While it is easy enough to see how a program to be monitored can be loaded and activated by a monitoring program using MT Icon, there's evidently no way for a monitor to get information about the program it is monitoring without additional facilities.

Producing Information for Monitoring

One possibility would be to insert code in the program to be monitored to activate the monitor and pass information to it. This *code intrusive* approach has many problems and serious limitations. Except for the most trivial kinds of monitoring, manual code insertion is impractical. Code insertion can be done automatically with a preprocessor (variant translators work nicely for Icon [2]). But the increase in source-code size for extensive monitoring makes this technique impractical for programs of even moderate size, and the penalty in execution speed is substantial. Furthermore, with source-code insertion there is no way to get infor-

mation about internal operations such as storage allocation and garbage collection.

Instrumentation

We have taken a different approach — one that provides detailed information about a program without having to modify it. Instead of putting code in the source program to report events, we've modified the Icon interpreter itself, adding instrumentation code that reports on events that occur when a program is running.

This instrumentation code originally wrote information about program events to a file, which could be piped into a monitor or saved for later “post-mortem analysis” [3, 4]. In this mode, monitors were entirely passive. They could not stop the execution of the monitored program when a specific event occurred, nor could they examine its data.

Recently we've changed this instrumentation and integrated it into MT Icon, so that a monitored program and a program that monitors it both run under the same invocation of the MT Icon interpreter. This allows monitors to get more information, get it with less overhead, and take an active role. Note that in this model, a monitor is necessarily an Icon program.

Monitoring

A monitor requests information about specific kinds of events in the monitored program. When such an event occurs in the monitored program, control is automatically transferred back to the monitor by co-expression activation, transmitting information about the event in the process. After processing the event, the monitor re-activates the monitored program.

The activation of the monitor by the monitored program is implicit and happens in the instrumentation code that is built into the implementation. The monitored program is not aware that this has happened. It effectively “goes to sleep” while the monitor is processing the event and then “wakes up” when the monitor re-activates it.

Of course, the execution of the monitored program is slowed down by monitoring. Checking for events isn't particularly time-consuming, but the monitored program isn't running at all while the monitor is active. However, unless the monitor does something intrusive to the monitored pro-

gram (which it can but normally doesn't), the computations performed by the monitored program are unaffected by the act of monitoring.

So far we've been talking in general terms. In order to understand what's going on, we need to talk about the process more concretely from the point of view of a monitor.

Monitors

A monitor is actively involved in requesting events and processing them. A monitor requests an event by calling a built-in function. When an event occurs in the monitored program, the function returns. The monitor then can process the event.

Note that the monitor "goes to sleep" when it requests information about events and is "awakened" when an event occurs in the monitored program. But from the point of view of the monitor, it just calls a function, which may take some time to perform the requested computation before it returns. This is no different, in principle, from calling any other function.

Events

Before going on, we need to say more about the way that events are characterized. An event has two components: an *event code* and an *event value*. These are ordinary Icon values. The event code identifies the nature of the event, such as an assignment, the allocation of a string, or a garbage collection. Event codes normally are one-character strings. The event value depends on the nature of the event. The event value associated with an assignment is a string that identifies the variable to which the assignment is being made. The event value associated with the allocation of a string is the number of bytes allocated for the string. The event value associated with a garbage collection is an integer that identifies the storage region in which an attempted allocation caused the garbage collection. And so on.

A monitor requests an event from a monitored program by

```
EvGet(mask)
```

where *mask* is a cset containing the event codes of interest. Only those events specified in *mask* are reported by the instrumentation. If *mask* is omitted or null, all events are reported. `EvGet()` fails

when the monitored program terminates.

When `EvGet()` returns, two keywords in the monitor are set: `&eventcode` and `&eventvalue`, corresponding to the code and value for the event. A simple example of monitoring is

```
while EvGet() do
  write(
    image(&eventcode),
    " : ",
    image(&eventvalue)
  )
```

This loop requests all events in the monitored program and writes out the corresponding event codes and values.

Setting up a Monitor

You may be wondering how a monitor gets started, how `EvGet()` knows what program is being monitored, and so on. We've developed a substantial amount of infrastructure to take care of such things in order to make monitors easy to write. Much of the infrastructure is written in Icon and is contained in a library named `evinit`.

A monitor links `evinit` and starts by calling the procedure

```
EvInIt(s)
```

where *s* is the name of the *icode* file to be monitored. `EvInIt()` loads *s* and performs various initialization tasks, including assigning commonly used values to global variables so that many aspects of monitoring can be specified symbolically. `EvInIt()` also assigns the co-expression for the program being monitored to `&eventsourc`. This co-expression is useful in accessing information in the monitored program using MT Icon functions.

A complete program, `evlist.icn`, to monitor all events as given above is

```
link evinit
procedure main(argl)
  EvInIt(argl[1]) | stop("*** can't load file")
  while EvGet() do
    write(
      image(&eventcode),
      " : ",
      image(&eventvalue)
    )
end
```

This monitor takes the name of the monitored program from the command line. For example,

```
evlist prog
```

lists all the events that occur when prog is run.

In many cases, the monitored program needs its own command-line arguments. In order to handle this, `EvInit()` can be called with a list instead of a string. In this case, the first element of the list is taken to be the name of the icode file and the rest of the list is passed to `main()` in the corresponding program. Thus, a more general form of `evlist.icn` might start as

```
procedure main(argl)
    EvInit(argl) | stop("*** can't load file")
    :
```

Then

```
evlist prog prog.dat prog.log
```

causes `evlist` to monitor `prog` as if `prog` had been called from the command line as

```
prog prog.dat prog.log
```

There's still the question about what to do about input and output, especially standard input, which may be needed by the monitored program, and standard output, which may be written by the monitored program. Like `load()`, `EvInit()` has three additional file arguments that can be used to specify standard input, standard output, and standard error output for the monitored program. These default to the corresponding files in the monitor. Consequently, in

```
evlist concord <sample.txt >sample.con
```

the output of both `evlist` and `concord` are written to `sample.con` — not exactly what you'd want.

There are various things that can be done about this. With visualization, and hence X-Icon, in mind, a good start is to use a window for the output of the monitor. Using this approach, `evlist.icn` might look like this:

```
link evinit
procedure main(argl)
    EvInit(argl) | stop("*** can't load file")
    window := open("evlist", "x") |
        stop("*** cannot open window")
    while EvGet() do
```

```
        write(window,
            image(&eventcode),
            " : ",
            image(&eventvalue)
        )
```

```
end
```

Incidentally, when X-Icon writes to a window, it scrolls when the window is full, so the list of events goes by much as it would on the screen of a typical terminal.

As you might imagine, just listing all the events that occur in a program is not very useful. For one thing, there are too many events to comprehend easily. In order to be useful, a monitor needs to be selective about events.

More About Events

The instrumentation in MT Icon is extensive. There are more than 100 different kinds of events. In addition to the kinds of events mentioned earlier, there are clock ticks, line and column changes corresponding to the location of execution in the source program, virtual machine instructions [5], structure references, string scanning events, and so on.

The `evinit` library provides a global variable for each kind of event. These global variables start with `E_` to distinguish them from other variable names. For example, the value of `E_Assign` is the event code for assignment, and the value of `E_String` is the event code for string allocation. Thus, monitors can be written symbolically, in terms of these global variables, without reference to the actual one-character strings for event codes.

The following monitor, which tabulates functions calls, illustrates the selection of a single kind of event:

```
link evinit
procedure main(argl)
    EvInit(argl) | stop("*** can't load file")
    fcalls := table(0)
    mask := cset(E_Fcall)
    while EvGet(mask) do
        image(&eventvalue) ? {
            ="function " &
            fcalls[tab(0)] += 1
        }
```

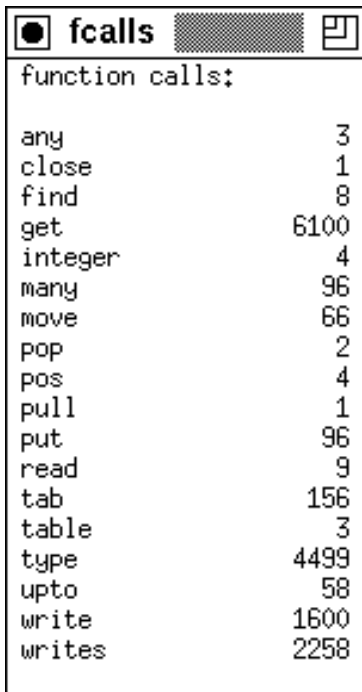


```

window := open("fcalls", "x",
  "lines=" || (*fcalls + 3),
  "columns=25") |
  stop("*** can't open window")
fcalls := sort(fcalls, 3)
write(window, " function calls:")
write(window)
while write(window, " ",
  left(get(fcalls), 15),
  right(get(fcalls), 8))
  XEvent(window)
end

```

Typical output from this monitor is:



function calls:	
any	3
close	1
find	8
get	6100
integer	4
many	96
move	66
pop	2
pos	4
pull	1
put	96
read	9
tab	156
table	3
type	4499
upto	58
write	1600
writes	2258

The monitor above is passive and, in fact, produces no output until the monitored program terminates. Monitors can take a much more active role and can affect the behavior of the monitored program if they wish.

Consider, for example, run-time errors [6]. It is possible, using the keyword `&error`, to have a run-time error converted into failure instead of having it cause error termination. This feature is, however, difficult to use, since it does not discriminate among different kinds of errors and applies to the entire program.

The instrumentation of MT Icon includes events for run-time errors. The corresponding values are the error numbers. Such events occur before possible error termination, so a monitor can watch for them and allow a user to decide whether to convert an error to failure or let it cause error termination. This can be done as follows:

```

while EvGet(E_Error) do {
  if keyword("error", &eventsourc) = 0 then {
    write(window,
      "Run-time error ",
      &eventvalue,
      ": ",
      keyword("errortext", &eventsourc)
    )
    write(window,
      "in ",
      keyword("file", &eventsourc),
      " at line ",
      keyword("line", &eventsourc)
    )
    writes(window, "Convert to failure? ")
    if read() == !"yY" then
      keyword("error", &eventsourc) := 1
    }
  }
}

```

If no run-time error occurs in the monitored program, this monitoring code does nothing. However, if a run-time error occurs in the monitored program and error conversion is not enabled there, the monitor provides information about the error to the user, who may then indicate that the error is to be converted to failure. In this case, setting `&error` to 1 in the monitored program causes the offending expression to fail.

Artificial Events

All the events described so far are produced by instrumentation in the MT Icon interpreter. A program also can produce events at the source level. The function

```
event(code, value)
```

in a monitored program produces an event in that program that transfers control to the monitor and sets its `&eventcode` to `code` and its `&eventvalue` to `value`.

Thus, "artificial" events can be produced by

inserting calls of `event()` in a program, either manually or with a preprocessor as suggested earlier. Since most kinds of events are covered by instrumentation in the interpreter, calls of `event()` in a program normally would be sparse and used only for special purposes.

The event codes produced by artificial events are not limited to the one-character strings used by the instrumentation. For example, the event code from an artificial event can be an integer or even a structure.

`EvGet()` has an optional second argument, which if non-null, allows reporting of events whose event codes are not one-character strings. Such event codes are reported regardless of the mask.

One use of artificial events is to allow a program to send “disabling” and “enabling” events that a monitor recognizes and uses to turn monitoring off and on. Thus, a program can restrict monitoring to a particular portion of itself. The disabling and enabling codes, given symbolically by `E_Disable` and `E_Enable`, are integers, so they are reported to `EvGet()` regardless of its mask.

Code in a monitor for disabling and enabling events can be written as follows:

```
while EvGet(mask, 1) do
  if &eventcode === E_Disable then {
    while EvGet("", 1) do
      if &eventcode === E_Enable
        then break
    }
  }
  else {
    ...          # regular monitoring
  }
```

Each event first is checked to see if it is a disabling event. If it is, `EvGet()` is called with an empty cset, turning off the reporting of all ordinary events until an enabling event occurs.

Note that the use of artificial events requires cooperation between a monitor and the program it monitors.

Discussion

With more than 100 kinds of events that extensively cover source-language semantics and also include many aspects of the implementation itself, there are many possibilities for program monitors. We'll discuss some of these in the next issue of the *Analyst*.

There's an even more exciting possibility — having several different monitors all monitoring the same program. MT Icon obviously allows several monitors to be loaded, but it's necessary to have some mechanism by which multiple monitors can get events from the same program and work cooperatively. In a subsequent article, we'll describe Eve, a program written by Clint Jeffery that serves as a controller for multiple monitors.

Acknowledgments

Clint Jeffery developed most of the features of MT Icon that are needed for monitoring. Ralph Griswold and Clint Jeffery provided the instrumentation in the Icon interpreter with help from Gregg Townsend and Ken Walker.

References

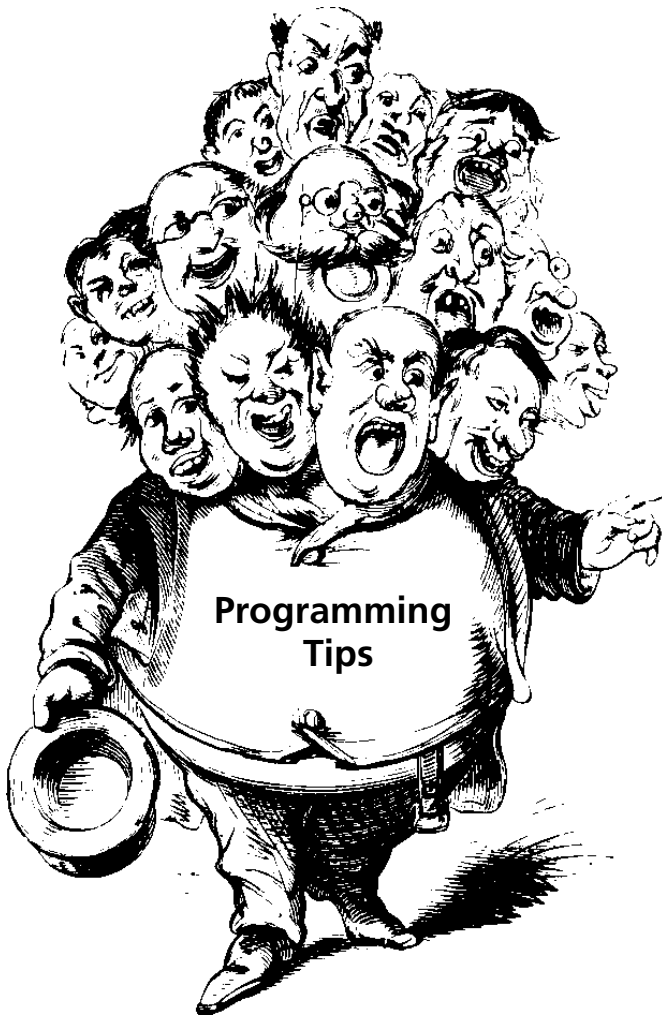
1. *The MT Icon Interpreter*, Clinton L. Jeffery, Icon Project Document IPD169, Department of Computer Science, The University of Arizona, 1992.
2. “Variant Translators”, *The Icon Analyst* 7, pp. 2-5.
3. “Memory Monitoring”, *The Icon Analyst* 1, pp. 7-10.
4. “Memory Monitoring”, *The Icon Analyst* 2, pp. 5-9.
5. *The Implementation of the Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Princeton University Press, Princeton, New Jersey, 1986, pp. 110-129, 264-278.
6. “Writing Bullet-Proof Programs”, *The Icon Analyst* 10, pp. 10-11.

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

FTP: cs.arizona.edu (cd /icon)



Efficiency

Most programmers are concerned about how fast their programs run. Execution speed usually depends most significantly on large-scale aspects of a program like its structure, the algorithms it uses, and the strategy with which the programming tasks are approached. But there are some lower-level matters that can help make programs run faster. If you tend to these lower-level aspects of programming as a matter of habit, you won't have to go back over a program to fine tune it for efficiency.

A few suggestions along these lines follow. We've mentioned some of these before, but we're repeating them here, since we continue to see programs that could benefit from their application.

1. Use the types that operations expect. Failure to do this results in type conversions that otherwise might be unnecessary. The most flagrant examples of poor usage in this regard occur

for literals, as in `upto("x")`, which should be written as `upto('x')`.

2. Let Icon do type conversions for you. If type conversion is needed, let the implicit mechanism do its work, don't do it yourself.

For example, if `n` is a string that represents a number, use

```
n + 1
```

not

```
numeric(n) + 1
```

Of course, these two expressions produce different results if `n` is a string that doesn't represent a number.

3. Order case clauses by likelihood of matching. Case clauses in a case expression are evaluated in the order they appear. If "yes" is more likely than "no", then put "yes" first, as in

```
case answer of {
  "yes": ...
  "no": ...
  :
}
```

In fact, it's worth knowing that a case expression is equivalent to a series of if expressions, as in

```
if answer === "yes" then ...
else if answer === "no" then ...
else if ...
```

Note that value comparison is used in case expressions.

4. Avoid unnecessary concatenation. Concatenation allocates storage and copies strings. Often strings to be written out can be constructed and written in order without concatenation. See Reference 1.

5. Use as few elementary operations as possible. There is a certain amount of overhead for every operation performed in Icon. Where there is a way of doing something that requires fewer operations than another way, it's generally faster to use the one that requires the fewer operations. For example,

```
s == ""
```

is faster than

```
*s = 0
```

for determining if a string is empty.

6. When possible, use element generation instead of subscripting when indexing through a structure. For example,

```
every x := !L do ...
```

is faster than

```
every x := L[1 to *L] do ...
```

The reason the first form is faster is not so much that it uses fewer operations but that element generation keeps track of where it is in a structure, while the second form must locate each element separately.

7. Avoid cset construction in loops. It is, of course, good programming practice to avoid any computation in a loop that can be moved outside. The problem with cset construction is that it's easy to overlook it. For example,

```
while tab(upto(~&letters)) do ...
```

constructs the complement of &letters every time through the loop.

You might ask why Icon doesn't take care of this automatically, since it's clear that ~&letters is a way of specifying a constant. Handling this at compile time is called constant folding. It's just something the implementation of Icon doesn't do.

8. Dispose of structures when they are no longer needed. Space for allocated data, which includes all structures in Icon, is automatically reclaimed by garbage collection if Icon needs more space.

Some programs build large data structures, use them, and then go on to other computations, leaving behind references to the data structures even though they are no longer needed.

The garbage collector has no way of knowing that data that logically can be referenced (such as the values of global variables) will not, in fact, actually be referenced in the future. The garbage collector must, therefore, keep such data. Furthermore, in the case of a structure, all the data in the structure must be kept, as must everything this data references, and so on.

Simply assigning a null value to a variable that references a structure that is no longer needed is all that's needed to "free" it and all the data it references. A simple expression like

```
database := &null
```

may do wonders to speed up subsequent program

execution.

9. Use an every loop to perform a task a specified number of times.

This may seem so obvious that it doesn't need mentioning, but persons who start using Icon after using another programming language sometimes fail to see the similarity between every and the for control structure of the other language. Instead, they write while loops, incrementing a counter and testing for a limit.

Reference

1. "String Allocation", *The Icon Analyst* 9, pp. 6.



What's Coming Up

In the next *Analyst*, we'll have an article on using program monitors to visualize program execution in Icon. We'll explain what program visualization is, describe some of the problems associated with it, and show some examples of our work so far.

Following up on the article on arrays in issue 14 of the *Analyst*, we'll describe how to use tables to implement sparse arrays in which only a small percentage of all possible elements are actually referenced.

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.