
The Icon Analyst

In-Depth Coverage of the Icon Programming Language

April 1994
Number 23

In this issue ...

Programmer-Defined Control Operations ... 1
Color in X-Icon ... 5
Subscription Renewal ... 7
Meta-Variant Translators ... 8
Programming Tips ... 11
What's Coming Up ... 12

Programmer-Defined Control Operations (continued)

In the last issue of the *Analyst*, we introduced the concept of programmer-defined control operations and showed examples of how they could be used to model existing control structures and to implement new ones. We'll continue in the same vein here.

More Examples

Alternation: In the last article, we showed how alternation could be written as a programmer-defined control operation:

```
procedure Alt(args)
  local x
  while x := @args[1] do suspend x
  while x := @args[2] do suspend x
end
```

We commented there that this procedure could be written more economically. Here's a more compact version that doesn't even need a local variable:

```
procedure Alt(args)
  suspend !@args[1]
  suspend !@args[2]
end
```

The repeated alternation control structure causes its argument co-expression to be repeatedly activated until it has no more results.

It's worth remembering that `!@C` generates all the results that the expression for `C` generates.

This procedure can be generalized to take an arbitrary number of arguments:

```
procedure Galt(args)
  local C
  every C := !args do
    suspend !@C
  end
```

For example, the result sequence for

```
Galt{1 to 3, !"abc", 4 to 5}
```

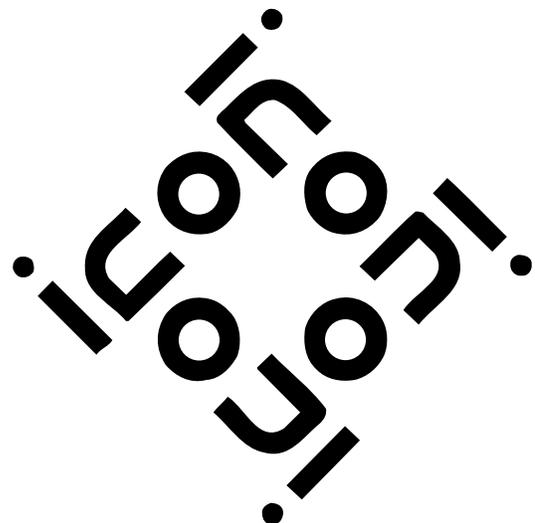
is {1, 2, 3, "a", "b", "c", 4, 5}.

It's tempting to go one step further, replacing

```
every C := !args do
  suspend !@C
```

by

```
suspend !@!args
```



This expression does not produce alternation, however. When repeated alternation is applied to `@!args`, `!args` suspends and is resumed, going on to the next co-expression in `args`. It's not until every co-expression has been processed once that repeated alternation starts over with the first co-expression to produce its second result. In fact, `!@!args` is a form of parallel evaluation, first producing the first result for each co-expression, then the second for each, and so on. Thus,

```
procedure Colseq(args)
  suspend !@!args
end
```

“collates” the sequences for its arguments. For example, the result sequence for

```
Colseq{1 to 4, !"abc", 4 to 5}
```

is `{1, "a", 4, 2, "b", 5, 3, "c", 4}`.

Notice that when there are no more results for one argument expression, the others keep going. Generation only stops when there are no more results for any argument expression, which causes repeated alternation to fail.

Goal-Directed Evaluation: Programmer-defined control operations can help clarify goal-directed evaluation. Here's a procedure that models the operation `expr1 + expr2`:

```
procedure Add(args) # e1 + e2
  local e1, e2

  while e1 := @args[1] do {
    args[2] := ^args[2]
    while e2 := @args[2] do
      suspend e1 + e2
    }
  }
end
```

This procedure illustrates that although the addition of two values produces only one result, an addition expression generates results if its argument expressions do. For example, the result sequence for `Add{1 to 3, 10 | 100}` is `{11, 101, 12, 102, 13, 103}`. The nested while loops show the last-in, first-out aspect of goal-directed evaluation. All the alternatives for the second argument expression are produced before another alternative for the first argument expression. And, for each alternative for the first argument expression, the second argument expression is evaluated anew, as shown

by the refreshing of its co-expression.

Contrast this with parallel addition, which generates the sums of corresponding alternatives for each argument expression:

```
procedure ParaAdd(args)
  local e1, e2

  while (e1 := @args[1]) &
    (e2 := @args[2]) do
    suspend e1 + e2
  end
```

Here the loop terminates when either of the argument expressions runs out of values. The result sequence for `ParAdd{1 to 3, 10 | 100}` is `{11, 102}`.

Goal-directed evaluation applies in the same way to the evaluation of the arguments of all operators and functions. Ignoring additional optional arguments and the handling of defaults, a programmer-defined control operation for `find(expr1, expr2)` has the same form as the programmer-defined control operation for `expr1 + expr2`:

```
procedure Find(args)
  local e1, e2

  while e1 := @args[1] do {
    args[2] := ^args[2]
    while e2 := @args[2] do
      suspend find(e1, e2)
    }
  }
end
```

Note that, unlike addition, `find(e1, e2)` itself may generate a sequence of results. This procedure illustrates that for any list of argument values, a function that is a generator produces all the results for those values before goal-directed evaluation supplies any new argument values.

This programmer-defined control operation can be generalized to handle any function of two arguments:

```
procedure Fnc2(args)
  local fnc, e1, e2

  while fnc := @args[1] do {
    args[2] := ^args[2]
    while e1 := @args[2] do {
      args[3] := ^args[3]
      while e2 := @args[3] do
        suspend fnc(e1, e2)
      }
    }
  }
```

```

    }
  }
end

```

Since the first argument of `Fnc2{}` is the function that is applied to the second and third arguments, the subscripts for `args` are shifted accordingly. Note that in this more general form, the expression for the function can itself be a generator. Thus,

```
Fnc2{upto | find, s1, s2}
```

is the programmer-defined control operation version of

```
(upto | find)(s1, s2)
```

See Reference 1 for remarks about this kind of construction.

This approach to writing programmer-defined control operations can be generalized to functions with any number of arguments. Without thinking just how this might be done, the general approach should come to mind: recursion [2].

String Scanning: String scanning provides a much more substantive example of modeling an existing Icon control structure:

```

procedure Scan(args)          # e1 ? e2
  local e1, e2
  local inner_pos, inner_subject
  local outer_pos, outer_subject
  while e1 := @args[1] do {    # e1
    # save outer environment
    outer_subject := &subject
    outer_pos := &pos
    # set up inner environment
    &subject := e1
    &pos := 1
    # start out fresh for expr2
    args[2] := ^args[2]
    while e2 := @args[2] do {  # e2
      # save inner environment
      inner_subject := &subject
      inner_pos := &pos
      # restore outer environment
      &subject := outer_subject
      &pos := outer_pos
      # suspend from scanning
      suspend e2
    }
  }

```

```

# update outer environment
outer_subject := &subject
outer_pos := &pos

# restore inner environment
&subject := inner_subject
&pos := inner_pos
}

# restore inner environment
&subject := outer_subject
&pos := outer_pos
}

```

end

We won't attempt to explain all the aspects of string scanning, including the maintenance of scanning environments, that this programmer-defined control operation handles. See Reference 3 for a detailed discussion, as well as a two-procedure model of string scanning that does not use programmer-defined control operations. It's worth nothing that the basic structure of goal-directed evaluation is the same for string scanning as it is for functions and operations.

Result Selection: We'll finish our examples of programmer-defined control operations with `Select{expr1, expr2}`, which selects the results of `expr1` according to positions specified by `expr2`. For example,

```
Select{!"abcde", 1 | 3 | 5}
```

generates the first, third, and fifth values generated by `!"abcde"` and has the result sequence `{"a", "c", "e"}`. We'll require that `expr2` generate positive integers in increasing order:

```

procedure Select(args)
  local i, j, x
  j := 0
  while i := @args[2] do {
    while j < i do
      if x := @args[1] then j += 1
      else fail
      if i = j then suspend x
      else stop("selection sequence error")
    }
  }
end

```

You might try your hand at rewriting `Select{}` to allow `expr2` to produce positive integers in non-decreasing order, so that

```
Select{!"abcde", 1 | 3 | 3 | 5}
```

would have the result sequence {"a", "c", "c", "e"}.

Limitations of Programmer-Defined Control Operations

As suggested at the end of the preceding article on programmer-defined control structures, there are limitations to what programmer-defined control operations can do. In fact, if you've tried using the programmer-defined control operations shown in that article, you probably already have stumbled across the major problem — the scope of identifiers in co-expressions.

When a co-expression is created, it includes *copies* of any local variables for the procedure in which it is created. It's necessary to copy local variables, since a co-expression may survive the call of the procedure in which it is created (for example, a co-expression can be returned from the procedure call in which it is created). In fact, every co-expression created in a procedure has its own copies of the procedure's local variables. As a result, although two co-expressions can have identifiers with the same names, these identifiers are distinct. As a consequence, local variables cannot be used to share data between co-expressions.

This has disastrous consequences for the looping control structures we've modeled, since data sharing often is essential in such control structures. Suppose, for example, that

```
every i := 1 to 10 do
  p(i)
```

is modeled by

```
Every{i := 1 to 10, p(i)}
```

Assuming *i* is a local variable, the two instances of *i* here are distinct, since they reside in two different co-expressions. Thus, incrementing *i* in the first argument expression has no effect on *i* in the second argument expression.

This problem can be overcome by using a non-local identifier, either global or static:

```
static i
...
Every{i := 1 to 10, p(i)}
```

Needless to say, this is not an attractive solution. For this reason, programmer-defined control

operations are not suitable for actual use in control structures in which argument expressions rely on sharing data via variables.

Another limitation of programmer-defined control operations is that they can't handle control structures that depend on the lexical context in which they occur. For example,

```
create break
```

is syntactically incorrect, since the `break` is not in a loop. Even if the `create` is in a loop, the co-expression containing the `break` could be exported out of the loop, so this kind of construction is not allowed. As a result,

```
While{s:= read(), if s == "stop" then break}
```

also is syntactically erroneous.

Conclusion

Despite some limitations, programmer-defined control operations are useful for understanding existing Icon control structures, experimenting with new ones, and for some kinds of expression evaluation, such as parallel evaluation, that otherwise are not available in Icon.

You'll find more examples in the Icon program library. Look at `pdco.icn` and `pdae.icn`. But you'll learn a lot more if you try to write some of your own. If you come up with programmer-defined control operations that you think are interesting, send them to us. If we get enough, we'll discuss them in a future article in the *Analyst*.

References

1. "Result Sequences", *Icon Analyst* 7, pp. 5-6.
2. "Programming Tips", *Icon Analyst* 13, pp. 10-12.
3. "Modeling String Scanning", *Icon Analyst* 6, pp. 1-2.

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.

Color in X-Icon

Editors' Note: Icon's graphics facilities are being redesigned and we expect to release a new version of Icon with these changes soon. This article, as well as previous ones about graphics, describe what's in Version 8.10. When the new version comes out, be sure to check the documentation for changes and new features.

Color Maps

If you get further into the use of color than just using few different colors for identifying objects and attracting attention to important situations, you'll run into one of the most troublesome aspects of dealing with colors.

Most modern color monitors are capable of displaying a very large number of different colors. 2^{24} colors (nearly 17 million) are common. On the other hand, the number of *different* colors that can be displayed at any one time is usually very limited. This limit is determined by the number of *planes* in the *frame buffer* used to drive the monitor. One plane is provided for each bit associated with a pixel on the screen. One bit, with one plane, supports only one color — a bi-level, black-and-white display. Color monitors typically have 8 planes, allowing $2^8 = 256$ different colors to be displayed at one time. The attribute depth gives the number of planes used in the colormap. For example if `XAttrib("depth")` returns 1, the display is bi-level. If it returns 8, 256 different colors (or maybe gray levels) are supported.

Because of the limited ability of the human eye to distinguish between colors, a number such as 256 is not as limiting as it might seem. Fairly realistic pictures can be composed from 256 different colors as long as there is a wide range of colors from which to choose. The real problem comes from the fact that the different available colors are shared by all the applications that use the screen. This includes colors your window system manager may use to decorate the title bars, as well as images belonging to other applications that are in the background when your Icon program is running.

On most platforms, the colors that appear on the screen are represented in a *color map* in which different colors are allocated. Different applications that use the same color share an entry in the color map. If the color map is full when a new color is needed, Icon creates a *virtual color map* that

initially has only the colors your Icon application needs and then allocates new colors in this virtual color map. This usually causes the colors for images associated with other applications to “go strange”, since they still reference the same places in the color map, but the colors may have changed.

Even with a virtual color map, the limitation on the number of different colors can become a problem for applications that display images with many colors, especially if the colors change.

The function `XFreeColor(s1, s2, ...)` frees the specified colors, allowing the corresponding entries in the color map to be reused. Another way to free colors is to close a window or completely erase it.

Using Color

Color specifications can be used to select the foreground and background colors when a window is opened and to change these colors using `XFg()`, `XBg()`, and `XAttrib()`.

When several different colors are used to indicate different situations to identify different kinds of objects, graphic contexts that differ only in their foreground colors can be useful.

For example, an application that displays different kinds of Icon values might use different colors to distinguish different types. A table of graphic contexts provides an easy and quick way to pick the appropriate color, as in

```
value := table()
value["integer"] := XBind(&window, , "green")
value["real"] := XBind(&window, , "blue")
value["cset"] := XBind(&window, , "brown")
...
write(value[type(x)], image(x))
```

Mutable Colors

Another problem with color arises if you want to change all the pixels of one color to another color. Even if you know where the pixels are, it may be complicated and time-consuming to change them.

Some color displays can dynamically change the RGB values for color map entries and hence change the displayed colors almost instantaneously. Color map entries used in this way are called *mutable*.

The function `XNewColor()` allocates a mutable color if one is available but fails if one isn't. `XNewColor()` returns a small negative integer that represents this mutable color. This integer then can be used in `XFg()`, `XBg()`, and `XAttrib()`. `XNewColor(s)` can be used to initialize the new color, where `s` is a string specification for the color.

The function `XColor(i, s)` can be used to change a mutable color, where `i` is a value produced by `XNewColor()` and `s` is a color specification.

Monochrome Portability

Applications that are designed for color screens can be ugly or unusable on monochrome screens unless some attention is paid to portability.

On a bi-level screen, `XColor()` can only choose black or white, and it returns whichever of these is closest to the requested color. On a multi-level monochrome screen, it chooses the closest gray.

Choosing black or white is about the best that can be done for drawing lines or writing text, but it doesn't work very well for colors that cover large areas. The Icon program library procedure `XShade(w, s)` addresses this problem. On a color screen, `XShade()` acts just like `XFg()`. On a bi-level display, `XShade()` sets the graphic context to use the halftone pattern that approximates the darkness of the requested color.

The best results usually are obtained by designing an application to consider at least two types of displays: bi-level and color. A test such as

```
if XAttrib("depth") = 1 then ...
```

might be used.

Note that this approach groups gray-scale displays with color displays — both have depths greater than 1 — on the assumption that approximating colors with grays looks better than using halftones.

Reading the Canvas

In some applications, you may need to know the colors of pixels on the canvas. The function `XPixel(x, y, w, h)` generates the pixel colors from the specified rectangular area. Colors are generated starting in the upper-left corner of the rectangular area, advancing down each pixel column before going to the next. Pixel colors are represented by integers. Ordinary colors are nonnegative integers while mutable colors are given by the

negative integers produced by `XNewColor()`. Ordinary colors are encoded in three eight-bit bytes. The most significant byte is zero, followed by bytes for the red, green, and blue components of the color. The 8-bit color values are the most significant bits of the regular 16-bit color values. Consequently color precision may be lost. (The encoding is a concession to efficiency and suffices for most situations.)

Printing Color Images

One problem that plagues the use of color is the need to produce printed copies of color images for use in technical reports, dissertations, journal papers, and so forth.

Although new technologies have lowered the cost and increased the ease of color printing, it's still impractical for most of us. Color images often are printed in black and white in hope of capturing at least some sense of the distinctions that color provides. Half-toning is used to convert colors to different shades of gray.

The trouble with this is that radically different colors may appear similar or even identical when printed. See Figure 1.



Figure 1. Colors Printed as Grays

Ways of dealing with this problem are beyond the scope of this article, but if you expect to

print color images in black and white, you may want to consider color selection in advance. Another possibility is to use an image-manipulation application to convert colors to ones more suitable for black-and-white printing when the time comes.

Conclusion

This concludes our discussion of color in Icon. Although we've described all the features, we've just scratched the surface of what's involved in using color.

There are many complications and lots of effort and experience are needed to use color effectively. But there also are many potential rewards, not the least of which is the fun involved and the satisfaction of getting really nice results.



Subscription Renewal

For many of you, the next issue is the last in your present subscription to the *Analyst* and you'll find a subscription renewal form in the center of this issue. Renew now so that you won't miss an issue.

Because of the way that the budget for the Icon Project is handled, we need to know soon what to expect for our next fiscal year. Your prompt renewal helps us plan.

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

© 1994 by Madge T. Griswold and Ralph E. Griswold
All rights reserved.

Meta-Variant Translators

In an earlier issue of the *Analyst* [1], we described variant translators, a system for constructing robust preprocessors for Icon programs. The variant translator specification system makes it easy to specify changes to programs, such as the one for modeling string scanning [2]:

```
expr1 ? expr2 → Escan(Bscan(expr1), expr2)
```

A single specification does the trick and works regardless of the complexity of the expressions involved:

```
Bques(x, y, z) "Escan(Bscan(" x "), " z ")"
```

Writing such specifications is fairly easy, once you learn a few rules. However, if a variant translation is complicated, its specification may be tedious to construct and prone to error. Furthermore, such specifications aren't feasible for specialized variant translations, such as for translating one procedure name differently from all other procedure names. Such translations can be accomplished by using C functions, but the kind of code that is needed is tedious to write, hard to modify, and requires proficiency in C.

This article describes a higher-level approach for producing variant translators, called meta-variant translators, that allows variant translations to be written in Icon instead of C.

An ordinary variant translator translates an Icon program into another Icon program, as shown in Figure 1. To change the translation, it is necessary to change the translation specifications and build a new variant translator, vt, as indicated by the asterisk.

A meta-variant translator translates an Icon program into another Icon program, which is translated and linked with a library of code-generation procedures, gen.icn, to produce the final Icon program, as shown in Figure 2.

The standard version of gen.icn contains procedures that perform an "identity" translation, so that the output is same to the input except for layout. A variant translation is accomplished by making changes to the code-generation procedures in gen.icn, rather than by changing the variant translator — that is, by using a variant gen.icn. Note that changing the translation does not require changing mvt.

Figure 3 shows a simple example of code generation for identity translation. In the second step, the output of the meta-variant translator is combined with gen.icn. Representative procedures from the standard version of gen.icn are shown in Figure 4.

As shown there, the standard procedure for translating scanning expressions is

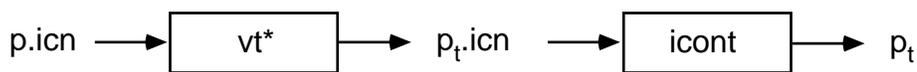


Figure 1. Variant translation

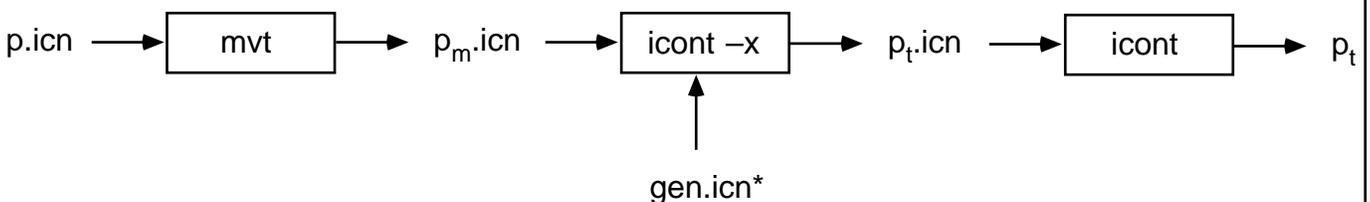


Figure 2. Meta-variant translation

```

procedure main()
  while line := read() do
    line ? process()
  end

```

Input to mvt

```

procedure program()
Proc_("main",)
Reduce_(While_Do_(Asgnop_(":=",Var_("line"),
Invoke_(Var_("read"),Null_())),Scan_(Var_("line"),
Invoke_(Var_("process"),Null_()))),)
End_()
end

```

Output of mvt

```

procedure main()
while (line := read()) do (line ? process())
end

```

Result of Running the Output of mvt

Figure 3. Meta-variant identity translation

```

procedure Scan_(e1, e2)
  return "(" || e1 || " ? " || e2 || ")"
end

```

To get the variant translation for modeling string scanning, it's only necessary to change this procedure to

```

procedure Scan_(e1, e2)
  return "Escan(Bscan(" || e1 || ")," || e2 || ")"
end

```

As another example, suppose you want a variant translation to convert calls of `map()` to calls of `Map()` so that you can trace the function `map()` by providing a procedure `Map()` that does the same thing. The declaration for `Map()` might be

```

procedure Map(s1, s2, s3)
  return map(s1, s2, s3)
end

```

The variant translation can be accomplished by adding the line

```
if s == "map" then s := "Map"
```

at the beginning of the procedure `Proc_()` as shown in Figure 4.

It's also necessary to get the procedure declaration for `Map()` into the final program. This can be done by adding the following lines to the beginning or end of `main()` in `gen.icn`:

```

write("procedure Map(s1, s2, s3)")
write("  return map(s1, s2, s3)")
write("end")

```

An alternative approach, which is more desirable in the case of more elaborate variant translations of this type, is to add

```
write("link libe")
```

at the beginning or end of `main()` in `gen.icn` and provide the code to be linked with the final program in `libe.icn`.

If you want to trace a generator, be sure to use `suspend` instead of `return`; otherwise your procedure won't produce all the results produced by the generator. For example, for `seq()`, the procedure would be

```

procedure Seq(i1, i2)
  suspend seq(i1, i2)
end

```

In fact, it doesn't hurt to use `suspend` for functions that aren't generators.

Conclusions

As described above, meta-variant translators allow you to write variant translators in Icon. The job is so easy that all kinds of things are worth doing that you'd probably not consider with standard variant translators and C.

There are, however, a few potential problems with meta-variant translators. Producing a translation, once you have `gen.icn` the way you want it, is somewhat more complex and slightly slower than for standard variant translators. The complexity can be hidden in a script and the loss in translation speed generally is insignificant, given the amount of *programming* time that it takes to craft a standard variant translator, as opposed to a meta-variant one.

```

# main() calls program(), which is produced by the
# meta-variant translation.
procedure main()
  program()
end
# Declarations
procedure Proc_(s, es[])          # procedure s(v1, v2, ...)
  local result, e
  if *es = 0 then write("procedure ", s, "()")
  result := ""
  every e := !es do
    if !e == "[]" then result[-2:0] := e || ", "
    else result ||:= (e | " ") || ", "
  write("procedure ", s, "(", result[1:-2], ")")
  return
end
procedure End_()                  # end
  write("end")
  return
end
  etc.
# Expressions
procedure Asgnop_(op, e1, e2)     # e1 op e2
  return "(" || e1 || " " || op || " " || e2 || ")"
end
procedure Binop_(op, e1, e2)     # e1 op e2
  return "(" || e1 || " " || op || " " || e2 || ")"
end
procedure Scan_(e1, e2)          # e1 ? e2
  return "(" || e1 || " ? " || e2 || ")"
end
procedure Invoke_(e0, es[])      # e0(e1, e2, ...)
  local result
  if *es = 0 then return e0 || "()"
  result := ""
  every result ||:= !es || ", "
  return e0 || "(" || result[1:-2] || ")"
end
procedure Null_()                # &>null
  return ""
end
procedure While_Do_(e1, e2)      # while e1 do e2
  return "while " || e1 || " do " || e2
end
  etc.

```

Figure 4. Standard code-generation procedures

A more serious problem is the amount of memory required to build the Icon program that produces the final translation. As illustrated in Figure 3, the output of mvt is considerably larger than the input to mvt. If the input to mvt is a large program, the output is a huge one. It's also necessary to link gen.icn with the output of mvt, adding to the size of the intermediate program. The memory needed usually is not a problem on platforms in the workstation class, but it certainly can be on personal computers.

As mentioned above, it's not necessary to change mvt (a standard variant translator) to change the variant translation. This is true as long as the input language is standard Icon. If the input language is different from Icon, as say, in the variant translator Seque for [3], then a different version of mvt is needed. If the output language is different from Icon, as it is in the translation of Rebus to SNOBOL4 [4], then a considerably different version of gen.icn may be needed.

Getting Meta-Variant Translators

The meta-variant translator system described here is available by anonymous FTP to cs.arizona.edu; cd /icon/meta and get READ.ME to see what to do next.

References

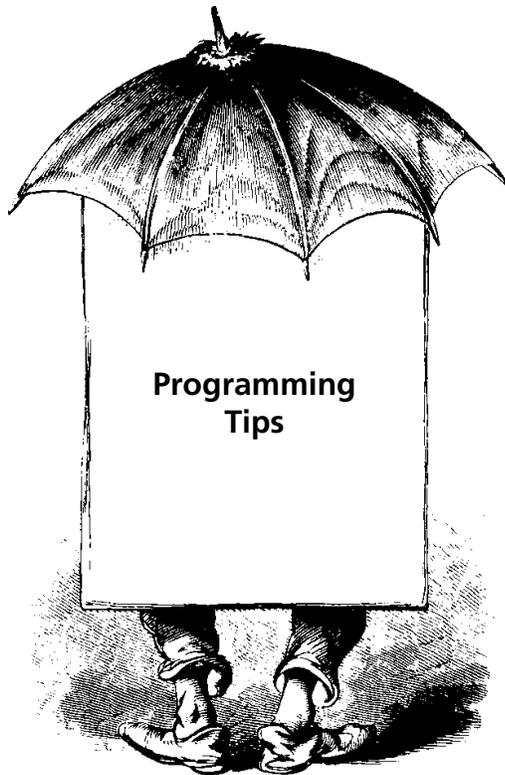
1. "Variant Translators", *Icon Analyst* 7, pp. 2-5.
2. "Modeling String Scanning", *Icon Analyst* 6, pp. 1-2.
3. "Lost Languages — Seque", *Icon Analyst* 19, pp. 1-4.
4. "Lost Languages — Rebus", *Icon Analyst* 18, pp. 1-4.

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

RBBS: (602) 621-2283

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



Breaking Out of Loops

You know that you can get out of a loop by evaluating `break` while inside the loop. This applies to any kind of loop: `every-do`, `repeat`, `until-do`, and `while-do`.

Two Icon programmers, Ray Pereda and Paul Abrahams, recently pointed out to us that the Icon book barely mentions that `break` has an optional argument. In looking through the book, we see only two places where `break` is used with an argument (one of which is rather amazing) and these are buried in a sample program at the back of the book [1].

We touched on this subject in a recent *Analyst* article [2] but hardly did it justice. We'll correct these omissions here.

The reason that you may not think about using `break` with an argument is that the argument is optional and defaults to the null value. Since `break` accomplishes its function of transferring control to the end of a loop without an explicit argument, it may not be obvious what use an argument could have.

In order to understand the function of an argument to `break`, it may be helpful to think of the execution of

```
break expr
```

as replacing the loop in which it occurs by *expr* as if it were a dynamic macro expansion.

Thus, in

```
while ... do {      # outer loop
  ...
  while ... do {    # inner loop
    ...
    break "break out of loop"
  }
  ...
}
```

the result of executing the `break` expression can be pictured as

```
while ... do {      # outer loop
  ...
  "break out of loop"
  ...
}
```

This example illustrates why you may not have thought of using an argument with `break` even if you knew it was possible: Since a loop fails when it terminates normally, it usually is written as an isolated expression whose outcome is ignored. So, in the case above, if `break` causes a loop to produce a value, there's nothing to use that value.

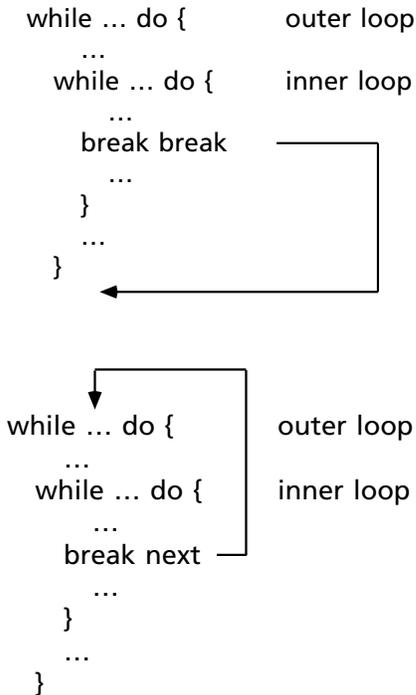
A glance at the example above suggests a use, however. Suppose you want to know if a loop is terminated by `break`. You would write

```
write(&errout,
  while ... do {
    ...
    break "break out of loop"
    ...
  }
)
```

If the loop terminates normally, it fails and nothing is written. If, however, the `break` is evaluated, `break out of loop` is written to standard error output.

There are more generally useful forms of `break` that you may have seen in programs but not thought much about. For example, the argument of `break` can be `break` or `next`. The expression `break break` breaks out of two levels of loop, while `break next` breaks out of the current loop and transfers control

to the beginning of the enclosing loop. These uses are illustrated in the diagrams below. You may find the macro expansion analogy useful in understanding these examples.



You'll even see expressions like `break break break` in some programs. (Every time we see this, we're reminded of the first line of one of Alfred Lord Tennyson's poems: "Break, break, break, on thy cold gray stones, O sea!") You're not likely to see very long strings of breaks, since loops rarely are nested many levels deep. If they are, it's hard for human beings to grasp the structure. But there's no limit to what a program that writes a program may do in this regard.

As mentioned above, most uses of `break` do not have arguments that supply an explicit loop value. Normally, for the reasons mentioned, the default null value doesn't cause any problems. If, however, a program relies on the failure of a loop, whether or not it terminates normally, you can assure this by using

```
break &fail
```

Incidentally, Dave Hanson introduced us to the idea of viewing conditional expression evaluation as a kind of dynamic macro expansion. At the time, we were considering the semantics of Icon's

```
if expr1 then expr2 else expr3
```

Originally, *expr2* and *expr3* were limited to one

result. The question was asked "Why perform this gratuitous limitation?" One person said "Something bad might happen if *expr2* or *expr3* could be generators." This seemed vague to others of us, especially since none of us could point to an actual example where generation would cause a problem. But we had nagging doubts and no really strong argument for the usefulness of generators in this situation.

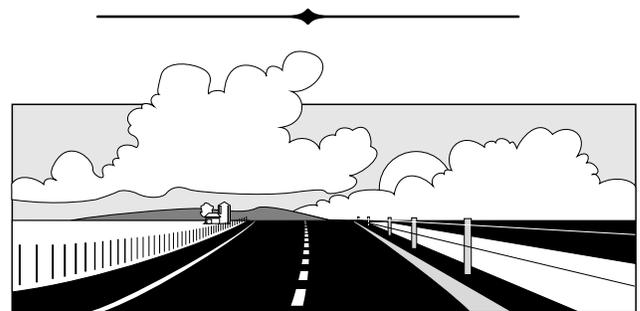
Dave suggested that we view the control structure as a macro, so that

```
if expr1 then expr2 else expr3
```

is "replaced" by *expr2* or *expr3*, depending on whether *expr1* succeeds or fails. Viewed that way, limitation seemed unnatural and we removed it. Nothing bad happened. And, although rarely used, there are situations in which generators in the arms of a selection expression are both natural and useful [3].

References

1. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, pp. 318-321.
2. "Programming Tips — Scanning Lines of a File", *Icon Analyst* 19, pp. 10-12.
3. "Programming Tips — Exploiting Expression-Based Syntax", *Icon Analyst* 11, p. 11.



What's Coming Up

In the next issue of the *Analyst*, we'll conclude the series of articles on graphics with a discussion of image files. In addition, we'll have an article on turtle graphics, which are based on a navigational view of drawing. And we'll start a new feature that describes some of the more useful programs and procedures in the Icon program library.