

---

---

# The Icon Analyst

---

*In-Depth Coverage of the Icon Programming Language*

---

October 1993  
Number 20

---

## In this issue ...

Exercises ...	1
Icon Made Difficult ...	1
Dealing with Windows in X-Icon ...	3
Piped Scanning ...	6
Solutions to Exercises ...	9
Programming Tips ...	11
What's Coming Up ...	12

## Exercises

It's been more than a year since we suggested some Icon programming exercises. Several readers have asked that we continue this feature of the *Analyst* so that they can develop their Icon programming skills. We've also been asked to include solutions in the same issue as the exercises. You'll find them starting on page 9.

If you're really interested in developing your Icon programming skills, do the best you can with these exercises before looking at the solutions.

If you have better solutions than we do or if you have comments, please send them to us; we'll discuss them in a later issue of the *Analyst*.

These exercises involve writing procedures related to Icon structures.

1. Write a procedure `revlist(lst)` that returns a list with the elements of the list `lst` in reverse order. Do not modify `lst`.

2. Write a procedure `keylist(tbl)` that returns a list of the keys in table `tbl` in sorted order.

3. Write a procedure `valset(tbl)` that returns a set whose members are the values in the table `tbl`.

4. Write a procedure `seteq(set1, set2)` that succeeds if and only if the sets `set1` and `set2` are "equivalent" — that is, if they contain the same members.

5. Write a procedure `setlt(set1, set2)` that succeeds if and only if set `set1` is "less than" `set2` — that is, if `set1` is a proper subset of `set2` (all members of `set2` are in `set1` but `set2` contains additional members).

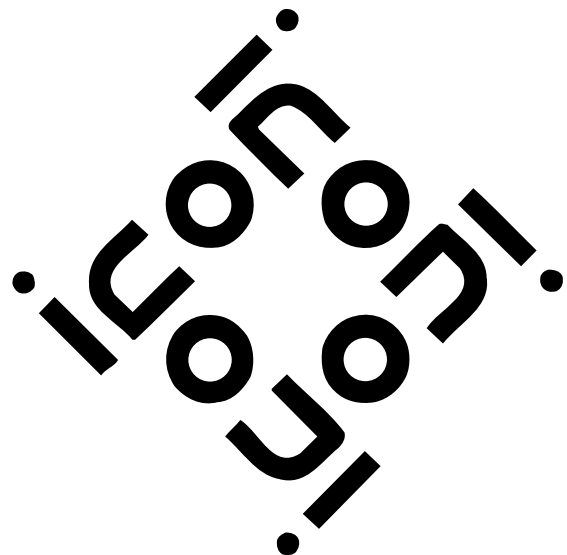
6. Write a procedure `tableq(tbl1, tbl2)` that succeeds if and only if tables `tbl1` and `tbl2` are "equivalent" — that is, if they have the same keys and the same corresponding values.

---

## Icon Made Difficult

Some time ago we came across a delightful little book titled *Mathematics Made Difficult* [1]. The spirit of the book is captured in this remark from its introduction: "Mathematicians always strive to confuse their audiences; where there is no confusion there is no prestige".

Sometimes it seems to us that programmers often apply the same principle. It is a temptation, although disorganized thinking and sloppy programming practices serve nicely even when there's no conscious intent to obfuscate.



Icon Analyst

We certainly don't advocate making programs difficult to read. In fact, we try very hard to make our own programs easy to understand, although at times we write unnecessarily clever and obscure code and pass it off as an idiom. But we thought it might be fun to show a few examples of "Icon made difficult".

Here's the expression that started us on this light-hearted if somewhat demented adventure:

```
&subject := &subject
```

At first glance, this looks like a "no-op". What possible use could there be to assigning the value of a variable to itself except to make a program unnecessarily long and slow? But assignment to `&subject` sets `&pos` to 1. So

```
&subject := &subject
```

is an obscure way of doing what

```
&pos := 1
```

does.

Now we've caught the obfuscation bug. How about

```
&subject <- &subject
```

That leads to

```
every &subject <- &subject
```

which sets `&pos` to 1 twice. Well, so does

```
&subject :=: &subject
```

or even

```
&subject <-> &subject
```

And, of course, there's

```
every &subject <-> &subject
```

which sets `&pos` to 1 four times!

There are other variations on our mad theme:

```
&subject ?:= tab(0)
```

Actually,

```
&subject ?:= &subject
```

does the same thing.

Enough, you say. I didn't subscribe to the *Analyst* to get cheap headaches from joke programming! Agreed. But we'll be good sports. If you have examples of unnecessarily obscure Icon code, send them to us. Surely, for example, you can

think of something to beat on other than `&subject`. We make no promises about publication, though. We don't dare.

## Reference

1. Carl E. Linderholm, *Mathematics Made Difficult*, World Publishing, 1972.

## 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

---

© 1993 by Madge T. Griswold and Ralph E. Griswold  
All rights reserved.

## Dealing with Windows in X-Icon

This is the fourth in our series of articles on X-Icon. Some features of X-Icon changed with the release of Version 8.10 of Icon, and this and subsequent articles refer to the new version. If you need documentation for X-Icon as of Version 8.10, it's available to you free as a benefit of subscribing to the *Analyst*. Ask for TR 93-3. Be sure to identify yourself as an *Analyst* subscriber.

### Windows

X-Icon provides many facilities related to windows: opening and closing them, where they are, how big they are, their foreground and background colors, and so forth. Like any function repertoire, there are things you may rarely or never need. On the other hand, knowing the facilities that are available may suggest possibilities that you otherwise might overlook.

### Window Managers

When you run X, there's a window manager that allows you to control the location and sizes of windows on the screen. The window manager also determines the appearance of some aspects of windows, such as the title bar that appears at the top of most windows.

Interaction between the user and the window manager typically occurs via the title bar. For example, if you press a specified mouse button when the mouse cursor is on the appropriate part of the title bar, your window manager may let you drag the window to another position on the screen. Pressing another button at another place may "iconify" the window, reducing it to a small image to free up screen space, and so on. Window managers also service requests from the client program — that is, your X-Icon program.

There are lots of different window managers and some of them can be configured in a wide variety of ways. We usually use twm (formerly called "Tom's Window Manager", but now officially known as the "Tab Window Manager"). The windows shown here illustrate the style used by twm.

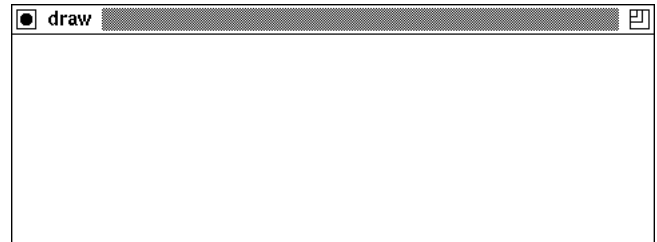
### Opening Windows

A window is opened by using "x" as the second argument of `open()`, as in

```
&window := open("draw", "x") |
stop("*** cannot open window")
```

which assigns a value of type window to `&window`. As with any use of `open()`, it's important to make provisions for the possibility that it may fail.

The first argument to `open()` provides a label, sometimes called a title, that identifies the window and typically appears on the title bar:



A window has many attributes associated with it. The defaults for these attributes produce a window of modest size that's good enough for most experiments. However, you'll generally want to customize your windows.

The attributes of a window can be specified as additional arguments to `open()` or set later on using `XAttrib()`. The attributes you're most likely to want to set when opening a window are its size, its location on the screen, and its foreground and background colors.

### Window Size and Position

The size of a window can be specified in two ways: by width and height in terms of pixels or by rows and columns in terms of the window's text font. For example,

```
&window := open("draw", "x",
"width=300",
"height=150") |
stop("*** cannot open window")
```

opens a window that is 300 pixels wide and 150 pixels high, while

```
&window := open("draw", "x",
"rows=40",
"columns=80") |
stop("*** cannot open window")
```

opens a window that can accommodate 40 lines of 80-column text in the default font.

The initial position of the upper-left corner of a window can be specified in terms of x,y pixel coordinates. The value of the `pos` attribute is an

integer pair of x-y coordinates measured relative to the upper-left corner of the screen. For example, the `open()` argument

```
"pos=100,200"
```

causes the window to be placed with its upper-left corner 100 pixels from the left edge of the screen and 200 pixels from the top of the screen.

The attributes `posx` and `posy` can be used to specify the coordinates individually, or the `geometry` attribute can be used to specify both the size and position of the window. The value of `geometry` has the form

```
widthxheight+xoff+yoff
```

where *width*, *height*, *xoff*, and *yoff* are integers that specify pixels. As you'd expect, *width* and *height* specify the size of the window and *xoff* and *yoff* specify offsets of the window's upper-left corner from the screen's upper-left corner. For example,

```
"geometry=100x200+0+0"
```

specifies a window that is 100 pixels wide, 200 pixels high, with its upper-left corner at the upper-left corner of the screen.

If you don't specify a window's size and position when it's opened, what happens depends on your window manager and how it's configured. For example, positioning a new window may be left to the user, who may be presented with a window outline to position and size manually.

Once a window is open, its size and position can be changed by the user via the window manager or by the program itself. For example,

```
XAttrib("width=600")
```

changes the width of the window to 600 pixels. Attributes also can be queried, as in

```
height := XAttrib("height")
```

which assigns to `height` the height of `&window`.

## Icons and Labels

As mentioned above, the window manager may allow a window to be iconified, with its contents hidden and reduced to a small box on the screen that typically shows only the window's label, as in



In this sense, a window has two possible states, "window" or "icon". The state of a window can be controlled by the program using the attribute `iconic`. For example,

```
XAttrib("iconic=icon")
```

iconifies `&window` and

```
XAttrib("iconic=window")
```

exposes `&window`, restoring it to its full size and showing its contents.

When a window is iconified, its contents still can be changed, although the change is not visible. When an iconified window is restored to its non-iconified form, the changes become apparent.

The attribute `iconpos` can be used to specify where the iconic form of a window is located. For example,

```
XAttrib("iconpos=0,0")
```

specifies that the icon for `&window` is at the upper-left corner of the screen. If the window already is iconified, it is moved to the specified position. If the window is not iconified, it moves to the specified position when it is iconified. Thus, the position of an icon can be specified in advance, whether the window is iconified by the program or by the user through the window manager.

The labels associated with a window and its icon can be changed by using the attributes `windowlabel` and `iconlabel`, respectively.

## Foreground and Background Colors

When a window is opened, its contents consist entirely of pixels in the background color, which defaults to white. Drawing and text written to the window appear in the foreground color, which defaults to black.

The foreground and background colors can be changed by `XBg()` and `XFg()`, respectively. For example,

```
XBg("blue")  
XFg("white")
```

change the background and foreground colors for `&window` to blue and white, respectively. These changes affect future operations but do not change the appearance of anything already displayed. Subsequent drawing and text operations are done in white. In the case of text, the area surrounding

the characters is drawn in the new background color, blue.

The function `XEraseArea(x, y, w, h)` clears all or a portion of a window to the current background color. The area cleared starts at `x` and `y` and extends for `w` and `h`. The arguments `x` and `y` default to 0. If `w` or `h` is omitted or 0, the cleared area extends to the edge of the window in that direction. Thus,

```
XEraseArea()
```

clears the entire window.

The function `XClearArea()` is similar to `XEraseArea()`, except that the portion of the window cleared is set to the background color that existed when the window was opened. In the example above, `XClearArea()` clears the window to white, while `XEraseArea()` clears the window to blue.

The function

```
XCopyArea(win1, win2, x1, y1, w, h, x2, y2)
```

copies a rectangular area within `win1` specified by `x1`, `y1`, `w`, and `h` to `win2` at offset `x1`, `y1`. `&window` is not a default for this function. The coordinates default to 0, and `w` and `h` default to the edge of `win1`. `win1` and `win2` may be the same.

## Stacked Windows

When several windows are on the screen at the same time, they may overlap so that one window obscures another. In this sense, the obscured window is behind another window. In some cases, a window may be completely obscured by another one.

The user can bring a window to the front using the window manager. For example, with some window managers, clicking on an exposed portion of an obscured window may expose the entire window, putting it in front of all other windows. If a window is completely obscured, of course, it may be necessary to move or resize other windows to make a portion of the obscured window visible.

A program also can change the order of its windows using the functions `XRaise()` and `XLower()` without any action on part of the user. `XRaise()` brings `&window` to the front, so that all other windows are behind it. Conversely, `XLower()` puts `&window` to the back of all other windows, possibly obscuring all or part of it.

## I/O Buffers and Synchronization

Some window systems buffer text and graphic output. Output to a window is automatically flushed when the window is waiting for input. The function `XFlush()` can be used to force pending output to be written to the window.

In the X client/server model, the client (that is the X-Icon program) may send requests to the server before the server has processed other pending requests. In this situation the display may lag behind the program. The function `XSync()` causes all output to be flushed and then waits for an acknowledgment from the server that all pending requests have been processed.

## Closing Windows

The function `close(w)` closes the window `w`, which then disappears from the screen. In the case of `&window`, the argument must be given explicitly.

All windows are closed automatically when a program terminates, so it is not necessary to close windows explicitly before a program terminates.

## A Final Word on Window Managers

Actions such as moving a window, resizing a window, iconifying it, and so forth are done by the window manager. Not all window managers honor all such requests from a program.

## Future Articles

In the next issue of the *Analyst*, we'll dig a little deeper into what a window is and reveal that it consists of a binding between a *canvas*, on which drawing is done, and a *graphic context*, which determines how drawing is done. Following that, we'll start on a topic that is both difficult and potentially rewarding — color. The last article on X-Icon we have planned is about images: how what appears in a window can be saved as a file and how an image can be brought into a window from a file.

## Reference

1. *X-Icon: An Icon Window Interface; Version 8.10* Clinton L. Jeffery and Gregg M. Townsend, Technical Report TR 93-9, Department of Computer Science, The University of Arizona, 1992.

## Piped Scanning

As we showed in an earlier article [1], string scanning expressions can be used in combination in several ways. One way is nested scanning, in which a string scanning expression occurs inside another, as in

```
s1 ? {  
  ...  
  s2 ? {  
    ...  
  }  
  ...  
}
```

String scanning expressions also can be used in conjunction, as in

```
(s1 ? { ... }) & (s2 ? { ... })
```

Neither of these kinds of combinations of string scanning expressions occurs frequently in practice, although they sometimes occur in ways that are not obvious, such as

```
s ? {  
  ...  
  p()  
  ...  
}
```

and

```
(s ? { ... }) & p()
```

where `p()` contains scanning expressions.

Recently we discovered another situation in which string scanning expressions can be used in combination:

```
s ? { ... } ? { ... } ? ... { ... }
```

Like most infix operators, the string scanning operator groups to the left, so the expressions above group as

```
((((s ? { ... }) ? { ... }) ? { ... }) ? ... { ... })
```

Parentheses do just as well as braces for grouping string scanning; we've used braces to show the right operands of the scanning operator in order to distinguish between these operands and the grouping of the scanning operators.

As a consequence of this grouping, the result of the left-most scanning operation provides the subject for the next scanning expression, and so on. The result is akin to a pipe with values "flowing"

from left to right through a sequence of scanning expressions.

Such an arrangement of scanning expressions may seem a bit peculiar to you and you may wonder how such a thing could be useful. We'll start with a simple example.

Suppose a file consists of lines with fixed-width fields and that the first field may begin with a command. If it does, the command is followed by a colon (colons may appear in other fields, too, but at most one colon appears in the first field). Now suppose you want a list of all the different commands in the file. You might use any of several different methods. We'll use string scanning (of course) and, of the many approaches there, use the following one:

```
command:= set()  
every !&input ? {  
  move(fw1) ? {  
    insert(command, tab(upto(':')))  
  }  
}
```

See Reference 2 for the way `every` is used here.

Notice that none of the braces in the expression above is needed to group the expressions. Suppose we remove all the braces:

```
every !&input ?  
move(fw1) ?  
insert(command, tab(upto(':')))
```

Has the removal of the braces changed anything? In one sense it has: The expressions now group from left to right instead of right to left. But since the result of

```
expr1 ? expr2
```

is the result of *expr2*, the two forms do the same thing.

But what is the advantage of the second form, aside from not having the braces? Braces aren't the point. The advantage is conceptual. Instead of nesting, which is difficult for human beings to understand and use correctly, the second form can be thought of as strings passing through a succession of independent scanning expressions — the pipe analogy mentioned above. Note that

```
every !&input
```

provides a source of subjects for the left-most scanning expression in the pipe. It "drives" the pipe.

To emphasize the independence of the components of a scanning pipe, a typographical device that keeps each component at the same “level” is important. That’s why we didn’t indent the expressions in the example without braces above. Since pipes can contain many components, stringing them out horizontally produces long lines — in fact, we can’t do that with the example above within the limitations of our two-column format. Furthermore, using braces to enclose the right operand of string scanning is good practice, since such operands often are compound expressions. Therefore, we’ve adopted the following typographical format for piped scanning:

```
source ? {
  expr1
} ? {
  expr2
} ? {
  ...
} ? {
  exprn
}
```

where *source* is an expression that drives the pipe.

In this form, the example above becomes

```
every !&input ? {
  move(fw1)
} ? {
  insert(command, tab(upto(':')))
}
```

Of course, such a layout is hardly necessary for this simple example, but as more components are added to the pipe, the vertical format with braces will be helpful.

Consider a variation on the example above where, instead of finding all the distinct commands, we count the number of lines whose first fields contain a command (as opposed to containing something else). This just amounts to changing one of the pipe components in the example above:

```
count := 0
every !&input ? {
  move(fw1)
} ? {
  upto(':') & count += 1
}
```

It’s worth noting in these examples that both `tab()` and `upto()` can fail. When a pipe component fails, nothing “goes forward”, of course.

This is simple enough, but if we insist on putting separate functionality in separate pipe components, we can do this:

```
every !&input ? {
  move(fw1)
} ? {
  upto(':')
} ? {
  count += 1
}
```

Do you see anything peculiar about this? The last pipe component doesn’t do any string scanning. There’s certainly nothing wrong with that; the right operand of string scanning can be any expression; there’s no requirement that it even consider the subject. But there’s something else here that deserves note. The pipe component `upto(':')` produces an integer, not a string. Yet the subject of string scanning must be a string. In this case, that’s not a problem, since the integer is converted to a string that serves as the (ignored) subject of the last pipe component.

This example points out that the values that pass through a pipe of scanning expressions must be strings or convertible to strings. This is a definite limitation to the use of scanning pipes (and suggests an interesting possibility for a language feature akin to string scanning but one that does not require the “subject” to be of any particular type).

The need to pass strings (or values convertible to strings) through scanning pipes suggests some special programming techniques.

Suppose, for example, you want to write only those fields that contain commands as well as counting them. This requires that the field be passed through if the test succeeds. Adhering to our separation of functionality into simple pipe components, this means that the component that contains `upto(':')` must pass on its subject. For the example above, this might take the form

```
every !&input ? {
  move(fw1)
} ? {
  upto(':') & tab(0)
} ? {
  write(&subject)
} ? {
  count += 1
}
```

There is some logic to a pipe component passing on its subject if it has no other value to pass on. Note that `write(&subject)` does the same thing, although the next pipe component doesn't use it.

You might think that

```
upto(":") & &subject
```

would work just as well as

```
upto(':') & tab(0)
```

but it doesn't and the result may be mysterious. The cause lies in the restoration of `&subject` when scanning is complete [3]. A good rule is never to produce `&subject` (or `&pos`) as the result of string scanning.

If you decide to use piped scanning, you may want to develop some guidelines for writing pipe components so that they will be "plug-compatible" over a variety of applications. One guideline might be the one suggested above: If a pipe component does not produce any useful value, but only serves as a "conditional", it should pass on its input.

Two other matters related to piped scanning deserve mention. One is that pipe components can be generators. In fact, the preceding examples contain generators, although the description of the data implies that generation will not occur — that the first field will contain only one colon. If this rule is violated in

```
every !&input ? {
  move(fw1)
} ? {
  upto(':')
} ? {
  count += 1
}
```

the expression `upto(':')` produces a value for every colon in its subject. As long as it is driven from above, it will generate these values and they will be counted, since `upto(':')` is the last suspended generator. It will be resumed before, for example, `!&input` at the head of the pipe.

Although generation is not expected for the data in the example above, it might occur unexpectedly. Limitation can be used to prevent this:

```
every !&input ? {
  move(fw1)
} ? {
  upto(':') \ 1
} ? {
```

```
count += 1
}
```

On the other hand, such generation can be useful. For example,

```
count := 0
every !&input ? {
  upto(':')
} ? {
  count += 1
}
```

counts the number of colons in the input file.

The other matter that deserves mention is that pipes also can be driven by suspend and hence easily can be incorporated into procedures. An example is:

```
procedure commands()
  suspend !&input ? {
    move(fw1)
  } ? {
    tab(upto(':'))
  }
end
```

which generates the commands from the first fields of lines of input as described above.

Procedures also can be useful as pipe components. For example, suppose you want a pipe component that generates the words from its subject. The typical method of finding words in a subject is to use a loop, as in

```
while tab(upto(&letters)) do {
  word := tab(many(&letters))
  ... # process word
}
```

The trouble is that this loop can't be used as a pipe component to generate words for the next pipe component — there's no way to export the words out of the loop. This can be done with a procedure, however:

```
procedure words()
  while tab(upto(&letters)) do
    suspend tab(many(&letters)) \ 1
end
```

Here, `words()` operates on the current subject and limitation is used to avoid unwanted backtracking [4].



```

Such a procedure can be used in a pipe, as in
every !&input ? {
  words()
} ? {
  # process word
  ...
}

```

## Conclusions

We won't blame you if you think piped scanning is a bit rarefied, if not contrived. But before you dismiss it, think about the value of having "plug-compatible" pipe components and being able to connect them together without all the worries of interactions and nesting that come from other organizations of string scanning.

Give them a try. Like any other new approach to programming, it takes a bit of time to get used to piped scanning. But once you get past that, you may benefit from "thinking pipes".

## References

1. "String Scanning", *Icon Analyst* 3, pp. 5-7.
2. "Scanning Lines of a File", *Icon Analyst* 19, pp. 10-12.
3. "Modeling String Scanning", *Icon Analyst* 6, pp. 1-2.
4. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, pp 88-89.

---

## Solutions to Exercises

### 1. revlist(lst)

One method that comes to mind for reversing a list is to create any empty list and push onto it elements generated from lst:

```

procedure revlist(lst)
  newlist := []
  every push(newlist, !lst)
  return newlist
end

```

A better method is to swap elements from opposite ends of a copy of lst:

```

procedure revlist(lst)
  newlist := copy(lst)
  every i := 1 to *newlist / 2 do
    newlist[i] := newlist[-i]
  return newlist
end

```

This method is about three times faster than the first and allocates only about one half the amount of storage.

### 2. keylist(tbl)

This one is quite simple. It's only necessary to recall that key(tbl), not !tbl, generates the keys in tbl:

```

procedure keylist(tbl)
  lst := []
  every put(lst, key(tbl))
  return sort(lst)
end

```

### 3. valset(tbl)

This one's just as simple:

```

procedure valset(tbl)
  set1 := set()
  every insert(set1, !tbl)
  return set1
end

```

### 4. seteq(set1, set2)

One way to tell if two sets have the same members is to check that the two sets are the same size and the same size as their intersection:

```

procedure seteq(set1, set2)
  if *set1 = *set2 = *(set1 ** set2) then
    return set2 else fail
end

```

You could also use the union:

```

if *set1 = *set2 = *(set1 ++ set2) then
  return set2 else fail

```

but this has the disadvantage that if the sets are not the same, the union is larger and takes longer to construct than the intersection.

Note that if seteq() succeeds, it returns set2.

That was not specified in the problem, but it fits nicely with the fact that all comparison operations in Icon return their left operands if they succeed.

You might consider checking whether the two sets are identical before doing anything else:

```
if set1 === set2 then return set2
```

Of course, it's probably unlikely that the sets are identical as opposed to being different but having the same members. Whether such a test is worthwhile is problematical.

Although the method used above is straightforward, it has the disadvantage of constructing a set, which can be relatively time-consuming operation that allocates a significant amount of storage, albeit collectable. Here's a method that does not construct a new set or allocate any storage:

```
procedure seteq(set1, set2)
  if *set1 ~= *set2 then fail
  every x := !set1 do
    if not member(set2, x) then fail
  return set2
end
```

If the two sets are the same size, but there is a member of set1 that is not in set2, the procedure fails. Otherwise it succeeds.

#### 5. setlt(set1, set2)

The second method above can be easily adapted to determine if set1 is a proper subset of set2:

```
procedure setlt(set1, set2)
  if *set1 >= *set2 then fail
  every x := !set1 do
    if not member(set2, x) then fail
  return set2
end
```

We'll leave it as a further exercise to do this with sizes and set operations.

#### 6. tbleq(tbl1, tbl2)

Determining whether two tables are equivalent is a bit more complicated than determining if two sets are equivalent. Not only are there both keys and values to worry about, but to be equivalent, two tables should have the same default value. We deliberately left that out of the problem statement to see if you'd think of it.

Determining the default value for a table is an interesting little puzzle. Of course, if you subscript a table with a value that's not a key (that is, for which a value has not been assigned), you get the default value. The problem is how to find such a value. One solution is to use a newly created value, such as a list, so that in:

```
x := tbl[[]]
```

x is assigned the default value for tbl. In tbleq(), a static variable can be used to avoid creating a new list every time the procedure is called:

```
procedure tbleq(tbl1, tbl2)
  static prod
  initial prod := []
  if *tbl1 ~= *tbl2 then fail
  if tbl1[prod] ~=== tbl2[prod] then fail
  else every x := key(tbl1) do
    if not(member(tbl2, x) |
      (tbl2[x] ~=== tbl1[x])) then fail
  return tbl2
end
```

Again, a test for identical tables could be added at the beginning:

```
if tbl1 === tbl2 then return tbl2
```

The loop that tests for equivalent values deserves examination. It might appear that

```
else every x := key(tbl1) do
  if tbl2[x] ~=== tbl1[x] then fail
```

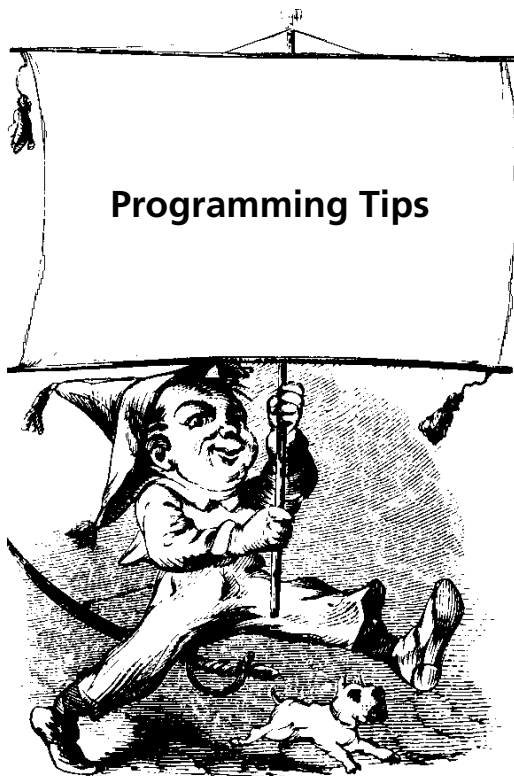
would do. After all, the tables are the same size and if one has a key the other doesn't, its value would be compared against the default value.

That's true, but there's no reason why the value for a key that's actually in the table can't be the default value. Consider

```
tbl1 := table(0)
tbl1["a"] := 0
tbl2 := table(0)
tbl2["b"] := 0
```

Here both tables are the same size and their single keys have the same value; the value of tbl1["b"] is 0, as is tbl2["a"]. So the membership test is necessary.

Just in case you didn't notice, value comparison is needed to determine if two values are the same; values can be of any type.



## Packaged Calls

The way you organize things when you program often makes all the difference in the results. We're not thinking of dividing a program into appropriate modules or designing good procedural abstractions, although both are important. Instead, we're thinking about conceptual formulation. This article describes an example of that.

In most programs, function calls and their arguments are written explicitly, as in

```
find(word[10], text)
```

Some functions, however, take an arbitrary number of arguments. The number may not be known when the program is written but instead may not be determined until the program runs. For example, in X-Icon the function `XFillPolygon()` takes an arbitrary number of arguments — the x,y coordinates of the vertices of the polygon. The number of vertices, and hence the number of arguments, may depend on run-time computations.

Icon provides a form of call in which the arguments are contained in an Icon list as opposed to being written out explicitly in the program. For example, the coordinates for the vertices of a polygon might be obtained from a file, as in

```
vertices := []
while xy := read(poly1) do
  xy ? {
    put(vertices, tab(upto(', '))) # x
    move(1)
    put(vertices, tab(0)) # y
  }
```

Then

```
XFillPolygon ! vertices
```

draws the filled polygon. Similarly,

```
XDrawCurve ! vertices
```

draws a smooth curve through the vertices.

Of course this form of call works for declared procedures as well as for built-in functions and for procedures and functions that have a fixed number of arguments.

We've mentioned this feature of Icon before. But we want to go a bit further here. Suppose you have a procedure that needs to call other procedures but the specific procedures called, as well as their arguments, are not known when the program is written.

We'll continue with our example from graphics because its easy to motivate, but the problem occurs in other contexts as well. Suppose you have a procedure `layout()` that draws a geometrical figure and perhaps performs some other computations. The figures it draws may vary — a polygon one time, a curve another, and so on. The procedure might have this form:

```
procedure layout(draw, args)
  ...
  draw ! args
  ...
end
```

and called as

```
layout(XFillPolygon, vertices)
```

or as

### 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.

```
layout(XDrawCurve, vertices)
```

Providing the arguments XFillPolygon and vertices for the formal parameters draw and args is equivalent to

```
draw := XFillPolygon
args := vertices
```

and

```
draw ! args
```

is equivalent to

```
XFillPolygon ! args
```

Using this approach, the drawing procedures and their arguments can be specified during program execution and the procedure layout() can be used in very general ways.

But there's a higher level of abstraction that's useful on occasion. The idea is to encapsulate the call — both the function and its argument list — in an object. A record provides the natural way of doing this:

```
record call(fnc, args)
```

so that

```
call(XFillPolygon, vertices)
```

creates an object for the call that can be assigned to an identifier, passed to a procedure, stored in a structure, and so forth. Imagine, for example, a table of calls corresponding to a library of drawings.

The conceptual value of this approach lies in being able to deal with a call as a single value rather than as a pair of values that are related only by context.

Using this approach, layout() can be recast as procedure layout(drawing)

```
...
drawing.fnc ! drawing.args
```

## Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

RBBS: (602) 621-2283

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

```
...
end
```

and used as

```
polygon := call(XFillPolygon, vertices)
...
layout(polygon)
```

Since calls may be used in a similar fashion in many places, it's useful to have a procedure to invoke them:

```
procedure invoke(call)
suspend call.fnc ! call.args
end
```

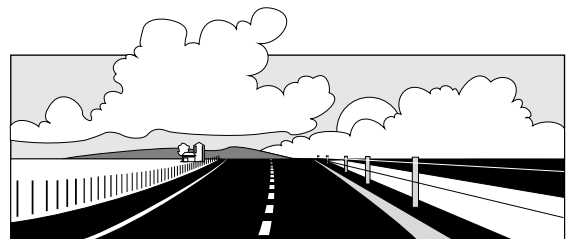
Using this procedure, layout() can be recast as

```
procedure layout(drawing)
...
invoke(drawing)
...
end
```

Whether or not treating calls as objects is worth the extra effort associated with creating and using records depends on the circumstances. This approach can be very helpful when what you're doing is naturally thought of in terms of calls as values. This approach may even suggest program features that take advantage of such run-time flexibility.

But you'll know you're heading for deep water when you find yourself using calls as arguments to other calls:

```
drawing := call(drawfig, [call(form1, [ ... ]), ... ])
```



## What's Coming Up

In the next issue of the *Analyst*, we'll have an articles on graphic contexts in X-Icon, procedures that have memory, and on returning multiple values from a procedure.