The Icon Analyst

In-Depth Coverage of the Icon Programming Language

October 1992
Number 14

In this issue …

Reader Feedback

We included a questionnaire in Issue 12 of the Analyst to give our subscribers an opportunity to tell us how they feel about the Analyst and to give us more information about their interests. A summary of the responses follows.

All persons responding said they were generally satisfied with the Analyst and several said they were very satisfied. All persons also said the Analyst was useful, at least some of the time. Its usefulness seems to be primarily related to programming in Icon.

Most persons responding said the technical level of the Analyst was about right. Only one person said it was too low. One person commented: "It's often somewhat over my head, but that's exactly how I would want it." Another person commented: "All items are fascinating, high and low. Sometimes I think everything is in Greek, but don't let that stop you."

The best-liked articles were those on program anatomies and case studies, with the programming tips a close second. There was no consensus on specific articles.

Most persons responding did not identify any kind of article they liked least, although a few said articles relating to implementation matters were of less interest than those on programming.

Although programming tips were frequently cited as being of interest, the specific responses about this feature of the Analyst were more varied and ranged from "very useful" to "potentially useful".

Our subscribers come from a wide range of professions: programmers, engineers, systems analysts, statisticians, literary analysts, researchers in the Humanities, computer scientists, accountants, physicists, … . The uses of Icon cited were even more varied than the professions and we'll not attempt to characterize them here.

The platforms on which subscribers use Icon follow the pattern of those reported in Issue 39 of the *Icon Newsletter*: predominantly MS-DOS, with strong representation for the Macintosh and UNIX workstations.

Among the general comments we received, there were several requests for more information about work in progress related to Icon.

Our general assessment of the responses to the questionnaire is that we are doing about as well as we could hope to be in providing what our readers want. We will try to give more coverage to programming and work in progress and less to implementation. Otherwise we'll generally continue to do what we've been doing.

By the way, we're running out of material for programming tips. If you have some suggestions, we'll be happy to consider them. And, of course, we'll give credit for any contributed material that we use.

We appreciate the time and effort that our subscribers took to respond to our questions. It isn't necessary to wait for another questionnaire to tell us what you think. Nor is it too late to return the questionnaire from Issue 12 of the Analyst. You can use any convenient means of communication: postal mail, electronic mail, fax, or telephone.

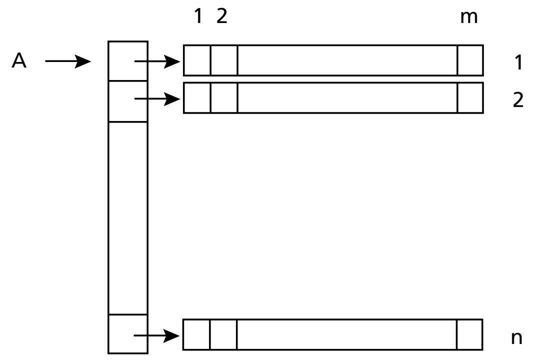
Arrays

One of Icon's weaknesses is the lack of an array data type. The closest thing Icon has is its list data type, which is one dimensional with a lower bound of 1. In this article, we'll describe how you can build arrays using lists.

The easiest way to create a two-dimensional array in Icon is to use a list of lists, as in

```
A := list(n)
every !A := list(m)
```

which can be thought of as an array with n rows and m columns. Such an array can be visualized as shown below. Note that there are m+1 lists and n*(m+1) elements in all.



Referring to an element of such an array is easy:

```
A[i][j] := x
```

This expression assigns x to the element in row i and column j. Similarly, write(A[i][j]) writes the value of the element in row i and column j.

The choice of what constitutes a row and column is arbitrary. The choice here corresponds to the conventional notation for subscripting an array: A[i, j]. Unfortunately, Icon doesn't support multiple subscripting like this, or didn't when we started working on this subject. See the note at the end of this article. Anyway, A[i][j] isn't *that* bad.

You can go a step further and create a three-dimensional array like this:

```
A := list(n)
every !A := list(m)
every !!A := list(k)
```

Generalizing this to create n-dimensional arrays is a bit more challenging. As usual, the key is the use of recursion. Here's a procedure that makes good use of variable-length argument lists and list invocation — much in the style of the recursive generators discussed in the last issue of the Analyst [1]:

```
procedure create_array(ubs[])
  local A

  A := list(get(ubs)) | stop("*** bad spec.")

  if *ubs > 0 then
    every !A := create_array ! ubs

  return A

end
```

For example,

```
A := create_array(5, 10, 20, 5)
```

creates a $5 \times 10 \times 20 \times 5$, four-dimensional array, and A[i][j][k][n] references an element of this array.

In the examples so far, the initial values of all array elements are null. This isn't convenient for, say, an array of integers, where the initial values might be zero and augmented by an expression such as

```
A[i][j][k][n] +:= 10
```

One approach to dealing with this problem is to have the last argument of create_array() be the initial value. For example,

```
A := create_array(5, 10, 20, 5, 0)
```

would create a four-dimensional array all of whose elements are 0 initially. A version of create_array() to do this is:

```
procedure create_array(args[])
  local ub, A

  if *args < 2 then stop("*** bad spec.")

  ub := get(args)

  if *args = 1 then
    return list(ub, args[1])
  else {
    A := list(ub)
    every !A := create_array ! args
    }

  return A

end
```

Using the last argument for the initial value is somewhat dangerous; if it is forgotten in a call of create_array(), the last upper bound is taken to be the initial value. An alternative is to specify the array upper bounds in a list when create_array() is called and to have the initial value be the second argument, as in

```
create_array([5, 10, 20, 5], 0)
```

In this case, create_array() is not declared with a variable-length argument list:

```
procedure create_array(ubs, value)
  local A

  A := list(get(ubs), value) |
    stop("*** bad spec.")

  if *ubs > 0 then
    every !A :=
      create_array(copy(ubs), value)

  return A

end
```

The list ubs must be copied because of Icon's pointer semantics, as discussed in Reference 2. Note that all the lists are given the initial value. Except for the last list, which actually holds all the values, this value is overwritten by the lists for the subsequent dimension. Doing it this way slightly simplifies the procedure and imposes no extra computational cost. Note also that if the second argument is omitted, all elements are null initially. This fits nicely with Icon's general handling of omitted trailing arguments.

So far, so good. But there's a thornier issue — the arrays constructed so far have lower bounds of 1 for all dimensions.

A general approach is clear — provide an offset for subscripting. However, it's hardly acceptable to require that all references to such an array specify the offsets, as in

```
A[i + i_off][j + j_off][k + k_off][n + n_off]
```

In fact, a procedure referencing such an array might not even know the offsets. Instead, the lower bounds need to be kept as part of the structure for the array and the necessary offset arithmetic needs to be handled automatically.

The first consideration suggests implementing arrays using a record such as

```
record array(structure, lbs)
```

where the first field contains the list structure that represents the array as described above and the second field contains a list of the lower bounds. Such a representation has the additional advantage of providing a type that allows arrays to be distinguished from other structures.

The procedure create_array() is only slightly complicated by these extensions, but it's necessary to decide how it will be called to provide the additional information needed. We'll use a form in which the first argument is a list of lower bounds, the second argument is a list of upper bounds, and the third argument is the initial value. For example, a three-dimensional array with lower bounds 0, –10 and 1 with corresponding upper bounds 5, 10, and 6 and with initial value 0, would be created as follows:

```
A := create_array([0, –10, 1], [5, 10, 6], 0)
```

Procedures to create such arrays are:

```
procedure create_array(lbs, ubs, value)
  local lengths, i

  if (*lbs ~= *ubs) | (*lbs = 0) then
    stop("*** bad spec.")

  lengths :=list(*lbs)

  every i := 1 to *lbs do
    lengths[i] := ubs[i] – lbs[i] + 1

  return array(create_struct(lengths, value),
    lbs)

end

procedure create_struct(lengths, value)
  local A

  lengths := copy(lengths)

  A := list(get(lengths), value)

  if *lengths > 0 then
    every !A := create_struct(lengths, value)

  return A

end
```

The more difficult problem is referencing an array element. It's no longer possible to just subscript an array. Instead, a procedure is needed to take care of accessing the record structure and to handle the offsets:

```
procedure ref_array(A, subscrs[])
  local offset, i

  if *A.lbs ~= *subscrs then
    stop("*** bad spec.")

  lbs := A.lbs
  A1 := A.structure

  every i := 1 to *subscrs – 1 do
    A1 := A1[subscrs[i] – lbs[i] + 1] | fail

  return A1[subscrs[–1] – lbs[–1] + 1]

end
```

Note that an out-of-bounds subscript causes failure.

Now an element of the array A is referenced as

```
ref_array(A, i, j, k)
```

For example,

```
ref_array(A, 3, 5, 2) +:= 2
```

increments the 3-5-2 element by 2. It is possible to assign to the result returned by ref_array(), since the result is a variable. This capability of Icon is not used often, but when it's needed, it's very handy. If it weren't possible, it would be necessary to pass the value as an argument of another procedure, and operations like augmented assignment would be much messier:

```
assgn_array(A, 3, 5, 2, ref_array(A, 3, 5, 2) + 2)
```

Comments

For the common case of two- and three-dimensional arrays in which all lower bounds are 1, simple methods suffice, although it probably is worthwhile to encapsulate the code in procedures. The more elaborate techniques for handling the general *n*-dimensional case with arbitrary lower bounds described at the end of this article are a bit more awkward to use. The procedures given above will be included in the next update to the Icon program library.

But what about really large arrays? It clearly is impractical to build even a two-dimensional 10,000 by 10,000 array using the techniques described in this article. But if only a small percentage of the 100,000,000 possible elements is ever referenced, it's quite feasible to have such an array. In a future issue of the Analyst, we'll have an article on how to implement such sparse arrays.

Now, about array subscripting. When writing this article, we mentioned in our local electronic mail how nice it would be if Icon allowed multiple subscripts for lists of lists, as in A[i, j]. The next morning we woke up to find Clint Jeffery had popped the feature into Icon's grammar. It's in Version 8.7.

References

1. "Programming Tips", The Icon Analyst 13, pp. 10-12.

2. "Pointer Semantics", The Icon Analyst 6, pp. 2-8.

Idiomatic Programming

The late Alan Perlis, one of the designers of Algol, allegedly said "an idiom is a trick you use twice". We haven't been able to find this aphorism in print, but in trying to verify it, we contacted Chris Fraser, one of Perlis' former students, who is now at Bell Labs. He sent us a copy of an APL idiom list that Perlis co-authored [1]. This idiom list inspired the material that follows.

APL is a programming language almost designed for idiomatic programming, and the APL idiom list is truly amazing. While Icon does not provide the fertile ground for idioms that APL does, we got to thinking about Icon idioms.

All programming languages, like all natural ones, have idioms. A widely used idiom, unlike a piece of tricky coding (which may yet be a budding idiom), may serve to express a computation concisely and make the nature of that computation instantly understandable. Of course, to an unknowledgeable person, an idiom may appear obscure, kinky, or even incomprehensible. Any person learning a new natural language encounters similar situations, yet a language cannot be mastered without learning its idioms and, in fact, thinking in them.

As with natural languages, idioms in a programming language develop over time. Many times we've seen an Icon expression that seemed unnatural and contorted, only later to see it instead as an elegant idiom.

This is the first of two articles that lists some Icon idioms — ones we use and ones we've found in programs written by others. We also are including some comments about coding style in general.

The instances of idiomatic usages are numbered at the right for reference and to distinguish them from other segments of code used in the discussion.

One thing we discovered in compiling this list is that it's hard to draw a line between idioms and simply standard programming practice. We've opted to include instances of expressions that are "simply the way to do it". We also do not have a good scheme for classifying idioms — hence there is a large collection of "odds and ends" at the end of the second article. Perhaps that's the nature of the beasts.

We also should comment that we don't like all the idioms listed here. We find some to be unnecessarily opaque. This is partly a matter of taste, but in some cases it can be argued that a non-idiomatic usage is better simply because it is clearer.

Using the Values of Comparisons

If you want to do something only if a value is, say, strictly between two other values, you can write something like this:

```
if (i < j) & (j < k) then …
```

You can write this more compactly (and more clearly) as

```
if i < j < k then …                    (1)
```

This works in Icon (but not in most other programming languages) for three reasons:

1. Icon comparison operations, if successful, return a value of the type compared (not a boolean value).

2. The value returned is the value of the right operand.

3. Comparison operations associate to the left.

Consequently,

```
i < j < k
```

groups as

```
(i < j) < k
```

and i < j returns the value of j if the comparison succeeds, leading to the subsequent comparison j < k.

Incidentally, the way comparison operations work in Icon is a matter of design, not a fortuitous accident.

There are other uses for these features of comparison. For example,

```
(i < j) & i
```

returns i if i is less than j. The parentheses are unnecessary and are provided only for clarity. Of course

```
j > i
```

does the same thing, but inverting the logic of a comparison sometimes interferes with understandability. The more general

```
(i < j) & x                            (2)
```

can be viewed as an idiomatic alternative to

```
if i < j then x
```

More generally,

```
expr1 & expr2
```

is an alternative to

```
if expr1 then expr2
```

provided that *expr1* cannot produce more than one result.

This is sometimes seen in the form

```
expr & return                          (3)
```

where it does not matter if *expr* can produce more than one result. Incidentally, we personally do not like this idiom, preferring

```
if expr then return
```

on the grounds that its intent is clearer.

Alternation and conjunction often play complementary roles, as in

```
(i < j) | i                            (4)
```

which produces the maximum of i and j, while

```
(i > j) | i                            (5)
```

produces the minimum of i and j.

These clearly are idioms. If you don't know what they do, you have to puzzle them out. If you recognize them as idioms, on the other hand, their intent is clear.

There are related idioms for selecting the first of several values greater than or less than a specified value:

```
i < (j | k | m | n | …)                (6)
```

and

```
i > (j | k | m | n | …)                (7)
```

Of course, the same idea applies to different kinds of comparisons.

An example of how such idioms can be used is

```
i < (2 | 3 | 5 | 7 | …)
```

which produces the least prime greater than i.

The sequence of values can be produced by a generator, as in

```
i < primeseq()
```

The same idiom works for other kinds of generators, as in

```
i < find(s)
```

which returns the first position of s that is greater than i.

Augmented assignment can be put to good use in comparisons. For example,

```
i <:= j                                   (8)
```

is equivalent to

```
i := (i < j)
```

and sets i to j if it is less than j. Used in a loop, this is an easy way to compute the maximum of a series of values, as in

```
max := 0
every max <:= !&input                     (9)
```

An elegant idiomatic combination of value comparison and augmented assignment appears in Anthony Hewitt's one-liner to filter out successive duplicate lines from a file:

```
every write(line ~===:= !&input)    (10)
```

Value comparison is used in place of string comparison so that line can be null initially and not require special handling for the first line read in. This particularly nice touch was suggested by Bob Alexander.

Filters and Transducers

One of the really powerful features that was first introduced in UNIX is the "pipe", which allows the output of one program to be fed into the input of another. This makes it easy to, among other things, "filter" the output of a program, passing through only those values that have some desired characteristic.

Icon's concepts of generators and failure make it easy to use the concept of filters in a program. Consider, for example,

```
integer(x)
```

which succeeds if x is an integer (or convertible to one) but fails otherwise. Applied to a generator, such a function becomes a filter, as in

```
integer(!X)                               (11)
```

which generates only those elements of X that can be converted to integers.

Actually, this expression is more than a filter since it produces integers, while the elements generated may, for example, be strings that represent integers. This motivates the term *transducer* used above.

It's not hard to produce a true filter, using Idiom 2:

```
type(x := !X) == "integer" & x
```

It might be clearer to use a procedure instead:

```
procedure Integer(x)

  return type(x) == "integer" & x

end
```

and use it as

```
Integer(!X)
```

Our preference for the body of the procedure, as indicated earlier, is

```
if type(x) == "integer" then return x
```

There are many possible variations on these ideas. For example,

```
procedure Integer(x, i)

  if type(x) == "integer" then return x
  else return i

end
```

could be used to replace non-integer values by i.

And, of course, it is possible to produce more than one output value for each input value, as in

```
procedure Intersperse(x, y)

  suspend x | y

end
```

which first produces x and then y, so that

```
Intersperse(!X, 0)
```

intersperses zeroes between the elements of X.

There are so many possibilities of this kind that it's probably better to think of filters and transducers as *idiomatic forms*, instead of thinking in terms of specific filtering and transducing idioms.

It's worth noting that thinking of expressions like the ones we've given here in terms of filters and transducers implies their evaluation in iterative contexts, such as

```
every write(integer(!X))
```

Such expressions have an even more idiomatic flavor when used in the context of goal-directed evaluation instead of iteration. For example,

```
i := integer(!X)                          (12)
```

assigns to i the first value of X that can be converted to an integer, leaving i unchanged if there is no such value.

In a similar vein,

```
i := integer(!X) | 0                    (13)
```

assigns to the first value of X convertible to an integer or zero if there is no such value.

Initializing Variables and Related Matters

If you want to initialize several variables to the same value, a compact way to do it is something like

```
x := y := z := 0
```

This works as intended because assignment associates to the right and returns its left operand as a variable. (These properties of assignment are by design.) This hardly classifies as an idiom.

But if you wish to assign, say, an empty list to each of several variables, then

```
x := y := z := [ ]
```

probably doesn't do what you actually want. It *does* assign an empty list to all three variables — in fact, it assigns the *same* empty list to all three variables.

The way to understand this is to realize that the expression [ ] is evaluated only once and, of course, before any assignment is made. This can be seen by making the grouping explicit:

```
(x := (y := (z := [ ])))
```

Another way of seeing this is to create the list before the assignments, as in

```
empty := [ ]
x := y := z := empty
```

A straightforward way of assigning a different empty list to each variable is, of course,

```
x := [ ]
y := [ ]
z := [ ]
```

There is, however, a more compact, idiomatic way to accomplish the same thing:

```
every (x | y | z) := [ ]                (14)
```

Here, the expression [ ] is evaluated for each variable generated by the alternation.

This technique can be used even when the same value is assigned to each variable, as in

```
every (x | y | z) := 0
```

although the idiomatic usage here is hardly justified.

By the way, there are various other idiomatic uses of iteration in combination with alternation. For example,

```
every close(f1 | f2 | f3)               (15)
```

closes files f1, f2, and f3.

It's often necessary to assign a value, such as zero, to a variable depending on whether or not the value of the variable is null. A method sometimes seen is

```
if type(x) == "null" then x := 0
```

This is unnecessarily complex and inefficient, since Icon provides operators specifically for testing for null and nonnull values. The expression above can be cast more simply as

```
if /x then x := 0
```

But an even more compact and idiomatic way is

```
/x := 0                                 (16)
```

This works because /x returns a variable if it succeeds. Similarly,

```
\x := 0                                 (17)
```

assigns zero to x only if x is nonnull.

Icon programmers frequently confuse these two operators. But, no, we don't have a good mnemonic device for distinguishing them. It's worth learning which is which, though, since these operators are very handy.

Incidentally

```
\x := &null
```

is a silly way to assign a null value to x. It's like saying if x is nonnull, make it null. It winds up being null whether or not the test succeeds. Instead, use

```
x := &null
```

There are numerous other idiomatic uses for the operator that succeeds only if its argument is nonnull. For example,

```
close(\f)                               (18)
```

closes f only if its value is nonnull, possibly to protect against the possibility that f might not have been assigned any value. This, of course, doesn't protect against f being some non-null value that is not a file.

While we're on the topic of assignment, there is a little trick (oops, *idiom*) for rotating the values of several variables to the right. For example,

```
x :=: y :=: z                (19)
```

results in y having the former value of x, z having the former value of y, *and* x having the former value of z. And it works the same way regardless of how many variables are involved in the exchanges. We'll let you work out why this is so. (Frankly, we don't want to have to explain it.)

This idiom works just as well for list and record references, as in

```
L[1] :=: L[2] :=: L[3]
```

If you want to rotate all the values in a list, however, there's a much better way:

```
push(L, pull(L))             (20)
```

This rotates the elements on position to the right with the same amount of computation whether the list contains only one element or whether it contains thousands of elements. There's an obvious, similar method for rotating elements to the right.

While we're writing about lists, it's worth noting that putting several values on a list is a common form of initialization. Again, alternation in combination with iteration provides useful idioms, as in

```
every put(L, x | y | z)      (21)
```

Next Time

In the next issue of the Analyst, we'll have the second and concluding article on Icon idioms. We somehow feel, however, that there's really no end to this.

Reference

1. A. J. Perlis and Spencer Rugaber, *The APL Idiom List*, Research Report #81, Department of Computer Science, Yale University, 1971.

Multi-Thread Icon

Multi-thread Icon, or MT Icon for short, is a version of the Icon interpreter that allows several programs to be loaded and run under the same invocation of the interpreter. Such programs can communicate with each other.

A brainchild of Clint Jeffery, MT Icon was motivated by our research in program visualization, in which program monitors, written in Icon, need information about an Icon program that is being monitored. We'll describe the use of MT Icon for program monitoring in subsequent articles. MT Icon has many other potential uses, however.

MT Icon is not a concurrent programming language, nor does it require or support multiprocessor hardware. Although several programs can be loaded under MT Icon, only one program is active at any time.

Transfer of control between loaded programs is done using co-expressions — one program activates another and hence relinquishes control to the other program.

Understanding MT Icon requires an understanding of co-expressions, including some features that are not used frequently. You may wish to review Reference 1 if you find some of the material about co-expressions here to be unfamiliar.

An Icon program that runs under MT Icon starts like any other program. In fact, the execution of a single program under MT Icon is just like its execution under the standard Icon interpreter. A program that runs under the regular Icon interpreter runs under the MT Icon interpreter without modification.

A program running under MT Icon can, however, load another MT Icon program and start its execution by activating it as a co-expression.

In order to support the execution of several programs under the same interpreter, MT Icon has additional functions and keywords. Some standard Icon functions also have extended capabilities under MT Icon.

Threads

As used here, the term thread means the execution state of a program running under the Icon interpreter. A thread consists of a set of co-expressions that share that program state. A single thread called the root is created when the interpreter starts execution. Additional threads can be created dynamically as needed.

Threads are created, referenced, and activated solely in terms of their member co-expressions. Threads are themselves implicit.

This definition of thread is related to but different from the one commonly used in operating systems. In operating systems, threads execute concurrently with their own stack and registers in a single process address space. In MT Icon, threads execute serially with their own stack, heap, and variables in a single address space.

Loading and Activating Programs

The function load(s, L) loads the icode file named s and returns a co-expression corresponding to the invocation of main(L) in the loaded icode file. (Recall that an icode file is the Icon interpreter's equivalent of an executable file that is produced by the compilation.)

As a simple example, suppose a program named example.icn consists of

```
procedure main(args)
  every write(!args)
  return
end
```

Then, using the MT version of the Icon interpreter,

```
mticont example
```

translates example.icn and produces an icode file named example. (On some platforms, the icode file is named differently. Under MS-DOS, for example, it is named example.icx.)

Now suppose the following program is named mttest.icn:

```
procedure main()
  prog := load("example", ["Hi", "mom"])
  @prog
  write("Good–bye")
end
```

then

```
mticont mttest –x
```

translates and executes mttest.icn. What happens?

The program mttest.icn starts like any other Icon program. But it uses the MT Icon function load() to load the icode file example. The value assigned to prog as a result is a co-expression for the call main(["hi", "mom"]) for example. The result of activating prog is to call main() in example with a two-element list as its argument, as if example had been run as

```
icont example –x Hi mom
```

The output produced is

```
Hi
mom
```

At this point, main() in example returns. Since it was invoked by @prog in mttest, control returns there. mttest then writes

```
Good–bye
```

before itself terminating, which ends the execution of MT Icon.

It's worth noting that the argument to main() that is passed to a loaded program in MT Icon can contain values of any type. They are not limited to strings, as in command-line invocation.

Data Spaces

Each program loaded under MT Icon has its own state and allocated storage regions. For example, a reference to &subject in a loaded program refers to its own subject of string scanning and is not affected by — nor does it affect — subjects of string scanning in other loaded programs.

Similarly, the allocation of space for a list in one loaded program has no affect on the storage regions of other loaded programs.

Each loaded program also has its own "name space" — its own global identifiers, keywords, and so on.

You may have noticed that the list given as the second argument of load() is a value in the *loading* program. This means that the loaded program has access to a data structure in the loading program. Furthermore, since the list passed as the argument of main() can contain values of any kind, it can contain, for example, other structures in the loading program. Thus, it is possible, and in fact necessary, for loaded programs to have access to structures in other loaded programs.

This makes it possible for one program to affect the storage regions of another program, although normally that is not done. It also means that when a garbage collection occurs, MT Icon must handle references between storage regions of different programs. You may be getting the idea that the implementation of MT Icon is anything but trivial.

Communication Among Programs

The example in the preceding section illustrates a simple form of communication among programs. mttest activates example much in the fashion of a procedure call. The return from main() in example causes control to return to the point of activation in mttest.

In co-expression activation, the flow of control need not be hierarchical. Furthermore, a co-expression can activate the co-expression that activated it, &source. In addition, a value can be transmitted to a co-expression when it is activated, using

```
x @ C
```

which activates C and transmits x to it. (@C is just an abbreviation for transmission of the null value.) This mechanism is the same whether a co-expression is being “called” or when a “return to it” is being made. This is illustrated by the program concord.icn:

```
procedure main()
   prog := load("textlist")
   @prog                       # wake up!
   while read() @ prog
   text := cofail(prog)
              ⋮
```

Suppose textlist.icn is:

```
procedure main()
   L := [ ]
   while put(L, @&source)
   L @ &source
end
```

The program concord loads textlist and activates it to get it underway. textlist then activates the program that activated it, concord, in a loop, putting successive values on a list. Meanwhile, concord reads lines of input, which it transmits to textlist. When there are no more lines of input, concord activates textlist using cofail(prog), an MT Icon function that causes failure of the activating co-expression. This terminates the loop in textlist. At this point, textlist activates concord to transmit the list it built.

Of course, it’s much easier to cast this functionality in terms of a procedure in concord and not use multiple programs and co-expressions at all. The example here is intended only to illustrate the way that programs can communicate under MT Icon.

MT Icon Functions

MT Icon provides several functions that allow one loaded program to access information in another loaded program. For example,

```
globalnames(C)
```

generates the names of the global identifiers in the thread that contains C. The values of identifiers in another loaded program can be obtained by using

```
variable(s, C)
```

where the second argument is an MT Icon extension that allows the thread to be specified. Thus, the names and values of global identifiers in the program prog could be written by

```
every ident := globalnames(prog) do
  write(
    ident,
    " : ",
    image(variable(ident, prog))
    )
```

Since the value of variable(s, C) *is* a variable, it is even possible to assign to the global variables in another program as in

```
variable("write", prog) := 1
variable("writes", prog) := 1
```

which has the effect of turning off output in prog.

Needless to say, changing the values of variables in another program is dangerous and should be done only in unusual circumstances.

The function keyword(s, C) accesses the keyword named s in the thread that contains C. For example,

```
write(keyword("line", prog))
```

writes the value of &line in prog.

For keywords that are variables, keyword() returns a variable. Consequently,

```
keyword("subject", prog) := text
```

assigns text to &subject in prog.

Standard Icon functions that have been extended in MT Icon to allow for the specification of a program are:

```
display(i, f, C)
name(x, C)
proc(x, i, C)
```

Additional functions specific to MT Icon include:

```
localnames(C)
paramnames(C)
staticnames(C)
```

These functions generate the names of the local variables, parameters, and static variables of the procedure currently active in the thread that contains C.

Input and Output

The function load() has three optional file arguments in addition to the two arguments mentioned earlier:

```
load(s, L, f1, f2, f3)
```

If f1, f2, and f3 are given, they become &input, &output, and &errout for the loaded program. If an argument is omitted, it defaults to the corresponding file in the loading program, and the loaded and loading program share that file.

The Relationship Among Loaded Programs

Any program can load another (including a copy of itself). A parent-child relationship is induced by loading. The result is a *load tree*. The program originally executed under MT Icon is the root of this tree.

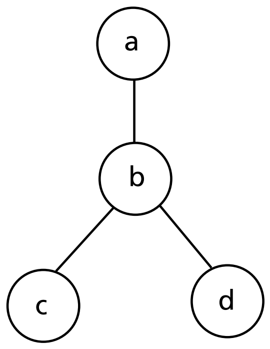
For example, if a.icn is

```
procedure main()
  @load("b")
       ⋮
```

and b.icn is

```
procedure main()
  load("c")
  load("d")
       ⋮
```

then the load tree after load("d") in b can be depicted as:



The function

```
parent(C)
```

produces the parent of C but fails if C is the root of

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

the load tree. The root can be found in any loaded program by using a procedure such as

```
procedure root()
  prog := &main
  while prog := parent(prog)
  return prog
end
```

Code Sharing

As noted earlier, there are several ways that data can be shared among loaded programs. Since procedures are first-class values in Icon, code can be shared by data sharing.

Suppose, for example, that several programs that are loaded together need to access the procedures gcd(), ximage(), and escape(). These procedures could be included in the root program (perhaps by linking ucode files). Then any loaded program that needs one of these procedures can get to it as follows:

```
global gcd
          :
library := root()
gcd := proc("gcd", , library) |
  stop("*** cannot get procedure")
          :
```

Subsequently, gcd() can be used as if it were in the loaded program. The global declaration is needed to make the procedure available throughout the program, since it is not otherwise declared in the program.

Comments

MT Icon may seem esoteric to you. And it may be difficult to imagine uses for it. As mentioned earlier, the motivation for MT Icon came from program monitoring. You can see how it might be useful to have several programs running under the same invocation of the Icon interpreter for such purposes. We'll describe how this works in the next issue of the Analyst.

We've not yet used MT Icon for many applications other than program monitoring, but there are all kinds of possibilities. Some are motivated by program organization — some programs are better thought of in terms of collections of sub-programs instead of collections of procedures. Existing programs sometimes can be used without modification by loading and "executing" them under the control of a master program. For example, a master root program can load a number of utility programs that normally run in a stand-alone fashion, with the master program providing a user interface and activating the utilities as needed.

It's also been our experience that facilities often are useful for purposes very different from those originally envisioned. Sometimes it takes a while for new uses to become apparent.

Having said all this, we also have to say that MT Icon is still somewhat experimental. We use it extensively in our program visualization research, but it's not ready for distribution yet. Given more experience with its use and more rigorous testing, we may include it in a future public release.

Acknowledgment

MT Icon was conceived, designed, and implemented by Clint Jeffery. Some of the material in this article was adapted from one of his reports [2].

References

1. *The Icon Programming Language*, second edition, Ralph E. Griswold and Madge T. Griswold, Prentice Hall, Englewood Cliffs, New Jersey, 1990, pp. 107-119.

2. *The MT Icon Interpreter*, Clinton L. Jeffery, Icon Project Document IPD169, 1992.

What's Coming Up

The next issue of the Analyst is largely a continuation of material that started in this one: more Icon idioms and applications of MT Icon, this time for program monitors that provide the foundation for program visualization.

We'll also have some tips on efficient programming.