The Icon Newsletter

No. 26 — March 1, 1988

Odds and Ends

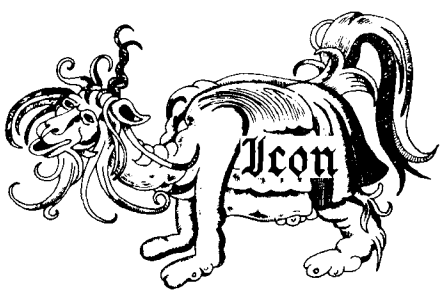
The Icon Extension Interpreter

The Icon extension interpreter (EI) described in *Newsletter* No. 25 sparked an unusual amount of interest. We would like to emphasize that EI is part of a research project at the University of Washington. The Icon Project does not have EI, nor is EI presently available for distribution. However, Bill Griswold at the University of Washington is planning to provide part of EI, the ability to call any C procedure from Icon, in a future version of Icon, which we then will be able to distribute. There is, as yet, no projected completion date for this extension to Icon.

Icon Clip-Art Contest

We've received several submissions to our clip-art contest. Those received by the deadline (January 15) appear scattered throughout this *Newsletter*. Credits are given at the end of the *Newsletter*.

We haven't selected artwork for our permanent logo yet. We'll continue to accept submissions for logos and other artwork related to Icon, and we will publish the most interesting material in subsequent *Newsletters*. We now have a scanner, so we can accept high-contrast black and white artwork in addition to material in machine-readable form.



Implementation News

Version 7 of Icon is Released

Version 7 of Icon is now available. This version contains several new features; the major ones are:

- New functions, ranging from inserting and removing tabs in strings to bit-wise operations on integers.
- Error traceback for run-time errors that shows procedure calls to the site of the error and the expression in which the error occurred.
- Procedures with a variable number of arguments.
- Correction of the handling of co-expression return points to support coroutine applications, co-expression tracing, and other new co-expression features.
- Correction of the handling of scanning environments so that the values of &subject and &pos are properly restored when a scanning expression is exited.
- Optional conversion of most run-time errors to expression failure, allowing a program to continue execution in situations that otherwise would cause termination.
- The ability to save an executable program image on UNIX 4.*n*bsd systems.
- New keywords that provide access to information on storage utilization.

Version 7 is now available for UNIX systems, VAX/VMS (VMS 4.6 or higher), MS-DOS (LMM), XENIX (LMM), and for porting to other computers. See the order form at the end of this *Newsletter*.

Version 7 implementations for the Amiga, Atari ST, the Macintosh (under MPW), and UNIX PC are in process and will be announced in future *Newsletters*.

One casualty of Version 7 is the so-called small-memory model implementation for MS-DOS, PC/IX, the PDP-11, and XENIX. Icon has been pushing the

limits of the small memory model for some time and finally exceeded the text-segment limit in a revision to Version 6. For a while, we tried to work around the problems by subsetting Icon, but it just didn't work.

We realize that persons whose personal computers have only a small amount of memory won't be able run the new version of Icon. A large majority of personal-computer users can run Version 7 of Icon, however, and many of those have clamored for the increased functionality that inevitably means a larger program. The future clearly lies in the direction of cheaper memory and access to more of it. In fact, most of the problems we see in the MS-DOS Icon user community relate to the 640 KB limitation in MS-DOS.

We will continue to distribute the small-memory-model implementation of Version 6 of Icon for MS-DOS, PC/IX, the PDP-11, and XENIX.

The Icon Newsletter

Madge T. Griswold and Ralph E. Griswold
Editors

The Icon Newsletter is published aperiodically, at no cost to subscribers. For inquiries and subscription information, contact:

Icon Project
Department of Computer Science
Gould-Simpson Building
The University of Arizona
Tucson, Arizona 85721
U. S. A.

(602) 621-2018

Electronic mail may be sent to:

icon-project@arizona.edu

or

... {allegra, ihnp4, noao}!arizona!icon-project

Update Policy

Vendors of commercial software make allowances for the cost of updates when determining the original purchase price. Our charges for software are intended only to recover costs, and we do not increase them to account for updates. However, we realize that it is unfair to expect a recent purchaser of Icon to have to buy a new version immediately. Consequently, we offer updates to Version 7 as follows: Persons who purchased Version 6 of Icon for UNIX, VAX/VMS, MS-DOS (LMM), or XENIX (LMM) after January 15, 1988 may obtain an update to Version 7, free of charge, by returning the original media (tape, cartridge, or diskettes). This offer is good until April 15, 1988. Please use the order form at the end of this *Newsletter* when requesting an update, marking it as "free".

Status of the Icon Program Library

The Icon program library is not yet ready for Version 7 of Icon. There are several reasons for this:

- We need to update the program library to use Version 7 facilities for maximum efficiency and functionality.
- We have received a large number of contributed programs. Adding them to the program library takes considerable time.
- We did not want to delay the Icon Version 7 distribution just because of the library.

Version 6 of the Icon program library, which is compatible with Version 7 of Icon, is included with the Version 7 UNIX and VMS Icon distributions. Version 6 of the Icon program library is also available separately for other implementations.

We expect that Version 7 of the Icon program library will be ready for distribution in the summer of 1988. Its availability will be announced in a forthcoming *Newsletter*.

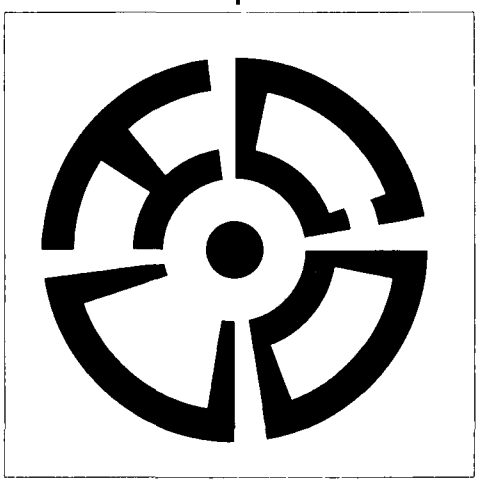
Icon for Prime Computers

Ed Feustel of Prime Computer has implemented Version 6 of Icon to run under their PRIMIX operating system. We hope to have copies available for distribution soon. If you are interested, send us a note.

ICEBOL3 in April

Once again, South Dakota State College is hosting the annual ICEBOL conference. It will be held April 21 and 22. Here's a description from their flyer:

ICEBOL3, the International Conference on Symbolic and Logical Computing, is designed for teachers, scholars, and programmers who want to meet to exchange ideas about non-numeric computing. In addition to a focus on SNOBOL, SPITBOL, and Icon, ICEBOL3 will feature introductory and technical presentations on other dangerously powerful computer languages such as Prolog and LISP, as well as on applications of BASIC, Pascal, and FORTRAN for processing strings of characters. Topics of discussion will include artificial intelligence, expert systems, desktop publishing, and a wide range of analyses of texts in English and other natural languages. Parallel tracks of concurrent sessions are planned: some for experienced computer users and others for interested novices. Both mainframe and microcomputer applications will be discussed.



For further information, contact:

Eric Johnson
ICEBOL Director
114 Beadle Hall
Dakota State College

Madison, SD 57402 U.S.A.
(605) 256-5270

eric@sdnet(bitnet)

Revision of the Icon Language Book

The Icon Programming Language (Prentice-Hall, 1983) is the only complete description of Icon. It describes Version 5, however, while the current version of the language is Version 7. We provide a technical report that supplements the book as an interim measure, but we are planning to revise the book itself to bring it up to date.

A book revision such as this takes a long time; in addition to the writing itself, time is required for production and printing. We expect that the revision will probably take at least a year to complete.

We intend to rewrite the book completely. In addition to including new features, we plan to introduce generators and string scanning earlier, extend the exercises, and substantially change the form of the reference material in the appendices.

If you have any suggestions about revising this book, please let us know.

Feedback

The discussion of expression failure in the last *Newsletter* provoked the following comments from three readers:

David Talmage:

The problem, it seems, is how to distinguish between an expression that produces no results at all, one that produces some that are useful (i.e. we want their values) but eventually stops producing any values, and one that produces values that we can't use until it runs out of values.

Suppose we use "fail" to describe an expression that produces no values at all. That seems reasonable to me.

Suppose we use "finished" to describe an expression that has produced some values we want and then produces no more. "Exhausted" would also work well in this case.

Suppose we use "lose" to describe an expression that produces no values we can use or that has yet to produce a useful value.

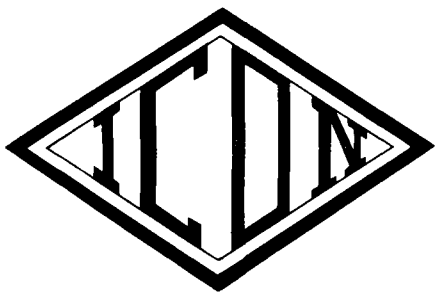
The difficulty here might be in changing the "type" of expression from "lose" to "finished". An expression could "lose" for its entire life, if you will, only to produce a value we want just before it expires. I would call that kind of expression "finished" or "exhausted".

Robert Gustafson:

Your essay on the semantics of "failure" and "fail" as used in Icon was interesting and illustrates a common dilemma: when there are a number of efforts moving forward independently (normal for research), the nomenclature does not get standardized for awhile (if ever). You might look into the realm of multiprocessor/parallel system languages for a more suitable

word. As an example, the language Occam uses the word SKIP as one way to indicate the termination of a process. In this context it means "do nothing and move on to another part of the program". In a wider context, SKIP might be construed as having the possibility of continuing after having skipped one or more times. To counter this thought, you might consider the word QUIT as an alternative.

I agree that implementation of either of these suggestions requires rewriting lots of documentation.



Davidson Corry:

The concept of "failure" as discussed in the first *Language Corner* started me thinking. When I first encountered the idea in SNOBOL2 back in the mid-60s, it seemed utterly natural (and a Godsend, for reasons I will discuss). But in trying to explain to colleagues what sets Icon apart from the mainstream of programming languages, I have had to recognize that some people don't grasp the concept immediately.

At this late date, it appears that terminology and teaching are the only tools left, since fail is inextricably a keyword (perhaps *the* key word) in the language.

Consider the following C function:

```
#define NULL 0
/* return a pointer to the position of character K in string */
/* S or NULL if K is not found in S */
char *index(s,k)
    char *s;
    char k;
    {
    for( ; *s; ++s)
        if (*s == k)
            return(s);
    return(NULL);
    }
```

The use of a NULL pointer is a typical C idiom to indicate "failure" of a search. The idiom leads to useful abbreviations of code because of two tricks designed into the language. First, a pointer to valid data is guaranteed to be nonzero (hence a zero-valued pointer is "reliably unreliable"). Second, conditional expressions are tested on zero/nonzero values, so the fragment

```
if (p = index(string,k))
  printf("%c in %s at %x",k,string,p);
else
  printf("%c not found in %s",k,string);
```

works as expected. These two tricks are specified as part of the design of the C language, and so programmers may use them comfortably — but they *are* tricks! Suppose that you need a function which returns a byte value, and all 256 possible byte values, including zero, are valid ("successful") values. The only way you can also indicate an invalid "failure" result is by using an auxiliary variable: a "flag" or "signal".

In a sense the idea of a signal is *not* incorrect. An Icon expression which succeeds gives (I am trying to avoid the Icon term "produces") a result which is an element of "value space" (that is, the result is &null, a string, a cset, a number or ...). An expression which fails "gives a result" which is *not* an element of "value space" (i.e. "none of the above"). Whether or not the computation concludes within "value space" is the *signal* result. You could say that an Icon expression potentially gives *two* results, one within "value space" and the other within "success space". "Success space" is a dimension orthogonal to "value space" and comprises only two states, "success" and "failure". A computation which reaches a conclusion must conclude either with success (and specifies a result in "value space") or with failure (and does *not* specify a result in "value space"). The only other possibility is a computation which does not conclude — a hung machine, which is in a class of tools I have not found useful.

This paradigm isn't much of a teaching aid — beside it, "failure" appears positively pellucid — but it is suggestive. "Success space" is reminiscent of a state machine, and that leads us in a useful direction.

Every Icon expression is a generator. This is obscured by the fact that expressions in traditional languages are all "degenerate" generators: they have a result sequence of length 1. We are so used to writing these degenerate expressions in other languages that we write them without thought in Icon — and wonder why the program doesn't operate in an Iconesque manner.

An expression whose result sequence consists of a single value can (and will) be viewed as complete in itself. Every "activation" (evaluation) of the expression re-starts the result sequence and exhausts it. There is no connection to the past or future. In fact, the

only way to "connect" a traditional function to its past results is to give it a "memory": static variables which preserve older states of a function activation and, in essence, import those states into the present via an explicit side-effect. The static variables are *in*formal parameters of the function.

What distinguishes a true generator from these "snapshot" expressions is that a generator produces a result *sequence*, not merely a "result". "Sequence" implies that the expression produces results over a period of time: the generator is an entity which has duration. Via the resumption mechanism, Icon provides a generator with a memory that is fully encapsulated, instead of relying on fragile and dangerous static-variable side effects. A generator is a module which modifies itself as appropriate to its experience. It evolves ... and *that* is our illuminating metaphor.

Life can be defined as the property of responding to stimulation. If an organism does not respond to stimulation, it is **dead**. "Stimulation" in the Icon context corresponds to evaluation or activation of an expression.

An Icon expression is **alive** until it **dies**. (It produces results from its result sequence until the sequence terminates. It succeeds until it fails.)

An expression may be **stillborn**. (Its result sequence is empty.)

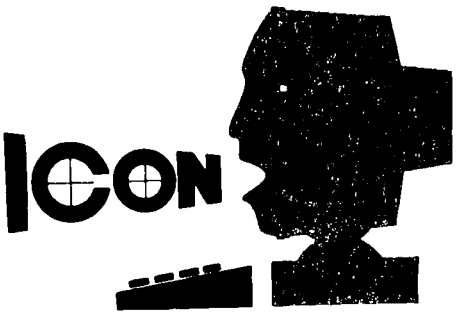
It may be **immortal**. (Its result sequence is infinite, does not terminate.)

A generator may be **killed** or (sic) **terminated**. (We might nickname the explicit limitation operator \ "Clotho".)

Death is **contagious**. (If the evaluation of an argument fails, the expression of which the argument is part also fails.)

A **dead** expression cannot be **reanimated**. (Once a result sequence has ended, further resumption does not produce results. In fact, it appears that Icon has no way to resume an expression which has failed — which makes sense.) However, it can be **reborn**. (By activating the expression again, which restarts the result sequence from scratch.)

The Icon keyword create **conceives** an expression/generator in **embryonic** form. (Creating a co-expression "primes" the expression but does not "trigger" it.) The embryo can be **cloned** (refreshing a co-expression produces a *copy* of the co-expression with local identifiers reset to their "primed"values) but the mature zygote cannot be cloned directly (there is no way to capture a co-expression in its "current" state, only in its "primed" state). And so on . . .



From Our Mail

I want to make copies of the Newsletter to give to my friends, but I notice it is copyrighted. How does this affect implementations (which I thought were in the public domain) and other Icon documents?

The *Newsletter* is copyrighted to protect our right to publish material in it at a later date. Our written permission is required to make copies of the *Newsletter*. We're generally willing to provide such permission, but we'll also be happy to mail copies of the *Newsletter* to interested persons — just send us their names and (postal) addresses. Copyright applies only to documents on which the copyright notice appears. Implementations of Icon are not copyrighted and may be duplicated freely. The same freedom applies to Icon documents that do not bear copyright notices.

I've tried to get Icon from Arizona via FTP, but it's very slow and I lose the connection before the transmission is complete. What's wrong?

If you're reaching arizona.edu via 192.12.69.1, slowness is par for the course. Try using 128.196.6.1 and see if you have any better luck. If you can't get satisfaction with 128.196.6.1, let us know.

Any news yet on Icon for the Apollo Workstation?

We have learned of at least one successful implementation. The problems previously reported seem related to the installation process rather than to Icon itself. The most recent word we have is that the implementation of Version 7 of Icon is proceeding without problems under the latest release of the Apollo operating system. We presently do not have detailed information or access to a working version of Icon for the Apollo, but we'll try to get more information for persons who are interested.

Can I get the source code for the public-domain version of make *that you distribute with MS-DOS Icon source code?*

Sure. In fact, we provide source code as well as executable binaries as part of our Version 7 MS-DOS source-code distribution. You can also get it from our BBS or by FTP to arizona.edu.

Does MS-DOS Icon compile under the new Microsoft 5.0 C compiler?

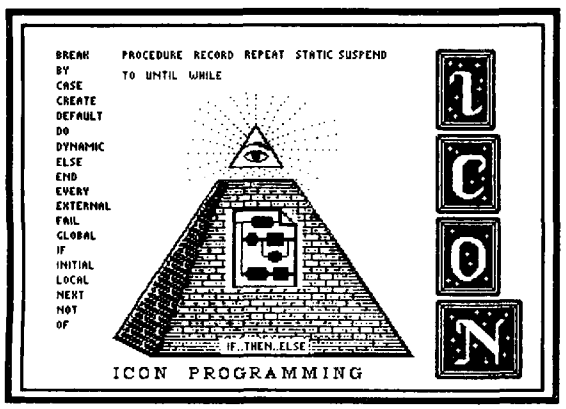
Version 7 of Icon compiles under Microsoft C 5.0. Version 6 doesn't, however. If you have Version 6 source code, we recommend that you upgrade to Version 7; it not only provides changes and workarounds needed for Microsoft C 5.0, but it also has many other corrections and improvements.

I've compiled Icon on the Atari ST, but with the Lattice 3.04 C compiler, producing smaller and faster code. Can you send me examples of assembler code that illustrate co-expressions and overflow checking so I can add these features to Atari Icon?

We have the co-expression and overflow code for Atari ST; these features will be available in Version 7 of Icon for the Atari when it's released. We also hope to have support for environment variables.

I got Icon on a data cartridge from you, but I can't read it on my HP computer. What's wrong? Help!

Data cartridges are a constant source of problems. For whatever reason, they aren't a reliable way to transfer data between different kinds of computer systems. Our cartridges are written on Sun Workstations (as raw devices) and generally can be read on other Sun Workstations without problems. They also have been successfully read on IBM RT PCs. Beyond that, cartridges should be considered a last resort in the absence of another available medium. We supply cartridges at the user's risk only. If anyone can provide more definitive information on the use of cartridges for data transfer between different systems, we will be happy to include it in a future *Newsletter.*



Does Icon run under OS/2?

Cheyenne Wills reports that Version 7 of Icon runs under OS/2 with the compatibility box. (He presently is working on a pure OS/2 implementation of Icon.)

I am interested in obtaining a source listing of Icon corresponding to the book ***The Implementation of the Icon Programming Language.***

The implementation of Icon is so large that distributing listings in printed form is impractical — even reduced to half size, they fill two good-sized binders. However, the source code easily fits on two standard diskettes. See the listing of available material at the end of this *Newsletter.* Incidentally, the current source code differs somewhat from the implementation described in the book. While we can provide source code that corresponds to the book, we recommend the current version.

In Support of Icon

Kelvin Nilsen Responds to an E-Mail Query

Someone recently enquired on behalf of a friend why one should program in Icon even though an enhanced Pascal language supporting high-level string functions might be available. Kelvin Nilsen responded as follows:

In my view, this challenge can be answered at two different levels. At the lowest level, Icon supports a large variety of high-level language features which are probably not provided by a library of high-level string processing functions. These language features include run-time typing, implicit type conversion determined at run time depending on context, high-level data types such as dynamically sized heterogeneous lists, csets, strings (which are dynamically sized), and heterogeneous associative tables. Perhaps of greatest significance are the benefits of garbage collection.

As with any high-level language, these features protect me from much of the detail of programming in lower level languages. In situations where the execution costs of my programs are not a serious concern, I almost always select Icon as my programming language of preference. By programming in Icon I avoid the possibility of having to explicitly allocate and free memory, or search for bugs in my C programs caused by type mismatches in parameter lists, or deal with core dumps that result from dereferencing dangling pointers. I also enjoy the freedom afforded me by Icon to build structures comprised of different types of objects (C or Pascal would, for example, re-

quire that each element of an array be the same type), and I need not concern myself with setting artificial limits on the sizes of buffers or checking for their overflow. All of this adds up to increased productivity from me.

At a higher level though, Icon offers even more advantages over programming in more conventional languages. Icon offers goal-directed evaluation as the standard expression evaluation mechanism. This means the same mechanism that guides backtracking in string scanning governs the evaluation of every expression in an Icon program. The concept of success and failure in the evaluation of each portion of an expression makes possible such constructs as:

```
a < b < c
```

which succeeds only if a < b and b < c, or

```
max <:= y
```

which assigns to max the value of y only if y is greater than max. Goal-directed evaluation, which builds upon the idea of success and failure, makes possible such expressions as:

```
find("ub", "Rub-a-dub-dub, Three men in a tub") = 8
```

which succeeds because the second occurrence of "ub" within the longer string appears at index position 8. In evaluating this expression, find returns 2 first, because "ub" appears at this position in find's second argument. But comparison of this returned value with 8 fails, causing find to be resumed in order to produce another index representing a different matching location. The second value produced by find is 8, which succeeds when compared with 8. This mechanism is the heart of string scanning. It provides many of the same capabilities as Prolog.

It is difficult to describe all of the benefits of a high-level language like Icon in a small amount of space. Icon is the sort of language that causes programmers to approach problems in entirely new and different ways from more conventional programming languages. My own mind has been warped to the point that the most natural solution to many problems pops into my head as a simple Icon idiom. To program the solution in any other language is uncomfortable and awkward. I know, for example, that C provides an index library function with capabilities similar to Icon's upto. I have never, however, felt comfortable using C's index function. I know, in the back of my head, that it's not the real thing. Its capabilities are more limited than those of upto, and the expression evaluation mechanism that governs its use is unable to deal with multiple matches and backtracking.

Undoubtedly, your friend can make do with Pascal and an enhanced library of string processing routines. However, s/he is probably missing out on an opportunity to learn new ways of thinking about and solving problems, ways that are not only different from his or her current thinking, but are also in some sense, better.

A Brief History of Icon — Continued

As described in the last *Newsletter*, the programming language SL5, which was designed as a successor to SNOBOL4, was the precursor to Icon. One of the goals in SL5 was to provide a wider range of control structures in pattern matching — to supplement search and backtracking control structures with more traditional control structures to make it easier to control the analysis of strings.

The idea that led to Icon was that pattern matching need not be a separate part of a programming language. Instead, pattern matching can be performed by "matching functions" like more traditional computations, provided that search and backtracking control structures are available as a general part of expression evaluation.

This idea offered a considerable simplification of some language mechanisms and a substantial reduction in vocabulary of SL5. These possibilities were so attractive that SL5 was abandoned and work began on the programming language that was to become Icon.

In retrospect, it is interesting that the focus of attention was on pattern matching, which became string scanning in Icon. It took us a long time to realize that the introduction of search and backtracking control structures as a general feature of expression evaluation had more impact on "traditional computation" than on string analysis. That is, we initially did not appreciate that the generalization of expression evaluation — which led to generators, goal-directed evaluation, and new control structures — would have such wide-spread implications and that string scanning would become an essentially trivial by-product.

The initial design for Icon resembles the present language in many respects. In addition to generators, goal-directed evaluation, matching functions, and string scanning (represented with reserved words instead of an operator symbol), there were several structure types: lists, stacks, tables, and records. Lists and tables could be opened and closed. Stacks offered last-in, first-out access to values.

Once the idea for Icon was launched, it was clear that the implementation of expression evaluation was a significant problem. The implementation of expression evaluation in familiar languages like FORTRAN, Pascal, and C is well understood and there are straightforward models to follow. This is not so with expressions that may produce many results and with the automatic generation of results to satisfy a "goal".

The first implementation of Icon, started in 1977, was written in FORTRAN, using the RATFOR preprocessor to allow better program structuring. The primary reason for using FORTRAN as an implementation language was portability. At that time, no other programming language was so widely available on so many computers. While the implementation was very portable in principle, FORTRAN isn't well suited for implementing other languages, and only a few FORTRAN compilers were robust enough to handle Icon. Versions 1 and 2 of Icon, which were implemented in this way, are still in use in some mainframe environments.

Meanwhile, the Icon language developed, mainly in the areas of control structures and data structures. Repeated alternation and limitation were added, and the original stack data type was merged with lists to produce the present version of lists that support both positional and deque access.

Next time: — A new implementation and the evolution of Icon to its present form.

Language Corner

The Null Value

At first sight, the null value may appear to be about the most useless feature in Icon. On the contrary, it can be very useful, provided you understand its role and how Icon treats it.

The null value came about because a variable has to have *some* value before one is explicitly assigned to it. Since Icon does not associate a specific type with a variable, an initial value such as zero or the empty string would be a bit arbitrary, and a "garbage" value would be unacceptable. Instead, every variable is given an initial value whose type is different from the types used in normal computation.

This has the effect of distinguishing variables that have been assigned values from those that have not. It provides a way to tell, for example, if a variable is being used for the first time in a procedure call. (Granted, a variable can be assigned the null value deliberately, but unless this is done, the distinguishing aspect of the null value holds.)

Since the use of a variable that has not been explicitly assigned a value probably is an error, the null value is treated as erroneous in most computations. This catches many errors. For example, in

```
procedure sum(f)
  local i
  while i +:= read(f)
  return i
end
```

a call of sum() terminates with an error when the first line is read from f, since i has the null value and the attempt to add to it is erroneous.

On the other hand, the null value is not erroneous in all computations. For example, write(&null) writes an empty line and is equivalent to write(""). This is a consequence of the way omitted arguments are treated and the provision of default values for the arguments of many functions.

An omitted argument is equivalent to a null-valued argument. Thus, write() is equivalent to write(&null). Furthermore, a null-valued argument to write() defaults to the empty string. This means that to write an empty line, all that is needed is write(). Behind the scenes, the omitted argument is supplied as a null value and the null value is taken to be the empty string.

Since many functions are used most frequently with specific arguments, this scheme makes it possible to use them without having to specify these arguments explicitly. And, since it's easier and more readable if omitted arguments are in trailing positions, the order of arguments in many functions is determined by defaults. For example, in trim(s,c) there is no plausible default for s, but trim usually is used to remove trailing blanks. Consequently, the default for c is a cset that contains only a blank and trim(s) produces the desired result. If the arguments were the other way around, you'd have to write trim(,s). These considerations explain the order of arguments in the lexical analysis functions. For example, in the function find(s1,s2,i,j)

the range of s2 to be examined is specified by the last two arguments. They default to 1 and 0 respectively, so that all of s2 is considered in the most common situation: find(s1,s2). For string scanning, s2 can be omitted and defaults to &subject. In this case, the default for i is &pos.

You can use the same ideas in designing procedures. Suppose, for example, the procedure sum given above has a second argument that is an initial value to which the numbers in f are to be added:

```
procedure sum(f,i)
  while i +:= read(f)
  return i
end
```

It may be that this initial value usually is zero, so that the procedure normally would be called as sum(f,0). This makes the second argument a good candidate for defaulting, so that the procedure can be called as sum(f) as shorthand for sum(f,0). If a procedure is called with fewer arguments than are declared, the omitted arguments are supplied as null values, as they are for functions. Thus, sum(f) is equivalent to sum(f,&null) and one way to write the procedure is:

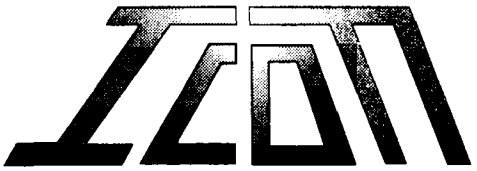
```
procedure sum(f,i)
  if type(i) == "null" then i := 0
  while i +:= read(f)
  return i
end
```

This way of testing for the null value is unnecessarily cumbersome. You might wonder if i === &null would do. It would, but Icon provides an operator specifically for testing for the null value: */expr*. Using this operator, the procedure above can be rewritten as:

```
procedure sum(f,i)
  /i := 0
  while i +:= read(f)
  return i
end
```

Since */expr* returns a variable if */expr* is a variable, assignment can be made on the spot. In fact, the operator serves to prevent the assignment by failing if the value is not null.

Sometimes the logic is reversed and what's needed is a test for a nonnull value. Suppose that count is a list produced by count := list(100). The function list has a second argument. Here the second argument is omitted and is equivalent to count := list(100,&null). This is not a default; it's just that an omitted argument is supplied as a null value. It might be clearer to give the null value explicitly in this case, but the idiom is so pervasive in Icon programs that an experienced Icon programmer probably would stop and wonder if the null value were provided explicitly.



Suppose now that some computations are performed and that, as a result, values are assigned to some elements of count but not to others. Assuming the assigned values are not null, the elements to which assignments have been made can be determined simply by looking for nonnull values. One way to do this is exemplified by

```
if not /count[i] then ...
```

This "double negative" is awkward. It's better to use Icon's operator that tests for nonnull values: *\expr*. With this operator, the test above can be written as

```
if \count[i] then ...
```

Similarly, to write only the nonnull values in count, you could use the following:

```
every write(\!count)
```

You need to be careful to avoid ambiguous failure when testing for nonnull values. It would not do to use

```
i := 0
while write(\count[i +:= 1])
```

In this case, the while loop terminates when the first null value is encountered, not just when the subscript exceeds the size of the list. (If you're not sure why every works and while doesn't, you probably have lots of company. The distinction is an important one, however, and illustrates one of the main advantages of generators used in an iterative context.)

Some Icon programmers have trouble remembering which operator tests for the null value and which operator tests for nonnull values. It may help to picture the null value as a small, flat "nothing" and to picture all other values as kinds of wheels. The expression *\expr* succeeds if the value of *\expr* can "roll out", flattening the "gate", while */expr* succeeds if its value lets the gate fall on it. This picture is a bit strained; you may prefer a different mnemonic device, but almost anything is better than having to stop in the middle of writing a program to consult a manual on a point of syntax.

A word on &null: It's common for persons talking about an Icon program to say things like "x is &null". What they really mean is that x has the null value. Granted, &null has the null value and can be used explicitly as in x := &null. The fact remains that "x is &null" is not literally true — it's just easier to say.

Incidentally, there is only one null value. So you should say "x has the null value", not "x has a null value". This isn't a very important point, except to note that if x and y are null-valued, then x === y always succeeds, which it would not if there were more than one null value.

Programming Corner

Timing Expressions

In the last Newsletter we promised more results from benchmarking Icon expressions to see how fast they execute. The timings that follow were obtained under Version 7 of Icon. In most cases, there should be little difference between Version 6 and Version 7 timings.

Of course, absolute timings vary greatly from computer to computer. The timings that follow are relative to a mythical "Icon execution cycle". Relative timings may vary somewhat depending on the computer used, the C compiler, and so forth. Although we have not collected data on differences in relative timings for Icon running on different computers, such differences usually should be small enough to ignore. (This remains to be shown, however.) For reference, the relative timings that follow were obtained from running Icon on a VAX 8650 under UNIX 4.3bsd.

To provide some values for timings, suppose the following assignments are made first:

```
i := 10
j := 20
r1 := 3.0
r2 := 2e10
```

Addition is about as simple and conventional an operation as you can imagine. Here are some timings:

```
i + j       4.0
r1 + r2     5.6
i + r1      7.3
```

You probably would expect real (floating-point) arithmetic to take a little longer than integer arithmetic, but that's not what accounts for the difference in timings for the first two expressions. Icon real numbers are stored in small blocks that have to be allocated (12 bytes each). It's the allocation time that accounts for most of the difference above. (The time required for possible garbage collection as a result of this allocation is not included in the figures above.) The reason why the addition of an integer and a real takes even longer is because of conversion of the integer to a real.

Now consider some operations on strings. Suppose

```
s1 := "abcdef"
s2 := string(&cset)
```

One simple operation is computing the size of a string:

```
*s1         2.3
*s2         2.3
```

As these figures suggest, the time it takes to determine the size of a string does not depend on how long the string is. In fact, the size of a string is computed when the string is created and is stored as part of the string value — it's right there when it's needed.

String comparison illustrates how subtle some operations are and how difficult it is to know how much time it takes to perform them. We'll use the following strings in lexical comparisons:

```
s1 := "abcdef"
s2 := "abc" || "def"
s3 := repl(s1,100)
s4 := repl(s1,100)
```

The reason for having duplicate strings becomes apparent in the timings:

```
s1 == s1    3.8
s1 == s2    5.6
s1 == s3    3.8
s3 == s4   60.6
```

What's going on here? It takes the same amount of time to compare a string to itself as it does to compare a string to a much longer string. But comparing two strings with the same value takes longer! The reason why there is a difference in the first two timings is that s1 and s2 are physically distinct, even though they have the same value. (That's a property of the way Icon is implemented, not of the language itself.) What happens in string comparison *is that two* checks are made right away. First, are the values physically the same, as in the first expression above? If so, comparison succeeds without even looking at the charac-

ters. Second, are the lengths different, as in the third case above? If so, the comparison fails immediately. If neither of these cases apply, the characters compared. The comparison is from left to right, character by character, until there is a mismatch (failure) or there are no more characters (success). Consequently, string comparison takes the longest when the two strings are physically distinct (produced in different computations), have the same length, and have a long common initial substring.

There's not much you can do about this when programming -- and you probably shouldn't try -- but this information may keep you from jumping to unwarranted conclusions and doing things that may be counterproductive.

If timings are not intuitive for simple expressions, what about something more complicated, like operations on a set? To begin with, how time consuming is it to construct a set?

```
set()        13.9
```

That's probably less than you'd expect. (The expression above uses a feature of Version 7 that allows the first argument of set to be defaulted rather than requiring an empty list, as in Version 6.) A word of caution, however — space has to be allocated for a set; while the figure above includes the time for allocation, it does not account for time that this allocation may subsequently incur in possible garbage collections.

The next obvious question is how long does it take to insert a member in a set. If we start with an empty set S, the timing is

```
insert(S,1) 15.3
```

As you might imagine, that figure doesn't mean much, since how long it takes to look up a value in a set must depend on how big the set is and what its members are. Suppose S contains the first 1,000 integers, as in

```
S := set()
every insert(S, 1 to 1000)
```

Here are some figures for looking up integers in S, as well as an integer that's not in S:

```
member(S,1)          9.2
member(S,500)        7.5
member(S,799)       12.5
member(S,1001)      14.1
```

To begin with, it should seem reasonable that it takes longer to find out that a value is not in the set — whatever technique is used for look up, one way or another, everything has to be checked, while if the value is in the set, it may be found more quickly. But why does it take longer to find 1 than 500? (The difference in timings is real, incidentally.) Can you guess anything about how sets are implemented from these figures? And, much more importantly, is it faster to look up integers than, say, strings?

Without trying to answer all these questions (rather we hope they will make you think and possibly dig deeper into the internal workings of Icon), we'll just comment that timings for a language like Icon with all its features in all possible combinations, are not really subject to reduction to a few simple formulas, guidelines, or tables of timings. We hope (someday, if there's ever time) to compile an extensive list of timings, but the result is more likely to be a curiosity than a useful tool for programmers.

Storage Allocation

Another dimension of expression evaluation relates to the storage that may be allocated. The timings above account for the time required for allocation, but they don't give an insight into how much storage is involved or the time that may be needed later on for garbage collection.

There are several factors involved here. The space allocated for an object may be transient and used only temporarily, until another value takes its place. That's true in expressions like

```
while line := read() do
  if check(line) then write(line)
```

where space is allocated for each string that is read and assigned to line, but then is replaced by the next string. Such transient allocation involves "storage throughput", in which space that is no longer needed can be reclaimed by garbage collection. On the other hand, a set or table may be used to hold many values that last from the beginning of program execution until the end, tying up storage all the time.

The interesting thing is that garbage collection spends most of its time working on storage that has to be retained; it barely notices "garbage" that it collects. For this reason, storage throughput, as exemplified by the loop above, is comparatively cheap in itself. But if there is a lot of it, it does cause garbage collections. Such collections are fast if there is a lot of garbage, but if there are a lot of "permanent" objects like sets and tables, they are paid for each time.

What all this means is that there is no simple formula for associating a timing penalty for garbage collection with storage allocation. It depends on the storage environment, and in a complicated way. It's worth noting that many programs run to completion

without ever doing a garbage collection. The allocation piper, as it were, is never paid.

Nonetheless, it may be interesting to know how much space various kinds of objects take in Icon. This is something that can be expressed in formulas and presented in tables (this is done in the Icon implementation book). However, there are some things about storage allocation that you might not expect.

For example, a string takes only as many bytes of storage as there are characters in it (unlike C, Icon's strings are not null-terminated). For example,

```
s := repl("x",100)
```

takes 100 bytes of storage. But what about the following expression?

```
s[1] := "y"
```

This expression does not actually change the former value of s (another variable might be sharing the value and must not have its value changed as the result of changing the value of s). Instead, the expression above is a shorthand notation for concatenation and assignment of a new value to s:

```
s := "y" || s[2:0]
```

Consequently, you'd expect this operation also to take 100 bytes of storage. It does take 100 bytes of string storage, but it also allocates 20 bytes for a *substring trapped variable* block that is used to keep track of the substrings involved and the variable to which the assignment is made. Something like this is necessary, since in the general case, a lot might go on between the subscripting operation and the final assignment. For example, in

```
s[1] := compute()
```

there's no telling what compute may do before an assignment is made to s.

Substring trapped variable blocks contribute to storage throughput, since they are needed only until the assignment is made, which usually is right away. Unfortunately, the present implementation of Icon is not smart enough to detect when substring trapped variables are not needed — it even allocates them when no assignment is involved, as in

```
write(s[1])
```

Nonetheless, such blocks are transient. They may cause garbage collection, but getting rid of them is fast. And the implementation could be improved so that substring trapped variable blocks would be allocated only when they are actually needed.

Clip-Art Credits

Page 1. This strange beast is taken from a 16th-century engraving by Noel Garnier. It was digitized from a book in the Dover Pictorial Archive Series and adorned to serve as Icon's temporary mascot.

Page 3 and 4. Submitted by Robert Gray, using Adobe Illustrator.

Page 5. Submitted by Vint Blackburn and Kelly Tracy of the *Mad Statter*, digitized line art.

Page 6. Submitted by Richard Colvard, using MacPaint/Superpaint/Canvas.

Page 7. Submitted by Mary Fletcher, using Postscript.

Page 9. Submitted by Benson Cardon, using Cricket Draw.

Page 10. Submitted by Vint Blackburn and Kelly Tracy of the *Mad Statter*, digitized line art.

Thanks to all who contributed! Credits at the "Icon Store" have been sent as described in the previous *Newsletter*.

Ordering Icon Material

Shipping Information: The prices listed on the order form at the end of this *Newsletter* include handling and shipping in the United States, Canada, and Mexico. Shipment to other countries is made by air mail only, for which there are additional charges as follows: $5 per diskette package, $10 per tape or cartridge package, and $10 per documentation package. UPS and express delivery are available at cost upon request.

Payment: Payment should accompany orders and be made by check or money order. Credit card orders cannot be accepted. Remittance *must* be in U.S. dollars, payable to The University of Arizona. There is a $10 service charge for a check written on a bank without a branch in the United States. Organizations that are unable to pre-pay orders may send purchase orders, but there is a $5 charge for processing such orders.

What's Available

Icon program material falls into four categories: UNIX, VMS, personal computer, and porting.

The UNIX package contains source code, the Icon program library, documentation in printed and machine-readable form, test programs, and related software — everything there is. It can be configured for most UNIX systems. The documentation includes installation instructions, an overview of the language, and operating instructions. It does not include either of the Icon books. Program material is provided on magnetic tape, cartridge, or diskettes.

The VMS package contains everything the UNIX implementation contains except UNIX configuration information and UNIX-specific software. However, the UNIX and VMS systems are configured differently, and neither will run on the other system. The VMS package also contains object code and executables, so no C compiler is required. The VMS package is distributed only on magnetic tape. *Note*: VMS Version 4.6 or higher is required to run Version 7 of Icon.

Icon for personal computers is distributed on diskettes. Because of the limited space that is available on diskettes, in most cases there are separate packages for the different components such as executable files and source code. Each package contains printed documentation that is needed for installation and use. *Note*: Icon for MS-DOS requires 512KB of RAM.

Icon for porting is distributed on MS-DOS format diskettes. There are two versions, one with a flat file system and one with a hierarchical file system. Both versions are available in either plain ASCII format or compressed ARC format.

There are two documentation packages that contain more than is provided with the program packages: one for the language itself and one for the implementation. These documentation packages contain the language and implementation books, respectively, together with supplementary material.

When ordering, use the codes given at the beginning of the descriptions that follow.

Program Material

Note: All the distributions listed below are for Version 7 of Icon. Earlier Version 6 implementations that are not supported for Version 7 are still available. If you wish to order a Version 6 implementation, ask for a Version 6 order form, which is free.

UNIX Icon:

UT-T: Tape, *tar* format (specify 1600 or 6250 bpi). $25.

UT-C: Tape, *cpio* format (specify 1600 or 6250 bpi). $25.

UC-T Cartridge, *tar* format, (DC 300 XL/P, raw mode only). $40.

UC-C: Cartridge, *cpio* format, (DC 300 XL/P, raw mode only). $40.

UD-M: *cpio* files: five MS-DOS formatted 2S/DD 5.25" diskettes. $40.

UD-X *tar* files: seven XENIX formatted 2S/DD 5.25" diskettes. $50.

VMS Icon:

VT: Tape, (specify 1600 or 6250 bpi). $25.

Icon for Personal Computers:

DE: MS-DOS (LMM) Icon executables: two 2S/DD 5.25" diskettes. $20.

DS: MS-DOS Icon source: two 2S/DD 5.25" diskettes. $25.

XE: XENIX (LMM) Icon executables: one 2S/DD 5.25" diskette. $15.

Icon for Porting:

PF-A: Flat file system, ASCII format: four 2S/DD 5.25" diskettes. $35.

PF-K: Flat file system, ARC format: two 2S/DD 5.25" diskettes, $25.

PH-A: Hierarchical file system, ASCII format: four 2S/DD 5.25" diskettes. $35.

PH-K: Hierarchical file system, ARC format: two 2S/DD 5.25" diskettes. $25.

Documentation

LD: Language documentation package. $29.

ID: Implementation documentation package. $40.

NL: Back issues of the *Newsletter*. $.50 each for single issues (specify numbers). $6.00 for a complete set (#1-25) There is no charge for overseas shipment of single back issues, but there is a $5.00 shipping charge for the complete set.

Order Form

Icon Project • Department of Computer Science • Gould-Simpson Building • The University of Arizona • Tucson, AZ 85721 USA
Ordering information: (602) 621-2018

name _____

address _____

_____

city _____ state _____ zipcode _____

(country) _____ telephone _____

☐ check if this is a new address

qty.	code	description	price	total

subtotal	
sales tax (Arizona residents*)	
extra shipping charges	
purchase-order processing	
other charges	
total	

Make checks payable to The University of Arizona

*The sales tax for residents of the city of Tucson is 7%. It is 5% for all other residents of Arizona.