

The Icon Newsletter

No. 32 – January 15, 1990



Address Change

Our Internet electronic mail address now has an additional qualification, cs, to distinguish our department from other organizations at The University of Arizona. For example, to send electronic mail to the Icon Project via Internet, you now should use

icon-project@cs.arizona.edu

The old address (without the cs) will continue to work for a while, but to be safe, you should use the new one. The same change applies to other addresses in our organization. For example, use cs.arizona.edu for FTP and address the Icon newsgroup as

icon-group@cs.arizona.edu

Uucp addresses are not affected by this change.

From Our Mail

I just received the C source code for Icon and I want to compile it on my MS-DOS system. Unfortunately, my C compiler, Let's C from Mark Williams, is not one of those supported for Icon. I've read your documentation and would prefer not to do all the work needed for a new C compiler. But I notice benchmark figures for Icon under Let's C in your recent newsletter. Can you put me in touch with the person who did the Let's C work?



We did the Let's C benchmarks here at the Icon Project. There is considerable work needed to get this implementation fully functional and into our distribution system. Recent additions to Icon for MS-DOS, how-

ever, are available well ahead of the regular distribution via our source update subscription service. See the order form at the end of this *Newsletter*. Support for Let's C was included in the last update.

I have an Icon program that uses the system() function on an MS-DOS system. All of a sudden, when I added some new procedures to the program, the system() function disappeared. What's wrong?

By "disappeared" we assume you mean "stopped working". This probably is the result of insufficient memory. As your program gets larger, it takes up more memory. The system() function executes command.com. If there is not enough memory for this, it fails silently. There is really nothing Icon can do about this — it's an operating-system and environment problem. You might try removing unnecessary drivers and resident programs that occupy memory. Even this may not work; the MS-DOS memory limit of 640KB is simply not enough for many large applications.

Sometimes Icon hangs my XT. Do you know why this happens? Is there anything I can do about it?

You may be getting stack overflow, which can wipe out important data and cause your system to malfunction in a variety of ways. On some implementations of Icon, the size of the stack can be specified on the command line when you run Icon. Check the user's manual for the version of Icon you are using to see how to do this. There is a more extensive discussion of this problem in an article on bugs later in this *Newsletter*.

Icon is a terrific language — powerful, rich, and most interestingly, intuitive. Modern interpreted languages, such as APL and Icon, demonstrate enormous flexibility that compiled languages cannot match. When will we have silicon that can execute this stuff?

Thanks for the compliments. As to "Icon in Silicon", we doubt you'll ever see that. In fact, the trend is away from casting software in silicon; the result is too hard to change and RISC architectures can offer comparable performance.

I was VERY disappointed when you discontinued the Icon compiler. Compilers are great for developers who want to distribute their software. I want the compiler back! I never used 5.0 or 6.0.

There was a compiler of sorts for Icon through Version 5, but only on a couple of computers running under UNIX. We didn't have the resources to maintain the compiler implementation, much less transport it to a wide range of computers. Instead, we opted for portability with the interpreter.

We realize that a compiler version of Icon would offer advantages in addition to speed. We are working on a compiler that generates C code (thus partially solving the portability problem).

Does Icon run on the NeXT machine?

Our current development version of Icon runs on the NeXT, as well as on several of the newer workstations, such as the Sun SPARCstation and the DecStation 3100. Support for these computers will be included in the next release of Icon, later this year.

I am trying to run Version 7.5 of Icon on my 512K Amiga 1000. When I attempt to compile even small programs, my machine issues a nasty "Software Error" message. Icon runs on my 512K MS-DOS machine with no problems.

You do not have enough memory on your Amiga to run Version 7.5 of Icon. AmigaDOS is larger than MS-DOS and the C compiler used for Amiga Icon generates somewhat larger code than the one for MS-DOS. You may be able to get some programs to compile by using environment variables to reduce the size of Icon's storage regions, but you'll probably just run out of memory when you try to execute programs.

I tried to download MS-DOS/386 Icon from your bulletin board, but it's not there.

That's a casualty of a misunderstanding about software licensing requirements. MS-DOS/386 Icon is built using "DOS extender" software that requires licensing. We were originally told that the DOS extender could be incorporated into public-domain software, but now we're told it cannot. Catspaw, Inc. has graciously offered to provide us with licensed copies of MS-DOS/386 Icon at a nominal price. See the order form at the end of this *Newsletter*. We cannot, however, provide the licensed product electronically.

I'd like to get a copy of the Yacc grammar for Icon to try on my PC.

The Yacc grammar for Icon presently is included only for UNIX systems. It's too large for most PCs. We've had several requests for it recently, however, and we're planning to make it available as part of all future source-code distributions.

ICEBOLA

The *Fourth International Conference on Symbolic and Logical Computing* was held at Dakota State University in Madison, South Dakota on October 4-5, 1989.

Twenty-two papers were presented, covering topics ranging from applications in the Humanities to programming language design and implementation. Attendance was good (more than 200 persons) and diverse in backgrounds and interests.

An increase in the presence of Icon was notable. Several of the papers related to Icon. Many attendees used Icon and several were expert Icon programmers.

Copies of the *ICEBOLA Proceedings* (390 pages) are available for \$35. Some copies of the earlier *ICEBOL3 Proceedings* (\$20) and the *ICEBOL86 Proceedings* (\$18) also are available. Orders should be sent to:

Conference Department
114 Beadle Hall
Dakota State University
Madison, SD 57042

The Icon Newsletter



Madge T. Griswold and Ralph E. Griswold
Editors

The Icon Newsletter is published three times a year, 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) 621-4049

FAX: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

...{uunet,allegro,noao}!arizona!icon-project

© 1989 by Madge T. Griswold and Ralph E. Griswold

All rights reserved.

Object-Oriented Icon

Editors' Note: An article by Bill Griswold on object-oriented features for Icon appeared in Newsletter 30. The following article by Clint Jeffery at The University of Arizona describes his recent work on the subject.

Idol is a preprocessor for Icon that implements a means of associating a piece of data with the procedures that manipulate it. The primary benefits to the programmer are thus organizational. The Icon programmer may view Idol as providing an augmented record type in which field accesses are made not directly on the records' fields, but rather through a set of procedures associated with the type.

Motivation

In Icon, after a program reaches a certain size it becomes difficult to understand and/or modify a data structure manipulated by one portion of the program without breaking the code somewhere else. When the structure in question is one of the built-in data types, even cross-referencing tools can be of little help. The Idol preprocessor assists the programmer in managing the larger-scale structures used in Icon programs.

Classes

Since Idol implements ideas found commonly in object-oriented programming languages, its terminology is taken from that domain. The augmented record type is called a "class". The syntax of a class is:

```
class foo(field1, field2, field3, ...)
    procedures to access class foo objects
    [code to initialize class foo objects]
end
```

In order to emphasize the difference between ordinary Icon procedures and the procedures which manipulate class objects, these procedures are called "methods" (the term is again borrowed from the object-oriented community). Nevertheless, the syntax of a method is that of a procedure:

```
method bar(param1, param2, param3, ...)
    Icon code that may access fields of a class foo object
end
```

Since execution of a class method is always associated with a given object of that class, the method has access to an implicit variable `self`, which is a record containing fields whose names are those given in the class declaration. References to the `self` variable look just like normal record references; they use the dot (`.`) operator.

Objects

Like records, instances of a class type are created with a constructor function whose name is that of the class. Instances of a class are called objects, and their fields may be initialized explicitly in the constructor in exactly the same way as for records. For example, after defining a class `foo(x, y)`, one may write:

```
procedure main()
    f := foo(1, 2)
end
```

The fields of an object need not be initialized by the class constructor. For many objects it is more logical to initialize their fields to some standard value. In this case, the class declaration may include an initially section after its methods are defined and before its end. For example, suppose one wished to implement an enhanced table type which permitted sequential access to elements in the order they were inserted into the table. This can be implemented by a combination of a list and a table, both of which would be initialized to the appropriate empty structure:

```
class taque(l, t)      # pronounced "taco"
    e.g. insert, lookup, foreach ...
initially
    self.l := []
    self.t := table()
end
```

In such a case one can create objects without including arguments to the class constructor:

```
procedure main()
    .
    mytaque := taque()
    .
end
```

Object Invocation

Once one has created an object with a class constructor, one manipulates the object by invoking methods defined by its class. Since objects are both procedures and data, object invocation is similar to both a procedure call and a record access. The dollar (\$) operator invokes one of an object's methods. It is used similarly to the dot operator used to access record fields. Using the `taque` example:

```
procedure main()
    mytaque := taque()
    mytaque$insert("greetings", "hello")
    mytaque$insert(123)
    every write(mytaque$foreach())
    if \mytaque$lookup("hello")
        then write(", world")
end
```

Note that direct access to an object's fields using the usual dot operator is not possible outside of a method of the appropriate class. Attempts to reference `mystack.l` in procedure `main()` would result in a runtime error (invalid field name). Within a class method, the implicit variable `self` allows access to the object's fields in the usual manner. The taque insert method is:

```
method insert(x, key)
  /key := x
  put(self.l, x)
  self.t[key] := x
end
```

Inheritance

In many cases, two classes of objects are very similar. In particular, many classes can be thought of simply as enhancements of some class that has already been defined. Enhancements might take the form of added fields, added methods, or both. In other cases a class is just a special case of another class. For example, if one had defined a class `fraction(numerator, denominator)`, one might want to define a class `inverses(denominator)` whose behavior was identical to that of a `fraction`, but whose numerator was always 1.

Idol supports both of these ideas with the concept of inheritance. When the definition of a class is best expressed in terms of the definition of another class or classes, we call that class a subclass of the other classes. This corresponds to the logical relation of hyponymy, or special-casing. It means an object of the subclass can be manipulated just as if it were an object of one of its defining classes. In practical terms it means that similar objects can share the code that manipulates their fields. The syntax of a subclass is

```
class foo : superclasses (fields ...)
  methods
  [optional initially section]
end
```

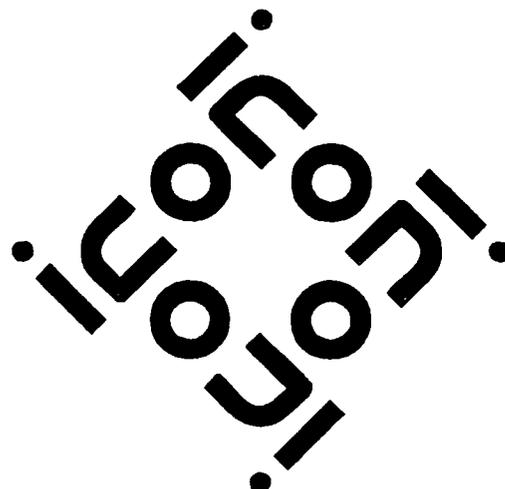
Multiple Inheritance

There are times when a new class might best be described as a combination of two or more classes. Idol classes may have more than one superclass, separated by colons in the class declaration. This is called multiple inheritance.

Invoking Superclass Operations

When a subclass defines a method of the same name as a method defined in the superclass, invocations on subclass objects always result in the subclass' version of the method. This can be overridden by explicitly including the superclass name in the invocation:

```
object$superclass.method(parameters)
```



Public Fields

Sometimes it would be really nice to access fields in an object directly, as with records. An example from the Idol program itself is the `name` field associated with methods and classes — it is a string which is intended to be read outside the object. One can always implement a method that returns (or assigns, for that matter) a field value, but this gets tedious. Idol currently supports read-only access to fields via the `public` keyword. If `public` precedes a fieldname in a class declaration, Idol automatically generates a method of the same name which dereferences and returns the field. For example, the declaration

```
class sinner(pharisee, public publican)
```

generates code equivalent to the following class method in addition to any explicitly defined methods:

```
method publican()
  return .(self.publican)
end
```

Miscellany

Idol supports some shorthand for convenient object invocation. In particular, if a class defines methods named `size`, `foreach`, and `random`, these methods can be invoked by a modified version of the usual Icon operator:

```
$*x is equivalent to x$size()
$?x is equivalent to x$random()
$!x is equivalent to x$foreach()
```

Other operators may be added to this list. If `x` is an identifier it may be used directly; if it is a more complex expression (such as a function call) it should be parenthesized, as in, `$(complex_expression())`. Parentheses are also required in the case of invoking an object produced by a complex expression:

```
(classes$lookup("theClass"))$name()
```

These requirements are artifacts of the first implementation and are subject to change.

Running Idol

Idol requires version 7.5 or higher of Icon. It runs best on UNIX systems. It has not been ported to all the various micros and operating systems on which Icon 7.5 runs. In particular, if your version of Icon does not support the `system()` function, or your machine does not have adequate memory available, Idol will not be able to invoke `icont` to complete its translation and linking.

Since Idol is untested on many systems, you may have to make small changes to the source code in order to port it to a new system.

Getting a Copy

Idol is in the public domain. It is available electronically from the Icon RBBS and by anonymous ftp from `cs.arizona.edu`. It is not available by mail from the Icon Project. Interested parties may contact the author (`cjeffery@cs.arizona.edu`):

Clinton Jeffery
Department of Computer Science
Gould-Simpson Building
University of Arizona
Tucson, AZ 85721
U.S.A.

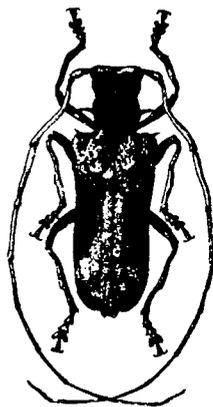
Bugs

Stack overflow can cause problems with some Icon programs. You probably think of this possibility for recursive procedure calls, but it also can happen as the result of many simultaneously suspended generators and during garbage collection.

Icon actually has two stacks: a system stack used by C, in which Icon is implemented, and an evaluation stack used for intermediate Icon results.

Procedure calls use space on the evaluation stack. There's an overflow check that usually is effective and causes program termination with an error message. If evaluation stack overflow is not detected, Icon data usually is overwritten.

Suspended generators use space on both stacks. There is no overflow check on the system stack on most implementations of Icon. It's possible on some systems, but it significantly slows program execution. If the system stack overflows, important system data



usually is overwritten. A personal computer may crash if this occurs.

Garbage collection also takes space on the system stack, especially if Icon data contains long chains of pointers from structure to structure. The likelihood of overflow during garbage collection depends on how much space is in use on the system stack at the time and hence is hard to predict.

On systems with a large amount of memory, system stack overflow rarely is a problem. While Icon for PCs is configured so that most programs run without problems, overflow should be suspected if your computer hangs while running Icon.

Co-expressions add another complication. Every co-expression has its own evaluation stack and system stack. Space for these stacks must come from available memory. Since memory is a limiting factor on many systems, the stacks for co-expressions generally are smaller than the main evaluation and system stacks. It's also more difficult to test for overflow in a co-expression. All this adds up to increased problems with stack overflow when evaluating in co-expressions.

What does all this mean? If you're using Icon on a system with a lot of memory, you probably won't have to worry about overflow, except in co-expressions. If you use co-expressions simply to control the production of results of generators, you probably won't have any trouble with them either. However, if your co-expressions perform recursive procedure calls or contain many simultaneously suspended generators, you may have problems and may need to modify your use of co-expressions.

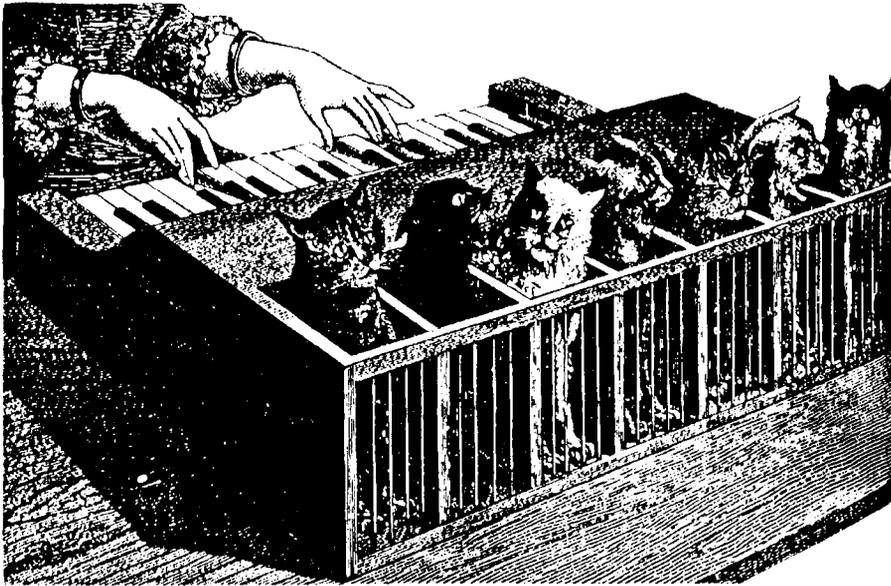
If you are using Icon on a system with a small amount of memory (MS-DOS is typical), watch out not only for the kind of co-expression usage described above, but beware of complex mutual evaluation. This is most likely to occur in string scanning, as in

```
line ? {  
    tab(upto(space)) &  
    tab(many(space)) &  
    move(1) &          # and many more conjunctions  
    :  
}
```

Bounded expressions release space used by suspended generators they bound. For example, it may be possible to rephrase the scanning expression above as:

```
if tab(upto(space)) then {  
    tab(many(space))  
    move(1)  
    :  
}
```

On some systems, the size of the system stack can be set when Icon is run. See user manuals for the default size and the method of changing it.



Language Corner

Returning from Procedures

The control structure `fail` causes a procedure call to return without producing a value — in other words, the procedure call fails just as the call of a function such as `integer(x)` can fail.

If the computation that a procedure performs is a conditional one (such as `integer(x)`), then the use of `fail` is quite natural. An example is:

```

procedure posint(x)
  if integer(x) & (x > 0) then return x
  else fail
end

```

Flowing off the end of a procedure body also causes a procedure call to fail. This feature is partly motivated by the view that unless a value is explicitly returned, a procedure call returns no value at all — and hence fails.

Some procedures perform unconditional computations but have no meaningful value to return. An example is:

```

global pcount

procedure print(s)
  initial pcount := 0

  write(s)
  pcount += 1
end

```

Since there is no explicit return, a call of `print()` fails. Often this makes no difference, as in:

```

while line := read() do
  print(line)

```

However, if this loop is written in a more compact form, such as

```

while print(read(line))

```

the failure of `print()` terminates the loop after the first line is read. While the problem is easy to see here, it's often difficult to find in more complicated situations, especially if a small change to a program that was working correctly introduces this kind of problem.

For this reason, it is good practice to place an explicit return at the end of such a procedure, as in:

```

procedure print(s)
  initial pcount := 0

  write(s)
  pcount += 1
  return
end

```

The omitted argument of `return` defaults to the null value, which usually is a safe value to return in contexts where no specific value is expected.

This problem would not exist if flowing off the end of a procedure body returned the null value instead of failing. However, this would cause another problem, and it has to do with generators. Consider the following procedure, which generates the elements of `L1` and `L2` that are the same:

```

procedure geneq(L1, L2)
  suspend !L1 == !L2
end

```

If flowing off the end of a procedure body returned the null value, such a generator would have to be written with a `fail` at the end to avoid producing a final, spurious value.

In other words, either interpretation for flowing off the end of a procedure body leads to potential problems. The interpretation is the way it is for the reason mentioned earlier — no value is synonymous with failure.

Generators

We're sometimes asked why a function like `read()` does not generate lines instead of returning one each time it is called.

The basic language design criterion in deciding if an operation should be a generator is whether or not the operation naturally has more than one result. The function `upto(c, s)` typifies such an operation.

Of course, it is not always clear whether this criterion applies. There also are pragmatic considerations. For example, the function `many(c, s)` is not a generator, although it is easy to see how it could have more than one result. In fact, at one point in the design of Icon, `many(c, s)` was made into a generator, first returning the position at the end of the longest initial substring of `s` consisting of characters in `c`, and then generating successively smaller positions. The result was a practical disaster, largely because failure of a subsequent expression produced all kinds of unwanted results in the contexts in which `many()` usually is used.

Economy suggests that an operation should not be a generator if there is an easy way to construct a generator from it. For example, to turn `read()` into a generator, just use `|read()`. Similarly, `|?X` turns the random-element operation into a generator.

Another reason for not making an operation a generator unless there is a good reason is the possibility of unexpected results from generation. For example, novice Icon programmers sometimes write loops like this one:

```
while read() ? expr do ...
```

The problem with such a formulation lies in the possibility of ambiguous failure: The loop terminates on an end-of-file but also if the scanning expression fails. Perhaps the scanning expression is not supposed to fail. Nevertheless, unexpected things happen. If `read()` were a generator, the loop above would be cast as

```
every read() ? expr do ...
```

This would (in some sense) have worse consequences if the scanning expression failed. The loop wouldn't necessarily terminate, and more (perhaps all) lines would be read, with mysterious results.

In some sense, a generator "goes off" in the presence of failure. Since it's generally easy enough to make a non-generator into a generator, it's prudent to avoid generation unless there is a good reason for it.

Graphic Credits

Graphics that first appeared in earlier *Newsletters* are credited there.

Page 5: *Petrognatha gigas* F., scanned image.

Page 6: Scanned image from *Music; A Pictorial Archive of Woodcuts and Engravings*, Dover Publications, 1980.

Back cover: Ralph Griswold, *Illustrator 88*; Icon logo filled with pattern from *Adobe Collector's Edition — Patterns and Textures*.

Faculty Positions

The Department of Computer Science at The University of Arizona invites applications for faculty positions at all ranks to begin in August, 1990. Applicants must have a doctorate in computer science or a closely related field.

Applicants for senior positions should have made substantial research contributions to the field, while applicants for junior positions should show promise of future excellence.

There are currently 14 faculty members, with plans to expand over the next few years.

Research is currently conducted in a variety of areas including algorithm design and analysis, complexity theory, databases, distributed and parallel computing, graphic and user interfaces, operating systems, programming languages, and scientific visualization.

Qualified individuals working in these areas as well as other areas, such as artificial intelligence, computer architecture, and performance analysis, are encouraged to apply.

The research program is supported by numerous grants to individual faculty as well as a second department-wide NSF infrastructure grant.

Computational facilities are diverse, including a Sequent Symmetry, dozens of Sun workstations, a VAX 8650, an Intel iPSC Hypercube, color graphics workstations, a NeXT machine, and dozens of Macintoshes. Other equipment includes numerous laser printers, a QMS color PostScript printer, and an L-300 imagesetter.

Send a complete resume and the names of at least three references to:

Udi Manber
Faculty Recruiting Committee Chairman
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Applications will be reviewed beginning January 15, 1990, but the positions will remain open until filled.

The University of Arizona is an equal opportunity/affirmative action employer.

Downloading Icon Material

Several implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

FTP: cs.arizona.edu (cd /icon)
(128.196.128.118 or 192.12.69.1)

Programming Corner



Correction

In the programming corner in the last *Newsletter*, the procedure `checkstr2()` contained the line

```
t := table(0)
```

This should have been

```
t := table()
```

since the null default value is used in the code that follows.

Cset Operations

Dave Cargo poses the following problem:

I wanted to know if the intersection of two csets is empty and wrote

```
if c1 ** c2 == " then ...
```

but then I realized I wasn't sure if == was the right operator to use.

The best operation for comparing two csets is

```
c1 === c2
```

This operation compares the bit vectors used to represent the characters in csets; it is fast and direct. The operation

```
c1 == c2
```

first converts `c1` and `c2` to strings and then compares the resulting strings. The conversion is time-consuming and unnecessary.

To determine if the intersection of two csets is empty, another formulation is

```
if *(c1 ** c2) = 0 then ...
```

Black Holes

In almost every programming language, it is possible (even easy) to get into an endless loop. While no one would intentionally write something like

```
while 1
```

in a real program, its equivalent happens to us all from time to time. Such problems are easy to understand if not always easy to find.

Icon with its generators has the potential for a some-

what subtler kind of problem — endless generation *within* the expression-evaluation mechanism. Consider, for example, the following suggested method for writing all the values in the list `L`:

```
every write(L[seq()])
```

(The function `seq()` generates an endless sequence of integers, 1, 2, 3,)

The problem here is that when the end of the list is reached, the `every` expression does not stop even though the list reference fails; `seq()` continues to generate, `L[seq()]` continues to fail, and so

Fortunately, such evaluation "black holes" rarely occur in Icon programs. If they were common, Icon would be a useless programming language. Incidentally, it's this problem that motivated the special termination condition for `|expr`, which stops if `expr` ever fails to produce a single result. Otherwise, evaluation of an expression such as `|read()` would never stop.

Records

Steve Wampler writes:

I just discovered that

```
record t(a, b, c)
procedure main(args)
  tmp := t()
  every !tmp := get(args)
  every write(!tmp)
end
```

works just fine. How nice! I just figured out a use for that.

Yes, even though records usually are accessed by their field names, their components can be subscripted, as in `tmp[1]`, and generated as you've observed. In fact the operations `X[i]`, `!X`, `?X`, and `*X` apply to all structure types with the exception of `X[i]`, which does not apply to sets, since there is no concept of order for the members of a set (although we could invent one ...).

Idiomatic Icon

Anthony Hewitt wrote this clever little program to filter out adjacent duplicate lines in a file:

```
procedure main()
  write(s := !&input)
  every write(s ~==:= !&input)
end
```

Trivia Corner

What's the shortest complete Icon program that will compile and execute without error?

Confusing hint: It's not

```
procedure main()
end
```

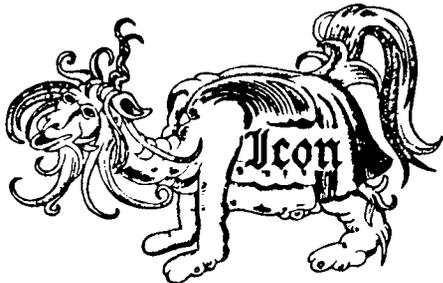
Ordering Icon Material

What's Available

There are implementations of Icon for several personal computers, as well as MVS, UNIX, VAX/VMS, and VM/CMS.

Source code for Icon is available. There is also a program library, as well as documentation both on the Icon programming language itself and on its implementation.

The current version of Icon is 7.5. All the program material here is for Version 7.5.



Icon Program Material

All program material is in the public domain except the MS-DOS/386 implementation of Icon, which is a commercial product that carries a standard software license.

Personal Computers: Executables and source code for Icon for personal computers are provided separately. Each package contains printed documentation that is needed for installation and use. *Note:* Icon for personal computers requires at least 640KB of RAM; it requires more on some systems.

MVS and VM/CMS: The MVS and VM/CMS packages contain executables, source code, and documentation in printed and machine-readable form.

UNIX: The UNIX package contains source code (but not executables), documentation in printed and machine-readable form, test programs, and related software. 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 available on magnetic tape, cartridge, or diskettes. *Note:* executables for XENIX and the UNIX PC are available separately.

VAX/VMS: The VMS package contains everything the UNIX package 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 a C compiler is not required. The VMS package is distributed only on magnetic tape.

Porting: Icon source code for porting to other computers 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.

Source Updates for MS-DOS

Updates to the Icon source code for MS-DOS are available by subscription. A subscription provides five complete updates. Updates are released about three times a year.

Icon Program Library

The Icon program library consists of Icon programs, collections of procedures, and data. Version 7 of Icon is required to run the library. The Icon program library is being issued in parts. Part 1 presently is available. *Note:* Version 7 of the Icon program library is available only on diskettes. The UNIX tape and cartridge packages and the VMS tape package presently contain an older version of the Icon program library. The Icon program library is not yet available for MVS or VM/CMS.

Documentation

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.

Shipping

Except as noted, the prices listed 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, money order, or credit card (Visa or MasterCard). Remittance *must* be in U.S. dollars, payable to The University of Arizona, and drawn on a bank with

a branch in the United States. Organizations that are unable to pre-pay orders may send purchase orders, subject to approval, but there is a \$5 charge for processing such orders.

Ordering Instructions

Legend: The following symbols are used to indicate different types of media:

-  9-track magnetic tape
-  DC 300 XL/P cartridge
-  360K (2S/DD) 5.25" diskette
-  400K (1S) 3.5" diskette
-  800K (2S) 3.5" diskette

All cartridges are written in raw mode. All 5.25" diskettes are written in MS-DOS format. 3.5" diskettes are written in the format appropriate for the system for which they are intended.

MVS and VM/CMS tapes are available only at 1600 bpi. When ordering UNIX or VMS tapes, specify 1600 or 6250 bpi (1600 bpi is the default). When ordering diskettes that are available in more than one size, specify the size (5.25" is the default).

Use the codes given at the beginning of the descriptions that follow when filling out the order form.

Program Material

Amiga:

AME:		executables	\$15
AMS:		source	\$15
AML-1:		library	\$15

Atari ST:

ATE:		executables	\$15
ATS:		source	\$20
ATL-1:		library	\$15

Macintosh/MPW:

ME:		executables	\$15
MS:		source	\$25
ML-1:		library	\$15

MS-DOS:

DE:	 (2) or 	executables	\$20
DS:	 (2) or 	source	\$25
DL-1:	 or 	library	\$15
DU:		source updates	\$50

MS-DOS/386: (not public-domain)

DE-386	 or 	executables	\$25
--------	--	-------------	------

MVS:

MT:		entire system	\$30
-----	--	---------------	------

OS/2:

OE:	 or 	executables	\$15
-----	---	-------------	------

UNIX:

UT-T:		entire system (tar)	\$30
UT-C:		entire system (cpio)	\$30
UC-T:		entire system (tar)	\$45
UC-C:		entire system (cpio)	\$45
UD-M:	 (6) or  (4)	entire system (cpio)	\$40
UL-1:	 or 	library (cpio)	\$15

UNIX - UNIX PC:

UPE:		executables	\$15
------	--	-------------	------

UNIX - XENIX:

XE:	 or 	executables	\$15
-----	---	-------------	------

UNIX - XENIX/386:

XE-386:	 or 	executables	\$15
---------	---	-------------	------

VAX/VMS:

VT:		entire system	\$30
-----	--	---------------	------

VM/CMS:

CT:		entire system	\$30
-----	--	---------------	------

Other systems (for porting):

PF-A:	 (5)	flat system (ASCII)	\$40
PF-K:	 (2)	flat system (ARC)	\$30
PH-A:	 (5)	hierarchical system (ASCII)	\$40
PH-K:	 (2)	hierarchical system (ARC)	\$30
PL-1:		library (ASCII)	\$15

Documentation

LD: Language documentation package. *The Icon Programming Language* (Prentice-Hall, 1983) and six technical reports. \$33.

ID: Implementation documentation package. *The Implementation of the Icon Programming Language* (Princeton University Press, 1986) and update. \$45.

NL: Back issues of the *Icon Newsletter*. \$.50 each for single issues (specify numbers). \$7.50 for a complete set (Nos. 1-31). There is no charge for overseas shipment of single back issues, but there is a \$5.00 shipping charge for the complete set.

