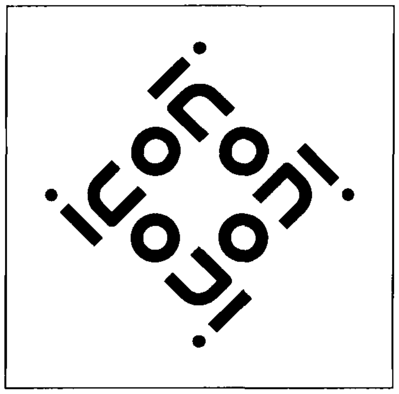
The Icon Newsletter

No. 27 — June 11, 1988

Logo!

We finally selected an official logo for Icon, and here it is:



This logo was selected from among many competitors and chosen for its simple elegance. Our thanks go to Gregg Townsend, who designed it.

We will continue to use other artwork to identify regular features of this *Newsletter*. We also welcome additional contributions of artwork (recent arrivals appear throughout this issue) and will provide credit at the "Icon Store" for all artwork we print (see *Newsletter* No. 25).

New Icon Implementations

Version 7 of Icon is now available for:

The Atari ST (executables only).

The Macintosh under MPW (executables and source).

The UNIX PC (executables; source already is available in the general UNIX distributions).

See the order form at the end of this issue.

Several other implementations are in the works: Version 7 for the Amiga, Version 7 source for the Atari ST, a 32-bit 386 protected-mode version for MS-DOS, and a Xenix/386 implementation.

Commercial Support for Icon?

We are occasionally asked how we feel about the possibility of a commercial version of Icon. These questions usually have the flavor of "would it be okay with you?" An example is the following electronic mail from Richard Goerwitz:

> A question that has been on my mind is this: Do you want some commercial outfit to pick up Icon some day? Or do you want to keep it "in house"? Do you see Icon primarily as a research tool? Or do you think of it as another, interesting and powerful, general-purpose programming tool?

In the first place, it really doesn't matter whether we would approve or disapprove of a commercial Icon venture. Icon is in the public domain. Furthermore, the source code for it is also in the public domain and anyone is free to use it as they like. Of course, we have some ideas about what we would and wouldn't like to have done with Icon, but we can't enforce them.

The main advantage of public-domain software is its low cost and unrestricted availability. The disadvantages are that, in the absence of financial resources, it can have only limited support and even its existence is known largely by word of mouth. Mark Olsen recently made the following comment in an electronic news group:

> I think that Icon is probably at the point in its life where it could be reasonably picked up by a small developer, like Catspaw or the developer of SPITBOL for the PC, and given real support at a fair price. I know academics are broke, but that does not mean that we should really try to get something for nothing.

We agree. And don't be surprised if you see such a venture before long. If it's well done and priced fairly, a commercial version of Icon could go a long way toward making the language more widely available and useful. If successful, such a venture could support enhancements — faster execution, a programming environment, and so on.

ICEBOL3

The third "ICEBOL" conference, officially called "The International Conference on Symbolic and Logical Computing", was held at Dakota State College in Madison, South Dakota on April 21-22 of this year.

The first conference focused on applications of SNOBOL4 in the Humanities, but its coverage has been broadened to include Icon, Prolog, and other programming languages as well as a wider range of applications.

This conference was well attended, with about 100 persons, including a good international representation. Interest in Icon was up. Of the twenty-five or so papers and panels, ten related to Icon in one way or another. Examples were "Elementary Cryptography Using Icon", "Analyzing Program Structures Using Icon", and "Programming with Sets in Icon".

Proceedings from the conference are in press and should be available soon. Copies are $20 and may be ordered from:

The Division of Liberal Arts
114 Beadle Hall
Dakota State College
Madison, SD 57042

A Brief History of Icon — Concluded

Until 1979, the implementation of Icon was done using facilities provided by the computer center at The University of Arizona — a DEC-10 and a CDC 6400. In 1979, the Department of Computer Science acquired the first computer of its own — a PDP-11/70 running UNIX. Compared to its current computer facilities, this PDP-11 seems puny in retrospect, but at the time it provided an environment that was much more conducive to software development than the university's computer center. The immediate effect of this new computer was to stimulate interest in a new implementation of Icon, written in C.

Along with this new implementation came new ideas for the language: functions as data objects, a generalization of expression evaluation, the fusion of lists and stacks into a single data type, and so on.

One of the initial concerns with the C implementation of Icon was whether it could be made to fit within the 128KB address-space limitation of the PDP-11/70. There are still remnants of this concern in the present implementation. At the time, it also was not clear that a C implementation would be useful beyond UNIX systems, and there was not the focus on portability as there had been in the earlier Fortran implementation.

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

Downloading Icon Material

Several of the implementations of Icon are available electronically:

BBS: (602) 621-2283

FTP: arizona.edu (/usr/ftp/icon)

(128.196.6.1 or 192.12.69.1)

With the success of the C implementation of Icon (starting with Version 3 in 1980), new features were added to Icon — repeated alternation, limitation, and co-expressions.

As Icon became more widely used, it became apparent that it was not just another interesting high-level programming language whose use would be limited to a small group of devotees; it was being used for "serious" work by an increasingly large user community. In some sense, Icon crossed a threshold and acquired a life of its own. As the user community increased in size, better documentation was needed, and the book describing Version 5 was published in 1983.

Soon folks were clamoring for implementations for new computers and operating systems. By this time, decent C compilers were available for many computers and it was feasible to adapt the C implementation of Icon to run under systems other than UNIX. One of the first of these was VAX/VMS. Many others followed, with MS-DOS being the most widely used (the implementation that proved, and still proves, the most difficult and frustrating). From 1983 to the present, much of the work on Icon has focused on implementation: making it more portable and adapting it to a variety of computer architectures, operating systems, and C compilers. With Version 6, most of the major problems were solved and the book on the implementation was published in 1987.

When the language book was written in 1983, it seemed like Icon was sufficiently mature and stable that few changes would be made to the language itself. However, the research program that originally spawned Icon continued to generate new ideas and the increasingly large user community asked (and still asks) for new features. Programmer-defined control operations came from research on control structures. Sets came from a project in a graduate course on "language internals". Many of the new features in Version 7 of Icon were provided in response to user requests and the recognition that more facilities were needed to support robust application programs.

Requests and suggestions for new features (as well as changes in old ones) continue to come in. Most of these would add to the size and complexity of the language. On the other hand, size — both linguistically and in terms of the implementation — tends to detract from the utility of a language as well. Good language design dictates compromises between facility, size, and complexity. Icon would be a godawful mess if everything everyone suggested were just thrown in. Yet there are good new ideas and possibilities for simplification as well.

On the other hand, the sheer size of the Icon user community and the number of computer systems on which it is implemented contribute a sizable amount of inertia. Version 7 of Icon was a major undertaking that took the better part of two years and still is in progress. Just providing new documentation requires a massive effort, since every system on which Icon is implemented has its own idiosyncrasies. Furthermore, while many users want new features, every new version creates work for users — installation, new things to learn, programs to recompile, and so forth. As the user community increases in size, the inertia increases. At some point, things stop moving.

Version 7 may or may not be the last version of Icon. Probably not. But there comes a time when, on balance, stability becomes essential.

Furthermore, the resources for developing, implementing, documenting, and distributing Icon necessarily are taken away from other possible endeavors — things like a true compiler for Icon or an Icon "machine" cast in silicon.

What all this means is that you may expect to see some future improvements and refinements, but nothing major, to Icon itself. However, there might be some surprises too.

From Our Mail

I downloaded Version 5.9 of Icon from a local BBS. I can't get it to run at all. Can you help me?

Not really. We hear of problems like this frequently. Since Icon is in the public domain, we have no control over its distribution. If the version you downloaded doesn't run at all, it probably was corrupted in transmission somewhere along the line. Also, Version 5.9 is very old. The current version is 7. If you get a copy of Version 7 from us and have problems with it, we'll try to help.

I'm interested in using Icon in computer-assisted composition. I would be most interested in being put in contact with others doing similar work.

While we know what some persons are doing with Icon, we have no way of knowing everyone working in a specific area. Furthermore, we do not give out information about persons without their permission. Since the kind of question you ask comes up fairly frequently, we've decided to provide a free "classified ad" feature in future issues of the *Newsletter*. Just let us know that you're willing to have your name, address, and interests published.

Is Version 7 of Icon available yet for the Atari ST? If so, please have someone transmit it to my computer at . . .

Yes, Version 7 of Icon is available. However, we can't undertake to transmit individual copies to other computers. You can pick it up from us via FTP or from our BBS. (See the **Downloading Icon** box on page 2 for details.)

I just got a copy of the book on the implementation of Icon. Please tell me how I can get the source code for Version 6.2 as suggested in the book.

The implementation book describes Version 6 of Icon and Version 6.2 corresponds most closely to the material in the book. At the time the book was written, it seemed like a good idea to suggest to readers that they get Version 6.2 of the source code. However, many improvements have been made to the source code since then, and we now think it is better for readers of the book to have the most current source, which is presently Version 7.0. See the order form at the end of this *Newsletter* for available formats. A document describing the differences between Version 6.2 and 7.0 of the source code is available, free of charge. Ask for IPD51.

Any progress on the "extension interpreter" described in Newsletter No. 25?

Yes, we have modifications to Icon so that it can call and be called by C functions. However, it's going to take a while to get it into the versions we distribute. We're presently estimating release some time early in 1989.

How's the next version of the Icon Program Library coming?

We were afraid someone would ask us that. There seem to be a *bezillion* things to do, and work on the program library keeps getting interrupted. While it's still possible that we'll get it out late this summer, don't count on it.

I haven't seen anything about an implementation of Icon for IBM 370 mainframes. Is anything in the works?

Quite a bit of work has been done to support such an implementation, including most of the code necessary to support the EBCDIC character set. However, there's not been much progress on an actual implementation. We've just recently been contacted by a person who has the resources to do the job. Maybe there will be good news by the time of the next newsletter.

Has anyone made any progress getting Icon to run on the Apollo Workstation?

Yes. In fact, on the Apollo Workstation running UNIX 9.7 it's said to be straightforward to configure UNIX Icon. We have the configuration files, but they're not yet included in our present UNIX distribution. We'll send them, if you like.

Does Version 7 of Icon run on the Sun-4 Workstation?

Yes. We recently got the configuration files, which are similar to those for the Sun-3 (although co-expressions and arithmetic overflow checking are not yet implemented).

We're installing a CDC Cyber system and would like to get Version 2 of Icon.

We no longer distribute Version 2 of Icon. Depending on what model of Cyber you are installing, it may be practical to port Version 7 of Icon to it.

Does Icon for MS-DOS compile under QuickC?

No; some modules are too large for QuickC.

Any progress getting Icon for MS-DOS to run under Turbo C?

The answer is still "no". We're hoping someone in the MS-DOS user community will tackle this one; we just haven't had time.

I want to run MS-DOS Icon on my PS/2, but it only has a 3.5" drive. Could you send me 3.5" diskettes instead of 5.25" ones?

We certainly understand your problem — we have it too. The problem is partly one of managing our distribution (we offer a *lot* of different diskettes) and partly one of media preparation. We're in an "in-between" situation. We distribute so many diskettes that making copies individually is painful, but we don't distribute enough to use commercial duplication services. We have a bulk diskette copier for 5.25" diskettes, which makes their production manageable. Bulk diskette copiers are very expensive, and we can't justify one for 3.5" diskettes. We'll do something about this, however, probably offering 3.5" diskettes where there is the most demand first.

Bugs

Like all complex software systems, Icon has bugs. (SDI would be no different.) Some we know about and, of course, some we don't. Some are longstanding and some are new. Some we know how to fix and some we don't (yet). Some we plan to fix and some we don't. Some are so esoteric that you need an expert knowledge of the implementation to understand them. Some things you might think of as bugs we prefer to think of as "features". This may make it sound like Icon is really buggy. It's not. It's just that it's large and complicated and some problems are inevitable.

We're starting a feature in the *Newsletter* related to bugs. There's no chance of running out of material, but we may not have the stomach for some of the esoterica. We'll start with a couple of problems of current interest.

Leap-Year Woes

Several users have reported that MS-DOS ER Icon gives the wrong date. This is a leap-year problem and it is the result of an error in the Lattice C 3.20 runtime library. There's nothing we can do about it until we produce a new ER version with a corrected version of the Lattice library.

String-Allocation "Botch"

There is a problem in Version 7 of Icon that can produce error termination with the message "`system error: string allocation botch`". Your chances of running into this problem are small, and if you do run into it there is a workaround. To understand this, you need to know how Icon allocates storage.

Each allocation request is preceded by a "predictive need request", which assures that enough space will be available when it's needed. A garbage collection may occur when such a predictive need request is made, but a garbage collection cannot occur when allocation is actually performed. This strategy allows requests to be made in advance at "safe" times, since at the time allocation is done, things the garbage collector needs may be in disarray.

That's fine as long as every routine that needs to allocate space remembers to make a predictive need request and requests the right amount of space. If a request isn't made or if the requested amount is too small, nothing bad will happen as long as enough storage is available when the allocation is done. However, if there is not enough storage available, this is detected as an internal error in Icon and the "botch" message occurs. Note that a "botch" can only occur when there is little space left to allocate. For more detailed information, see the implementation book.

The bug in Version 7 of Icon is in the new function `char`, which fails to make a predictive need request for the one character of storage it needs.

The workaround is *ad hoc*, but amounts to adding some operation that assures enough space is available. The heavy-handed (expensive), but obvious approach is to add a call of `collect` to assure that plenty of space is available. (Garbage collection always assures there is extra space available.) A less-straightforward and non-guaranteed approach is to add other operations that allocate strings in the vicinity of the trouble spot in the hope that these operations will trigger a garbage collection when space gets low. Note that since `char` only needs one character, the chances of a "botch" are small. (But it's easy to write a program that always shows this bug — try it.)

Another new function, `detab`, also has a problem with predicting its storage needs. The situation that causes it is obscure, and the resulting error message is different. See if you can produce it.

These "predictive needs" bugs will be corrected in the next release of Icon. No date for a new release has been set yet.

Language Corner

Failure and Errors

The concept of failure — that an expression may not produce a result — is central to expression evaluation in Icon (see *Icon Newsletter* No. 25). Failure does not mean that something is wrong or that an error has occurred but rather that a computation cannot be performed or that there is no reasonable result. Examples are an end-of-file when attempting to read data and an out-of-bounds list subscript.

Using failure to control computation and program flow makes Icon programs compact and concise. An example is

```
while write(output,read(input))
```

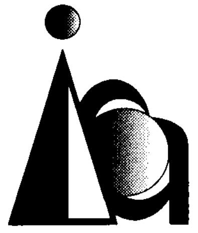
which copies input to output and terminates when an end-of-file is encountered on input.

Icon discriminates between failure and errors, which cause program termination. For example, an error occurs if data cannot be written (usually because of lack of space on the output device). Similarly, an attempt to perform arithmetic on nonnumeric data is erroneous, as in

```
i + "a"
```

The distinction between failure and error is largely a matter of language design. Some conditions, like an end-of-file, are expected, do not indicate a programming mistake, and are useful for controlling program flow. Other situations, such as an attempt to perform an arithmetic operation on nonnumeric data, indicate a programming mistake. Similarly, the inability to write data indicates a serious problem external to the program itself.

Sometimes the distinction is not so clear. Should repl(s,0) be an error, should it fail, or should it just produce an empty string? What about repl(s,−1)? Icon treats the former as a reasonable computation, since the empty string is well-defined, but it treats the latter an error. This is a fine distinction; while it probably would not be a good idea to have repl(s,−1) produce a result, failure is not unreasonable. Some decision has to be made, however.



One advantage of failure over error termination is that failure at least allows the program to retain control. The disadvantage is that a programming mistake may go undetected. Different programmers have different views about what should be treated as errors. You might prefer to have "unusual" situations treated as errors during program development and debugging, but you probably would be annoyed if you wrote an application program that simply crashed, because of lack of disk space, in the middle of updating a user's database.

Unfortunately the situation is complicated, and there is no easy solution. There are a lot of places where failure or error termination may occur in Icon's large repertoire of functions and operations. At present there are more than 400 places in the implementation where failure is signaled and nearly 300 places where an error is signaled. It isn't really practical to give a programmer fine control over the distinction between failure and error termination. You can't expect to say "I want an out-of-bounds list subscript to be an error in this procedure, but I also want attempts to add nonnumeric values to fail". (If you're interested in programming language design and implementation, you might think about how such a facility could be provided.)

The most serious problem is having a program terminate when you don't expect it to or want it to. As suggested above, this is a particularly serious problem in programs that modify files. It also is a general problem in application programs that are used by persons who may know nothing about Icon or its error messages. To provide a way around some of these problems, Version 7 of Icon has a method for converting errors to failure. This method is not subtle or flexible, but it does allow a program to retain control in all but the most extreme situations.

Errors are converted to failure using the keyword &error. If the value of &error is nonzero when an error occurs, the erroneous condition is converted into the failure of the expression in which the error otherwise would occur. This process is called *error conversion*. When an error is converted, the value of &error is decremented. Thus, it behaves in the same fashion as &trace: assigning the value 2 to &error allows two errors to be converted, after which error conversion is turned off, while assigning −1 to &error allows unlimited error conversion.

Suppose, for example, that you want to sum the numbers from a file. You might write this as

```
i := 0
while i +:= read(f)
```

If one of the lines in f is nonnumeric, your program will terminate with a run-time error. You could provide a test to avoid this and provide diagnostic output in the following way:

```
i := 0
while x := read(f) do
  if not(i +:= numeric(x)) then
    write(x," is not numeric")
```

You could shorten this somewhat by using error conversion:

```
&error := -1
i := 0
while x := read(f) do
  (i +:= x) | write(image(x)," is not numeric")
```

While this illustrates the use of error conversion, it probably is not good programming style in this case. After all, you don't really *need* error conversion to do the job, as the previous procedure illustrates. Worse, it is dangerous. Since error conversion applies to all errors, if data cannot be written, the warning about bad data will be lost. That probably isn't a practical problem in this case, but consider what may happen if error conversion is in effect in the following loop:

```
while write(output,read(input))
```

If there is not space to write the data, it is simply lost, silently — one of the worst things that can happen.

Basically you should use error conversion only when absolutely necessary, not just as a short-cut. If you do use it, take extra care to detect failure that may be induced by error conversion at places you normally would not expect to find it. For example, the loop above might be recast as

```
while line := read(input) do
  write(output,line) | stop("*** write failed")
```

In other words, if you use error conversion properly, your programs should be *longer* and more detailed than if you don't.

It's worth noting that the idea for error conversion came from problems that resulted because write formerly did not detect that data was not actually written. This wasn't a practical problem on large computer systems with a lot of secondary storage, but it was a real problem with personal computers, where floppy disks have very limited capacity and hard disks fill up quickly. To handle this, problems with writing data are detected and treated as Icon run-time errors in Version 7 of Icon. But if this happens, the program loses control. Failure, on the other hand, was too dangerous. So error conversion was introduced as a means of giving the programmer the control needed to detect errors but maintain control of program execution.

There are several other keywords related to error conversion:

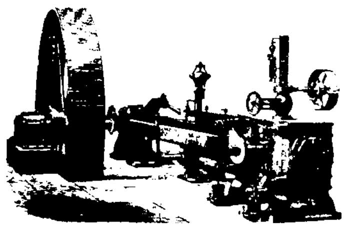
- &errornumber is set to the error number when an error is converted to failure.
- &errortext is set to the text for the error message when an error is converted to failure.
- &errorvalue is set to the offending value when an error is converted to failure. If there is no relevant value, the evaluation of &errorvalue fails.

The use of these keywords is illustrated by a procedure, ErrorCheck, that could be called at places in a program where errors may occur. An example of such a situation might be

```
while line := read(input) do
  write(output,line) | ErrorCheck(&line,&file)
```

The following version of this procedure could be used to report errors to the person running a program and giving that person the opportunity to terminate execution or let it continue:

```
procedure ErrorCheck(line,file)
  write("\nError ",&errornumber," at line ",line, "in file ",file)
  write(&errortext)
  write("offending value: ",image(&errorvalue))
  writes("\nDo you want to continue? (n)")
  if map(read()) == ("y" | "yes") then return
  else exit(&errornumber)
end
```



Inside Icon

Ideally, users of a programming language should not have to know much, if anything, about its implementation. In practice, some knowledge usually is needed and more may be helpful, so we're starting *yet another* semi-regular feature in the *Newsletter* devoted to this aspect of Icon.

Our earlier remarks about how lists are implemented (see *Icon Newsletter* 25) prompted Dave

Gudeman to design and implement an alternative implementation of lists, which is incorporated in Version 7 of Icon. What follows is his description of the problem and the improved implementation.

> The push() and put() functions in Icon extend the length of lists, so that the implementation of these functions has to allocate memory space for new elements. Memory is allocated in "blocks". Each block has sixteen words of memory allocated for various bookkeeping data and two words for each list element that it can hold. To save space, push() and put() formerly allocated blocks big enough for eight elements so the next seven push() or put() calls didn't have to allocate any new memory. The problem with these eight-element blocks is that half of the memory allocated in list-element blocks was just for bookkeeping data, essentially wasted space.
>
> Obviously we would like to save some of this wasted bookkeeping space by allocating blocks that hold more elements. We could just allocate space for (say) twenty elements, but that means that nineteen elements are wasted if there aren't any more push() or put() calls (that's nineteen unused elements at two words per element). A better solution is to allocate blocks that get bigger as the list gets bigger. Remember that half of the space is wasted for the old method. So for the new method, why not allocate a block that doubles the number of elements in the list? This wastes a little more space than the old method when new blocks are allocated, but once a few more elements are put in the new block, the new method becomes better than the old one. And it continues to improve for each put() until a new block is allocated again. So on the average the new method uses less space than the old one.
>
> Another advantage of this method is that it makes access faster for large lists that have been built up by push() and put(). For example, take a list that started as an empty list and had all of its elements added by put(). The blocks are chained together, and to access the *i*th element we have to go through all the blocks up to the block that contains the element we are looking for. For the old method, this means we have to traverse $i/8$ blocks. For the new method, we only have to traverse $log(i/4)$ blocks — a considerable improvement. Experimentation shows that access is already noticeably faster for a list with less than five hundred elements. And the bigger the list gets, the more speedup we see with the new method.
>
> There are a couple of details yet. First is the problems that may show up for lists that are built with both push() and put() instead of just one of them. A lot of space can be wasted if there are huge mostly empty blocks at both ends of the list. Experimentation shows that if we allocate new blocks that are half the current size of the list, then we save more space than we do with either the old method or the method where we allocate blocks the full size of the list. Also, the half-size method seems to be just as fast as the full-size method for accessing list elements.
>
> The second detail involves what happens when a program runs out of memory. Suppose a program does a push() to a large list that has to allocate a large block, but there isn't enough memory. It seems like a good idea to try a smaller block, hoping that the program will be able to finish without allocating any more list blocks. So if we try to allocate a large block and cannot do it, we divide the size by four and try again. We keep trying this until we can allocate a block or the block gets so small it would be useless.
>
> Here is the final algorithm for allocating a new list-element block for list L. The procedure new_block(i) allocates a block with i elements if there is enough memory. If there isn't enough memory, new_block(i) fails.

```
i := (8 < *L) | 8
until block := new_block(i) do
   # not enough memory for i elements
   i /:= 4
   if i < 8 then stop("out of space")
```

Clip-Art Credits

Graphics that first appeared in *Newsletter* No. 26 are credited in that issue.

Page 1. Gregg M. Townsend, using programmer-defined Postscript font.

Page 4. Jacques Nel, using Cricket Draw.

Page 5. *Blackburnium cavicolle*, Geotrupidae, scanned image.

Page 5. Jack Radley, using Adobe Illustrator.

Page 6. Jacques Nel, using Cricket Draw.

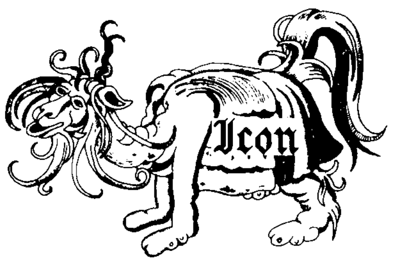
Page 7. From the Dover Pictorial Archive Series, scanned image.

Once again, thanks to all who contributed. Credits at the "Icon Store" have been sent as described in *Newsletter* No. 25.

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 personal computers requires at least 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 books *The Icon Programming Language* (Prentice-Hall, 1983) and *The Implementation of the Icon Programming Language* (Princeton University Press, 1986), 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:

ATE: Atari ST Icon executables: one single-sided 3.5″ diskette. $15.

DE: MS-DOS Icon executables: two 2S/DD 5.25″ diskettes. $20.

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

ME: Macintosh (MPW) Icon executables: one 800K 3.5″ diskette. $15.

MS: Macintosh (MPW) Icon source and test programs: two 800K 3.5″ diskettes. $25.

UPE: UNIX PC Icon executables: one 2S/DD 5.25″ diskette. $15.

XE: XENIX 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. $30.

ID: Implementation documentation package. $40.

NL: Back issues of the *Icon Newsletter*. $.50 each for single issues (specify numbers). $6.00 for a complete set (Nos. 1-26). 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.