

# The Icon Newsletter

No. 30 – June 4, 1989



## Credit Card Orders

We can now accept MasterCard and Visa for the purchase of Icon material. See the ordering information at the end of this *Newsletter*. If you place an order by telephone, be sure to have your card handy.

## Icon Program Library

Version 7 of the Icon program library is now available – at least the first part of it. See the order form at the end of this *Newsletter*.

Part 1 of the Icon program library contains both complete programs and collections of procedures that may be useful in composing other programs. There are 45 complete programs and 137 additional procedures. The programs range from simple utilities to sophisticated text generators.

While the library is useful in its own right, it also provides many examples of Icon programming techniques that may be particularly helpful to persons who are new to Icon or who want to improve their Icon programming skills.

Part 2 of Version 7 of the Icon program library is in preparation. It will contain larger and more complex program packages as well as some programs designed for specific operating systems.

Contributions to the Icon program library always are welcome. Guidelines for submission of new material are contained in the documentation that accompanies Part 1 of the library.

## Implementation News

### Version 7.5

We've brought several more implementations of Icon into the Version 7.5 "stable": the Amiga, the Atari ST, Macintosh MPW, the UNIX PC, XENIX, and

XENIX/386. Version 7.5 source code also is now available for the Atari ST and Macintosh MPW.

Since the differences between 7.0 and 7.5 are minor as far as the language itself is concerned, there is no compelling reason to upgrade if you have an earlier Version 7. If you encounter a problem, however, you may need to get 7.5, so that we can provide help. If you are working with the source code, however, you should get Version 7.5, since there are many changes and improvements in the 7.5 implementation.

Thanks to Bob Alexander, Mic Bowman, Ronald Florence, Clint Jeffery, and Steve Wampler, who helped with Version 7.5 upgrades.

### 3.5" Diskettes

We can now provide 3.5" diskettes (720K) for MS-DOS executables, MS-DOS source, and UNIX source. The default for filling orders still is 5.25" diskettes; if you want 3.5" diskettes, be sure to specify them when ordering.

## Communicating with the Icon Project

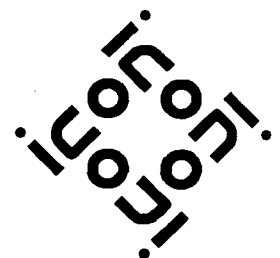
You can communicate with the Icon Project in several ways. Electronic communication often is fastest and easiest. In fact, most Icon program material and some documentation is available electronically.

### Getting Material Electronically

There are two methods for getting program material and documents electronically: network file transfer (FTP) and our electronic bulletin board (RBBS).

#### Network File Transfer

If you have access to FTP, that is by far the fastest and most reliable way to get Icon material. FTP to arizona.edu. When you are asked to log in, enter anonymous. When you are asked for a password, enter any non-empty string. Then



cd icon  
get README

Look through README to see what's available.

If arizona.edu does not work for you, there are two numerical network addresses that you can use: 128.196.128.118 and 192.12.69.1.

### Electronic Bulletin Board

RBBS access is via the telephone. It provides Xmodem, Xmodem(CRC), windowed Kermit (which includes unwindowed Kermit), and plain ASCII transfers.

The RBBS number is (602) 621-2283. It uses a Hayes 2400-baud modem. You can connect at either 1200 or 2400 baud. RBBS asks for your first and last names and the city and state from which you are calling. It is set to reject certain names it recognizes as bogus, so you should give your real name. After getting on, using the bulletin board is simply a matter of navigating through the menus.

For the greatest flexibility in using the file downloading capabilities, once you've logged on, set your serial port to 8-bit characters, no parity, and 1 stop bit.

RBBS is normally available from 5 p.m. to 8 a.m. Mountain Standard Time on weekdays and all day on weekends and national holidays. It may be available at other times, but it is subject to unscheduled interruption then.

Notes: RBBS is available only for downloading Icon material. Uploading is not supported and messages are not answered.

### *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.

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

All rights reserved.

RBBS contains several files that are much too large for general downloading. These files are there for compatibility with the contents of our FTP area and for persons in our local calling area who can download large files via telephone successfully. Use your own judgement about the practicality of downloading large files.

Most problems related to file transfer from RBBS seem to be related to bad telephone connections. In some cases there may be problems with modem incompatibilities. If you experience persistent problems, you probably should get Icon material another way, such as by ordering physical copies — it's usually less expensive and faster in the long run.

### *Electronic Mail*

Electronic mail is the fastest way to get information and help. We do not, however, provide program material or documentation by electronic mail.

Our electronic mail addresses are:

icon-project@arizona.edu (Internet)

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

Electronic mail addressed to icon-project is usually answered the day it is received.

Although the reliability of electronic mail has improved considerably over the last few years, there still are failures. In addition, the mail path back for responses sometimes fails. If you send electronic mail to icon-project and do not get a prompt response, you should assume it was not received or that the answer did not reach you. The first time you send electronic mail to the icon-project, you should include a postal mailing address in case we are unable to reach you electronically. This will also put you on the list for a free subscription to the *Icon Newsletter*, if you're not already on it.

Responses to electronic mail addressed to icon-project may come from one of several persons who handle various aspects of Icon. You may wish to exchange subsequent electronic mail about a specific topic with that person rather than with icon-project, but later mail on a different topic should be addressed to icon-project to assure it reaches the appropriate person and to get a prompt response in case an individual is away.

### *Electronic Newsgroup*

We provide an electronic newsgroup for the redistribution of electronic mail to persons who are interested in Icon.

To become part of this newsgroup, send a request to icon-group-request in place of icon-project in the addresses given for electronic mail previously. You should

also use icon-group-request to unsubscribe to the newsgroup.

All electronic mail sent to icon-group at Arizona is automatically distributed to newsgroup subscribers. Any subject of general interest related to Icon is suitable for the newsgroup. However, since mail to icon-group is redistributed widely, that address should not be used for requests for information that the Icon Project can handle. Use icon-project for this.

### **Voice Contact**

Information of a nontechnical nature about Icon may be obtained by calling (602) 621-2018. This number also can be used for credit card orders of program material and books. Be sure to have your credit card handy for such orders. You can also leave messages for individuals at this number.

### **Facsimile Transmission**

Our FAX number is (602) 621-4246. FAX may be used for technical matters and for credit-card orders. For credit-card orders, provide the card type, card number, expiration date, cardholder name, and signature, stating that you authorize the order to be charged to your credit card. See the order form at the end of this *Newsletter* for an example of what is required.

### **Postal Correspondence**

Postal correspondence to the Icon Project should be addressed as follows:

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

We answer all postal correspondence, usually within one week of receipt. Persons who request routine information about Icon may receive an appropriate document rather than a letter.

### **Summary of Addresses and Numbers:**

FTP:

arizona.edu (128.196.128.118 or 192.12.69.1)

RBBS:

(602) 621-2283

Electronic mail:

icon-project@arizona.edu  
...{uunet,allegra,noao}!arizona!icon-project

Newsgroup subscription:

icon-group-request@arizona.edu

...{uunet,allegra,noao}!arizona!icon-group-request

Newsgroup mail:

icon-group@arizona.edu

...{uunet,allegra,noao}!arizona!icon-group

Voice contact:

(602) 621-2018

FAX:

(602) 621-4246

### **From Our Mail**

*I'm trying to read binary data in Icon on my IBM AT clone. I'm using reads(f,n) as recommended in the last Newsletter, but characters with hexadecimal code 0d vanish. What's wrong?*



Our answer to a related question in the last *Newsletter* was incomplete. The reason to use reads(f,n) to read binary data is to enable you to read fixed-sized blocks and also not have linefeeds discarded as read(f) does. On an MS-DOS system, you must also open f in untranslated (binary) mode; as in

```
f := open(fname, "u")
```

Otherwise, by default the file is considered to be text and line terminations (carriage return/line feed) are converted into linefeeds on input and conversely on output. This default translation is a property of CI/O libraries on MS-DOS. This is also true for the Atari ST.

*Will Version 7.5 of Icon compile and run under the MS-DOS MIX C compiler? (We're repeating this question from Newsletter 29 to provide a more complete answer.)*

No. Icon requires 32-bit pointers, like those found in the large memory model. MIX C does not provide this model.

*Why can't I get The Icon Newsletter electronically? It seems to me it would save you a lot of money.*

The way the *Newsletter* is prepared does not permit electronic distribution. In the first place, it's done using a desktop publishing system. While the material in the *Newsletter* is in machine-readable form, it requires the desktop publishing system to format and print it. Some persons recognize that Icon is printed on a PostScript printer and have asked for the PostScript electronically. The PostScript file would be *very* large. More important, it contains code, copyrighted by others, that we cannot distribute.

*I've just gotten a copy of Icon, but unfortunately the Prentice-Hall book I need is out of print. Can you help me?*

The Prentice-Hall book (*The Icon Programming Language*) is not out of print. Once or twice in the past few years it has been temporarily out of stock at the publishers because of unexpected demand. If you can't find a copy locally, you can order it from the Icon Project as described at the end of this *Newsletter*.

*When is Icon for VM/CMS going to be available?*

We don't know for sure. There are a couple of problems that need fixing and there's no user documentation yet. Unfortunately, we don't have the facilities to work on it locally. We are getting outside help and hope to have something available by this fall.

*Will VM/CMS Icon run under MVS?*

We think it will be fairly simple to transport VM/CMS Icon to MVS, but no one has tried it yet.

*I want a copy of the source code for Icon. Please break it up in small pieces and send it to me by e-mail.*

We're happy to make Icon material available electronically. However, you'll have to download it from our RBBS or use FTP, as we don't undertake to initiate transfers. Incidentally, the source code for Icon is too large to send via electronic mail.

*Professor Donald Knuth has developed a language pre-processor, WEB, that significantly aids both program development and documentation. There are public-domain versions of WEB for C and Pascal. Is there an Icon version?*

Not as far as we know. Perhaps one of our readers will respond if such a thing is in the works.

*What's the status of the "extension interpreter" for Icon? I have an Atari ST and would like to be able to access system routines for graphics and sound from Icon programs.*

We have the code that will allow Icon programs to access C library routines. However, we have not incorporated it into Icon yet. We hope to get to this; it's one of several projects that are pending, waiting for time and resources. However, accessing library routines for Icon has two significant problems. One problem is passing data, which in many cases is different in format between Icon and library routines. The other problem is that overhead in communication may be so large that operations using low-level library functions will not run acceptably fast.

*Why do I have to pay for Icon material in US dollars? It's a lot of trouble for me; can't you just convert my currency (pounds)?*

The problem is the cost of conversion. It can be \$30 or more, regardless of the amount converted, and it's also unpredictable. In fact, our bank refuses to handle conversion in many cases. Credit cards, which we now can accept, provide a good way around this problem.

## Object Icon

*Editors' Note: Object-oriented programming continues to receive a lot of attention and we're frequently asked about such features for Icon. This article is contributed by Bill Griswold at the University of Washington. It describes experimental work he's done on the subject.*

Object Icon is an extension of Icon that permits defining types so that a value can carry its operations with it. This permits users to define polymorphism per type rather than per procedure. It does not support encapsulation. It is fully compatible with Icon. The additions to the translator using the variant translator facility were simple, and no changes were required in the interpreter. Many features and aspects of the implementation are imperfect; Object Icon is only a prototype.

To define a good object-oriented Icon might require creating a new language, rather than extending Icon. For example, a raw record type is a little out of place in Object Icon. Also, decisions regarding existing (i.e., non-object oriented) types in Icon require careful thought. A language with two type models is a little awkward, but so are integers implemented in an object-oriented style.

## The New Features

A new user type (class) is defined by the class declaration

```
class type(field1,field2,...)
```

This defines `type` as a class possessing each `field[i]` as an attribute. For example, the following definition defines a stack class with attributes `rep`, `push`, `pop`, `size`, and `init`.

```
class stack(rep,push,pop,size,init)
```

Each attribute is actually an instance variable. Any value can be stored in an instance variable, including class procedures (methods). The syntax for defining a method is

```
procedure type::name(arg1,arg2,...)
```

If `name` is an attribute of `type`, creating an instance (object) of `type` causes the instance variable `name` to be initialized to the procedure defined by `type::name`. When called, a class procedure has an implicit local variable `self` that contains the value of the instance that is invoking the operation. This permits a class procedure to access the instance variables of the invoking instance. Consider the following method:

```
procedure stack::push(value)
  push(self.rep,value)
  return self
end
```

This method uses `self` to access the instance variable `rep` to perform a push operation on the list stored in it. A method is accessed through the `$` operator. For example

```
mystack$push("hello")
```

invokes `stack::push` for `mystack`, pushing the value "hello" on it. A class method can also be accessed directly with the `::` syntax. For example

```
mystack.pop := stack::pop
```

assigns the method `stack::pop` to the `pop` field of `mystack`. The operation can also be invoked directly, as in

```
stack::push("world")
```

However, this is unlikely to be meaningful if the push operation accesses the `self` variable (which in this case would be null). However, this syntax is useful for the object-creation operation:

```
mystack := stack::stack()
```

The `self` variable in `stack::stack` is null on entry to the method but is initialized during its execution. Here is the implementation of the constructor `stack::stack`.

```
procedure stack::stack()
  self := new_stack()
  self.rep := []
  return self
end
```

The method `new_stack` is defined implicitly by the system. It calls the allocation function `stack`, and initializes all the instance variables to the methods that are defined for them (based on the correspondence of type and field names).

There is no inheritance declaration such as

```
class stack : aggregate (...)
```

which would declare that a `stack` has all the instance variables and procedures of `aggregate`, plus any additionally defined by `stack`. This might be forthcoming, but to be effective it must be general, allowing multiple superclasses and redefinition of inherited procedures. This sharing can be handled in Object Icon explicitly by overlapping instance variable names and assigning a borrowed procedure directly into the overlapping instance variable. See the implementation section for details on how it might be implemented.

## Using Object Icon

Icon's field access operations are polymorphic. This means that the field reference `x.y` is defined as long as the value `x` is a record that has a field `y`. The same is true

for class instances. This means that instance variables of the same name in different classes are implicitly and automatically shared (it cannot be prevented). In many cases this overlap is meaningful. For example consider a class `array` and the above class `stack`.

```
class array(rep,size,in,out,clear)
```

Both classes define a procedure `size` that returns an integer representing the size of the invoking object:

```
array::size()
  return *self.rep
end

stack::size
  return *self.rep
end
```

If some variable in the program knows it has an aggregate of some kind, but does not care which kind, it can invoke the `size` method and find its size, regardless of whether it's an `array` or `stack`:

```
agg_size := some_agg$size()
```

Not only can actual fields be shared, but code implementations can as well. For example, if the `stack` and `array` were actually implemented as above, the sharing would be easy. Suppose `array` just reused the implementation of `stack::size` by borrowing it in its constructor function:

```
procedure array::array()
  self := new_array()
  self.rep := []
  self.size := stack::size
  return self
end
```

This works as expected.

## Implementation

The implementation is accomplished with a variant translator. The variant translator facility, available only under UNIX, is an Icon-source-to-Icon-source translator that can be modified easily to accept Icon variants and generate Icon code to support the variants. A variant is described by modifying the grammar and actions of the Icon Yacc description, and adding any necessary semantic processing routines. This is aided by macro processors that take a high-level description of actions and map it into the actual description.

The Object Icon source is translated into Icon and then compiled as an Icon program. This translation results in a number of names being changed, possibly obfuscating tracing. Line numbers can be preserved, but currently are not.

The class declaration is translated directly to a record declaration of the same name. This means that `type(mystack) == "stack"`. It also means that calling the constructor function `stack` directly does not initialize stack instances meaningfully. This is why class object constructors such as `stack::stack` call the object initializer defined by the variant translator when class `stack` is declared:

```

procedure new_type()
  return type(type_field1, ...)
end

```

If `type_field1` is undeclared, it has the null value as expected.

The system does not check that a defined procedure actually corresponds to a class definition. For example, the procedure `stack::stack` defined above does not have a corresponding attribute in the class `stack` declaration. This means spurious procedures can be defined, but it also allows for definition of the constructor without actually storing it in each instance.

Defining the method `type::name(arg1,arg2,...)` results in defining the Icon procedure

```

type_name(self,arg1,arg2,...)

```

Defining a procedure of `type_name` directly can result in a compile-time error due to the multiple definitions.

The call `expr$name(arg1,arg2,...)` results in defining the call

```

(_self_temp := expr).name(_self_temp,arg1,arg2,...)

```

or some equivalent expression.

### Example

This is a complete Icon program implementing a stack class.

```

class stack(rep,push,pop,size,init)
procedure main()
  mystack := stack::stack()
  mystack$push("hello")
  mystack$push("world")
  size := mystack$size()
  write("My stack is size ",size)
  write("popping ",mystack$pop(),
    " ", mystack$pop())
end
procedure stack::stack()
  self := new_stack()
  self.rep := [ ]
  return self
end

```

```

procedure stack::push(value)
  push(self.rep,value)
  return self
end
procedure stack::pop()
  return pop(self.rep)
end
procedure stack::size()
  return *self.rep
end
procedure stack::init()
  self.rep := [ ]
  return self
end

```

### Implementing Inheritance

To implement class inheritance, such as in a declaration like

```

class type : supertype1 : supertype2 : ... (field1,field2,...)

```

it is feasible to substitute the fields of supertype in front of the fields defined for `type`. Redeclaring a field could cause an error, or mask the old field (i.e., its initial value). Masking initial method values can be done in the `new_type` operation as follows:

```

procedure new_type()
  return type(type_field1 | \supertype_field1 | ... , ...)
end

```

resulting in taking the "closest" implementation to the type being defined. This type of sharing does not prohibit the sharing of class attributes exploited in the current definition of Object Icon, meaning that there are two ways to share. This can compromise clarity. It may be better to leave Object Icon the way it is. Also, note that the current technique permits sharing class procedure implementations across types, independent of field names. This means a procedure can be borrowed without the names being the same. This is important, since `type` is considered to be independent of implementation.

Another problem with inheritance is that inherited instance variables must get initialized in some way. For variables holding methods this is not difficult, but it is for normal values. Usually this should be done by the constructor of the superclass, but the current method does not allow for this in a transparent fashion. However, any solution could be used to initialize method variables as well, and the above method could be abandoned.

*Note:* Object Icon is an experimental system. It is not available for distribution.

## Programming Corner



### Pointer Semantics, Graphs, and Sets

Icon uses pointer semantics for structure values. What this means is that a list, for example, is a pointer to a collection of the values that the list contains. A

pointer, which is a memory address in implementation terms, is small and fixed in size, even though the collection of values in the list may be arbitrarily large.

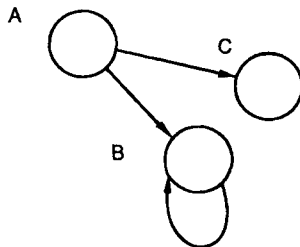
An assignment in Icon, such as

```
L := list(1000)
```

merely copies the pointer produced by list(1000) into L; the 1,000 values are irrelevant as far as the assignment is concerned — they are not even referenced.

The use of pointer semantics has several consequences. Some are good (efficiency in handling structure values) and some are (at least potentially) bad, such as unintentional sharing of structure values via pointers. These issues are discussed in the Icon language book.

What may not be obvious is how pointers to structures in an Icon program can be used to reflect, in a natural way, structures that exist in the problem domain. Consider, for example, a simple directed graph:

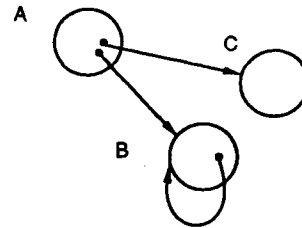


In order to perform computations on such a structure, it is necessary to represent them in some way in a program. One way to do this is to represent each node in the graph by a set. Then the values in the set are pointers: arcs to the nodes to which the node points. For example, the program structures for the graph shown above are:

```
A := set()
B := set()
C := set()
insert(A,B)
insert(A,C)
insert(B,B)
```

The important conceptual point is that a set is a pointer to a collection of pointers to other sets. A

slightly different visualization of the structures in the programming domain illustrates this:



Thus, an arc is represented by a (pointer to) a set and a node is represented by the values in the set.

The ease of manipulating this program representation of graphs is illustrated by procedures to compute the transitive closure of a node (the node and all nodes reachable from it by a succession of arcs):

```
procedure closure(n)
  S := set()
  insert(S,n)          # start with the node itself
  return more(S,n)
end

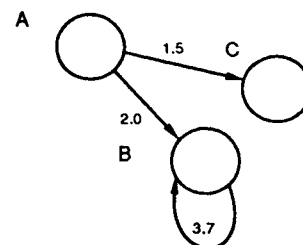
procedure more(S,n)
  every n1 := !n do    # process reachable nodes
    if not member(S,n1) then { # skip ones already in
      insert(S,n1)         # add new node
      more(S,n1)           # recurse
    }
  }
  return S
end
```

Note that a set is used to keep track of nodes already accumulated.

There are several problems that arise in computations on graphs that may require a somewhat more sophisticated representation of structures. For example, if the values are associated with arcs (or if there may be more than one arc between two nodes), the set-of-sets approach is inadequate. In such cases, a record type can be used for arcs, as in

```
record arc(value,node)
```

where the value field contains the value associated with the arc and the node field contains the set to which the arc points. Suppose, for example, that the arcs in the example above are weighted:



Then the graph can be represented in the program as follows:

```
insert(A,arc(2.0,B))
insert(A,arc(1.5,C))
insert(B,arc(3.7,B))
```

*Exercise:* Modify the procedure closure given above to handle this representation of directed graphs.

### Two-Way Tables

Programs that manipulate graphs generally need to be able to read a representation of a graph in string form and write results in string form. For example, the (unweighted) form of the graph above might be represented as

```
A→B
A→C
B→B
```

*Exercise:* Write a procedure to read this representation of unweighted graphs and build the corresponding program structures.

One problem is associating labels for the nodes with corresponding program structures. The natural solution in Icon is to use a table in which the keys (entry values) are the labels and the assigned values are the corresponding sets. Written out explicitly for the graph above, this might be:

```
Node := table()
Node["A"] := A
Node["B"] := B
Node["C"] := C
```

Consequently, Node["A"] produces the node (set) labelled A. Such a table might be used, for example, in constructing a graph from its string representation as posed in the exercise above.

On the other hand, the converse may be needed. For example, in writing out the results of a computation on a graph (such as the transitive closure of a node), the labels associated with nodes may be needed.

Since any kind of value can be used as a key into a table, a table like the one above, with the keys and entry values reversed, can be used:

```
Label := table()
Label[A] := "A"
Label[B] := "B"
Label[C] := "C"
```

It is not necessary to have two tables, however. Since the keys in a table need not all be of the same type, the same table can be keyed with both the labels and the nodes (sets):

```
Graph := table()
Graph["A"] := A
Graph["B"] := B
Graph["C"] := C
```

```
Graph[A] := "A"
Graph[B] := "B"
Graph[C] := "C"
```

Such a "two-way" table keeps all the information needed to associate labels with nodes and vice versa in one structure. Subscript it with a label to get the corresponding node and subscript with a node to get the corresponding label.



## The ProIcon Group Announces First Language Release

ProIcon, the first commercially produced version of Icon, was released May 8 for Apple Macintosh computers. The language is produced by The ProIcon Group, which is a cooperative effort of Catspaw, Inc., Salida, Colorado and The Bright Forest Company, Tucson, Arizona.

ProIcon is an enhanced version of Icon Version 7.5. It is integrated into the Macintosh environment and does not require MPW to run. It features an environment that allows program development, testing, and debugging without leaving the application. ProIcon also has many features not available in standard Icon, including function tracing, an optional termination dump, and several new functions, including ones for manipulating the screen and windows. Finished applications may be distributed to others using a royalty-free run-time system.

For more information, contact

The ProIcon Group  
P.O. Box 1123  
Salida, Colorado 81201-1123  
U.S.A.

719-539-3884

## Graphics Credits

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

Page 8. Charles Richmond, Atari ST, printed output, scanned and traced with Illustrator 88, converted to PostScript font with Keymaster.

Page 10. Ralph Griswold, *Planet Icon*, section of graphic on back page "spherized" in Graphist II; gray-scale editing done in ImageStudio.

Back page. Ralph Griswold, *Icon Fractal Space*, Illustrator 88, based on a design by Mark Emmer of Catspaw, Inc.



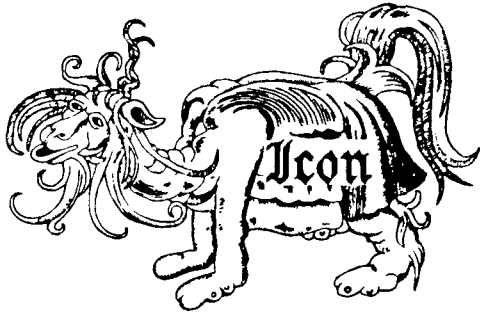
## Ordering Icon Material

### What's Available

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

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

The current version of Icon is 7. All the program material here is for Version 7.



### Icon Program Material

**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 512KB of RAM; it may require more on some systems.

**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.

**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.

## 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.

### 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.

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. 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, 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.

When ordering 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.

The symbol  identifies material that is new since the last *Newsletter*. The symbol  identifies material that has been updated to Version 7.5 since the last *Newsletter*.

### Program Material

#### Amiga:

 AME:  executables \$15


#### Atari ST:

 ATE:  executables \$15

 ATS:  source \$20


#### Macintosh/MPW:

 ME:  executables \$15

 MS:  source \$25

#### MS-DOS:

DE:  (2) or  executables \$20

DS:  (2) or  source \$25


#### MS-DOS/386:


DE-386  or  executables \$15

#### OS/2:


OE:  or  executables \$15



#### UNIX:

UT-T:  tar format \$25

UT-C:  cpio format \$25

UC-T:  tar format \$40


UC-C:  cpio format \$40

UD-M:  (6) or  (4) cpio format \$40

#### UNIX - UNIX PC:

 UPE:  executables \$15

#### UNIX - XENIX:

 XE:  or  executables \$15

#### UNIX - XENIX/386:


 XE-386:  or  executables \$15


#### VMS:


VT:  \$25

#### Source for Porting:

PF-A:  (5) flat system, ASCII \$35

PF-K:  (2) flat system, ARC \$25

PH-A:  (5) hierarchical system, ASCII \$35

PH-K:  (2) hierarchical system, ARC \$25

### Icon Program Library (Part 1)

#### Amiga:

 AML-1:  \$15

#### Atari ST:

 ATL-1:  \$15

#### Macintosh/MPW:

 ML-1:  \$15

#### MS-DOS and OS/2:

 DL-1:  or  \$15

#### UNIX:

 UL-1:  or  cpio format \$15

#### Others:

 PL-1:  ASCII \$15

### Documentation

LD: Language documentation package. *The Icon Programming Language* (Prentice-Hall, 1983) and three technical reports. \$32.

ID: Implementation documentation package. *The Implementation of the Icon Programming Language* (Princeton University Press, 1986) and update. \$40.

NL: Back issues of *The Icon Newsletter*. \$50 each for single issues (specify numbers). \$7.00 for a complete set (Nos. 1-29). There is no charge for overseas shipment of single back issues, but there is a \$5.00 shipping charge for the complete set.





