The Icon Newsletter

No. 43 – November 15, 1993

Contents

Implementation News ... 1

An Icon-Based Parser Generator ... 2

Icon for Humanists Out of Print ... 5

Holiday Closing ... 5

From Our Mail ... 6

Icon in the Classroom ... 7

Graphics Credits ... 8

Ordering Icon Material ... 9

Implementation News

Recent Updates

Bob Alexander has updated Icon for the Macintosh running under MPW to Version 8.10. Both executables and source code are available.

Alan Beale has updated Icon for MVS to Version 8.8. Source code is not included in the present distribution tape.

Icon for NT

Clint Jeffery, who is implementing Icon for NT, reports that the work is coming along nicely. All the regular features of Icon are working, including co-expressions. At present the working graphics features include opening a window and drawing simple graphics and text. A considerable amount of work remains to be done to provide Icon's complete graphics and interface repertoire.

He expects that the NT implementation also will run under Windows 3.1 using the Win32 libraries.

It's too early to predict when Icon for NT will be available. If you're interested in this implementation, let us know.

Work in Progress

We're currently working on Version 8.11 of Icon. New features include the ability to add several elements to a list in a single function call.

The major thrust of Version 8.11, however, is in the area of graphics facilities, which will be substantially improved.

We're not sure yet when Version 8.11 will be available — next summer, perhaps.

Source Updates

We are constantly updating the source code for Icon to correct errors, add new features, and bring platform-specific code up to date.

Because of the amount of time and effort it takes to prepare a full-blown source-code distribution, we only do that when a significant number of changes have accumulated or when there's a major addition to the language itself. This usually happens every two years or so.

Persons who are interested in having the latest version of the source code can subscribe to our source update service, which provides the latest source about three times a year.

The source code for Icon is the same for all platforms, so these updates can be used for any implementation of Icon.

Updates now are available on MS-DOS disks in LHarc and compressed tar formats. See the order form at the end of this newsletter.

An Icon-Based Parser Generator

Editors' Note: The following article, which describes a Yacc-like parser generator, was provided by Dr. Richard Goerwitz, a frequent contributor to the Icon program library.

What is Ibpag2?

Ibpag2 is a so-called "parser generator", that is, a tool for automating the process of generating a recognizer and/or parser from an abstract structural specification of an input language. This description might seem to thrust Ibpag2 into the category of highly specialized tools, but in fact the purpose of parsers is simply to infer structures from an input stream — for example, words from alphabetic sequences, sentences from texts, or equations from series of symbols. Although programmers may not always label them as such, parsers are as much a part of their work as sorting, searching, or anything else. An automated tool for creating them is therefore of potential use to almost anyone who writes computer programs.

Despite their general utility, parser generators like Ibpag2 have significant limitations. Virtually all (Ibpag2 included) use a family of algorithms that cannot easily accommodate natural languages and older programming languages like FORTRAN. For these, one must use a more powerful (and less efficient) system, such as a chart parser. Ibpag2, in fact, does come equipped with a secondary system that gives it this power. Using this system, however, constitutes an advanced topic, and will not be discussed here. For more details, consult the README file that comes with the Ibpag2 distribution (see the end of this article).

A Bit of Background

During the 50s and 60s, linguists, mathematicians, and engineers became quite interested in the formal properties of languages: Could they be described as a series of logical structures that computers could recognize and manipulate efficiently? Linguists, in particular, came to realize that the amount of structural complexity, ambiguity, and pure noise in natural language would render it computationally intractable. Mathematicians and engineers, however, found that many of the formalized notations they dealt with could, in fact, be (re)designed in such a way that efficient computer processing was — at least in principle — achievable.

Principle, in this case, did not meet reality until viable parser generation tools came into being. Parser generation tools convert abstract structural descriptions of formal notations or "languages" to working computer code. Ideally, the designer simply makes assertions like:

An expression is composed of either

(1) a term (for example, 10), or

(2) an expression, a + or –, and another expression.

Parser generator systems translate these assertions (the "grammar") into a machine, that is, automaton, that can recognize and/or manipulate input streams that conform to the "language" so described.

Let me dwell, for a moment, on the toy expression grammar offered above. Note that it describes a set of simple mathematical constructs like:

```
9 + 3
9 + 3 - 8
```

According to the specifications given, the 9, 3, and 8 alone constitute terms — which are also expressions (via rule 1). Because these terms are also expressions, 9 + 3 can be reduced to a larger expression by rule 2. The same is true for 9 + 3 – 8, except that there rule 2 must apply twice — once for 9 + 3, and then again for that and the remainder of the line.

If we join addition and subtraction actions to the above grammar specification, we can create a calculator-like automaton. Traditionally, LR-family automata (like Ibpag2's) contain a parser, one or more stacks, and a set of action tables. The parser reads from an input stream segmented into "tokens" (for example, TERM, +, –), and then manipulates its stacks according to directives contained in its tables. As the parser reads the input stream, it matches rules with action code specified by the programmer. For example, rule 2 above might be matched with code that added/subtracted the expressions on either side of the +/– operator, and produced (in calculator style) the result. Alternatively, it might be matched with code that generated an equivalent construct in another language. The result is a little machine that understands and processes input.

In general, creating code for such a machine is difficult and the resulting files are not easily main-

tained. What Ibpag2 and other tools like it do is allow the programmer to automate the coding process. Ideally he or she need only declare tokens and language specifications, then flick a switch, so to speak. The parser generator creates the appropriate automaton with little, or no, human intervention. If changes are made to the grammar, this process can be repeated, making the actual automaton quite easy to change and maintain. It is the ease and simplicity tools like Ibpag2 bring to the automaton-creation process that made, and in fact still makes, them critical to the development of not just theoretically feasible, but truly "practical" computer language design systems.

Using Ibpag2

To recode the above toy expression grammar in Ibpag2 format is relatively simple, especially if we omit the actions. We need only a set of token declarations and three rules:

```
%token TERM, '+', '–'
%%
exp : TERM
exp : exp, '+', exp
exp : exp, '–', exp
```

TERM, and the addition and subtraction operators, are the tokens (that is, the terminal symbols out of which the grammar is constructed — the things into which the input stream is segmented). The colon means "is composed of". The double percent sign separates token declarations from the grammar proper.

Adding in our actions (that is, directions on what to do with the language as it is recognized) requires just a few extra lines of Ibpag2 action code enclosed in braces. Repeated left-hand sides of rules are indicated by a vertical bar:

```
%token TERM, '+', '–'
%%
exp :TERM {return arg1}
    | exp, '+', exp {return arg1 + arg3}
    | exp, '–', exp {return arg1 – arg3}
```

The arg*n* above refers to the *n*th element of the right-hand side of the preceding rule. So, for example, the action {return arg1 + arg3} above means: "When you have found an expression consisting of a sub-expression, a plus operator, and another sub-expression, use the value of sub-expression 1 plus the value of sub-expression 2 as the value for the expression as a whole". Were we to find 1 + 3 in the input stream, this action would cause the parser to produce 4.

One serious problem with the set of specifications above is that the operators – and + group left to right. We human beings take this for granted. The computer, though, has to be told exactly how to group such expressions. Without explicit instructions, the parser does not know, after it has read 9 + 32 and is looking at a minus sign, whether to shift the minus sign onto the stack, and eventually try to group it as, say, 9 + (32 – 4), or to reduce 9 + 32 to an expression and group as (9 + 32) – 4. This ambiguity is rectified by replacing

```
%token TERM, '–', '+'
```

with

```
%token TERM %left '–', '+'
```

Adding in the unary minus sign to our grammar (for example, –4) takes only a little extra machinery. The main difficulty here is that the minus sign is also used for subtraction, and yet the two expression types have different precedences and associativities (that is, expressions with the unary minus sign get done before subtraction expressions, and group right to left). To get around the operator conflict we use a "dummy" token declaration, and a %prec declaration:

```
%token TERM
%left '+', '–'
%right UMINUS
%%
exp   :TERM {return arg1}
      | exp, '+', exp {return arg1 + arg3}
      | exp, '–', exp {return arg1 – arg3}
      | '–', exp %prec UMINUS {return – arg2}
```

The %prec declaration simply tells the parser that, even though the rule contains a – operator, the rule should be handled as if the operator were UMINUS.

Let us now add in multiplication and division operators to our calculator specifications:

```
%token TERM
%left '+', '–'
%left '*', '/'
%right UMINUS
%%
```

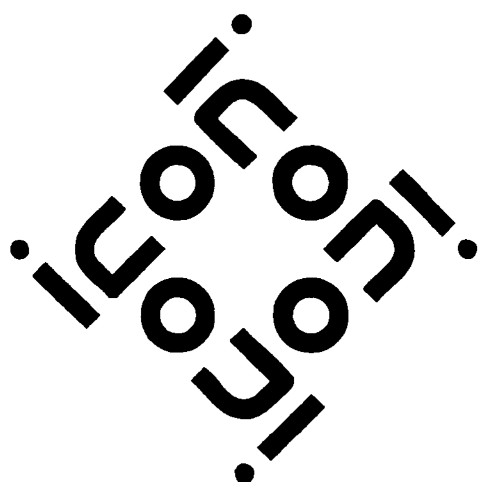
```
exp  :TERM {return arg1}
     | exp, '+', exp {return arg1 + arg3}
     | exp, '–', exp {return arg1 – arg3}
     | exp, '*', exp {return arg1 * arg}
     | exp, '/', exp {return arg1 / arg3}
     | '–', exp %prec UMINUS {return – arg2}
```

Note that the multiplication and division operators were defined *after* the addition and subtraction operators. The reason for this is that, technically speaking, the grammar itself is ambiguous. If we treat all operators identically, the parser will not be able to tell whether 9 + 1 * 3 should be parsed as (9 + 1) * 3 or as 9 + (1 * 3). As we all know from our high-school algebra, multiplication has a higher precedence than addition. To tell the parser this, we declare * after +.

The only fundamental problem remaining with the above grammar is that it assumes that the end of the input coincides with the end of the line. To make our grammar accept arbitrarily many lines (printing a result at the end of each), we simply add another set of productions to the grammar. Note that only the first rule has an action field and that epsilon stands for the empty string:

```
lines   :lines, exp, '\n' {write(arg2)}
        | lines, '\n'
        | epsilon
```

The rules above ensure that if there is no input (epsilon), nothing gets printed, because lines : epsilon has no action field. If the parser sees an expression and a newline, the parser takes this as an instance of epsilon, plus an expression, plus a newline. This, then, becomes the first component of rule 1 if another expression plus a newline follows, or if just a newline occurs. Every time an instance of rule 1 occurs, the action {write(arg2)} is executed, that is, the value of the expression gets printed. (In actual practice, this is not nearly as arcane as it sounds.)



Note that the goal of our parse no longer is just an exp, but rather lines. lines, that is, is our "start symbol". By default Ibpag2 assumes that the left-hand side symbol of the first rule is the start symbol. This may be overridden with a %start declaration in the tokens section.

With our new, multi-line start symbol in place, the only piece that needs to be added, in order to make our calculator specification a full working input to Ibpag2, is a tokenizer. A tokenizer is a routine that reads input from a file or from some other stream (for example, the user's console), and then segments this input into %tokens its parser can understand. Literal values for tokens are specified by setting the global variable, iilval. For example,

```
iilval := 10
suspend TERM
```

For operators there is no need to set iilval. One simply returns these "literally" as integers (usually via suspend ord(c)).

The tokenizer routine is best $included or appended to the grammar after another double percent sign. Everything after this second double percent sign is copied to the output file. Ibpag2 demands that the tokenizer be called iilex, that the tokenizer take a single file argument, that it be a generator, and that it fail when it reaches end-of-input. Combined with our lines productions, the addition of an iilex procedure to our calculator grammar yields the following Ibpag2 input file:

```
%token TERM
%left '+', '–' %left '*', '/'
%right UMINUS
%start lines
%%
exp  :TERM {return arg1}
     | exp, '+', exp {return arg1 + arg3}
     | exp, '–', exp {return arg1 – arg3}
     | exp, '*', exp {return arg1 * arg3}
     | exp, '/', exp {return arg1 / arg3}
     | '–', exp %prec UMINUS {return – arg2}
lines :lines, exp, '\n' {write(arg2)}
     | lines, '\n'
     | epsilon
%%
```

```
procedure iilex(infile)
 local nextchar, c, num
 nextchar := create !(!infile || "\n" || "\n")
  c := @nextchar | fail
  repeat {
    if any(&digits, c) then
      if not (\num ||:= c) then num := c
    else {
      if iilval := \num then {
      suspend TERM
      num := &null
      }
      if any('+-*/()\n', c) then {
        iilval := c
        suspend ord(c)
        } else {
          if not any(' \t', c) then
            # deliberate error; handled
            suspend &null
        }
      }
      c := @nextchar | break
    }
  if iilval := \num then {
    return TERM
    num := &null
  }
end

procedure main()
  return iiparse(&input, 1)
end
```

If you like, copy the above code into a temporary file (say tmp.ibp) and then feed it to Ibpag2 by typing

```
ibpag2 -f tmp.ibp -o tmp.icn
```

Ibpag2 will turn your grammar specifications and actions into a procedure called iiparse. When Ibpag2 is finished creating its output file (tmp.icn above), compile that file the way you would compile any other Icon program (for example, icont tmp). Finally, run the executable. You should be able to type in various simple arithmetic expressions and have the program spit back answers each time you hit a return.

There are certainly additional issues that might be covered here — for example, error detection, handling, and recovery; debugging; natural languages; efficiency concerns, and many others. These are all covered in the Ibpag2 README file, included as part of the standard Ibpag2 distribution.

Technically, Ibpag2 is still in beta testing. However, it has been in continuous use by me for some months now, and I have completed several substantial projects using it. The most recent version can be obtained either from me at goer@midway.uchicago.edu or from The University of Arizona's RBBS or its FTP server. Ibpag2 should function on any platform that has Icon Version 8.10 with co-expressions. Please report problems to me via e-mail, or by the post:

Richard Goerwitz
5410 S. Ridgewood Ct., 2E
Chicago, IL 60615

Editors' Note: To get Ibpag2 *via FTP, connect to* cs.arizona.edu, cd /icon/contrib, *set binary mode, and* get ibpag2.lzh *or* ibpag2.tar.Z *(compressed tar).* ibpag2.lzh *also is available from our RBBS at the corresponding location.*

Holiday Closing

As a cost-saving measure, The University of Arizona plans to shut down during the holiday period from December 24, 1993 through January 2, 1994.

During this period, there will be no one at the Icon Project to receive telephone calls, faxes, or postal mail. We think electronic mail and FTP will remain in operation, but we can't be sure of this.

If you're planning to place an order for Icon material, we suggest you do so by early December to assure delivery before the holiday period.

Icon for Humanists Out of Print

Alan Corré's book, *Icon Programming for Humanists,* went of out print in August.

We have only five copies of this book left in stock. If you want a copy, we suggest you call us and reserve one.

Downloading Icon Material

Most implementations of Icon are available for downloading electronically:

BBS: (602) 621-2283

FTP: cs.arizona.edu (cd /icon)

From Our Mail

Wow! I didn't know I was going to get so much stuff when I ordered the Icon program library. Neat! But I'm a bit lost. Any hope of a guided tour to help folks like me find the best things?

We're glad to hear you're enjoying the library. We think it's a great resource and one that many Icon programmers overlook. It does contain a lot of material, and we continue to add more material through updates that come out three or four times a year. You might consider subscribing to our update service. As to locating things in the library, you'll find keyword-in-context listings among the documents that come with the library. These should tell you where to start looking. There's also a program called ibrow that lets you look through the library interactively. You'll also find pointers to some of the most useful library material in articles in this *Newsletter* and the Analyst. Still, your point is well taken. We're thinking about other ways to make it easier to locate material in the library. Suggestions along this line are welcome.

The new preprocessor in Version 8.10 of Icon is great. But it sure would be nice to have a few more features like macros with arguments and the ability to define and undefine macros on the command line. Any chance?

We'd also like such features in the preprocessor. The problem is finding the resources to implement them. We're pretty well occupied with other work that has higher priority. We won't say extensions to the preprocessor will never happen, but we have no immediate plans along these lines.

I'm using Icon on a UNIX machine and would like to keep up to date on the source. It looks like your source update service is what I need, but it seems to apply only to MS-DOS.

As mentioned on page 1 of this *Newsletter*, the source code for Icon is the same for all platforms. The problem is distribution format and platform-specific configuration files. We now offer the source updates in both LHarc and compressed tar formats. For UNIX, you'll also need configuration files. If you already have Version 8.10 of the source code for UNIX, you have these files. If you don't, you should first get the 8.10 source and then subscribe to the update service.

I'm forming a local fan club for Icon called the Iconoids. Would you please autograph this letter and send it back so I can show it to my computer buddies?

Uhh, sure ...

One of my friends has an Icon t-shirt with the chessboard from the cover of your book. He says he bought it from a professor in a computer class he took, but he doesn't know where to get another one. I'd really like to have one of these t-shirts; can you tell me where I can get one?

Catspaw, Inc. designed the t-shirt you're talking about. We got some to sell to interested students at The University of Arizona. There aren't any more of these t-shirts left, as far as we know. We've thought about designing a new t-shirt, or even getting coffee mugs with the Icon logo. There are, however, lots of restrictions on commercial activities in an academic context. We're working on it, but we can't give you much hope of seeing anything soon.

In Newsletter 41 you mentioned problems with financing the Icon Project. I did my bit by subscribing to the Analyst. (I should have done this long ago and I now regret what I missed.) How are things going? Will you be able to stay out of bankruptcy?

We appreciate your support and are glad to hear you got something useful in return. We're doing moderately well. Our income has picked up a bit, thanks to persons like yourself. Keeping the *Newsletter* going should be no problem for the immediate future. We just hope we don't have to replace a hard disk drive on one of the Icon Project computers.

How's the book on graphics in Icon coming?

It's coming along, but more slowly than we'd hoped. For one thing, we keep coming up with new features to add to the graphics capabilities of Icon. We think we're near the end of such problems, but we've been wrong about this before. Our goal is to have the book complete in draft in about a year.

How old are you?

The Icon Project is 13 years old. The editors of this *Newsletter* are both 39, like Jack Benny.

Icon in the Classroom

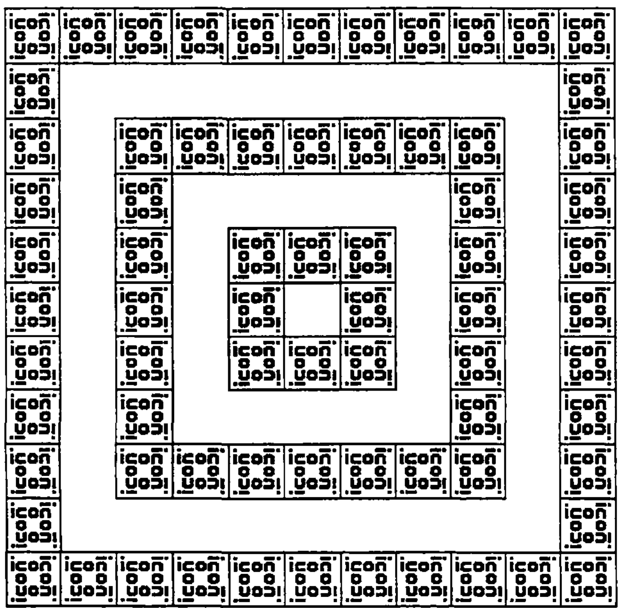
As you might expect, we use Icon in several courses in our Department of Computer Science at The University of Arizona. Here are some comments on our experience.

Comparative Programming Languages

Like many computer science departments, we have an undergraduate course called Comparative Programming Languages. This course is designed to educate students about programming language features and give them some experience with a variety of different languages. At present, the course covers Icon, LISP, Prolog, and Idol (the object-oriented version of Icon). Our experience using Icon in this course has been quite positive. Students find Icon interesting and fun to use. The main problems they encounter are with pointer semantics for structures and with string scanning. In string scanning, the main problem seems to be coming to grips with strings as data values in their own right, as opposed to arrays of characters (most students' prior programming experience is limited to Pascal at this point in their program). These difficulties are not necessarily bad; the course is designed to introduce new ideas.

String and List Processing

We have taught a graduate-level course in string and list processing for over 20 years. About five years ago, an upper-division undergraduate section was added.



This course, which was originally used SNOBOL4 as a programming vehicle, has used Icon for the last 10 years or so. The course is designed to introduce students to programming problems and techniques that involve text and structured data. In many cases, this is an entirely new experience for students, and the high-level features of Icon lead students to use programming paradigms that are quite different from ones in their previous experience. If there's any problem with this course, it's that some students have so much experience with a lower-level language (at this point, it's usually C) that they find it hard to *think* in a new language.

In recent years, students have been given a large project in lieu of a final examination. Each student selects and carries out a project individually. This project approach has worked out well. Students learn about a specific problem area, get significant programming and problem-solving experience, and have the satisfaction of seeing a substantial finished project (at least in most cases).

Two years ago we introduced graphics as an option for projects. Graphics fit nicely with string and list processing, where many applications benefit from the visual display of information.

Despite the added work inherent in using graphics, most students elect this option — 29 of 32 students the last time the course was offered.

Many of the projects have been quite ambitious and several produced first-class results (although only rarely has a project been so robust and well-documented that we've distributed it as part of the Icon program library). It has been evident both in the projects and the demonstrations of them that graphics was a strong motivating factor. In many cases students who used graphics in a significant way became more involved and got deeper into the subject area of their projects than other students.

Compiler Design

Last semester students were given the choice of using Icon or C in parts of a compiler-design course. All but one student chose Icon. The instructor estimates that this student had to write four times the amount of code that students who used Icon did. Most of the additional C code involved implementing structures like sets that are built into Icon.

As in the comparative programming languages course, students new to Icon had trouble with pointer semantics at first, but only because it caught them by surprise. After some experience, students gave Icon very good ratings.

Graphics Programming

The success of using graphics in the string-and-list-processing course has led us to design an undergraduate graphics programming course. The idea of this course is to introduce students to the use of graphics and user interfaces as a regular part of programming; it is not intended to be a computer graphics course or a course on interface design.

Not surprisingly, Icon will be the programming language used. With its high-level graphics facilities, students will not have to deal with the tedious detail that is so discouraging and that so severely limits the coverage of subject matter when a language like C is used.

Students that take this course will have had only a little experience in Icon — perhaps from our comparative programming languages course. What else they need to know about Icon, will be covered in the course.

This course is still in the planning stage and won't be taught until next year.

Icon as a First Programming Language

Every so often, someone raises the issue of using Icon as the programming language in a first course on computing. Our department has considered this possibility when trying to find an alternative to Pascal, which it presently uses.

Factors in favor of Icon are ease of use, being able to get to interesting material quickly, and the possibility of giving students more interesting kinds of assignments than are possible in a language like Pascal.

Arguments against Icon are the lack of an introductory text, whether Icon might "spoil" students who later will have to program in C, the fact that there is little precedent for using Icon as a first programming language, and differing opinions about the technical characteristics a first programming language should have so as to encourage students to approach programming "correctly".

After a lot of discussion, we decided to try Scheme as an alternative to Pascal, but the issue seems far from settled.

Graphics Credits

The images on Page 7 and the back cover were construced with Icon, using the ProIcon icon designed by Mark Emmer. The image on the back cover then was modified in Adobe Photoshop using a filter from Kai's Power Tools.

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. To subscribe, contact

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

voice: (602) 621-8448

fax: (602) 621-4246

Electronic mail may be sent to:

icon-project@cs.arizona.edu

or

...uunet!arizona!icon-project

and

The Bright Forest Company
Tucson Arizona

© 1993 by Madge T. Griswold and Ralph E. Griswold
All rights reserved.

Ordering Icon Material

What's Available

There are implementations of Icon for several personal computers, as well as for CMS, MVS, UNIX, and VMS. Source code for most implementations is available. *Note:* Icon for personal computers requires at least 640KB of RAM; it requires more on some systems.

There also is a program library that contains a large collection of Icon programs and procedures. written in Icon.

Icon Program Material

Icon programs provided by the Icon Project are in the public domain.

All program material is accompanied by documentation in printed and machine-readable form that describes how to install and use Icon. This documentation does not, however, describe the Icon programming language in detail. A book is available separately.

Personal Computers: Executable files and source code are provided in separate packages. Source code for MS-DOS includes the Icon optimizing compiler, configurations for several C compilers, and also OS/2. *Note:* Personal computer distributions are stored in compressed format, and most diskettes are nearly full. It therefore is necessary to have a second drive to extract the material.

CMS: The CMS package contains executable files, source code, test programs, and the Icon program library.

MVS: The MVS package contains object files, source code and test programs, but not source code.

UNIX: The UNIX package contains source code (but not executable files), test programs, related software, and the Icon program library. UNIX Icon can be configured for most UNIX platforms.

VMS: The VMS package contains executable files, source code, test programs, and the Icon program library.

Update Subscriptions: Updates to the Icon source code and the Icon program library are available by subscription.

Source-code updates are distributed on MS-DOS diskettes in LHarc and compressed tar format. Each update normally provides a completely new copy of the source. A source-code subscription provides five updates. Updates are issued about three times a year.

Icon program library updates are available for MS-DOS, the Macintosh, and UNIX. A library subscription provides four updates. Updates are issued three or four times a year.

Documentation

In addition to the installation guides and users' manuals included with the program packages, there are three books on Icon. One contains a complete description of the language, another describes the implementation of Icon in detail, and a third is an introductory text designed primarily for programmers in the Humanities.

There are two newsletters. *The Icon Newsletter* contains news articles, reports from readers, information of topical interest, and so forth. It is free and is sent automatically to anyone who places an order for Icon material. There is a nominal charge for back issues of the *Newsletter.*

The Icon Analyst contains material of a more technical nature, including in-depth articles on programming in Icon. There is a subscription charge for the Analyst.

Payment

Payment should accompany orders and be made by check, money order, or credit card (Visa, MasterCard, or Discover). The minimum credit card order is $15. 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.

Prices

The prices quoted here are good until February 28, 1994. After that, prices are subject to change without further notice. Contact the Icon Project for current pricing information.

Extra Payment

If you wish to support the Icon Project by making an additional payment, a line is provided at the bottom of the order form for this.

Versions

Version information is shown in parentheses. The symbol ✻ identifies recently released material.

Ordering Instructions

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

- [tape symbol] 9-track magnetic tape
- [cartridge symbol] data cartridge
- [5.25" symbol] 5.25" LD diskette
- [3.5" LD symbol] 3.5" LD diskette
- [3.5" HD symbol] 3.5" HD diskette

Tapes are written at 1600 bpi. Cartridges are written in QIC-24 format. 5.25" diskettes are 360K. 3.5" LD diskettes are 720/800K except as noted; HD are 1.44 MB.

Diskettes are written in MS-DOS format except for the Amiga, the Atari ST, and the Macintosh. When ordering diskettes that are available in more than one size, specify the size (the default is shown first). In some cases, there are several diskettes in a distribution.

Shipping Charges: The prices listed include handling and shipping by parcel post in the United States, Canada, and Mexico. Shipment to other countries is made by air mail only, for which there are additional charges as noted in brackets following the prices. For example, the notation $15 [$5] means the item costs $15 and there is a $5 shipping charge to countries other than the United States, Canada, and Mexico. UPS and express delivery are available at cost upon request.

Order Codes: When filling out the order form, use the codes given in the second column of the list to the right (for example, DE, MSM, ...).

Item	Code	New	Media	Price	Shipping
Executables					
Acorn Archimedes (8.0)	ARE		3.5" LD or 5.25"	$15	[$5]
Amiga (8.0)	AME		3.5" LD	$15	[$5]
Atari ST (8.0)	ATE		3.5" LD[1]	$15	[$5]
MS-DOS (8.10)	DE		3.5" LD or 5.25"	$15	[$5]
MS-DOS 386/486 (8.10)	DE-386		3.5" LD or 5.25"	$15	[$5]
Macintosh (8.0)	MET		3.5" LD	$15	[$5]
Macintosh/MPW (8.10)	MEM	✻	3.5" LD	$15	[$5]
OS/2 (8.10)	OE		3.5" LD	$15	[$5]
Source					
Amiga (8.0)	AMS		3.5" LD	$15	[$5]
Atari ST (8.0)	ATS		3.5" LD	$15	[$5]
MS-DOS & OS/2 (8.10)	DS		3.5" LD or 5.25"	$30	[$5]
Macintosh (8.0)	MST		3.5" LD	$15	[$5]
Macintosh/MPW (8.10)	MSM	✻	3.5" LD	$25	[$5]
Updates, LHarc (5)	SU-L		3.5" LD or 3.5" HD	$60	[$15]
Updates, comp. tar (5)	SU-T		3.5" HD	$60	[$15]
Complete Systems					
CMS (8.0)	CT		tape	$30	[$10]
MVS (8.8)	MT	✻	tape	$30	[$10]
UNIX (8.10)	UD		3.5" HD	$25	[$5]
UNIX (8.10)	UT		tape	$30	[$10]
UNIX (8.10)	UC		cartridge	$45	[$10]
VMS (8.10)	VT		tape	$32	[$11]
Program Library					
MS-DOS (8.10)	DL		3.5" LD or 5.25"	$15	[$5]
Macintosh (8.10)	ML		3.5" LD	$15	[$5]
UNIX (8.10)	UL		3.5" LD or 5.25"	$15	[$5]
MS-DOS updates (4)	LU-D		3.5" LD or 5.25"	$30	[$12]
Macintosh updates (4)	LU-M		3.5" LD	$30	[$12]
UNIX updates (4)	LU-U		3.5" HD	$30	[$12]
Books					
The Icon Programming Language	LB			$40	[$13]
The Implementation of Icon + update	IB			$53	[$14]
Icon Programming for Humanists + diskette	HB			$38	[$10]
Newsletters					
The Icon Newsletter (complete, 1-42)	INC			$19	[$5]
The Icon Newsletter (back issues, each)	INS			$1	[$2[2]]
The Icon Analyst (1 year, 6 issues)	IA			$25	[$10]
The Icon Analyst (first three years, 1-18)	IAC			$60	[$3]
The Icon Analyst (back issues, each)	IAS			$5	[$2[2]]

[1] 400K

[2] Per order, regardless of the number of issues purchased.

Order Form

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

Ordering information: (602) 621-8448 • Fax: (602) 621-4246

name ____________________

address ____________________

____

city ____________________ state ________ zipcode ____________

(country) ____________________ telephone ____________________

❒ check if this is a new address

qty.	code	description	price	shipping*	total
	XP	Support for the Icon Project			

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.

❒ Visa ❒ MasterCard ❒ Discover ❒ check or money order

I hereby authorize the billing of the above order to my credit card: ($15 minimum)

card number ❒❒❒❒❒❒❒❒❒❒❒❒❒❒❒❒ exp. date ❒❒❒❒

name on card (please print) ____________________

signature ____________________

MasterCard VISA DISCOVER

*Shipping charges apply only to addresses outside the United States, Canada, and Mexico