THE UNIVERSITY OF ARIZONA

TUCSON, ARIZONA 85721

DEPARTMENT OF COMPUTER SCIENCE

Icon Newsletter #12

Ralph E. Griswold

July 14, 1983

The Icon Compiler Versus the Interpreter

Version 5 of Icon has both a compiler and an interpreter. The two are compatible with respect to source programs, with a few exceptions. Only the compiler allows external functions to be declared. Such functions can be used to add to the built-in repertoire of Icon. On the other hand, there are a few experimental extensions that are only implemented in the interpreter.

Many users assume that the compiler is much faster than the interpreter and it is common when Icon is brought up at a new site for only the compiler to be installed.

In fact, the compiler is only slightly faster in execution than the interpreter — 5% to 10% is typical. On the other hand, the interpreter bypasses the assembly and loading phases of the compiler and hence gets into execution much more quickly. The interpreter typically is about five times faster than the compiler in this respect.

Other considerations are that the executable files produced by the interpreter generally are much smaller than those produced by the compiler. On the other hand, interpreted files generally take more memory to run than compiled files.

For most programmers, however, the main consideration in program development and debugging in an interactive environment is the speed in getting into execution. Therefore the interpreter is generally preferable to the compiler, unless external functions are needed. Furthermore, most programs developed under the interpreter can be compiled to get better execution speed.

Icon Program Library

We have developed a library of Icon programs, procedures, and external functions. The programs range from utilities to games. The procedures include, among other things, operations on structures, programmer-defined control operations, and SNOBOL4-style pattern matching. The external functions include UNIX-interface operations, mathematical routines, and so forth.

The Icon Library is documented in TR 83-6 and the library itself is included in the Version 5.8 distribution of Icon as described below.

Transporting Icon to UNIX Environments

Version 5 of Icon presently is available for the PDP-11, VAX-11, and Onyx C8002 operating under UNIX*. There are many other computers that support UNIX, and their number is increasing rapidly.

Bill Mitchell has done extensive work preparing Version 5 of Icon for transporting to UNIX environments and has documented the porting process in detail in TR 83-10.

Programs to support the porting of Icon are available with the 5.8 distribution.

*UNIX is a trademark of Bell Laboratories.

Version 5 of Icon for the Onyx C8002

Recently we have had several inquiries about Icon for the Onyx. Here is the latest information:

Orders should be addressed to

John D. Polstra & Co., Inc.
4522 S.W. Andover Street
Seattle, Washington 98116

Payment in the form of a check for $75.00, payable to 'John D. Polstra & Co., Inc.' must accompany the order. The software is distributed on a standard Onyx cartridge tape (Scotch DC-300A or equivalent) and includes all source as well as a set of binary files ready for installation. Instructions for installing the software are included. The price of the system covers the cost of the tape, printed materials, first class postage, and handling. Language manuals and related Icon technical reports are not included. There are no restrictions on the software; persons are welcome to sell or give away copies as they see fit.

Version 5.8 of Icon

Version 5.8 of Icon for the PDP-11 and VAX-11 is now available. This version corrects a few bugs in Version 5.7 and adds a link directive to make it more convenient to include separately translated procedures in programs.

As mentioned above, the Version 5.8 distribution contains the Icon Library and support for porting Icon to other UNIX environments.

Use the form at the end of this Newsletter to request Version 5.8.

Status of Version 5 of Icon for VAX/VMS

The VAX/VMS implementation of Version 5 of Icon has taken longer than anticipated. However, substantial work has been done by one person and another person has now taken over the project.

While we are hopeful that this system will be finished, we can make no promise of a completion date. Several persons have already sent us tapes for the VMS version, which we are still holding. However, we ask others *not* to send tapes at this time.

When the VMS implementation of Version 5 of Icon is available, we will send out a notice promptly.

Icon Discussion Group

We have established an electronic mail group for the discussion of topics related to Icon. Participation in this group is available to anyone with access to CSNET. Simply send your request to

icon-project.arizona@rand-relay

Version 2 Versus Version 5 of Icon

Version 2 of Icon is available on several computers and operating systems for which Version 5 has not been implemented. Although Version 2 differs from Version 5 in many respects, it still embodies the fundamental language features that characterize Icon.

Since the Icon book describes Version 5, users of Version 2 only have a reference manual to rely on. To alleviate this situation, a document describing the differences between the two versions has been prepared. This document, TR 83-5, is keyed to the Icon book. It points out the detailed differences between the two versions and also includes the procedures from the Icon book translated into Version 2.

Recent Icon Documents

New documents include the technical reports mentioned in the preceding sections, a list of all known implementations of Icon, and reprints of two recent journal articles. See the request form at the end of this Newsletter.

Anyone who is interested in receiving this Newsletter is welcome to do so; just write and ask to be placed on the Icon mailing list. Back issues of Newsletters #2 through #11 are available on request.

Programming Corner

Solutions to the Problems in Newsletter #11:

N-Queens: Numerous solutions to the 8-queens problem have been presented to show different backtracking techniques. Icon lends itself nicely to such problems as illustrated by the following program*:

```
procedure main()
   write(q(1), q(2), q(3), q(4), q(5), q(6), q(7), q(8))
end

procedure q(c)
   suspend place(1 to 8, c)          # look for a row
end

procedure place(r, c)
   static up, down, row
   initial {
      up := list(15, 0)
      down := list(15, 0)
      row := list(8, 0)
      }
   if row[r] = down[r + c - 1] = up[8 + r - c] = 0
   then suspend row[r] <- down[r + c - 1] <-
      up[8 + r - c] <- r             # place if free
end
```

The heart of this solution lies in the mutual goal-directed evaluation of q(1), q(2), ..., q(8). Each set of values for which these procedure calls mutually succeed corresponds to a solution. Note that all the queens are 'equal'.

Clearly, however, this type of solution does not lend itself to parameterization, since each of the q(i) must appear explicitly in the program.

One approach to the general problem is to use a list to contain the solutions and a hierarchical, recursive scheme in which each queen controls the invocation of the next one. Thus the first queen dominates the second, and so on. A program of this kind, due to Steve Wampler, is

```
global n, solution

procedure main(x)
   n := x[1]                         # number of queens
   solution := list(n)               # list of column solutions
   every q(1)                        # start by placing queen in first col.
end
```

*Ralph E. Griswold and Madge T. Griswold, *The Icon Programming Language*, © 1983, page 153. Reprinted by permission of Prentice–Hall, Inc., Englewood Cliffs, New Jersey.

```
procedure q(c)
   local r
   static up, down, rows
   initial {
      up := list(2 * n - 1, 0)
      down := list(2 * n - 1, 0)
      rows := list(n, 0)
      }
   every 0 = rows[r := 1 to n] = up[n + r - c] = down[ r + c - 1] &
      rows[r] <- up[n + r - c] <- down[r + c - 1] <- 1 do {
         solution[c] := r           # record placement
         if c = n then show()       # all queens placed, show positions
         else q(c + 1)              # try to place next queen
         }
end

procedure show()
   every writes(!solution)
   write()
end
```

Here the value of *n* is supplied on the command line when the program is run. The version of the program given here is stripped down to conserve space. The complete program with error checking and a display of the queens on a chessboard is included in the Icon library.

This approach can also be cast in terms of co-expressions, as illustrated by the following program by Steve Wampler:

```
global n, nthq, solution

procedure main(x)
   local i
   n := x[1]
   nthq := list(n + 2)              # list of queens
   solution := list(n)              # list of solutions
   nthq[1] := &main                 # 1st queen is main routine
   every i := 1 to n do             # 2 to n + 1 are real queen placements
      nthq[i + 1] := create q(i)    # one co-expression per column
   nthq[n + 2] := create show()     # n + 2nd queen is display routine
   @nthq[2]                         # start by placing queen in first col.
end
```

```
procedure q(c)
   local r
   static up, down, rows
   initial {
      up := list(2 * n - 1, 0)
      down := list(2 * n - 1, 0)
      rows := list(n, 0)
      }
   repeat {
      every 0 = rows[r := 1 to n] = up[n + r - c] = down[r + c - 1] &
         rows[r] <- up[n + r - c] <- down[r + c - 1] <- 1 do {
            solution[c] := r
         @nthq[c + 2]               # try to place next queen
         }
      @nthq[c]                      # tell last queen try again
      }
end

procedure show()
   repeat {
      every writes(!solution)
      write()
      @nthq[n + 1]                  # tell last queen to try again
      }
end
```

A somewhat different form of solution — and one that requires a better understanding of Icon — is illustrated by the following solution to the *n*-rooks problem by Tom Slone.

```
global n

procedure main(x)
   n := x[1]                        # number of rooks
   every show(rook(n))
end

procedure show(a)
   every writes(!a)
   write()
end

procedure rook(i)
   static a
   initial a := list(n)
   if i = 0 then return
   if i < n then
      suspend (a[i] := r()) & rook(i - 1)
   else suspend (a[i] := r()) & rook(i - 1) & a
end

procedure r()
   suspend place(1 to n)
end
```

```
procedure place(r)
   static col
   initial col := list(n, 0)
   if col[r] = 0 then suspend col[r] <- r
end
```

A Puzzle: In the last Newsletter, the behavior of the following program was posed:

```
procedure main()
   write(
      "abcde" ? {
         p() := "x"
         write(&subject)
         &subject
         }
      )
end

procedure p()
   suspend &subject[2:3]
end
```

In the first place, the argument of write in main is a scanning expression. The scanning expression consists of three sub-expressions. The first is a call to p that returns a substring of &subject. Since this is a substring of a global variable (&subject), it is not dereferenced when p returns, and the subsequent assignment changes the value of &subject to axcde, which is written. The last sub-expression of the scanning expression is simply &subject; this is the result of the scanning expression and hence the argument of the outer write. Since &subject is a global variable, it is not dereferenced when it is produced by the scanning expression. Hence write gets the *variable* &subject. However, when the scanning expression returns, it restores the former value of &subject (in this case, the empty string, the initial default value of &subject). So, by the time write dereferences &subject, its value is the empty string. Thus this program writes axcde followed by an blank line!

This example illustrates two things: (1) dereferencing is a serious language design problem, and (2) programmers may encounter some mysterious results if they write programs that rely on the side effects of assignment to global variables like &subject.

Superficially, this program looks like it ought to write axcde twice (actually, the write in the scanning expression got there as a result of trying to find out why the outer write produced a blank line). The problem can be solved by explicitly dereferencing &subject in the scanning expression, as in

```
procedure main()
   write(
      "abcde" ? {
         p() := "x"
         write(&subject)
         .&subject
         }
      )
end
```

Now the value returned by the scanning expression is axcde — the value of &subject before its former value is restored at the end of scanning.

New Problems:

Scanning an Entire File: Since Icon has a string data type and many operations on strings, it is natural to process strings as whole objects, rather than character by character as in most lower-level languages. String scanning raises the operations on strings to an even higher level, providing a subject on which scanning functions operate, matching functions that move the position of attention in the subject, and backtracking to the previous position in case a matching function fails.

One annoying problem in trying to process strings as whole objects occurs in situations like lexical analysis, in which the object to be processed is really a file with interspersed line terminators. The result of read in Icon is a string consisting of a line up to a line terminator. Therefore it is natural to process files in terms of their lines. However, a construction to be processed may span several lines (comments are typical). Thus the natural input to string scanning may not contain the whole string to be processed. Although it is possible to make the entire input into a single string, this is generally inefficient and often impractical.

There is a device that often can be used to overcome these problems. If an expected match fails (for example, searching for the closing token of a comment), the subject of scanning can be reset *during* scanning to be the next line of input. This is illustrated by the following program by Tom Slone for stripping 'white space' out of Pascal programs:

```
procedure main()
   local line
   while line := read() do
      line ? {
         remwhite()
         write("" ~== tab(0))
         }
end

procedure remwhite()
   while tab(many(' \t')) | (="(*" & comment()) | (pos(0) & newline())
end

procedure comment()
   while not ="*)" do
      if pos(0) then newline()
      else move(1) & tab(many(~'*'))
   return
end

procedure newline()
   (&subject := read()) | stop("unexpected end-of-file.")
   return
end
```

This technique is applicable only if the part of the subject already scanned can be discarded — once a new value has been assigned to the subject, the old value of the subject is lost and the automatic backtracking in string scanning does not apply to it.

The program above is not complete — it has at least two defects. Correcting these is left as an exercise.

Random Numbers: A linear congruence method is used for generating pseudo-random numbers in Icon. There is only one sequence, which is used for producing random integers, real numbers, and elements of strings and structures. Thus the operation ?x is valid for values of x of different types. The pseudo-random sequence is computed from the value of &random, which is initially zero and which changes with each use of ?x. &random also can be set, so that a sequence can be repeated or started at an arbitrary place. A problem arises, however, if two or more independent random sequences are needed — the use of one inevitably affects the other — or does it?

For starters, write a procedure random() whose result sequence consists of the successive values of &random as they are produced by successive uses of ?x (note that is it not necessary to know how Icon actually performs the pseudo-random computation). Assume that there is no other use of random generation in the program in which random() is used.

Now remove the assumption, so that the result sequence produced by random() is not affected by uses of ?x elsewhere in the program.

Request for Icon Documents

Please send the documents checked below to:

____

____

____

____

____

☐ *Implementations of Icon*
☐ *Differences Between Versions 2 and 5 of Icon*, TR 83-5
☐ *The Icon Program Library*, TR 83-6
☐ *Porting the UNIX Implementation of Icon*, TR 83-10
☐ Co-Expressions in Icon, reprinted from *Computer Journal*
☐ Programmer-Defined Control Operations, reprinted from *Computer Journal.*

Return this form to:

Icon Project
Department of Computer Science
University Computer Center
The University of Arizona
Tucson, Arizona 85721
U.S.A.

Request for Version 5.8 of Icon for UNIX

Note: This system can be configured for either PDP-11s with separate I and D spaces or VAX-11s. The Icon program library is included.

☐ Check here if you also want test programs related to transporting Icon.

Contact Information:

name: ______________________________

address: ______________________________

____

____

____

telephone: ______________________________

electronic mail address: ______________________________

cable/telex: ______________________________

All magnetic tapes are written in 9-track *tar* format.
Please specify your preferred tape recording density:

☐ 1600 bpi ☐ 800 bpi

Return this form to:

Icon Project
Department of Computer Science
University Computer Center
The University of Arizona
Tucson, Arizona 85721
U.S.A.

Enclose a magnetic tape (at least 600') or a check for $15.00 payable to the University of Arizona.