THE UNIVERSITY OF ARIZONA

TUCSON, ARIZONA 85721

DEPARTMENT OF COMPUTER SCIENCE

Icon Newsletter #13

Ralph E. Griswold

August 31, 1983

Version 5 of Icon for VMS

The Version 5 Icon interpreter has been implemented for VAX/VMS by Bob Goldberg of Multicomputing, Inc. Thanks go to Steve Brown, president of Datalogics, Inc. in Chicago for his contribution of system resources to this project. This VMS implementation is being distributed by the University of Arizona. The distribution includes executable binaries, object modules, com files, sample programs, and the procedures from the Icon book. Source code is not provided. (Persons interested in seeing how Icon is implemented can get source code of the UNIX* implementation as described in previous Newsletters.)

The VMS implementation should run on VMS Version 3.2 and later systems and on VAX 730s, 750s, and 780s, although installations to date have been limited to VMS Versions 3.2 and 3.3 on 780s.

To get a copy of this system, use the request form at the end of this Newsletter.

Programming Corner

Random Numbers: In the last Newsletter, the question of independent random sequences was raised. One of the problems posed there was to write a procedure random() whose result sequence consists of the successive values of &random as it changes as a side effect of the evaluation of ?x, assuming there are no other uses of random generation in the program in which random() is used. This is easily done, and such a procedure is

```
procedure random()
   repeat {
      suspend &random
      ?0
      }
end
```

The only point here is the observation that each evaluation of ?0 (or any other valid form of ?*e*) changes &random.

Now the problem is how to achieve this same sequence of results if there are other occurences of ?*e* in the program. Some method of remembering the value of &random is needed. This is not hard to do in a procedure, but it requires a little care. A solution is

*UNIX is a trademark of Bell Laboratories.

```
procedure random()
   local i
   suspend i := 0
   repeat {
      &random :=: i
      ?0
      i :=: &random
      suspend i
      }
end
```

Here, it is necessary to know that the initial value of &random is 0, since there may be random generation before random() is called. The local identifier i keeps track of the value of &random, which may be changed between suspensions and resumptions of random(). The values of &random and i are exchanged so that random() does not affect other uses of ?*e* — independence works two ways.

So far, so good. But what about an *expression* that generates the successive values of &random without using a procedure? The procedure provides a loop from which values are produced, but there is no corresponding expression form in Icon. For example, there is no way to deliver a value out of a **while** or **every** loop. To do this with an expression requires rephrasing the procedural solution as a mutual evaluation expression, in which backtracking is used to assure that &random has the correct value on each resumption. Such an expression is

```
    (i := 0) | (i :=: &random, |?0, i <-> &random)
```

The first expression in the alternation sets i to the initial value of &random which is known to be 0, and also produces this as the first value of the whole expression. The second expression in the alternation is a mutual evaluation of three expressions. The first of these expressions exchanges the values of i and &random, so that i now contains whatever value &random had outside the expression and &random is properly initialized. The second expression changes the value of &random (the reason for repeated alternation will become apparent in a moment). Next, the values of i and &random are exchanged again. Since the exchange operation returns its left argument, the value of the entire mutual evaluation expression is the desired result.

The interesting aspect of this expression occurs when it is resumed to produce another value. The last expression, the reversible exchange, is resumed. This causes the values of i and &random to be restored to what they were before the reversible exchange was evaluated. Specifically, &random is restored to the value it had after the preceding evaluation of |?0, regardless of what happened while the mutual evaluation expression was suspended. Since the reversible exchange does not produce a second result when it is resumed, but only reversed the exchange, |?0 is resumed next. Here the reason for repeated alternation is evident — it always produces another result and as a side effect advances the random sequence. With this new result, the reversible exchange is *evaluated* again to produce the next value of &random, and so on. Note that the first expression in the mutual evaluation is never resumed; it serves only as initialization and the repeated alternation provides a barrier to backtracking, since it always produces another result.

Admittedly, this mutual evaluation expression is somewhat arcane. However, once the concepts are grasped, such techniques become idiomatic.

Getting away from the problem of independently generating the values of &random, which was chosen only to make the problem simple, there are cases in which other independent random sequences are useful. One is a random "production/consumption" kind of process. This is illustrated in the following program, which creates randomly positioned stars on a terminal screen, building up an initial field of stars, after which the oldest stars are destroyed in the order in which they were created. Finally, creation ceases and destruction continues until all the stars are gone. To accomplish this, two identical sequences of co-ordinate positions are used, one for creation, and one for destruction. Creation is started up first and allowed to proceed until the destruction process is started. There is no need to provide storage for the co-ordinate positions, since this is done in the expressions as described above. Co-expressions are used so that the creation and destruction of stars can be controlled. The program is:

```
procedure main(x)
   local i, j, r, ran1, ran2
   i := x[1] | 10                      # time for creation/destruction (default 10)
   j := x[2] | 50                      # steady state time (default 50)
   r := 0
   ran1 := create (r := 0, &random :=: r, rplot("*"), &random <-> r)
   ran2 := create (r := 0, &random :=: r, rplot(" "), &random <-> r)
   clear()                             # clear the screen
   every 1 to i do @ran1               # create the universe
   every 1 to j do {                   # steady state condition
      @ran2
      @ran1
      }
   every 1 to i do @ran2               # destroy the universe
   home()                              # home the cursor (screen is clear)
end
```

Note that the times for startup/destruction and the steady-state times can be provided optionally as command line arguments. The identifiers r in the co-expressions for ran1 and ran2 are distinct, since the creation of a co-expression creates independent copies of local identifiers.

The procedures rplot(s), clear(), and home() are terminal-dependent. The last two clear the screen and home the cursor, respectively. The interesting routine is rplot(s), which is a generator that on successive resumptions writes s at randomly selected spaces on the screen. For the DataMedia 3025, rplot is:

```
procedure rplot(s)
   static row, col
   initial {
      row := string(&cset[33+:24])
      col := string(&cset[33+:80])
      }
   suspend |writes("\^[Y", col[?80], row[?24], s)
end
```

Question: Can the repeated alternation in

```
   suspend |writes("\^[Y", col[?80], row[?24], s)
```

be placed at any other position other than in front of writes?

Acknowledgement

Independent random sequences of the kind described above were used by Lorraine Callahan in a C program on which the Icon program library game *kaleid(6)* is based. Steve Wampler wrote *kaleid(6)* and also *worm(6)*. Steve Wampler suggested this problem for the Programming Corner.