THE UNIVERSITY OF ARIZONA

TUCSON, ARIZONA 85721

DEPARTMENT OF COMPUTER SCIENCE

Icon Newsletter #6

Ralph E. Griswold

May 1, 1981

1. Portable Icon

1.1 Implementations

As mentioned in earlier Newsletters, there are running implementations of Version 2 of the portable Icon system for the DEC-10, the CDC Cyber/6000, the IBM 370 and compatible computers, the VAX-11/780, and the CRAY-1. The implementation for the ICL 2900, mentioned in the last Newsletter, is now also complete. The portable system and the CDC Cyber/6000 and DEC-10 implementations are still available from us. Request forms are contained in Newsletter #5.

An implementation for the PRIME 400 has recently been completed. For information, contact

Dr. V. J. Rayward-Smith
School of Computing Studies and Accountancy
University of East Anglia
Norwich NR4 7TJ
England

Claude Finn has completed an implementation of Version 1.3 on the Data General MV8000. He comments

I have now written about 5 medium sized programs (50-200 lines each), doing research on the syntax and grammar of on-line documentation. The power of Icon doing this research can be seen by the following comparison: My first Icon program (58 lines long) replaced a 1250 line PL/I program. In defense of PL/I, however, it must be pointed out that about 600 lines of PL/I are usable in other programs which do syntax scanning. Thus, my calculated "density ratio" is about 10:1 in favor of Icon over PL/I for this particular exercise.

Finn plans to update his system to Version 2.0 and then make it available to interested users. If you are interested, contact him:

Mr. Claude Finn
Principal Member Technical Staff
Data General Corporation
Route 9
Westboro, Massachusetts 01581

1.2 Word-Size Limitations

When we designed the portable implementation of Icon, we hoped that it could be made to run on 16-bit machines. We now know that expectation was unrealistic. One of the problems is the limitation that 16-bit addresses place on the size of Icon's various data regions. Real arithmetic and some internal computations also present problems on a 16-bit machine. If anyone who has tried to implement Icon on a 16-bit machine has found a way around these problems, we would be happy to stand corrected.

1.3 Updated Corrections to Version 2

We have recently made a number of corrections to the Version 2.0 source program. Most of these relate to the portable aspects of the system. A list of the new corrections is available. See the document request form at the end of this Newsletter.

2. The UNIX Implementation of Icon

2.1 Version 3

Version 3.2 of Icon for UNIX systems is still being distributed. The list of known bugs in this version has been updated recently. See the document request form at the end of this Newsletter.

Paul Eggert at the University of California has converted Version 3 to run on a VAX-11 780 under Berkeley UNIX. For those interested, his address is

Mr. Paul Eggert
Department of Computer Science
University of California
Santa Barbara, California 93106

2.2 Version 4

The inevitable "next version" is nearly done. Version 4, the successor to Version 3 for UNIX, is near completion and should be ready for distribution some time this summer (its availability will be announced in the next Newsletter).

Version 4 contains quite a number of differences from Version 3 more differences than there have been between earlier versions. The major differences are:

- Limitations on goal-directed evaluation have been removed from a number of control structures in order to provide more uniform and general evaluation of expressions. In addition, new control structures have been added to allow generators to be used in a more flexible manner.
- Co-expressions (the approximate equivalent of co-routines on the expression level) have been added to allow generators to be encapsulated and hence activated at any time and place in a program rather than only at the site where they appear. Co-expressions and some of the new control structures in Version 4 are described in TR 81-1. See the document request form at the end of this Newsletter.
- List and stacks have been unified into a single data structure that can be accessed as a list, stack, or queue.
- The null value is no longer convertible to other types and is illegal in most computations. This allows detection of the use of uninitialized variables.

3. Current Research

3.1 Sequences and Expression Evaluation

Some expressions in Icon, such as `x + y`, produce a single result and correspond to conventional computational expressions found in most programming languages. Other expressions, such as `x < y`, may not produce any result and correspond to conditional expressions in SNOBOL4. An expression such as `find(s1,s2)` may produce several results and is called a *generator*. Generators, of course, subsume ordinary computational and conditional expressions.

Considerable insight into expression evaluation in Icon may be obtained by considering the sequence of results that expressions may produce and how such sequences relate to goal-directed evaluation and the control structures of Icon.

For example, the sequence of results that may be produced by

```
find("th", "this is the thesis")
```

is {1,9,13}. The results that are actually produced by such an expression depend, of course, on the context in which it is evaluated. In general

every *e*

forces *e* to produce its entire sequence.

The value of using sequences to describe control structures is illustrated by alternation:

```
e1 | e2
```

This control structure simply produces the sequence produced by *e1* followed by the sequence produced by *e2*.

A notation for characterizing result sequences and results of investigating this characterization of expression evaluation are contained in TR 81-2. See the document request form at the end of this Newsletter.

3.2 Models of String Pattern Matching

An earlier report (TR 80-25) described how SNOBOL4-style pattern matching might be implementated in Icon.

The ideas in that work now have been developed more fully and provide the basis for various models of string pattern matching. The results of this research are described in TR 81-6 and conclude with some suggestions for design of an Icon-like language with a pattern-matching facility. See the document request form at the end of this Newsletter.

3.3 Generators in C

In the last Newsletter, a project to add Icon-style generators to the C programming language was described briefly. The results of that work have been encouraging and an implementation of a full-blown preprocessor to translate "C-with-generators" (Cg) into standard C is underway. The intent is to use Yacc for the preprocessor, with semantic actions simply transcribing the source program intact except where generator constructs appear.

The runtime system to support generators in C is complete and as soon as the preprocessor and documentation are finished, the system will be made available to interested persons.

4. Programming Corner

4.1 An Idiom

Every programming language has a number of particularly apt idioms. Consider the expression

x [illegible] x

At first sight, this expression appears to be a curiosity. However, when used in a conjunction expression, it serves as a stack with automatic pushing of the value of x when it is evaluated and automatic popping of the value of x during backtracking. Thus in

e1 & (x [illegible] x) & *e2*

if *e1* succeeds, the value of x is pushed and *e2* is evaluated. If *e2* fails, the value of x is popped and *e1* is reactivated.

In situations in which several expressions are connected by conjunction to obtain the first-in, last-out sequencing provided by goal-directed evaluation, this reversible-assignment idiom is both concise and (once it is understood) clearly indicates its purpose.

4.2 Solutions to Questions Posed in Newsletter #5

In the programming corner of Newsletter #5, several programming questions were posed. These questions are restated below with their answers. Asterisks indicate material for Version 3 only.

Problem 1:

Q: What is the output produced by each of the following expressions?

```
(a)    every write((0 | 0) to 7)
(b)    every write(0 to 3,0 to 7,0 to 7)
(c)    every write(1 | 2 to 3 | 4 by 1 | 2)
(d)    every 1 to 3 do every write(1 to 3)
```

A: These expressions illustrate the use of every to force generators through all their results. The left-to-right, last-in first-out order of results is shown by the output below. Ellipses are used to compress long sequences where the output follows an obvious pattern.

```
(a)     0       (b)     000     (c)     1       (d)     1
        1               001             2               2
        2                .              3               3
        3                .              1               1
        4                .              3               2
        5               076             1               3
        6               077             2               1
        7               100             3               2
        0               101             4               3
        1                .              1
        2                .              3
        3                .              2
        4               176             3
        5               177             2
        6               200             2
        7               201             3
                         .              4
                         .              2
                         .              4
                        276
                        277
                        300
                        301
                         .
                         .
                         .
                        376
                        377
```

Problem 2:

Q: For arbitrary procedures f(x,y) and g(x,y), what is the sequence of calls produced by

```
every (f | g)(1 to 3, 4 | 5)
```

A:

```
f(1,4)
f(1,5)
f(2,4)
f(2,5)
f(3,4)
f(3,5)
g(1,4)
g(1,5)
g(2,4)
g(2,5)
g(3,4)
g(3,5)
```

Problem 3*:

Q: Given

```
s1 := "aeiou"
s2 := "abecaeioud"
```

what are the outcomes of

```
(a)     (find | upto)(s1,s2)
(b)     (find | upto)(s2,s1)
(c)     (if size(s1) < size(s2) then upto else find)(s1,s2)
```

A: The answer to this question illustrates that functions are data objects in Version 3 and that function application involves applying the value of a (function-valued) expression, such as (find | upto). In addition, goal-directed evaluation applies to such expressions themselves. In fact, an expression such as *e0*(*e1*, ..., *en*) involves the mutual goal-directed evaluation of *e0*, *e1*, ..., *en* in which the value of *e0* is applied to *e1*, ..., *en*. The outcomes for the expressions above are

(a)	5
(b)	1
(c)	5

Problem 4:

Q: What are the outcomes of the following expressions? (Note any that produce errors.)

```
(a)     (x | y) := 3
(b)     (x & y) := 3
(c)     (1 & x) := 3
(d)     (x & 1) := 3
(e)     (x + 1) := 3
```

A: The term *outcome* is used in the technical sense here. As indicated, the outcomes of the first three expressions are variables, since assignment returns its left operand as a variable.

(a) x (assigned the value 3)
(b) y (assigned the value 3)
(c) x (assigned the value 3)
(d) *error* (variable expected)
(e) *error* (variable expected)

Problem 5:

Q: Given the procedure

```
procedure drive(x)
  fail
end
```

What is the output produced by

(a) `drive(write(1 to 7))`
(b) `drive(write(0 to 7,0 to 7))`

A: This problem illustrates the relationship between goal-directed evaluation and the control structure **every**, which forces generators to produce all their results. The same effect can be produced by a procedure that only fails, hence forcing goal-directed evaluation to produce all the results of its argument.

```
(a)   1        (b)   00
      2              01
      3              02
      4              .
      5              .
      6              .
      7              06
                     07
                     10
                     11
                     12
                     .
                     .
                     .
                     .
                     .
                     .
                     66
                     67
                     70
                     71
                     72
                     .
                     .
                     .
                     76
                     77
```

Problem 6*:

Q: What does the execution of the following program do?

```
procedure main()
   f(f :  write)
end
```

(Note: this program is slightly different from that given in Newsletter #5, where a second argument to f was accidentally included.)

A: This one is tricky. Since the program has no procedure declaration for f, one might suppose the execution of the program is an error. Recall Problem 3 above, however, noting that function and procedure applications are evaluated the same way. Furthermore, in Version 3, variables are not dereferenced until all the arguments are evaluated. This applies to the "zeroth" argument, which is the function or procedure to be applied. Evaluation of the first argument assigns a function value to f (i.e., the value of write). Hence this expression is equivalent to write(write) and produces the output

```
function write
```

(The form of the output is a consequence of "imaging" a non-string value for the purposes of output. This is the same imaging that is used in tracing procedure calls.)

Problem 7:

Q: The following procedure is proposed as a generator of "words" strings of consecutive letters in the lines of the input file. It does not work properly, however. What does it actually do and what are the causes of the problems? Rewrite the procedure to work properly.

```
procedure genword()
   local line
   static letters
   initial letters :  &lcase ++ &ucase
   while line :  read() do
      scan line using
         while tab(upto(letters))
            do suspend tab(many(letters))
end
```

A: There are two things wrong with this procedure, one more subtle than the other. The less subtle error is that the procedure simply terminates after the while loop. In this case, the procedure would return a final null value a spurious result following the words it is supposed to produce. There should be a fail before the end.

A more subtle problem lies in the suspend expression itself. suspend is like every it forces its argument to generate all its results. Although neither tab nor many have alternative results, tab does restore the value of &pos if it is re-activated to produce a second result. Hence, &pos is always restored to its position prior to the first word and this procedure loops, continually returning the first word of the first line of input!

There are two ways of circumventing this problem: use of an auxiliary identifier or explicitly preventing generation of alternatives in the suspend expression, and hence the backtracking done by tab. Thus the while loop can be rewritten as

```
while tab(upto(letters)) do {
  t :  tab(many(letters))
  suspend t
  }
```

or

```
while tab(upto(letters)) do
   suspend {tab(many(letters))}
```

Incidentally, this problem is sufficiently insidious that it deserves attention in the design of Icon. The subtlety of the problem lies in the fact that, except for reversible assignments, Icon does data backtracking only in tab and move.

Request for Icon Documents

Please send the documents checked below to:

____

____

____

____

____

☐ Corrections to Version 2 of Icon (updated April, 1981)

☐ Corrections, Changes, and Known Bugs Related to Version 3.2 of Icon (updated April, 1981)

☐ New Control Structures for Icon, TR 81-1

☐ Sequences and Expression Evaluation in Icon, TR 81-2

☐ Models of String Pattern Matching, TR 81-6

Return this form to:

Ralph E. Griswold
Department of Computer Science
University Computer Center
The University of Arizona
Tucson, Arizona 85721
USA