THE UNIVERSITY OF ARIZONA

TUCSON, ARIZONA 85721

DEPARTMENT OF COMPUTER SCIENCE

Icon Newsletter #14

Ralph E. Griswold

January 17, 1984

Survey

It is time to get some feedback from persons who receive this Newsletter (there are now over 600 persons on the mailing list). A questionnaire for this purpose is attached to the back of this Newsletter.

Please take the time to fill out this questionnaire and return it. If you are on the mailing list, your mailing label is affixed to the questionnaire. Please make any necessary corrections. Anyone who is not presently on the mailing list is welcome to copy the questionnaire and send it back with their address. It will be added to the mailing list automatically.

There is a return address on the back of the questionnaire so that it can be folded and taped (which is preferred to stapling) and mailed without an envelope.

Responses to the survey will be compiled and published in the next Newsletter.

Programming Corner

Answer to a Previous Query: In Newsletter #13, the expression

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

was used to generate a sequence of displays of s at randomly selected positions on a terminal screen. The question that was posed was whether the repeated evaluation in the expression could be placed at any place other than in front of writes.

In fact, the repeated evaluation can be placed anywhere in the expression as long as both random selection operations are evaluated subsequently to the evaluation of the repeated alternation. One possibility is

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

When the argument of the **suspend** expression is resumed to produce another result, the arguments of writes are resumed from right to left. None produce a result until |"\^[Y" is resumed, which produces "\^[Y" again. The arguments to its right are then evaluated again, giving new positions, after which writes is called again to write s at another position. The repeated alternation in this case serves as an endless generator of a constant value for an argument, serving as a kind of barrier for the left-to-right resumption. Other possible locations for the repeated alternation are

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

The results are the same in all cases. The last expression is the most efficient. Why?

Note that the following expression does not produce the desired effect:

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

Although a new row position is produced, the previous column position is used again and s is always written in the same column. The barrier occurs too far to the right.

On the other hand,

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

does not produce the desired effect at all. In fact, it generates only one display. Why?

Returning More than One Value from a Procedure: Persons have asked us about ways to return multiple values from a procedure.

Icon argument transmission is strictly by-value and there is no way, *per se*, to return multiple values from a procedure. One way around this problem is to return a structure that contains several values as in

```
        procedure p()
                  .
                  .
                  .
           return [expr1, expr2, expr3 ... exprn]
        end
```

In some cases, a record may be more appropriate than a list.

The values then may be obtained from the structure that is returned, as in

```
           a := p()
           x := a[1]
           y := a[2]
           z := a[3]
                  .
                  .
                  .
```

Note that this approach allows a procedure not only to produce multiple values, but also to produce an arbitrary number of values, possibly varying from call to call. In this case, a more general method must be used to access the values that are returned.

Since structures in Icon are pointers to data objects, something akin to call-by-reference can be obtained by passing a structure as an argument to a procedure, as in

```
           a := list(n)
           p(a)
```

with

```
        procedure p(a)
                  .
                  .
                  .
           a[1] := expr1
           a[2] := expr2
           a[3] := expr3
                  .
                  .
                  .
           a[n] := exprn
           return
        end
```

When p returns, the values are in the list a and can be accessed as before.

An entirely different approach to returning multiple values is to generate them in sequence:

```
procedure p()
      .
      .
      .
   suspend expr1 | expr2 | expr3 | ... | exprn
end
```

This method fits naturally into Icon's expression evaluation mechanism. The values can be put into a list, if desired, by

```
a := []
every put(a,p())
```

The following expression does the same thing:

```
put(a := [],p())
```

Note that the number of values that p produces need not be known. On the other hand, there is a problem if the generated values are to be assigned to separate variables. This can be done by making explicit assignments as above, using

```
x := a[1]
y := a[2]
z := a[3]
      .
      .
      .
```

but it is tempting to avoid the list and to use iteration, as in

```
every (x | y | z | ...) := p()
```

This does not work as intended, since evaluation is left-to-right and resumption is right-to-left. Thus the alternation expression first produces x and p() is called, producing the value of $expr_1$, which is assigned to x. However, p() is resumed next, producing the value of $expr_2$, which is also assigned to x, and so on. (What values are assigned to y and z?)

The lack of "parallel" evaluation in Icon can be circumvented by using a co-expression:

```
e := create p()
every (x | y | z | ...) := @e
```

The resumption of a co-expression activation does not produce another value. Consequently, the first value is assigned to x, the alternation is resumed, y is produced, and the second value produced by p() is assigned to it by the second activation of e, and so on.

While this approach is a bit oblique for this simple situation, it illustrates the control that co-expressions provide over the production of results.

Initial Assigned Values in Tables: When a table is created, as in the expression

```
t := table(x)
```

the value of x is the initial assigned value for new entries in t. This initial assigned value is used for references to entries that are not already in the table. For example, if a count is being kept of strings, the initial assigned value might be 0, as in

```
count := table(0)
```

Now suppose the following expression is evaluated:

```
count[s] +:= 1
```

This expression produces the same result as

```
count[s] := count[s] + 1
```

Suppose s is not in the table. Then the reference to

```
count[s]
```

in the addition operation produces 0, the initial assigned value. (The augmented assignment operation is preferable to addition and assignment, since the latter expression requires that s be looked up twice in the table.)

But suppose that the initial assigned value is not known. How can it be determined? The problem is that there is not, in general, any way of knowing what entries there are in a table, short of converting it to a list by sorting it.

One approach is to pick some unlikely entry value (perhaps a value that is not a string, assuming all the entries in the table are strings). This may work in practice, but for an arbitrary table with arbitrary entries, what value can be *guaranteed* not to be in the table?

The answer is easy — use an entry value that cannot have *existed* before the test. Any newly created structure will do, but the following expression is particularly simple:

```
x := t[[]]
```

Since a list creation expression creates a new structure, the entry value [] cannot be in t and the expression above assigns the initial entry value of t to x — absolutely guaranteed!

Matching Expressions: Matching expressions, which are analogous to patterns in SNOBOL4, provide a way to elevate string scanning in Icon to a higher conceptual level. By definition, a matching expression is an expression that may change &pos and always returns the substring of &subject between the new and old values of &pos. A matching expression that fails also must restore &pos to its old value. For example, tab(i) and move(i) are built-in matching expressions, but &pos := i is not a matching expression, since it does not return the substring of &subject between the former and new values of &pos.

Suppose that $expr_1$ and $expr_2$ are matching expressions. Then which of the following also are matching expressions?

$expr_1$ & $expr_2$
$expr_1$ | $expr_2$
$expr_1$ || $expr_2$
x := $expr_1$
if $expr$ then $expr_1$ else $expr_2$
while $expr_1$
every $expr_1$

A matching procedure is a procedure whose call is a matching expression. An example is

```
procedure Arb()
   local pos, s
   pos := &pos
   every s := &subject[pos:&pos := &pos to *&subject + 1] do suspend s
   &pos := pos
end
```

which corresponds to the SNOBOL4 pattern ARB. This procedure can be written considerably more compactly; this will be discussed in the next Newsletter.

The requirements that an expression must meet in order to be a matching expression can be modified to produce a variety of different classes of "patterns". One possibility is suggested by John Polstra:

> A useful extension to the idea of the matching procedure is what I call the transforming procedure. A transforming procedure is just like a matching procedure, except that it returns a *variable* which is a substring of &subject. Assignment to a call of a transforming procedure can then be used for modifying the subject, as could assignment to tab(i) and move(i) before Version 5 of Icon. A useful general transforming procedure is the following:

```
procedure xform(p)
   suspend &subject[.&pos:p() & &pos]
end
```

If p is a parameterless matching procedure, then p() is the corresponding matching expression and xform(p) is the corresponding transforming expression. Using co-expressions, a more general version (allowing parameters) can be written.

The remark "before Version 5" refers to Version 4 of Icon, which is now obsolete. Version 2, which is still in use, does not allow assignment to tab(i) and move(i).

More on this subject will appear in the next Newsletter.

Problems with Dereferencing: The arguments of functions are not dereferenced until all arguments are evaluated. Consequently, the expression

```
write(s,s := "a")
```

writes aa, regardless of the value that s had prior to the evaluation of this expression.

Such expressions generally are considered to be bad style and they rarely occur in programs, at least in such an obvious form. More subtle and perplexing problems may occur because subscripting expressions in Icon are variables and are polymorphic. Consider

```
x[y] := z
```

Here x may be a string, a list, a table, or even a record. Now consider the expressions

```
x := "hello world"
x[3] := (x := "abc")          "ababc"
```

What happens when the value of x changes between the time it is subscripted and the time the assignment to the subscripted variable is made? What about

```
x := "hello world"
x[3] := (x := "ab")          err 205       out of range
```

Here the subscript of x is in range when it appeared on the left side of the assignment but is out of range when the assignment is made? What about

```
x := "hello world"
x[3] := (x := [1,2,3,4])     err 103       string expected
```

where the type of the value of x is changed? Or even

```
x := "hello world"
x[3] := (x := 397)          "39397"
```

where the type is changed, but to one that is coercible to the previous type?

Granted that such expressions are unlikely tto occur in "real" programs, they must be accounted for by the semantics of Icon and implementations must handle them properly.

If you have Version 5 of Icon available, try these expressions and see if the results are what you expect. (Version 2 performs dereferencing differently from Version 5, with consequently different results.)

Some answers and more discussion on these matters will appear in the next Newsletter.

Syntactic Pitfalls: The Icon translator automatically inserts semicolons between expressions on adjacent lines in cases where the token at the end of the first line is legitimate as the end of an expression (an "ender") and the token at the beginning of the second line is legitimate as the beginning of an expression (a "beginner"). Thus semicolon insertion makes it possible to write programs without having to put semicolons at the ends of expressions.

All prefix operators are beginners. Since many prefix operators also are legal in infix operators, semicolon insertion sometimes can produce unexpected results. For example,

```
x
| y
```

is translated as if it had been written

```
x; | y
```

not as

```
x | y
```

(Identifiers are both beginners and enders.) Note that

```
x; | y
```

is syntactically correct, if a bit unlikely. It is advisable to guard against unexpected translations of infix operations by putting the infix operator at the end of the first line, not at the beginning of the second. Thus

```
x |
y
```

is translated as

```
x | y
```

since no operator is an ender and hence a semicolon is not inserted.

The semicolon insertion mechanism does not take into account situations in which a line ends with an ender and the next line begins with a beginner, but in which the lines do not form complete expressions. Thus

```
write(i
+ j)
```

is translated as if it were

```
write(i;
+j)
```

and is diagnosed as a syntactic error, although

```
i
+ j
```

is translated as if it were

```
i;
+ j
```

which is syntactically correct.

In the case of expressions with optional arguments, semicolon insertion may produce mysterious effects if care is not taken. For example,

```
return
   x
```

is translated as if it were

```
return;
x
```

This occurs because the argument of **return** is optional, making it an ender. When this code segment is evaluated, the null value is returned and x is never evaluated. If the problem is not recognized, it appears that x always has the null value, even though it may obviously have a different value.

As another example, consider the following code segment that was produced by a SNOBOL4 programmer who is used to having an omitted right argument of assignment default to the empty string:

```
s[1] :=
return s
```

On the face of it, this is erroneous in Icon. However, since := is not an ender, a semicolon is not inserted and this code segment is translated as if it were

```
s[1] := return s
```

Although this is a strange expression, it is both syntactically and semantically correct. When the right argument of the assignment is evaluated, a return occurs and the assignment is never completed. Unless the translator's interpretation of this expression is recognized, the effect during program execution may appear mysterious.

Fortunately, semicolon insertion works well in practice. Observation of the rules of program layout given above avoids all these problems.

Trivia Corner: What is the longest string of distinct prefix operators which, when applied to a value, might compute a meaningful result? (You may assume any value that you wish.) What if the prefix operators need not be distinct?

Recent Icon Documents

Several recently published documents related to Icon are now available.

One report, TR 83-14, describes a facility for displaying pattern matching activity. This report includes listings of several moderately sophisticated Icon programs, which may be of interest to programmers, independent of the subject matter of the report.

Work on sequences and their relationship to the expression evaluation mechanism of Icon has led to the development of an experimental programming language for manipulating sequences as program data objects. The formal basis for this language is described in TR 83-15 and the language itself is described in TR 83-16.

There also is a reprint of a recent paper on result sequences.

Single copies of all these documents are available, free of charge. Use the document request form on the next page. The form may be returned with the questionnaire.

Request for Icon Documents

Please send the documents checked below to:

____

____

____

____

____

☐ *Understanding Pattern Matching — A Cinematic Display of String Scanning*, TR 83-14

☐ *The Description and Manipulation of Sequences*, TR 83-15

☐ *Seque; An Experimental Language for Manipulating Sequences*, TR 83-16

☐ "Result Sequences", reprinted from *The Journal of Computer Languages*

Return this form to:

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