Sequences and Expression Evaluation in Icon*

Stephen B. Wampler

TR 81-2

March 1981

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS79-03890.

Sequences and Expression Evaluation in Icon

1. Introduction

This paper introduces a notation for the static description of expression evaluation in Icon [7] and uses this notation to detail the operation of various control mechanisms in Icon. This notation gives insight into the relationships between Icon control mechanisms and those found in conventional languages. Insights gained from this notation have also resulted in the design of several new control mechanisms, a number of which have been implemented in Version 4 of Icon (see the Appendix). This paper refers to expression evaluation in Version 4.

2. Expression Evaluation

Most programming languages contain expressions that are evaluated to produce *results*. A result is either a value or a variable. Expression evaluation in conventional languages such as Algol always produces exactly one result. For example, the result of evaluating 1 + 3 is the value 4. Control structures in such languages are driven by the values of their control expressions. In the Algol expression

```
if x = y then z := 0
```

the comparison of x and y produces a boolean value that is used to determine whether or not the assignment to z is performed.

Expression evaluation need not produce a result. In SNOBOL4 [9] expression evaluation may produce no result at all. For example, the expression

```
EQ(1,0)
```

fails to produce any result. The concept of *failure* in SNOBOL4 is equivalent to failure of an expression to produce a result, while *success* corresponds to the production of a result.

SNOBOL4 lacks conventional control structures such as those found in Algol, and instead relies upon the presence or absence of results to control a conditional branching mechanism, as in

```
EQ(X,Y)                    :S(LABEL1)F(LABEL2)
```

which causes a branch to LABEL1 if EQ(X,Y) produces a result, and a branch to LABEL2 otherwise.

Failure is also used to control completion of the evaluation of enclosing expressions. For example, in evaluation of

```
Z = EQ(X,Y) 0
```

assignment of 0 to Z is performed only if X and Y are numerically equal.

Expressions in Icon are capable of generating more than one result in sequence and for this reason are termed *generating expressions* [8]. Context determines whether evaluation of an expression produces more than one result. The expression every *e* is used to produce the entire sequence of results for *e*.

For example, the expression 1 to 10 is capable of producing the results 1, 2, ..., 10. Evaluation of

```
every write(1 to 10)
```

writes the results 1, 2, ..., 10. However, in the expression (1 to 10) >= 3, only the results 1, 2, and 3 are produced during evaluation of 1 to 10, since the context only requires a result greater than or equal to 3.

There are no constraints on the length of a sequence of results in Icon. The sequence may be empty, as in evaluation of 1 = 0, or it may contain an infinite number of results, as in the case of ?100, where each result is an integer randomly selected from the range 1 to 100.

The *outcome* of expression evaluation in Icon is either a result or failure to produce a result. Expressions are evaluated in Icon with the goal of producing a result as the outcome [11]. If the outcome of an expression evaluation is failure to produce a result, then the evaluation is said to *fail*, otherwise the evaluation *succeeds*. As in SNOBOL4, Icon control structures are driven by the presence or absence of results. For example

```
if x = y then z := 0
```

assigns 0 to z if the expression x y succeeds.

While several of the control structures in Icon resemble control structures in Algol, the use of success or failure to drive control structures involves some subtle differences [4]. In particular, control structures in Algol are driven by boolean values, and hence their control clauses must be expressions that produce boolean results. Thus,

```
x < y
```

produces a boolean result in Algol. This, in turn, renders expressions such as

```
x < y < z
```

meaningless in Algol.

However, the outcome of an Icon expression is either failure or a computationally useful result. For example, the expression

```
x < y
```

fails if x is not less than y, and produces y otherwise. This makes it possible to write such expressions as

```
if x < y < z then z := x else z := y
```

Failure may be used as it is in SNOBOL4 to abort the evaluation of enclosing expressions. For example, the function read fails when attempting to read past the end of a file. If read fails during evaluation of

```
write(read())
```

the function write is not invoked, and the entire expression fails. Hence

```
while write(read())
```

copies the input file to the output file.

The expression evaluation mechanisms in both Algol-like languages and SNOBOL4 are in a sense subsets of the Icon expression evaluation mechanism. Expressions in conventional languages such as Algol correspond to Icon generators that produce exactly one result. Except in pattern-matching, SNOBOL4 expressions correspond to generators producing at most one result. The pattern-matching component of SNOBOL4 constitutes a sublanguage [6] in which some patterns function as generators during the pattern-matching process. The patterns for ARB, BAL, and the patterns produced by BREAKX(S) [2] and P1 | P2 are generators capable of producing more than one result.

3. Side effects

Some expressions are not evaluated for their results, but rather for the side effects of their evaluation. For example, the function write produces its last argument as its result, but it is almost always used to produce output to a file. Further, some operations rely upon side effects to work properly. read, for example, advances a file pointer each time it produces a result, so that subsequent evaluation produces a new result.

While side effects play a very real and important role in expression evaluation, they also tend to obscure more fundamental aspects of an expression evaluation mechanism. The notation presented in this paper is not intended to describe side effects, but rather to describe the static aspects of expression evaluation.

4. Result Sequences

A *result sequence* is defined as the sequence of results that an expression is capable of producing. This section presents a notation for result sequences and describes operations used in the manipulation of result sequences.

The result sequence produced by the expression e is denoted by

$$\mathcal{S}(e) = \{s_1, s_2, \ldots, s_n\}$$

where s_i is the ith result generated during the evaluation of e. For example, the result sequence for 1 to 10 is {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

When more than one result sequence is needed, subscripts are used. For example, the result sequences for expressions e_1 and e_2 are denoted by

$$\mathcal{S}(e_1) = \{s_{1_1}, s_{1_2}, \ldots, s_{1_n}\}$$

$$\mathcal{S}(e_2) = \{s_{2_1}, s_{2_2}, \ldots, s_{2_m}\}$$

The *null sequence*, {}, is a sequence containing no elements and is denoted by Φ. Another common sequence is { • }, which contains the null value •, and is denoted by Λ.

Subsequences are denoted using subscripts and superscripts, as in

$$\mathcal{S}_i^j(e) = \{s_i, \ldots, s_{\min(j,n)}\}$$

If $i = 1$ the subsequence is abbreviated as

$$\mathcal{S}_1^j(e) = \mathcal{S}^j(e)$$

If $i = j$ the subsequence is abbreviated as

$$\mathcal{S}_i^i(e) = \mathcal{S}_i(e)$$

The *size* of a sequence is the number of elements it contains.

$$|\mathcal{S}(e)| = n$$

where $0 \le n \le \infty$.

Concatenation of sequences is denoted by

$$\mathcal{S}(e_1), \mathcal{S}(e_2) = \{s_{1_1}, \ldots, s_{1_n}, s_{2_1}, \ldots, s_{2_m}\}$$

Concatenations of a sequence with itself are represented using exponential notation. For example,

$$\mathcal{S}(e), \mathcal{S}(e) = \mathcal{S}(e)^2$$

$$\mathcal{S}(e), \mathcal{S}(e), \cdots = \mathcal{S}(e)^\infty$$

Note that $\mathcal{S}(e)^i$ is not, in general, the same as $\mathcal{S}^i(e)$.

The null sequence is the identity element with respect to concatenation of sequences:

$$\Phi, \mathcal{S}(e) = \mathcal{S}(e), \Phi = \mathcal{S}(e)$$

Concatenation of result sequences from a number of expressions, as in

$$\mathcal{S}(e_1), \mathcal{S}(e_2), \ldots, \mathcal{S}(e_n)$$

is denoted by

$$\prod_{i=1}^{n} \mathcal{S}(e_i)$$

The *dot product* of two sequences produces a sequence of pairs as defined by

$$\mathcal{S}(e_1) \cdot \mathcal{S}(e_2) = \{(s_{1_1}, s_{2_1}), (s_{1_2}, s_{2_2}), \ldots, (s_{1_k}, s_{2_k})\}$$

where $k = \min(n, m)$. This may be generalized to the dot product of n sequences to produce a sequence of n-tuples.

The *cross product* of two or more sequences produces a sequence of n-tuples. For example, the cross product of two sequences produces a sequence of pairs as given by

$$\begin{aligned}\mathfrak{S}(e_1)\times\mathfrak{S}(e_2) = \{&(s_{1_1}, s_{2_1}), (s_{1_1}, s_{2_2}), \ldots, (s_{1_1}, s_{2_m}),\\ &(s_{1_2}, s_{2_1}), (s_{1_2}, s_{2_2}), \ldots, (s_{1_2}, s_{2_m}),\\ &\ldots\ldots\\ &(s_{1_n}, s_{2_1}), (s_{1_n}, s_{2_2}), \ldots, (s_{1_n}, s_{2_m})\}\end{aligned}$$

The *application* of an arbitrary operation ρ to a sequence yields the concatenation of the sequences resulting from applying ρ to every element in the original sequence. Application of ρ to a sequence is denoted by $\rho::\mathfrak{S}(e)$, and application of ρ to a sequence element is denoted by $\rho{:}s$. That is

$$\begin{aligned}\rho::\mathfrak{S}(e) &= \mathfrak{S}(\rho{:}s_1), \mathfrak{S}(\rho{:}s_2), \ldots, \mathfrak{S}(\rho{:}s_n)\\ &= \mathop{\Xi}_{i=1}^{n} \mathfrak{S}(\rho{:}s_i)\end{aligned}$$

5. A Unified Syntax for Expression Evaluation

The syntax of Icon is intended to assist the user in the development of clear, understandable algorithms. This is accomplished by providing both mnemonic forms of various control mechanisms and a concise representation for expressions. Conciseness is gained by providing an implicit representation for the goal-directed evaluation control mechanism. Goal-directed evaluation is such an inherent aspect of Icon expression evaluation that a visible syntax would be overbearing. The use of mnemonic forms for control mechanisms acts as syntactic 'sugar' to enhance program readability. Because this syntax is designed for program development, it is called the $\mathfrak{L}$-syntax. Expressions written in $\mathfrak{L}$-syntax are referred to as $\mathfrak{L}$-expressions.

Nevertheless, when discussing the expression evaluation mechanism in Icon, these same syntactic features serve only to obscure the underlying concepts. This section presents a description of an alternative syntax for Icon. This syntax is intended as a descriptive device to simplify subsequent discussions of the semantics of the Icon expression evaluation mechanism, For this reason it is called the $\mathfrak{S}$-syntax. Expressions written in $\mathfrak{S}$-syntax are referred to as $\mathfrak{S}$-expressions.

5.1 Language Primitives

Literals, identifiers, and operators constitute the language *primitives* in Icon. Aside from syntactic considerations, there is little that distinguishes operators from function- and procedure-valued identifiers. The value of an operator is the function that the operator performs. In $\mathfrak{S}$-syntax, operators, functions, and procedure calls all have the same syntax, and are collectively termed *functions*. Operators are replaced by a name for the function that performs the indicated operation. A function is represented in $\mathfrak{S}$-syntax as an n-tuple, where the first element of the n-tuple is the function, and the remaining elements are the arguments of that function. For example, the $\mathfrak{L}$-expressions

```
write("hello world")
show(x,y,z)
1 + 2
1 to 5
```

are written respectively in a LISP notational style as the $\mathfrak{S}$-expressions

(write, "hello world")
(show, x, y, z)
(*add*, 1, 2)
(*to*, 1, 5)

The sans-serif italic font is used to distinguish actual literal functions from the names of Icon identifiers. For example, the value of the variable write is the function *write*.

5.2 Control Regimes

Control structures in Icon are simply syntactic representations of operations in a metalanguage. These operations are termed *control regimes*.

A control regime corresponds to a particular method of expression evaluation. For example, sequential processing of expressions separated by semicolons is a control regime in which expressions are evaluated from left to right.

The $\mathfrak{E}$-syntax for control regimes is

regime : *arguments*

where *regime* is the name of some control regime and *arguments* is a list of expressions constituting the operands to that control regime. The names for specific control regimes are chosen to be indicative of the corresponding control structure, and are shown here in an Old English typeface. Thus the $\mathfrak{L}$-expression

if e_1 then e_2 else e_3

is represented in $\mathfrak{E}$-syntax as

$\mathfrak{If} : e_1, e_2, e_3$

and the $\mathfrak{L}$-expression

every e_1 do e_2

has $\mathfrak{E}$-syntax

$\mathfrak{Every} : e_1, e_2$

Goal-directed evaluation, which has no explicit representation in $\mathfrak{L}$-syntax, is the control regime $\mathfrak{Goal}$ in $\mathfrak{E}$-syntax. Thus the $\mathfrak{L}$-expression

1 to 10

has $\mathfrak{E}$-syntax

$\mathfrak{Goal} : to, 1, 10$

Arguments that are omitted in $\mathfrak{L}$-syntax are denoted by ϕ in the $\mathfrak{E}$-syntax. Hence the $\mathfrak{E}$-expression

$\mathfrak{If} : e_1, e_2, \phi$

corresponds to the $\mathfrak{L}$-expression

if e_1 then e_2

Brackets are used for any necessary grouping in the $\mathfrak{E}$-syntax, so that the $\mathfrak{L}$-expression

every 1 to 10

is written in $\mathfrak{E}$-syntax as

$\mathfrak{Every} : [\mathfrak{Goal}: to, 1, 10], \phi$

and the $\mathfrak{L}$-expression

```
if x > y then 1 to x
```

is the 𝔊-expression

```
𝔍f : [𝔊oal:greaterthan,x,y],[𝔊oal:to,1,x],ϕ
```

6. Result Sequences for Language Primitives

6.1 Literals and Identifiers

The result sequence for a literal or identifier consists of the identifier or the value of the literal. Hence the result sequences for of x and 3 are {x} and {3}, respectively.

6.2 Functions

Functions are evaluated by the *function invocation* mechanism, Γ. Γ is applied to a function to produce a result sequence.

For example, the conjunction operator, &, has as its value a function *conj* that returns its right operand. Conjunction is invoked as in

```
Γ:(conj,x,y)
```

which has the result sequence {y}.

The application of Γ is one step of goal-directed evaluation, and as such, has no representation in either 𝔓- or 𝔊-syntax. No attempt is made here to describe the actual evaluation of functions, except to note that Γ produces a (possibly empty) sequence of results from a function. For example, the result sequence for

```
Γ:(to,1,5)
```

is

```
{1,2,3,4,5}
```

and the result sequence for

```
Γ:(greaterthan,3,4)
```

is Φ.

6.3 Control Regimes

A control regime takes a series of expressions as arguments and specifies the order in which these expressions are evaluated. For example, one control regime may evaluate its arguments from left to right, while another regime may use the outcome of evaluating its first argument to select another argument to evaluate.

Like primitive operations, control regimes have result sequences. However, unlike primitive operations, control regimes deal with result sequences, not results. For example, the result sequence for

$$\mathfrak{If}: e_0, e_1, e_2$$

depends upon the size of the result sequence for e_0 and determines whether the result sequence for this expression is the result sequence for e_1 or the one for e_2.

7. Result Sequences and Control Regimes

Let Ψ represent an arbitrary control regime and Ψ:S be that control regime with a list of arguments S. The evaluation of Ψ:S may be characterized by the result sequence for Ψ. The result sequences for some representative control regimes in Icon are given below.

Sequential Processing

Sequential processing, represented by the $\mathfrak{L}$-expression

$$\{e_1;e_2;\cdots;e_n\}$$

and $\mathfrak{S}$-expression

$$\mathfrak{Sequence}:e_1,e_2,\ldots,e_n$$

is a control regime that evaluates its arguments in order. Its result sequence is the result sequence of its last argument. Hence

$$\mathfrak{S}(\mathfrak{Sequence}:e_1,e_2,\ldots,e_n)=\mathfrak{S}(e_n)$$

Goal-Directed Evaluation

The *goal-directed evaluation* control regime forms the cross product of the result sequences for its arguments, producing a sequence composed of n-tuples. Function invocation is then applied to the elements of that sequence to produce a result sequence. Formally,

$$\mathfrak{S}(\mathfrak{Goal}:e_0,e_1,\ldots,e_n)=\Gamma::\mathfrak{S}(e_0)\times\mathfrak{S}(e_1)\times\cdots\times\mathfrak{S}(e_n)$$

For example,

$$\mathfrak{Goal}:to,1,5$$

has the result sequence

$$\begin{aligned}\mathfrak{S}(\mathfrak{Goal}:to,1,5)&=\Gamma::\{to\}\times\{1\}\times\{5\}\\&=\Gamma::\{(to,1,5)\}\\&=\{\Gamma:(to,1,5)\}\\&=\{1,2,3,4,5\}\end{aligned}$$

As another example, the $\mathfrak{L}$-expression

```
3 < (1 to 5)
```

is written in $\mathfrak{S}$-syntax as

$$\mathfrak{Goal}:lessthan,3,[\mathfrak{Goal}:to,1,5]$$

From above,

$$\begin{aligned}\mathfrak{S}(\mathfrak{Goal}:lessthan,3,[\mathfrak{Goal}:to,1,5])&=\mathfrak{S}(\mathfrak{Goal}:lessthan,3,\{1,2,\ldots,5\})\\&=\Gamma::\{lessthan\}\times\{3\}\times\{1,2,\ldots,5\}\\&=\Gamma::\{(lessthan,3,1),(lessthan,3,2),\ldots,(lessthan,3,5)\}\\&=\{\Gamma:(lessthan,3,1)\},\{\Gamma:(lessthan,3,2)\},\ldots,\{\Gamma:(lessthan,3,5)\}\\&=\Phi,\Phi,\Phi,\{4\},\{5\}\\&=\{4,5\}\end{aligned}$$

As a final example of goal-directed evaluation, consider

```
(1 to 5) & x
```

with $\mathfrak{S}$-syntax

$$\mathfrak{Goal}:conj,[\mathfrak{Goal}:to,1,5],x$$

From above,

$$\mathfrak{S}(\mathfrak{Goal}{:}conj,[\mathfrak{Goal}{:}to,1,5],x) = \Gamma{::}\{conj\}\times\{1,2,3,4,5\}\times\{x\}$$
$$= \Gamma{::}\{(conj,1,x),(conj,2,x),(conj,3,x),(conj,4.x),(conj,5,x)\}$$
$$= \{\Gamma{:}(conj,1,x)\},\{\Gamma{:}(conj,2,x)\},\{\Gamma{:}(conj,3,x)\},\{\Gamma{:}(conj,4,x)\},\{\Gamma{:}(conj,5,x)\}$$
$$= \{x,x,x,x,x\}$$

Alternation

Alternation, $e_1 \mid e_2$, is the control regime $\mathfrak{Alternation}$ that simply concatenates the result sequences for its arguments. That is

$$\mathfrak{S}(\mathfrak{Alternation}{:}e_1,e_2) = \mathfrak{S}(e_1),\mathfrak{S}(e_2)$$

If-then-else

The $\mathfrak{If}$ control regime uses the size of the result sequence of its first argument to determine which of the two remaining arguments, or *arms*, to evaluate. The result sequence of $\mathfrak{If}$ is the result sequence of the selected arm. That is

$$\mathfrak{S}(\mathfrak{If}{:}e_0,e_1,e_2) = \begin{array}{ll} \mathfrak{S}(e_1) & \text{if } \mathfrak{S}(e_0) \neq \Phi \\ \mathfrak{S}(e_2) & \text{otherwise} \end{array}$$

Iteration

The iteration control regimes are evaluated for side effects and produce the result sequence Λ.

$$\mathfrak{S}(\mathfrak{Every}{:}e_1,e_2) = \mathfrak{S}(\mathfrak{While}{:}e_1,e_2) = \mathfrak{S}(\mathfrak{Repeat}{:}e_1) = \mathfrak{S}(\mathfrak{Until}{:}e_1,e_2) = \Lambda$$

None of the iteration control regimes is fundamental. The operation of any of these regimes can be described in terms of other control regimes. These and other equivalences are described in Section 8.

Outcome inversion

$\mathfrak{Invert}$ is the *outcome inversion* control regime denoted by the $\mathfrak{L}$-expression

not e

and the $\mathfrak{E}$-expression

$\mathfrak{Invert} : e$

The result sequence for $\mathfrak{Invert}$ is

$$\mathfrak{S}(\mathfrak{Invert}{:}e) = \begin{array}{ll} \Lambda & \text{if } \mathfrak{S}(e) = \Phi \\ \Phi & \text{otherwise} \end{array}$$

That is, the result sequence for $\mathfrak{Invert}$ is empty if its argument has the null sequence, and Λ otherwise.

Repeated evaluation

The *repeated evaluation* control regime, with $\mathfrak{L}$-syntax

|e

and $\mathfrak{E}$-syntax

$\mathfrak{Reval} : e$

has the result sequence

$$\mathfrak{S}(\mathfrak{Reval}{:}e) = \mathfrak{S}(e)^{\infty}$$

In practice, side effects are relied upon to limit the size of the result sequence for Reval. $\mathcal{S}(e)^{\infty}$ is the infinite series of concatenations $\mathcal{S}(e)_1, \mathcal{S}(e)_2, \mathcal{S}(e)_3, \ldots, \mathcal{S}(e)_{\infty}$. If the ith term in this expansion has the result sequence Φ, then evaluation of Reval terminates after evaluating the ith term. Unless $\mathcal{S}(e)_1$ is Φ, only side effects can cause $\mathcal{S}(e)_i$ to be Φ for $i>1$.

Limitation

Limitation, with $\mathcal{L}$-syntax

$$e_1 \setminus e_2$$

and $\mathcal{E}$-syntax

$$\text{Limit:}e_1,e_2$$

has the result sequence

$$\mathcal{S}(\text{Limit:}e_1,e_2) = \prod_{i=1}^{|\mathcal{S}(e_2)|} \mathcal{S}^{s_i}(e_1)$$

where $\mathcal{S}(e_2) = \{s_1, s_2, \ldots, s_m\}$. Note that if $|\mathcal{S}(e_2)| = 1$ and $s_{2_1} = k$, then the result sequence of limitation is simply a subsequence of the result sequence of e_1. That is,

$$\mathcal{S}(\text{Limit:}e_1,e_2) = \mathcal{S}^k(e_1)$$

8. Equivalences among Control Regimes

Some control regimes can be expressed in terms of others. The control regimes

Goal
If
Alternation
Reval
Limit

are sufficient, in combination with the conjunction function *conj*, to describe a number of other control regimes. The control regimes listed above are referred to as the *basic* control regimes.

8.1 Supplementary Control Regimes

The following control regimes do not exist in Icon, and hence have no $\mathcal{L}$-syntax. They are described here to provide a convenient notational shorthand for the description of subsequent control regimes.

Limitation to Exactly One Result

One is a control regime that produces exactly one result. One : e is equivalent to

(e | &null) \ 1

or, in $\mathcal{E}$-syntax

Limit : [Alternation:e ,•],1

Alternation is needed to guarantee that the result sequence for One has at least one element, since $\mathcal{S}(e)$ may be Φ. Limitation provides the first result from this sequence, so that

$$|\mathcal{S}(\text{One:}e)| = 1$$

Limitation to No Results

The control regime None always fails to produce a result. That is, the result sequence for None is

$$\mathcal{S}(\text{None}) = \Phi$$

The evaluation of None is equivalent to the evaluation of

Limit:e,0

where e is any expression. The keyword &fail is also equivalent to None.

Repeated First Result

First : e is a control regime that is equivalent to the $\mathcal{L}$-expression

$$|(e \setminus 1)$$

and the $\mathcal{G}$-expression

Reval : [Limit:e,1]

First has a result sequence consisting of an infinite repetition of the first element in the result sequence for its argument. That is, the result sequence for First is given by

$$\mathcal{S}(\text{First:}e) = \mathcal{S}_1(e)^{\infty}$$

subject to the same termination conventions as Reval.

Multiple Conjunction

As stated before, the conjunction operator is a way to provide goal-directed evaluation over two expressions. This generalizes to any number of expressions, but quickly becomes notationally unwieldy in $\mathcal{G}$-syntax. For example, the $\mathcal{L}$-expression

$$e_1 \& e_2 \& e_3 \& e_4$$

has the equivalent the $\mathcal{G}$-expression

Goal : *conj*,e_1, [Goal:*conj*,e_2, [Goal:*conj*,e_3,e_4]]

Mulconj is a control regime for expressing this type of conjunction series. For example, the above expression is denoted

Mulconj : e_1,e_2,e_3,e_4

The result sequence for Mulconj is

$$\mathcal{S}(\text{Mulconj:}e_1, e_2, \ldots, e_n) = \mathcal{S}(e_n)^{\prod_{i=1}^{n-1} |\mathcal{S}(e_i)|}$$

where $\prod_{i=1}^{n} s_i$ is the usual notation for a product of n values.

8.2 Formulations for Icon Control Regimes

Sequential Processing

By definition, Sequence evaluates its arguments from left to right and has the result sequence of the last argument. All the arguments except the last one are limited to exactly one result. Thus

Sequence : e_1,e_2, . . . , e_n

may be expressed as

Mulconj : [One:e_1], [One:e_2], . . . , [One:e_{n-1}], e_n

Outcome Inversion

Invert:e may be expressed as

If : e ,None,•

Iteration

All of the iteration regimes can be expressed in terms of the basic regimes. However, the approach is simplified by first expressing the iteration regimes in terms of each other.

The evaluation of Repeat only terminates because of side effects. That is,

Repeat : e

is expressible as

While : [One:e],ϕ

Similarly,

Until : e_1,e_2

is expressed as

While : [Invert:e_1],e_2

Expressing While in terms of Every is simplified by the use of First. That is,

While : e_1,e_2

is expressed as

Every : [First:e_1],e_2

or

One : [Multconj : [First:e_1],[Limit:e_2,1],None]

Finally,

Every : e_1,e_2

can be expressed

One : [Multconj:e_1,[Limit:e_2,1],None]

It is interesting to note the similarity between the basic regime formulation for While and the formulation for Every. The only difference between the two is that the first argument to While always evaluates to its first result.

9. Conclusion

The use of result sequences to characterize the operation of control structures leads to a better understanding of the operation of Icon expression evaluation. For example, both alternation and conjunction have had rather complex descriptions in the Icon literature, despite the fact that result sequences show both to be simple operations.

Result sequences also suggest several additional control regimes to provide increased flexibility in the use of generators. While not implemented in Version 4, they suggest possibilities for further experimentation.

Alternation and Conjunction

Alternation traditionally has a rather complex explanation. For example, the Version 3 reference manual [1] has the following explanation:

The most fundamental generator is alternation

expr1 | *expr2*

This expression first evaluates *expr1*. If *expr1* succeeds, its result becomes the result of the alternation expression. If *expr1* fails, however, the outcome of the alternation expression is the outcome of evaluating *expr2*. For example,

(i = j) | (j = k)

succeeds if i is equal to j or if j is equal to k.

Alternation has an important additional property. If *expr1* succeeds, but the expression in which the alternation occurs would fail, the alternation operator then evaluates *expr2*. For example,

x = (1 | 3)

succeeds if x is equal to 1 or 3.

This explanation is not only involved, it is inaccurate! Let

$\mathfrak{Or}{:}e_1,e_2$

be the $\mathfrak{S}$-expression for the alternation operation as described above. The result sequence for $\mathfrak{Or}$ is

$\mathfrak{S}(\mathfrak{Or}{:}e_1,e_2) = \mathfrak{S}_1(e_1),\mathfrak{S}(e_2)$

That is, only the first result from e_1 is ever used from its result sequence. This is not how alternation works. As shown earlier, alternation simply concatenates the result sequences for its arguments, so that all of the results are accessible by a surrounding expression. Finally, note that the explanation given in the Version 3 manual does not describe what happens if *expr2* succeeds, but the surrounding expression fails.

$\mathfrak{Or}$ may be constructed from the basic control regimes as

$\mathfrak{Alternation} : [\mathfrak{Limit}{:}e_1, 1],e_2$

Another interesting form of alternation is "normal" alternation as found in SUMMER [10], and described as "forward" alternation by Doyle [3]. This form of alternation is equivalent to the $\mathfrak{L}$-expression

$(e_1 \mid e_2) \setminus 1$

and the $\mathfrak{S}$-expression

$\mathfrak{Limit} : [\mathfrak{Alternation}{:}e_1,e_2], 1$

which produces at most one result.

Several mysterious properties have been ascribed to conjunction. As recently as the Version 2 reference manual [5], it was considered to have special properties with respect to goal-directed evaluation. However, conjunction is the simplest of all binary operators, merely returning its right operand [4]. The "mysterious properties" are shown in Section 7 to be the normal operation of goal-directed evaluation, independent of the conjunction operation.

Possible Extensions to Icon

Generating expressions provide a notationally concise form for expressing a great deal of computational power. However, their use is often hampered by the lack of sufficient "generator-based" control regimes. For example, the expression

every f(!alist)

invokes f on every element of alist, but to invoke f on just the even indexed elements, or on the tenth through twentieth elements, requires a radically different expression.

Result sequences have suggested a number of interesting control regimes that provide additional control over generators. This section describes several of these control regimes and presents possible $\mathfrak{L}$-syntax forms for these regimes.

The control regime $\mathfrak{Subsequence}$ with $\mathfrak{L}$-syntax

$e_1 \setminus [e_2 : e_3]$

and &-syntax

$\mathfrak{Subsequence} : e_1,e_2,e_3$

has a result sequence that is a subsequence of the result sequence for its first argument. For example,

$\mathfrak{S}(\mathfrak{Subsequence}:e_1,5,10) = \mathfrak{S}_5^{10}(e_1)$

Note that Subsequence is a generalization of Limit. That is,

$\mathfrak{Limit} : e_1,e_2 = \mathfrak{Subsequence} : e_1,1,e_2$

For convenience, the &-expression

$e_1 \setminus [e_2:0]$

is the &-expression

$\mathfrak{Subsequence} : e_1,e_2,\infty$

so that

(1 to 10) \ [7:0]

has the result sequence {7, 8, 9, 10}. Finally,

every f(!alist \ [10:20])

invokes f on the tenth through twentieth elements of alist.

A generalization of Subsequence is Newsequence with &-syntax

$e_1 \setminus\setminus e_2$

and &-syntax

$\mathfrak{Newsequence} : e_1,e_2$

The result sequence for Newsequence consists of elements of the result sequence for e_1 as specified by the elements of the result sequence for e_2. Formally,

$\mathfrak{S}(\mathfrak{Newsequence}:e_1,e_2) = \mathfrak{S}_{s_1}(e_1),\mathfrak{S}_{s_2}(e_1), \ldots, \mathfrak{S}_{s_m}(e_1)$

where $\mathfrak{S}(e_2) = \{s_1, s_2, \ldots, s_m\}$ consists of integers in strictly increasing order.

For example, the &-expression

$e_1 \setminus\setminus (e_2 \text{ to } e_3)$

is same as

$e_1 \setminus [e_2:e_3]$

except when the result of e_3 is 0.

As a final example,

every f(!alist \\ (2 to *alist by 2))

invokes f on the even-indexed elements of alist.

One of the difficulties in controlling generators in Icon is that there is no mechanism for limiting the result sequences for individual operations without also limiting the arguments to that operation. For example, consider the following &-expression

every find("icon", line := !&input) do write(line)

which outputs any input line containing the string icon. However, if an input line contains more than one occurrence of icon, that line is output more than one time. The problem is that the result sequence for find(s1, s2) contains the locations of all occurrences of s1 in s2.

Attempting to limit find to at most one result in the above example also limits !&input to at most one result. Thus,

```
every find("icon", line := !&input) \ 1 do write(line)
```

outputs only the first line containing icon.

One solution to this problem would be an alternative form of function invocation that permits limitation of specific operations, without limiting the arguments. Let Υ denote this alternative function invocation mechanism, with the following relationship to Γ:

$$\Upsilon\text{:}function = \mathfrak{L}\text{imit}\text{:}[\Gamma\text{:}function\,],1$$

That is, Υ produces at most one result from the invocation of a function.

Υ may be applied during goal-directed evaluation in place of Γ to limit a particular operation without limiting the arguments to that operation.

An operation that is to be invoked with Υ instead of Γ is "tagged" in $\mathfrak{L}$-syntax with a grave accent (`). For example, the following $\mathfrak{L}$-expression outputs every input line that contains the string icon, with no line being output more than once.

```
every ` find("icon", line := !&input) do write(line)
```

Acknowledgement

I am indebted to Cary Coutant, Ralph Griswold, Dave Hanson, and Tim Korb for their discussions and criticisms. Cary Coutant provided invaluable assistance with the implementation of the new Version 4 features.

References

1. Coutant, Cary A., Ralph E. Griswold, and Stephen B. Wampler. *Reference Manual for the Icon Programming Language; Version 3.* Technical Report TR 80-2, Department of Computer Science, The University of Arizona. May 1980.
2. Dewar, Robert B. K. *SPITBOL Version 2.0.* Technical Report S4D23, Illinois Institute of Technology. February 1971.
3. Doyle, John D. *A Generalized Facility for the Analysis and Synthesis of Strings, and a Procedure-Based Model of an Implementation.* Technical Report S4D48, Department of Computer Science, The University of Arizona. February 1975.
4. Griswold, Ralph E. *Expression Evaluation in Icon.* Technical Report TR 80-21, Department of Computer Science, The University of Arizona. August 1980.
5. Griswold, Ralph E. and David R. Hanson. *Reference Manual for the Icon Programming Language; Version 2.* Technical Report TR 79-1a, Department of Computer Science, The University of Arizona. January 1980.
6. Griswold, Ralph E., and David R. Hanson. "An Alternative to the Use of Patterns in String Processing", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2 (April 1980). pp. 153-172.
7. Griswold, Ralph E., David R. Hanson, and John T. Korb. "The Icon Programming Language; An Overview", *SIGPLAN Notices*, Vol. 14, No. 4 (April 1979). pp. 18-31.
8. Griswold, Ralph E., David R. Hanson, and John T. Korb. "Generators in Icon", *ACM Transactions on Programming Languages and Systems*, To appear in April, 1981.
9. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language.* Second Edition. Prentice-Hall, Inc. Englewood Cliffs, New Jersey. 1971.
10. Klint, P. "An Overview of the SUMMER Programming Language", Conf. Rec., 7th Annual ACM Symp. on Principles of Programming Languages. Las Vegas, Nevada (January 1980). pp 47-55.
11. Korb, John T. *The Design and Implementation of a Goal-Directed Programming Language.* Technical Report TR 79-11, Department of Computer Science, The University of Arizona. June 1979.
12. Wampler, Stephen B. *New Control Structures in Icon.* Technical Report 81-1, Department of Computer Science, The University of Arizona. January 1981.

Appendix - Version 4 of the Icon Programming Language

Version 4 Icon is an extension of Version 3 incorporating some novel control mechanisms that exploit generators. In addition to these new control mechanisms, Version 4 differs from Version 3 in several aspects. This appendix briefly describes those differences that are pertinent to this paper. A more complete discussion may be found in [12].

1. The places in which generators are limited to at most one result have been significantly reduced in Version 4. For example, the arms of the if-then-else control structure are no longer limited to one result. Thus

   ```
   every write(if x < y then 1 to x else 1 to y)
   ```

   outputs 1, 2, . . . , min(x,y) in Version 4, rather than simply 1, as in Version 3.

2. Braces in Version 3 serve to both group expressions and to limit generators. In Version 4, braces serve only to group expressions and do not limit generators.

3. The lone suffix operation in Version 3, fails, has been replaced with the prefix control structure not, to help relieve the confusion caused by having a suffix operation.

4. There is a new keyword, &fail, whose evaluation always fails.

5. The size of an object is computed with the size operator, *x, rather than the function, size(x).

6. random(i) has been replaced by the random sequence operator, ?x. ?x is an infinite generator that produces random elements from x. If x is an integer greater than 0, then integers between 1 and x are produced. ?0 produces real values between 0.0 and 1.0.

7. repeat e is now an infinite loop. The only way to terminate evaluation of repeat is with break.

8. The control structure

   $e_1 \setminus e_2$

   limits the evaluation of e_1 to at most e_2 results. For example,

   ```
   every writes(?100 \ 3 )
   ```

   outputs three random integers between 1 and 100.

9. Version 4 provides a control structure for constructing generators at the expression level. This control structure, termed *repeated evaluation*, is

   ```
   |e
   ```

   and repeatedly evaluates e. For example,

   ```
   every writes(|(1 to 3))
   ```

   outputs the infinite sequence

   ```
   123123123...
   ```

Repeated evaluation terminates if an evaluation of e fails to produce any result. Each time 1 to 3 is evaluated in the above example, it produces at least one result (three results, in fact), so that evaluation of

```
|(1 to 3)
```

produces an infinite sequence of results. However,

```
|(1 = 0)
```

terminates immediately, and

```
|read()
```

terminates when read fails.