Seque: An Experimental Language for Manipulating Sequences*

Ralph E. Griswold

TR 83-16

November 30, 1983

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS81-01916.

Seque: An Experimental Language for Manipulating Sequences

1. Introduction

A companion report [1] presents a notation for describing and manipulating sequences. This notation is intended as a basis for programming language facilities for treating sequences as data objects. In particular, it is oriented toward sequences that are produced over a period of time by computations as well as the more traditional storage-oriented sequences, where the elements of the sequences exist in memory. This report describes an experimental programming language called Seque. Seque is embedded in the Icon programming language [2].

Embedding new features in an existing language is much easier than designing and implementing a new language from scratch. This approach is particularly appropriate here, since the ideas are experimental and likely to be changed as the result of experience with their use. By embedding the new features in an existing language, a full repertoire of computational facilities is available to support the new facilities without any extra implementation effort. Icon is particularly suitable for the embedding language, since its expression mechanism is capable of producing sequences of results, which is closely related to the new features. Furthermore, Icon has the necessary facilities for creating and manipulating sequences as data objects.

The implementation of Seque is accomplished via a preprocessor that translates new syntactic constructions into standard Icon with calls to procedures in a runtime support library. The implementation is described in a separate report [3].

One disadvantage of embedding Seque in Icon is that concessions have to be made to the existing syntax of Icon. As a result, some aspects of the notation for manipulating sequences are represented in slightly awkward ways. More seriously, some of the semantic features of Seque do not co-exist well with those of Icon. This leads to potential problems in programming, which are discussed in Section 3.

2. Seque Language Features

The language features of Seque are based on the concepts described in [1] with a few additions and extensions for programming considerations. The features from [1] are represented in Seque as described in the following section.

2.1 Basic Features

Sequences are data objects of type `Sequence`. In the sections that follow, identifiers that begin with uppercase letters are used to indicate sequence-valued expressions.

The following global identifiers have predefined sequences as values:

`Phi`	$\Phi \equiv \{\}$
`Izero`	$\mathcal{I}_0 \equiv \{0, 1, 2, 3, \ldots\}$
`Iplus`	$\mathcal{I}_+ \equiv \{1, 2, 3, 4, \ldots\}$

The following expressions are available for manipulating sequences:

feature	*formal notation*	*Seque syntax*
explicit sequence	$\{x_1, x_2, x_3, \ldots\}$	seq{x1, x2, x3, ... }
concatenation	$X \oplus Y$	Cat(X, Y)
subsequence	$X_{i:j}$	Subseq(X, i, j)
pre-truncation	$X \downarrow i$	X %% i (%X is an abbreviation for X %% 1)
post-truncation	$X \uparrow i$	X ^^ i
generator	$[I : \lambda(j)X]$	gen[: I : lambda(j) X]
sequence length	$\mid X \mid$	Length(X)
element selection	$X \mathbin{!} i$	X!i
reduction	$Red_{\circ}(X)$	Red(X, p)

Except for sequence length, element selection, and reduction, these expressions all produce sequences.

The precedence and associativity of ^^ is the same as that of ^, and the precedence and associativity of %% is the same as that of % . The operator ! associates to the left and has a precedence greater than that of \ but less than that of unary operators.

A few points deserve note:

- Expressions can be used in all places where identifiers are shown above, except for the identifiers in lambda expressions. An example is

  ```
  seq{i, i + 1, i - 1, i + 2, i - 2}
  ```

- gen, lambda, and seq are reserved words.
- The generation sequence and the lambda expression are optional in generators. All the following forms are allowed:

  ```
  gen[X]
  gen[ : I : X]
  gen[lambda(i) X]
  ```

  Note the additional colon in the Seque representation of generators. It is necessary to avoid syntactic ambiguities.
- The second argument in reduction must have a procedure value or a string value that represents a binary operator or procedure. Examples are

  ```
  Red(I, "+")
  Red(S,trim)
  ```

- Sequences need not be homogeneous with respect to type and sequences can be elements of sequences. An example is

  ```
  seq{"a", 1, seq{1, 2}}
  ```

- The argument X in X ^^ i, X %% i, %X, and X!i is limited to at most one result.
- X!i is not a variable. For example, X!i := x results in runtime Error 111.
- The *define* facility described in [1] is not supported in Seque because of the availability of Icon procedures. See Section 2.3 for a description of recurrence declarations, however.

2.2 Additional Features

The following procedures related to sequences also are available in Seque:

- Compress(X) converts a sequence containing sequences as elements to a sequence of scalar values. For example, if

  ```
  I := seq{1, seq{2, 3, seq{3, 4}}, 4, 5, seq{seq{seq{5, 6}, 7}, 8}}
  ```

  then Compress(I) produces a sequence that corresponds to

  ```
  seq{1, 2, 3, 3, 4, 4, 5, 5, 6, 7, 8}
  ```

- Copy(X) creates a copy of the sequence X.
- Empty(X) succeeds and produces X if X is an empty sequence but fails otherwise.
- Image(X, i) produces a string image of X, limited to i elements. The default for i is 5.
- Read(f) produces a sequence of values resulting from reading file f.
- Trace(X, i) writes the result of Image(X, i) and produces the value of X.
- Write(X) writes the elements of X with separating linefeeds.
- Writes(X) writes the elements of X without separating linefeeds.

In addition, gen{*expr*} produces a sequence corresponding to the Icon result sequence for *expr*. For example,

```
gen{1 to 5}
```

produces a sequence that is equivalent to

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

To differentiate gen[...] and gen{ ... }, the former is referred to as a *Seque generator* and the later is referred to as an *Icon generator*.

2.3 Recurrence Declarations

Recurrence relations provide a natural and intuitive way of specifying many commonly used sequences. The well-known Fibonacci sequence is typical:

$$Fib(1) = 1$$

$$Fib(2) = 1$$

$$Fib(i) = Fib(i-1) + Fib(i-2), \; i = 3, 4, \ldots$$

A more complicated nested recurrence from [4] is:

$$G(i,k) = 0, \; i < 1$$

$$G(i,k) = i - G(G(i-k,k),k), \; i = 1, 2, 3, \ldots; \; k > 0$$

Note that this procedure has a parameter, k, whose value characterizes a particular recurrence in a family of recurrences. Note also that G is defined to be a constant for all values of i less than 1, independent of k.

Many recurrences can be characterized in terms of at most five components:

1. A generation variable, i, that takes on values from $\mathcal{I}_+$.
2. A fixed number j of initial values for $i = 1, 2, \ldots j$.
3. A constant value for $i < 1$.
4. A generation expression *expr* in the generation variable i.
5. Parameters that appear in *expr* that allow the specification of a particular recurrence from a family of recurrences.

Seque provides a recurrence declaration for those recurrences that can be characterized in the form described above. This declaration produces a procedure that generates the corresponding sequence, using tabulation techniques for efficient computation [2].

A recurrence declaration has the form

```
recur name ( generation variable ; [ parameters ] ; [ constant ] ; [ initial values] )
   expr
end
```

As in other declarations, **recur** is a reserved word. The name identifies the sequence. The generation variable is an identifier that appears in *expr* and takes on values from $\mathcal{I}_+$. The parameters consist of a list of identifiers separated by commas. The constant value is used as the value of any instance of the recurrence that has not been previously computed in the generation process. This usually occurs for values of the generation variable that become less than 1 in the computation (see $G(i, k)$ above), but may also occur for previous uncomputed combinations of the generation variable and the parameters. The initial values consist of a list of expressions $expr_1$, $expr_2$, ... that provide the initial values of the sequence. Note that the parameters, constant, and initial values are all optional.

The following recurrence declaration for the Fibonacci sequence provides an example:

```
recur Fib(i; ; ; 1, 1)
   Fib(i - 1) + Fib(i - 2)
end
```

The nested recurrence given above illustrates the use of a default value and a parameter:

```
recur G(i; k; 0; )
   i - G(G(i - k, k), k)
end
```

The invocation of a procedure produced by such a declaration is typically used in the form

```
F := gen{Fib()}
```

which assigns to F an expression that generates the Fibonacci sequence. Note that the generation variable is not specified in the call — it is merely part of the definition. Any parameters in the recurrence declaration are specified as arguments in the call, however. An example is

```
G2 := gen{G(2)}
```

which supplies the value 2 for the parameter k and produces a sequence corresponding to

```
seq{1, 2, 2, 2, 3,  ... }
```

Recurrences are not limited to the generation of integer values. For example, the "Fibonacci strings" as given in [5] are declared by

```
recur Fibs(i; ; ; "a", "b")
   Fibs(i - 1) || Fibs(i - 2)
end
```

Another example is given by the "chaotic strings", derived from the chaotic integer sequence given in [6].

```
recur Qs(i; ; ; "a", "ab")
   Qs(i - *Qs(i - 1)) || Qs(i - *Qs(i - 2))
end
```

The constant and initial values can be expressions. These expressions are inserted in the code for the corresponding procedure. For example, the initial values can be parameterized, as in

```
recur Qs(i; x, y; ; x, x || y)
   Qs(i - *Qs(i - 1, x, y), x, y) || Qs(i - *Qs(i - 2, x, y), x, y)
end
```

Thus,

```
gen{Qs("c","d")}
```

specifies the sequence of chaotic strings with the initial values c and cd. Note that the parameters x and y occur as arguments to all instances of Qs. This is necessary, since the tabulation of computed values depends not only on the values of the generation variable but also on the values of the parameters.

Note that sequences specified by recurrence declarations produce values over $\mathcal{I}_+$. No other generation sequence can be specified. The recurrence also must be self contained. For example, mutual recurrences, such as the following pair of "married sequences" [6] cannot be handled by recurrence declarations:

$$F(0) = 1$$

$$M(0) = 0$$

$$F(i) = i - M(F(i - 1)), i > 0$$

$$M(i) = i - F(M(i - 1)), i > 0$$

2.4 Expression Evaluation

There are two kinds of expression evaluation available in Seque programs: ordinary Icon evaluation with the usual operations and functions, and Seque evaluation, in which expressions are limited to at most one result and operations and functions are extended to apply to values of type Sequence. The type of evaluation used depends on context, which is determined as follows:

- Seque evaluation applies in procedures that are declared with the reserved word procedure. Such procedures are called *Seque procedures*.
- Within a Seque procedure, Seque evaluation applies in all expressions with operator syntax and in function and procedure calls in which the function or procedure is given by an identifier that begins with an initial lowercase letter[1]. Note that Seque evaluation applies to the built-in repertoire of Icon.
- Icon evaluation applies in procedures that are declared with the reserved words icon procedure. Such procedures are called *Icon procedures*. For example,

  ```
  icon procedure main()
            :
  end
  ```

  declares the main procedure to be an Icon procedure.
- Icon evaluation applies in recurrence declarations.
- Seque evaluation applies in explicit sequences and Seque generators, regardless of the context in which such expressions occur.
- Icon evaluation applies in Icon generators, regardless of the context in which such expressions occur.
- Icon evaluation applies all assignment operations, regardless of where they occur.
- Control structures behave in the way they do in Icon, regardless of the context in which they appear.

Note that it is possible to write Seque programs using exclusively Seque or Icon evaluation. Icon evaluation is the exception, however. Unless otherwise stated, Seque evaluation is assumed in the remainder of this report.

In Seque evaluation, arguments of operations and functions are limited to at most one result. Consequently,

```
every write(repl("a" | "b", 2 | 3))
```

only writes aa, instead of aa, aaa, bb, and bbb as in Icon evaluation. However, in a Seque procedure, Icon evaluation can be obtained by using function-valued identifiers with initial uppercase letters, as in

[1]Procedure and function calls may have at most four arguments in Seque evaluation contexts This is an implementation limit only.

```
Xwrite := write
Xrepl := repl
every Xwrite(Xrepl("a" | "b", 2 | 3))
```

which writes **aa**, **aaa**, **bb**, **bbb**. Since Seque evaluation applies only to functions given by an identifier with an initial lowercase letter, Icon evaluation can also be obtained by disguising the name, as in

```
every (write)((repl)("a" | "b", 2 | 3))
```

which also writes four values as in the example above. Similarly, Icon evaluation for operators can be obtained by using string invocation [7], as in

```
every write("+"(find(s1, s2), find(s1, s2)))
```

In Seque evaluation, operations and functions, when applied to arguments whose values are sequences, produce sequences of the corresponding operations. That is, the operations are "distributed" over the corresponding sequences. For example,

```
gen{1 to 3} + gen{1 to 3}
```

produces a sequence corresponding to

```
seq{2, 4, 6}
```

As defined in [1], the length of such a sequence is the minimum of the lengths of the sequences of the arguments. Thus

```
gen{1 to 3} + gen{1 to 1000}
```

produces the same sequence as the example above. Note that this expression is erroneous in an Icon evaluation context, since the Icon addition operation does not accept sequences as arguments.

In contexts where values of type Sequence are expected, scalar values are automatically converted to the corresponding unit sequences. For example,

```
Cat(3, Iplus)
```

produces a sequence that corresponds to

```
seq{3, 1, 2, 3, 4, ... }
```

In the case of operations and functions that are "polymorphic" with respect to sequences, scalar values are coerced to sequences only if at least one argument is a sequence. For example,

```
1 + 3
```

produces the scalar value 4, but

```
seq{1, 2} + 3
```

produces a value of type Sequence corresponding to the unit sequence

```
seq{4}
```

There is one important exception to the coercion of scalars to corresponding unit sequences in sequence contexts: the null value is coerced to the empty sequence, Phi. Thus,

```
X!i
```

fails if X is null-valued. This interpretation of the null value in sequence contexts causes uninitialized variables to be treated as empty sequences rather than as unit sequences containing null values.

2.5 The Evaluation of Sequences

The elements of sequences can be specified by values, as in

```
seq{1, 3, 17}
```

or they can be produced by the evaluation of expressions, as in

```
gen{1 to 1000}
```

The elements of a sequence are produced only when then are needed. This is essential to the use of infinite sequences.

When a sequence is created, none of its elements is computed. As elements are required, they are computed and stored. Thus previously computed elements can be obtained without recomputing them. The evaluation of the *i*th element of a sequence requires the evaluation of elements 1, 2, ..., *i−1*.

If an element is produced by an expression that has side effects, these side effects do not occur until the element is first evaluated. Subsequent references to that element do not cause its producing expression to be re-evaluated.

Normally, the elements of a sequence are not specified by expressions that have side effects, so this aspect of the evaluation of sequence elements is not noticeable. Consider, however,

```
X := seq{write(1), write(2), write(3)}
```

The creation of this sequence and its assignment to X does not evaluate the calls of the write function and there is no output. However, the expression X!2 causes the first and second elements of X to be evaluated, which cause 1 and 2 to be written. Subsequent references to X!1 or X!2 do not cause output, however.

All operations that produce sequences, such as Cat(X, Y) and Copy(X), create sequences that are distinct from their arguments. None of the elements in these sequences is evaluated when the new sequences are created, nor are values that were previously computed and stored used in the new sequences. Instead, the values of the elements in the new sequences are produced as needed from the original expressions.

When a value of type Sequence is created, any identifiers in it are bound to the values they have at the time the sequence is created. Thus,

```
x := 3
X := seq{x, x + 2, x + x}
x := 0
Write(X)
```

writes 3, 5, and 6.

If an element of a sequence is specified by an expression that fails when it is evaluated, no corresponding value is produced in the sequence. For example,

```
seq{1, \k, 2}
```

is equivalent to

```
seq{1, k, 2}
```

if the value of k is nonnull, but is equivalent to

```
seq{1, 2}
```

if the value of k is null.

3. Programming Considerations

Because of the important distinction between values of type Sequence and other values, it is helpful to use identifiers with initial uppercase letters as a mnemonic device to indicate where sequences are expected in Seque programs.

The runtime support library uses identifiers that end in underscores for internal purposes. To avoid possible name collisions, such identifiers should not be used in Seque programs.

As a general rule, it is not advisable to mingle Icon-style use of result sequences with Seque sequences. Where Icon result sequences are needed, the gen {*expr*} expression should be used. For example,

```
I := gen{1 | (1 to 3) + (2 to 4)}
```

provides a natural bridge between Icon result sequences and Seque sequence values, but

```
I := seq{1, (1 to 3) + seq{2 to 4}}
```

does not produce the sequence that it may appear to represent.

It is advisable to avoid the use in sequences of expressions that may have side effects when they are evaluated.

Obviously care must be taken when dealing with infinite sequences. For example, Length(Iplus) does not terminate. There is a more serious problem with termination, however. Consider

```
Ip := gen[I!i > 0]
```

for an arbitrary sequence I. Although Ip may be used in a context in which only one value is used, the generator itself may continue indefinitely without producing that value.

Seque employs a heuristic to avoid this problem in the case of finite sequences. In a Seque generator, this heuristic terminates generation from a sequence X if X!i fails[1].

4. Examples

Seque Generators

Seque generators provide ways of expressing many sequences in concise ways. For example,

```
gen[X]
```

is the repeated concatenation of X with itself.

The generation sequence in a Seque generator need not be integer-valued. An example is

```
gen[:Read():lambda(s) p(s)]
```

which is a sequence consisting of the procedure p applied to the lines of standard input.

Use of Procedures

Procedures provide a way of parameterizing Seque generators so that they can be used in a variety of contexts. For example, the following procedure "filters" X, returning a sequence in which all instances of x are omitted:

```
procedure Remove(X, x)
   return gen[x ~=== X!i]
end
```

The generator uses the fact that Icon comparison operations return the value of their right argument if they succeed. Note that comparisons that fail contribute nothing to the resulting sequence.

Another example of a parameterized generator is

```
procedure Interleave(X, Y)
   return gen[seq{X!i, Y!i}]
end
```

[1]This heuristic is analogous to the termination of the Icon expression |*expr* if *expr* fails.

which produces the result of interleaving the elements of X and Y. If one sequence is longer than the other, the trailing elements of the longer sequence are appended at the end of the result.

The library of "built-in" Seque procedures has been kept small deliberately, since many operations on sequences can be formulated as procedures in Seque. An example is the reversal of a (finite) sequence:

```
procedure Reverse(X)
   if Empty(X) then return X
   else return Cat(Reverse(%X), X!1)
end
```

Reversal can also be done iteratively, as follows:

```
procedure Reverse(X)
   local Y
   Y := Phi
   while Y := Cat(X!1, Y) do
      X := %X
   return Y
end
```

Reduction

An interesting use of reduction is illustrated by the following program, which writes the maximum of numbers given in standard input:

```
procedure main()
   write(Red(Read(), ">"))
end
```

Like Remove above, this program uses the value returned by successful comparisons. Note that unsuccessful comparisons do not terminate the reduction.

Variations on Cross Product Evaluation

As described in [1], cross-product evaluation can be constructed using nested generators. An example is

```
procedure Kross(X, op, Y)
   return gen[gen[lambda(j) op(X!j, Y!i)]]
end
```

This procedure applies op to X and Y to produce a cross-product sequence in which the elements of the first sequence, X, are selected first. For example, the result of

```
U := gen{!"ABC"}
L := gen{!"ab"}
Write(Kross(U, "||", L))
```

produces

```
Aa
Ba
Ca
Ab
Bb
Cb
```

Note that Icon-style cross-product evaluation is given by

```
gen[gen[lambda(j) op(X!i,Y!j)]]
```

An Example of Indexing Sequences

As defined in [1], an indexing sequence consists of the indices of a sequence for which some condition is satisfied. A use of indexing sequences arises in [4], where the question of the locations of the zeroes of a difference sequence arises for the recurrence $G(i, k)$ given in Section 2.3. The following program illustrates the computation of the indexing sequence for different values of the parameter k:

```
procedure main(a)
   local j, k, n, Gk
   j := a[1] | 5
   n := a[2] | 5
   every k := 1 to j do {
      write("parameter=", k)
      Gk:= gen{G(k)}
      Write(Zeros(%Gk - Gk) ^^ n)
      write("——")
      }
end

recur G(i;k;0;)
   i - G(G(i - k, k), k)
end

procedure Zeros(l)
   return gen[if l!i = 0 then i]
end
```

In this program, the values of k and n can be specified on the command line when the program is run. The default values for k and n are 5. The sequence Gk is given by a recurrence declaration as noted previously. The sequence

```
%Gk - Gk
```

is the "backward difference" sequence for Gk — that is, it consists of values of the form $G(i + 1, k) - G(i, k)$. The procedure Zeros(l) produces a sequence consisting of the indexes for which l is zero.

Output of this program for k = 2 and n = 5 is

```
parameter=1
1
4
6
9
12
——
parameter=2
2
3
8
9
12
——
```

5. Running Seque

Seque is run by the command

```
Seque [options] file [-x] [arguments]
```

where *file* is the name of a file containing a Seque program. Such file names must end in the suffix .seq. The result of running Seque is an executable version of the Seque program in the base name file corresponding to deleting the .seq suffix from the source-program file name. For example,

```
Seque model.seq
```

produces an executable file model.

The −x option, which occurs *after* the program file name, is analogous to the corresponding option for Icon [8] and causes the program to be executed automatically after it is translated. Any arguments that appear after the −x option are passed as a list to the main procedure.

There are two options that may appear before the file name:

−i saves the result of preprocessing the Seque program in a file with the base name and the suffix .icn. For example,

```
Seque -i model.seq
```

produces a file model.icn. It is the .icn file that is actually run, and having the source available is useful for locating errors from runtime diagnostics.

−g causes the −x option to be effective even if there are syntax errors in the Seque program. This option is intended for system debugging.

In addition, any other options that appear before the file name are passed onto Icon and used when the .icn file is translated. For example,

```
Seque -t -u model.seq -x
```

causes tracing to be set and undeclared identifiers to be noted.

Seque is available only with Version 5.8 of Icon including the experimental extensions [7], running on the VAX-11 under UNIX[1].

6. Potential Problems in Using Seque

The major problem likely to be encountered with the use of Seque is the computational resources that it requires. In Seque evaluation, Icon operations and functions are processed by procedures in the runtime library instead of being evaluated directly. Co-expressions, which are used in values of type Sequence, are large and impose a substantial overhead in terms of storage throughput.

To improve runtime speed of Seque programs:

- Do not create sequences unnecessarily.
- Do not use parameters in recurrence declarations unless they are necessary.
- Use Icon procedures instead of Seque procedures where appropriate. This is often possible even in cases where sequences are being manipulated.

The heuristic used to prevent endless generation is not always effective in nested generators. It is advisable to limit generation sequences explicitly in such cases, as in

```
gen[ : Iplus ^^ 100 :  X ]
```

The use of co-expressions in sequences precludes the use of the Icon control structures break and next in expressions that appear in explicit sequences, Seque generators, and Icon generators. Similarly, the scope of identifiers in such expressions is confined to the sequence. See Reference 9 for details.

[1]UNIX is a trademark of Bell Laboratories.

Because co-expression stack overflow is not checked in Icon, it is possible that Seque programs may malfunction mysteriously. If this problem arises, it is likely to be related to excessive recursion. The size of co-expression stacks can be specified as an option to Seque. See Reference 8.

Since Seque translates a Seque program into an Icon program, which is then translated and run, runtime error messages refer to the Icon program, not the Seque program. The use of co-expressions in the implementation also tends to obscure the locality of runtime errors.

Linguistically, Seque evaluation and Icon evaluation do not co-exist comfortably. This problem is most likely to be manifested as apparent malfunction of sequence computations or mysterious runtime errors. Observing the precautions given in Section 3 is strongly advised.

The implementation of Seque is somewhat arcane. It is very likely that it contains errors.

Acknowledgements

Dave Hanson, Bill Mitchell, and Steve Wampler provided a number of helpful suggestions on the design of Seque and on the presentation of the material in this report.

References

1. Griswold, Ralph E. *The Description and Manipulation of Sequences*, TR 83-15, Department of Computer Science, The University of Arizona. 1983.

2. Griswold, Ralph E. and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1983.

3. Griswold, Ralph E. The Implementation of an Experimental Language for Manipulating Sequences, TR 83-20, Department of Computer Science, The University of Arizona. 1983.

4. Downey, Peter J. and Ralph E. Griswold. *On a Family of Nested Recurrences*, Technical Report TR 82-18, Department of Computer Science, The University of Arizona. 1982.

5. Knuth, Donald E. *The Art of Computer Programming*, Vol. 1. Addison-Wesley Publishing Company, Reading, Massachusetts. 1968. p. 85.

6. Hofstadter, Douglas R. *Gödel, Escher, Bach: An Eternal Golden Braid.* Basic Books, New York. 1979. pp. 137-138.

7. Griswold, Ralph E. and William H. Mitchell. *Experimental Extensions to Version 5.8 of Icon*, technical report, Department of Computer Science, The University of Arizona. 1983.

8. *ICONT(1)*, manual page for *UNIX Programmer's Manual*, Department of Computer Science, The University of Arizona. 1983.

9. Griswold, Ralph E. and Michael Novak. "Programmer-Defined Control Operations", *The Computer Journal*, Vol. 26. No. 2 (May 1983). pp. 175-183.