Reference Manual for the Seque Programming Language*

Ralph E. Griswold and Janalee O'Bagy

TR 85-4

March 21, 1985

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants DCR-840131 and DCR-8320138.

Reference Manual for the Seque Programming Language

1. Introduction

Seque is an experimental language designed to explore the possibility of programming with sequences as data objects. Such data objects are called *streams*. A separate report [1] presents the fundamental concept of a stream and provides an introduction to programming with streams in Seque. This document is a reference manual for the Seque programming language.

Seque is implemented via a preprocessor that translates Seque syntax to standard Icon syntax. Icon, as the embedding language, provides versatility and expressive power in the specification of streams.

Most of the repertoire of Icon is available in Seque, and both the string capabilities and the semantics of expression evaluation are fully exploited. This report requires an understanding of Version 5 of Icon [2].

2. Basic Concepts

A *stream* is a data object that is capable of producing values. The values are produced on demand and, as such, constitute a sequence of values produced in time. The values that a stream can produce are specified by a sequence of computations.

Streams retain the values that they have computed. These values can then be accessed by *position*, according to the order in which they were computed. Thus, there is the notion of the first, second, ... *n*th values of a stream. Positional access does not require the recomputation of values that have already been computed. However, if a value that has not been computed is referenced by position, the necessary computations are performed to produce this value.

Streams may be finite or infinite. The *length* of a stream, denoted by |X|, is the number of values a stream is capable of producing.

Any variable in Seque that is not stream-valued is called a *scalar*. Thus integers, strings, and lists are scalars in this sense.

The values of a stream are produced in time and are specified by the expressions that are capable of producing them. Throughout this report, the following notation is used to denote the values that a stream is capable of producing:

$$S \rightarrow < v_1, v_2, v_3, \ldots, v_n, \ldots >$$

The arrow is read "may produce."

3. Built-in Streams

There are three predefined streams in Seque:

```
Phi   → < >
Izero → < 0, 1, 2, 3, 4, ... >
Iplus → < 1, 2, 3, 4, 5, ... >
```

Phi is the empty stream, which consists of no values. The integer streams Iplus and Izero are useful in the construction of other streams. Note that |Phi| = 0, while Izero and Iplus are infinite.

4. Creating Streams

4.1 Explicit Streams

Streams can be created explicitly by enclosing a sequence of expressions in braces:

{ $expr_1$, $expr_2$, $expr_3$, ..., $expr_n$ }

The expressions are evaluated from left to right and each may produce a number of results. For example,

```
{ 1 to 3, !"abcd" } → < 1, 2, 3, a, b, c, d >
```

An expression in a stream also may fail to produce a value. For example, if s1 is not a substring of s2, then

```
Indices := { 0, find(s1, s2) > 10 } → < 0 >
```

Notice that streams may be non-homogeneous with respect to type. Streams may also have streams as values, as in

```
{ { 0 to 100 by 2 } , { 1 to 100 by 2 } }
```

which is a stream of two streams consisting of first the even and then the odd integers up to 100.

Sometimes it is useful to be able to have an infinite stream in which only the first few values are specified. This may be done by placing ellipses at the end of an explicit stream, as in

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

which is an infinite stream whose first three values are 1, 2, and 3. The remaining values are null values.

4.2 Derived Streams

Streams may be specified by the computation of an expression over a given sequence of values. Such streams are called *derived streams* and have the following syntax:

```
[ : S : lambda(s) expr ]
```

S is the *controlling stream* and may be any stream-valued expression. The variable s is the *bound variable* and takes on successive values from the controlling stream. All occurrences of s in *expr* are bound. The expression *expr* is evaluated for every value in the controlling stream. For example,

```
Lowers := { !&lcase }
[ : Lowers : lambda(s) s || s ] → < aa, bb, cc, dd, ee, ... >
```

When not specified, the controlling stream defaults to the value of the keyword &Control, which is initially set to Iplus. The bound variable defaults to i. Some examples of derived streams using the default controlling stream Iplus are:

```
[ i - 2 ]  → < -1, 0, 1, 2, 3, 4, 5, 6,  ... >
[ i % 4 ] → < 1, 2, 3, 0, 1, 2, 3, ... >
[ "b" ] → < b, b, b, b, b, ... >
[ repl("a", i) ] → < a, aa, aaa, aaaa, ... >
```

If the expression in the controlling stream is stream-valued, the derived stream produces the repeated concatenation of this stream. For example,

```
[ { 1, 2, 3 } ] →  < 1, 2, 3, 1, 2, 3, 1, 2, 3, ... >
```

Unbound variables may occur in the derived stream expression. The values of these variables are inherited from the context in which the derived stream occurs. For example,

```
[ i + j ] →  < 1 + j, 2 + j, 3 + j, ... >
```

Section 10 explains in more detail the scoping and binding rules for variables occurring in stream expressions.

Derived streams may be nested. As an example, consider the following expressions, which produce the Cartesian product of the values of two streams:

```
U := { "A", "B", "C" }
L := { "a", "b", "c" }
[ : U : lambda(s1) [ : L : lambda(s2) s1 || s2 ] ]
```

For each value of U, the concatenation is performed with each value of L, producing

```
Aa, Ab, Ac, Ba, Bb, Bc, Ca, Cb, Cc
```

The computation of a derived stream terminates when the values of the controlling stream are exhausted. See Section 11 for an explanation of a heuristic that also terminates derived streams.

4.3 Concatenation

The concatenation of two streams is denoted by X -> Y and is a stream consisting of the stream X followed by the stream Y. For example,

```
{ 1, 2, 3 } -> { "a", "b", "c" } → < 1, 2, 3, a, b, c >
```

Concatenation may involve streams that are infinite. Thus

```
{ -3, -2, -1 } -> Izero → < -3, -2, -1, 0, 1, 2, 3, ... >
```

4.4 Sectioning

Streams consisting of portions of existing streams are created by sectioning operations. Sectioning is denoted by X{i:j}, which produces a stream consisting of the values i through j of X, inclusive, where $1 \le i \le j \le |X|$. For example,

```
Iplus{3:5} → < 3, 4, 5 >
```

In X{i:j}, if $i < 1$ or $i > |X|$, the resulting stream is empty.

A common form of sectioning involves eliminating either initial or final values of a stream. These forms of sectioning are called pre-truncation and post-truncation, and are denoted respectively by

$$X \,\%\% \, i \;\rightarrow\; < x_{i+1}, x_{i+2}, x_{i+3}, \ldots, x_{|X|} >$$
$$X \setminus\setminus \, i \;\rightarrow\; < x_1, x_2, x_3, \ldots, x_i >$$

For example,

```
Izero %% 10  → < 11, 12, 13, 14, ... >
```

and

```
Iplus \\ 10 → < 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 >
```

In both pre-truncation and post-truncation, the empty stream is produced when $i < 1$. %X is an abbreviation for X %% 1.

4.5 Coercion of Scalar Values to Streams

In all situations where a stream is expected, a scalar value is coerced to the corresponding unit stream. For example,

```
{ 1, 2, 3 } -> 4 → < 1, 2, 3, 4 >
```

5. Values of Streams

5.1 Referencing Values

The values of a stream are computed as they are needed. The values of a stream are referenced by position in this computational sequence by X!i, which produces the ith value of X. For example, if

```
Alpha := { !lcase, !ucase }
```

then Alpha!2 is b and Alpha!29 is C.

The values of a stream are saved as they are produced; values are never computed more than once. X!i requires values 1 through i of the stream X be computed, if necessary.

If the integer i in X!i is negative or larger than the length of the stream, the reference fails. Notice that since the length of a stream is not known *a priori*, the latter case forces computation of all values.

There are also three functions related to accessing values of a stream. The function Index(X) returns the position of the most recently computed value of stream X. The function Next(X) computes the next value of stream X and returns it as a value. The function Last(X) computes all the values of X and returns the last value of the stream. Finally, the function Length(X) computes the length of stream X, causing all values of X to be computed.

5.2 Element Generation

The values of a stream are generated by the operation @. For example,

```
Comp := { "Bach", "Beethoven", "Brahms" }
every write(@Comp)
```

writes the strings Bach, Beethoven, and Brahms.

5.3 Assignment of Stream Values

As mentioned previously, the values of a stream are saved as they are computed. A previously computed value may be changed by assignment. For example, consider the following expressions:

```
X := {"r", "a", "d", "a", "r" }
       :
       :
X!1 := X!5 := "m"
```

Initially the concatenation of the values of X forms the word radar. After the assignments, it forms the word madam.

Notice that self-referential streams may be created using assignment. Such a stream is recursively infinite, as in the example below:

```
A := { "a", "b" }
A!2 := A
```

After the assignment, A is a stream whose first value is a and whose second value references itself. The structure of A corresponds to

```
{ "a", { "a", { "a", { ... } } } }
```

6. Seque Functions on Streams

Swrite(X)

The function Swrite(X) writes the values of a stream, one per line, to standard output. Swrite(X) expects the values produced by X to be scalars whose types are legitimate for writing. Note that

```
Swrite(X) ≡ every write(@X)
```

Simage(X,i)

The function Simage(X, i) produces a string that is the image of X limited to i results. The image of a stream is the image of its values surrounded by braces. The default value for i is 5. For example,

```
Simage(|plus)
```

produces the string

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

Simage constructs the image of a stream recursively. For example,

```
Message := { { !"Hello" }, { !"World"} }
Simage(Message)
```

produces the string

```
{ { "H", "e", "l", "l", "o" }, { "W", "o", "r", "l", "d"} }
```

Compress(X)

The function Compress(X) "flattens" a stream that contains streams as values, producing a stream of scalars. For example,

```
Simage(Compress(Message))
```

produces the string

```
{ "H", "e", "l", "l", "o", "W", "o", "r", "l", "d" }
```

Copy(X)

The function Copy(X) creates a copy of the stream X. The stream created by Copy produces the same values as the original stream X.

Empty(X)

The function Empty(X) determines if a stream is empty. Empty(X) succeeds and produces X if X is empty and fails otherwise.

Red(X,f)

Reduction over a binary operation f is produced by the function Red(X, f). Reduction applies f to the values of stream X and produces a stream of the accumulated results. The second argument f may be a procedure or a string representing a binary operation.

For example, let l be the stream { 1, 2, 3, 4 }. Reduction of l over addition produces a stream that consists of a running sum of the values in l. That is,

```
Red(l, "+") → < 1, 3, 6, 10 >
```

Similarly, if the operation is concatenation, the result is the concatenation of all the values of produced by the stream. For example, if

```
X := { "a", "b", "c", "d" }
```

then

```
Red(X, "||") → < a, ab, abc, abcd >
```

7. Operations Extended over Streams

7.1 Scalar Operations on Streams

The unary and binary operators of Icon are implicitly extended over streams in Seque. In the case of a binary operation, the operation is distributed pairwise over the values of the streams, producing the dot product of the operands. In general, if ⊕ is a binary operator, then

$$X \oplus Y \rightarrow < x_1 \oplus y_1, x_2 \oplus y_2, x_3 \oplus y_3, \ldots >$$

For example,

```
{ 1, 2, 3 } + { 4, 5, 6 }  → < 5, 7, 9 >
```

If the lengths of the streams are unequal, the operation is performed for the first i values, where i is the minimum of the two lengths, and the resulting stream has length i. Thus,

```
Iplus + { 4, 5, 6 } → < 5, 7, 9 >
```

If one of the operands is a scalar, it is coerced to the corresponding unit stream. That is,

```
{ 1, 2, 3 } + 4 ≡ { 1, 2, 3 } + { 4 } → < 5 >
```

Operations also are applicable to streams that have streams as values. Consider the following example:

```
{ { 1, 2, 3 }, 6, 7, 8 } + { 1, 2, 3 } ≡ { { 1, 2, 3 } + 1, 6 + 2, 7 + 3 } ≡ { { 2 }, 8, 10 }
```

Unary operations defined for scalars also are applicable to streams, and in a similar manner, are applied successively to each value in the stream. For example, if

```
X := { 1, 2, 3, 4, 5 }
```

then

```
-X → < -1, -2, -3, -4, -5 >
```

Similarly, a stream consisting of the lengths of the lines from the file f is

```
*{ !f }
```

The semantics of a comparison operator distributed over streams is affected by the meaning of comparison operations in Icon. Recall that operations such as i=j produce their right operand, j, if the comparison succeeds, but fail otherwise, producing no result. If two streams are compared in this way, the resulting stream terminates on the first failure of a comparison. For example,

```
{ 2, 3, 4} = { 2, 8, 4 } ≡ { 2 = 2, 3 = 8, 4 = 4 }  → < 2 >
```

Since the second comparison fails, the computation of the stream terminates, having produced only one value.

7.2 Functions over Streams

Although the Icon operators are distributed over streams, functions are not. The Seque function Compose is available for applying functions to stream arguments. As with operations, the function is applied in parallel to each value in the stream arguments. For example, if Vowels is defined by

```
Vowels := { !"aeiou" }
```

then

```
Compose(repl(Vowels, Iplus)) → < a, ee, iii, ooo, uuuu >
```

The length of the resulting stream is the minimum of the lengths of the argument streams. In the example above, the resulting stream is finite, since the stream Vowels is finite.

Notice that Compose results in the application of a function to the values of streams in parallel. The implicit evaluation mechanism of Icon is not parallel, but rather a first-in, last-out combination of all

possibilities. To achieve this evaluation order with functions on stream arguments, iteration and value generation may be combined. For example,

```
every write(repl(@{ !"abc" }, @{ 1, 2, 3 }))
```

writes the following strings:

```
a
aa
aaa
b
bb
bbb
c
cc
ccc
```

8. 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 [3] 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 lplus.
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 results in a procedure that produces the sequence of values specified by the recurrence relation.

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 lplus. 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 procedure corresponding to such a declaration is typically used in the form

```
F := Fib()
```

which assigns to F a stream whose values constitute 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 := G(2)
```

which supplies the value 2 for the parameter k, so that

```
G2 → <1, 2, 2, 2, 3, ... >
```

Recurrences are not limited to the generation of integer values. For example, the "Fibonacci strings" as given in [4] 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 [5].

```
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,

```
Qs("c","d")
```

specifies a stream of the 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 streams specified by recurrence declarations produce values over Iplus. 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" [5] cannot be handled by recurrence declarations:

$$F(0) = 1$$

$$M(0) = 0$$

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

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

9. Streams and Icon Generators

In Icon, expressions may produce more than one result and are called generators. Generators may produce sequences of values of any type. Streams are data types and so may be produced by an Icon generator, that is, Icon results sequences may contain streams.

To illustrate, if

```
X := { !"abcd" }
```

then the result sequence for

```
X \\ (1 to Length(X))
```

consists of the substreams X \\ 1, X \\ 2, X \\ 3, ... In the context of iteration, the post-truncation operation is invoked for each result in the expression

```
1 to Length(X)
```

The Icon result sequence for the expression consists of the substreams of X and is generated by

```
every write(Simage(X \\ (1 to Length(X))))
```

which writes the strings

```
{ "a" }
{ "a", "b" }
{ "a", "b", "c" }
{ "a", "b", "c", "d" }
```

10. Scoping and Binding Times

Within a stream such as

$$\{ expr_1, expr_2, expr_3, \ldots, expr_n \}$$

or

$$[expr]$$

the environment needed to evaluate the expressions is a component of the stream. The values of all local identifiers occurring in the expressions are copied into this environment. These values are then used during the production of the stream. For example, consider

```
modulus := 3
Multiples := [ if (i % modulus == 0) then i ]
```

Two identifiers occur within the stream expression. The variable i is the bound variable for the derived stream and is associated with the values of Iplus. Assume that modulus is a local identifier in the procedure in which these expressions occur. Then when Multiples is created, the value of modulus is copied into the environment for Multiples. Since the value of modulus is 3,

```
Multiples → < 3, 6, 9, 12, ... >
```

Subsequent assignments to modulus in the enclosing procedure do not affect the stream Multiples, since Multiples has an environment of its own.

Since a stream is created with expressions that represent the potential values of the stream, the meaning of a stream cannot be altered *after* its creation if all expressions involved only refer to local identifiers. As with co-expressions, however, global variables may be used to change potential values during stream computation, or to communicate between two or more streams.

In the example above, if the identifier modulus is global, it is possible to change the values produced by Multiples by assigning another value to the identifier modulus.

11. Termination

Many problems are simplified by the ability of Seque to manipulate infinite streams. However, as noted previously, some computations on infinite streams do not terminate. A simple example is Length(Iplus). Derived streams with infinite controlling streams are subtler in behavior.

In the case of derived streams, Seque employs a general heuristic for determining the logical termination of stream computation. If a positional reference operation occurs in the stream expression, the computation terminates when the reference fails. For example,

```
[ X!i + 10 ]
```

terminates if X!i fails. Although such a derived stream terminates with reference failure, it is better programming practice to write the above stream as

```
[ : X : i + 10 ]
```

Finite controlling streams are preferred, since they avoid the pitfalls of infinite computation.

Although failure of a reference expression such as X!i causes the computation of a derived stream to terminate, the failure of an Icon expression does not terminate the computation of a derived stream. For example, the following derived stream might be thought to consist of the first nine integers:

```
[ 10 > i ]
```

However, the failure of the comparison 10 > 10 does not cause the derived stream to terminate. Consequently, this stream, if used in a context that requires all of its values, does not terminate.

To terminate a derived stream in the case that its expression fails, a caret before the expression can be used:

```
[ ^ expr ]
```

Then

```
[ ^ (10 > i) ] → < 1, 2, 3, 4, 5, 6, 7, 8, 9 >
```

12. Syntax Changes to Icon

With a few exceptions, the features of Icon are available in Seque. Two Icon constructs were changed, however, to accommodate the syntax of Seque.

Seque uses braces and brackets for stream construction, which conflicts with the syntax for list construction and compound expressions in Icon. When using Seque, Icon lists must be constructed as follows:

```
list[ ]
```

The reserved word list is used to avoid conflict with derived streams. Compound expressions, which are surround by braces in Icon, are delimited by the reserved words begin and nigeb in Seque.

In addition to the two syntactic changes above, co-expressions are not available in Seque. Note, however,

that the explicit stream creation of Seque provides features that are very similar to those of co-expressions. All results can be obtained in succession with the stream value generation operation @, or values may be obtained one at a time with the function Next.

13. Programming Considerations

Seque is implemented by translating programs with Seque syntax into standard Icon and linking this program with library routines that implement Seque operations. As a result, the line numbers in translation and run-time error messages correspond to the transformed Seque program, not to the original source code file. This makes tracking even syntax errors difficult.

The Seque run-time library uses procedure names ending in an underscore. To avoid collision with the library routines, Seque procedures should not follow this naming convention.

14. 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 [6] 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 run-time 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 on to Icon and are 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 5.9 of Icon and its the experimental extensions [7], running on the VAX-11 under UNIX[1].

[1] UNIX is a trademark of AT&T Bell Laboratories.

References

1. R. E. Griswold and J. O'Bagy, *Seque: A Language for Programming with Streams*, The Univ. of Arizona Tech. Rep. 85-2, Jan. 1985.
2. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
3. P. J. Downey and R. E. Griswold, "On a Family of Nested Recurrences", *Fibonacci Quarterly 4*, 1 (Nov. 1984), 310-317.
4. D. E. Knuth, *The Art of Computer Programming, Volume I*, Addison-Wesley, 1968, p. 85.
5. D. R. Hofstader, *Gödel, Escher, and Bach; An Eternal Golden Braid*, Basic Books, 1979, pp. 137-138.
6. R. E. Griswold and W. H. Mitchell, *ICONT(X)* manual page for *UNIX Programmer's Manual*, The Univ. of Arizona Tech. Rep., Aug. 1984.
7. R. E. Griswold, R. K. McConeghy and W. H. Mitchell, *Extensions to Version 5 of the Icon Programming Language*, The Univ. of Arizona Tech. Rep. Tech. Rep. 84-10, Aug. 1984.

Appendix

Stream Specifications

{ $expr_1$, $expr_2$, $expr_3$, ..., $expr_n$ }	explicit stream
{ $expr_1$, $expr_2$, $expr_3$, ... }	infinite explicit stream
[:S: lambda(s) *expr*]	derived stream
[:S: lambda(s) ^*expr*]	derived stream with failure termination

Operations

X -> Y	concatentation
X{i:j}	sectioning
X \\ i	post-truncation
X %% i	pre-trunctation
%X	abbreviation for X %% 1
X!i	positional reference
@X	stream value generation

Keywords

&Iplus	read-only version of Iplus
&Izero	read-only version of Izero
&Control	default controlling stream for derived streams, initially Iplus

Functions

Length(X)
Computes the length of stream X, forcing the computation of all values of X.

Next(X)
Computes the next value in stream X and returns it as a value.

Last(X)
Computes all the values of stream X and returns the last value.

Index(X)
Returns the position of the most recently produced value of stream X.

Compose(f, X1, X2, ..., Xn)
Applies the function f to the values in the argument streams in parallel.

Compress(X)
Creates a stream of scalar values from X.

Copy(X)
Creates a copy of stream X.

Red(X,f)
Applies f to the values in stream X and produces a stream of the accumulated results.

Simage(X,i)
Produces a string image of X limited to i results.

Swrite(X)
Writes the values of X, one per line.